VDMJ Design Specification

Author	Nick Battle
Date	12/01/10
Issue	0.8

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] DBGP - A common debugger protocol for languages and debugger UI communication, http://xdebug.org/docs-dbgp.php.

[7] Overture - Open-source Tools for Formal Modelling, http://www.overturetool.org/.

[8] Modelling and Validating Distributed Embedded Real-Time Control Systems, Marcel Verhoef, PhD Thesis.

0.3. Document History

Issue 0.1	22/10/08	First release.
Issue 0.2	28/11/08	Added comments from PGL. Added AST converter.
Issue 0.3	04/03/09	Added PO generator section, vdm.traces and misc other changes.
Issue 0.4	28/05/09	Added the DBGp protocol section, and extended runtime to discuss class initialization.
Issue 0.5	17/09/09	Added detail about VDM-RT implementation.
Issue 0.6	02/10/09	Updated CT description for TraceVariables
Issue 0.7	09/12/09	Added GPL copyright section.
Issue 0.8	12/01/10	Added section 3.

0.4. Copyright

1. Overview

VDMJ provides tool support for the VDM-SL, VDM++ and VDM-RT specification languages, written in Java [4][5][8]. The tool includes a parser, a type checker, an interpreter, a debugger and a proof obligation generator. It is a command line tool only, though it is accessible from graphical environments like Eclipse [7].

1.1. Package Overview

The implementation is divided into 19 Java packages, which are all sub-packages of org.overturetool.

Packages	
vdmj	The main VDMJ class and supporting classes.
vdmj.lex	Classes that implement the lexical analyser and its tokens.
vdmj.syntax	Classes that implement the syntax analyser.
vdmj.ast	Classes that translate an Overture AST to VDMJ's internal tree.
vdmj.types	Classes that represent static types during type checking.
vdmj.expressions	Classes that represent VDM expressions.
vdmj.statements	Classes that represent operation statements.
vdmj.patterns	Classes that represent patterns and binds.
vdmj.traces	Classes that represent trace definitions.
vdmj.definitions	Classes that represent VDM definitions.
vdmj.modules	Classes that represent VDM-SL modules and their import/export definitions.
vdmj.typechecker	Classes that support the static type checker.
vdm.pog	Classes that support the proof obligation generator.
vdmj.runtime	Classes that implement the interpreter.
vdmj.values	Classes that represent runtime values in the interpreter.
vdmj.commands	Classes that read and execute commands from standard input.
vdmj.messages	Classes that hold VDMJ error and warning messages.
vdmj.debug	Classes that implement the DBGp protocol.
vdmj.util	Utility classes and library routines used by all other packages.

The vdmj package contains the “main” abstract class for the suite, with two subclasses to parse, type check and interpret a specification in the VDM-SL, VDM++ or VDM-RT dialect.

The vdmj.lex package contains all classes to do with the lexical analysis of specifications. This includes the lexical token reader, plus a set of value classes for representing the various types of lexical token.

The vdmj.syntax package contains all the syntax analysis classes. This includes eight “readers”, which implement a recursive descent parser, based on a stream of lexical tokens. The readers are all sub-classes of an abstract Reader class.

The vdmj.ast package contains classes to translate an Overture parsed AST into VDMJ's internal tree format. This is to permit the Overture AST to be used by VDMJ, but without changing the type checking and runtime classes.

The vdmj.types package contains value classes that represent the various static types that can be contained in a VDM specification. They are all sub-classes of an abstract Type class.

The vdmj.expressions package contains value classes that represent the various types of expression that can be defined in a specification. They are all sub-classes of an abstract Expression class, which defines methods for an expression to be type checked and evaluated. Similarly, the vdmj.statements package defines a set of value classes that subclass Statement and represent the different statements in a specification.

The vdmj.patterns package includes value classes to represent the various patterns and binds in a specification. They are all subclasses of an abstract Pattern or Bind class.

The vdmj.traces package includes classes that represent the possible trace definitions that can be declared in a VDM++ or VDM-RT class.

The vdmj.definitions package contains a set of classes representing the definitions in a specification. They are all sub-classes of an abstract Definition class. Definitions are nested, so for example a class definition may contain function definitions, which in turn may contain function definitions for their pre and post condition functions. All definitions implement methods to perform type checking, and to generate runtime values representing their content.

The vdmj.modules package contains value classes that represent the modular structure of VDM-SL specifications, including their import and export declarations.

The vdmj.typechecker package includes classes which control the type checking of specifications. Most of the type checking is performed by the definition, expression and statement classes that collectively describe the specification, but the typechecker package defines the supporting classes to invoke the type checking methods of the other objects, and to represent the static environment in which type checking is performed.

The vdmj.pog package contains classes that support the proof obligation generator. Like type checking, the actual process of proof obligation generation is performed by the definitions, expressions and statements which form the specification, but the pog package contains classes to represent proof obligations.

The vdmj.runtime package defines the abstract interpreter, and its subclasses to interpret specifications. It includes classes to represent the runtime execution context, exceptions which can be generated at runtime, and the debugging and thread control classes.

The vdmj.values class contains a set of classes to represent runtime values in the interpretation of a specification. They are all subclasses of the abstract Value class. All values are immutable, with the exception of the UpdatableValue class hierarchy, which has a set method and is used to implement all state variables in the system.

The vdmj.commands package contains the command line readers that implement the interactive actions of the suite. This should be the only package that interacts with the user terminal.

The vdmj.messages package contains values classes and exceptions for holding error and warning messages.

The vdmj.debug package contains classes which implement the DBGp protocol described in [6]. This is used by the Eclipse debugger in Overture to debug specifications using VDMJ.

The vdmj.utils package contains common utilities used by all other packages, as well as the native code used by the "stdlib" VDM libraries.

2. Package Detail

2.1. vdmj

Class Summary	
VDMJ	The main class of the VDMJ parser/checker/interpreter.
VDMPP	The main class of the VDM++ parser/checker/interpreter.
VDMSL	The main class of the VDM-SL parser/checker/interpreter.
VDMRT	The main class of the VDM-RT parser/checker/interpreter.
ExitStatus	An exit status code.
Settings	A class holding flags for -pre, -post, -inv and -dtc, as well as the dialect.

The vdmj package contains the “main” abstract VDMJ class for the suite, plus three concrete subclasses to parse, type check and interpret a specification in the VDM-SL, VDM++ or VDM-RT dialect.

These classes just collect together a sequence of other classes to parse, type check and interpret the specification, depending on the command line arguments. The following (working) example illustrates the principles, using VDMJ classes to create a minimal interactive VDM-SL program:

```
public static void main(String[] args) throws Exception
{
	Settings.dialect = Dialect.VDM_SL;
	File file = new File(args[0]);
	LexTokenReader ltr = new LexTokenReader(file, Dialect.VDM_SL);
	ModuleReader mr = new ModuleReader(ltr);
	ModuleList modules = mr.readModules();

	if (mr.getErrorCount() == 0)
	{
		TypeChecker tc = new ModuleTypeChecker(modules);
		tc.typeCheck();

		if (TypeChecker.getErrorCount() == 0)
		{
			ModuleInterpreter interpreter =
				new ModuleInterpreter(modules);
			interpreter.init(null);
			CommandReader reader =
				new ModuleCommandReader(interpreter, "$ ");

			List<File> files = new Vector<File>();
			files.add(file);
			reader.run(files);
		}
	}
}
```

The example would be almost exactly the same for VDM++ or VDM-RT with "Class" instead of "Module" in the various names (ClassReader, readClasses, ClassTypeChecker etc.), and the dialect constant changed from VDM_SL to VDM_PP or VDM_RT.

VDMJ has a *-o* option which causes the parsed and type-checked specification to be written out to the filename specified by the argument to *-o*. Note that the entire tree is written to the file (if there are no type check errors), so if VDMJ is invoked with four specification files, and one *-o* option, the classes

specified by all four files are written to the one output file. Any input file called "*.lib" (rather than the more normal *.vpp or *.vdm) is assumed to be a pre-compiled library created by *-o* and is loaded as such without repeating the type checking. This can be faster for very large specifications. The example below shows the creation and use of IO.lib.

```
$ vdmpp -o IO.lib stdlib/IO.vpp
Parsed 1 class in 0.266 secs. No syntax errors
Type checked 1 class in 0.015 secs. No type errors
Saved 1 class to IO.lib in 0.125 secs.
```

```
$ vdmpp -i hello.vpp IO.lib
Loaded 1 class from IO.lib in 0.219 secs
Parsed 1 class in 0.422 secs. No syntax errors
Type checked 1 class in 0.015 secs. No type errors and 1 warning
Initialized 2 classes in 0.0 secs.
Interpreter started
> p new A().op()
Hello world!
= true
Executed in 0.015 secs.
```

2.1.1. Comments

The structure of these classes isn't very flexible. In particular, if the "load" command is given to the CommandReader, it is awkward to recover if parse/type errors are discovered in the new set of filenames (the List<File> passed to the run method is updated by the command reader to pass the new file names back to be parsed and checked).

The *-o* option was not as successful as I'd hoped. The idea was to serialize the tree after type checking, and load that back in quickly rather than re-parsing and re-checking the specification. Unfortunately, even though the serialization is passed through a gzip compression, the files produced are quite large and take a significant amount of time to decompress and de-serialize. For very small specifications it is faster to re-parse and check them, though for larger specifications there does appear to be an advantage.

Note that the serialization of the tree has to be for complete specifications (no unresolved references). This is because there is no link-editing available to combine partial specifications, and VDM++ and VDM-RT do not have a way to identify externals (VDM-SL has module imports, but VDM++ and VDM-RT have nothing similar). You can load multiple library files, or a mixture of library and source files.

2.2. vdmj.lex

Class Summary	
LatexStreamReader	A class to filter out LaTeX markup and #ifdefs from an input file.
BacktrackInputReader	A class to allow checkpoints and backtracking while parsing a file.
LexBooleanToken	A class to represent a boolean token.
LexCharacterToken	A class to represent a character literal token.
LexIdentifierToken	A class to represent an identifier.
LexIntegerToken	A class to represent an integer literal token.
LexKeywordToken	A class to represent keyword tokens.
LexLocation	A class to hold the location of a token.
LexNameList	A class to hold a list of LexNameTokens.
LexNameToken	A class to hold a name.

LexQuoteToken	A class to represent a quote type token.
LexRealToken	A class to represent a real literal token.
LexStringToken	A class to represent a string literal token.
LexToken	The abstract parent class for all lexical token types.
LexTokenReader	The main lexical analyser class.

Enum Summary	
Dialect	An enumeration to indicate the VDM dialect being parsed.
Token	An enumeration for the basic token types.

Exception Summary	
LexException	An exception class for lexical analyser exceptions.

The vdmj.lex package contains all classes concerned with the lexical analysis of specifications. This includes the lexical token reader, plus a set of value classes for representing all types of lexical token.

The base class of the lexical system is BacktrackInputReader, which allows a stack of markers to be pushed within a stream of characters, returning the read pointer to the previous marker when a pop operation is performed. The class allows truly random movement within the stream – which is held as an array of Unicode chars, once it has been loaded. It opens input files with a LatexStreamReader object (extends InputStreamReader), which strips out LaTeX markup and #ifdefs, while preserving the line numbers (turning LaTeX and #ifdef lines into blank lines). The #ifdef names available are the same as the Dialect constants (eg. #ifdef VDM_PP ... #else ... #endif). Ifdef statements must occur on a line by themselves.

LexTokenStream extends BacktrackInputReader. Its purpose is to make the input file look like a continuous stream of LexTokens. The nextToken method returns the next token from the stream, and getLast will (repeatedly) return the last token read. Push and pop mark the stream and return to a mark respectively; unpush removes a marker without returning to it; the retry method does a pop followed by a push.

The LexLocation class is used throughout the system to represent a location within source code, for error message reporting. The toString method of the class produces a string which can be appended to another message:

```
"in <class/module> (<filename>) at <line>:<column>"
```

All value objects in the system which have a sensible position in the source code have a LexLocation associated with them. The class is also used to implement execution coverage tracking, with static methods to return the list of executable lines, and the locations hit or missed since the last time they were reset.

There are several subclasses of the abstract LexToken to represent tokens that contain a value which is more naturally represented by a Java primitive type. For example, LexRealToken includes a double field, and LexBooleanToken contains a boolean.

The Token enumeration is the basic label for all token types. The enumeration includes a lookup method which, together with a Dialect, decides whether a given string is a token or not. Note that the dialect affects this: “static” is a token in VDM++ but is a legal variable name in VDM-SL, for example.

Lexical analysis throws a LexException if there are any problems. Recovery is left to the syntax analysis layer.

2.2.1. Comments

To report accurate line positions, the token reader has to know the width of tabs. This is currently fixed at 4 characters by the TABSTOP field of LexTokenReader. It should probably be a settable field.

There is some modest ugliness concerned with the parsing of certain symbol sequences that look like other tokens, eg. “mk_mod`name”. Naively, this would be a name (a LexNameToken) with a module part of “mk_mod” and the identifier part of “name”, but actually this is parsed as a single identifier, so that the syntax analyser can remove the “mk_” part and reveal the actual name.

Note that LexLocations have both start and end location information, though this is not used in VDMJ (it is used in the Overture editor though). The intention is to be able to identify blocks of source code that could be highlighted (eg. a whole statement or function, rather than just the start of one).

Recovering from a lexical error by raising an exception up to the syntax analysis may not be a good idea. The only recovery the syntax layer can do is to read up to some sort of safe point (eg. a semi-colon), and proceed from there. This gives comparatively poor error messages in general. It might be better to inject a likely token into the lexical stream, rather than throw an error and interrupt the stream.

LexNameTokens have a module/class name part and a simple name part (ie. they represent a grammatical *name* such as C`xyz). Whenever a name is created (as opposed to an identifier), it therefore has to include its class or module name, even if none was actually used in the specification (eg. simple parameter names are identifier patterns that are characterized by a name token, so all parameter names are held as LexNameTokens like C`x). That would be fine, except that in VDM++ the presence or absence of the class qualifier in a name can have semantic significance – explicitly identifying a member in a class hierarchy for example, with "object.X`name()". So LexNameTokens have an *explicit* flag, which means that the class name was explicitly specified by the caller, not implicitly filled in by the parser, and that flag is used during VDM++ type checking and runtime to identify the correct definition for the name. This works, but it may be over complicated. There are places where the *explicit* flag is set for very obscure reasons, which is a maintenance hazard. It may be better to take the Overture AST approach and allow names that simply don't have a class/module definition, and then take account of this when looking up definitions.

To enable VDM++ and VDM-RT function/operation overloading, LexNameTokens optionally include a TypeList qualifier, representing the parameter types of the function or operation. The qualifier, if present, is used in the equals method of the class, and when searching for a name in a function/operation apply, the name sought is qualified with the actual types of the arguments. The equals function uses the TypeComparator to make the test, so a function declared with an int parameter will match one sought with a nat1 argument, etc. This system for managing overloading is not without its problems. Firstly, the use of the TypeComparator makes the equals method quite heavy. Secondly, it means that names cannot naively be used in Java maps because the hashCode of a function declaration name, and a function apply name may not be the same (if the argument types are not identical to the parameter types). Thirdly, it causes trouble when functions are applied via function variables (ie. lambda values) – such values can either be qualified with their parameter types or not, but since a function variable can either be applied (where argument types can be deduced) or just passed on (where they cannot), one or other of the name lookups will fail. To compensate, VariableExpression (which resolves simple names) will perform an unqualified "plain" name lookup if a qualified one fails. But a qualified lookup may succeed inappropriately by picking up an outer definition from the environment, when an inner unqualified name exists. This is tricky to solve.

2.3. vdmj.syntax

Class Summary	
SyntaxReader	The parent class of all syntax readers.
ExpressionReader	A syntax analyser to parse expressions.
TypeReader	A syntax analyser to parse type expressions.
DefinitionReader	A syntax analyser to parse definitions.
ClassReader	A syntax analyser to parse class definitions.
ModuleReader	A syntax analyser to parse modules.
PatternReader	A syntax analyser to parse pattern definitions.
BindReader	A syntax analyser to parse set and type binds.
StatementReader	A syntax analyser to parse statements.

Exception Summary	
ParserException	A syntax analyser exception.

The vdmj.syntax package contains all the syntax analysis classes. This includes eight “readers”, which implement a backtracking recursive descent parser, based on a stream of lexical tokens. The readers are all sub-classes of an abstract Reader class.

Every SyntaxReader subclass is constructed by being passed a LexTokenReader object. The reader is then responsible for returning one or more syntactic elements that it is designed to parse. For example, an ExpressionReader can be attached to a lexical stream, and be used to read an expression, or a comma separated expression list. Readers typically have methods called “read<something>” to preform the actual parse (eg. the minimal example above calls readModules from a ModuleReader).

Note that one type of reader usually needs other types of reader to complete its job. So for example, when a DefinitionReader is parsing an explicit function definition, it will use the raw lexical stream to read the function name, it will use a TypeReader to read the function’s type signature, a PatternReader to read the parameter patterns, and an ExpressionReader to read the body of the definition and any following pre or post conditions. All readers cache instances of the other readers used, and methods like getTypeReader either return the previous instance or create a new one. Note that there is no positional state information held in a reader therefore – it must depend on the LexTokenReader it contains to (say) determine the last token read.

Several parts of the VDM grammar cannot be parsed unambiguously by reading lexical tokens in a strict sequence. For example, several parts of the grammar use a <pattern bind> symbol, which is defined to be either a pattern or a bind, but it is not possible to distinguish a pattern from a bind by looking at the next token in the stream – a type bind is a <pattern>:<type> for example, so it looks like a pattern to start with but turns out to be a bind. To overcome this, the parsers are able to backtrack. That is, the start of the <pattern bind> is marked (a push operation on the lexical stream), and an attempt is made to parse the “longest” possibility, which is a bind in this case; if that fails, the stream is popped back to the marker, and the other possibilities are tried. If all possibilities fail, there is clearly an error, though it is not clear which branch contains the error (eg. is it a pattern that is malformed or a bind that is malformed?). In this case, the parser reports the error from the branch which managed to consume the most tokens after the marker before failing. This is assumed to be the most helpful error, though it is not certain what the user intended.

The following backtrack code pattern is used frequently by the readers. Note how the ParserException is used to carry “depth” information about how far the parser progressed, and the two depths are compared at the end to see which exception to throw. The code calling this method may itself be backtracking, having pushed its own markers in the stream.


```
public PatternBind readPatternOrBind()
      throws ParserException, LexException
{
      ParserException bindError = null;

      try
      {
            reader.push();
            Bind b = readBind();
            reader.unpush();
            return new PatternBind(bind.location, b);
      }
      catch (ParserException e)
      {
            reader.pop();
            e.adjustDepth(reader.getTokensRead());
            bindError = e;
      }

      try
      {
            reader.push();
            Pattern p = getPatternReader().readPattern();
            reader.unpush();
            return new PatternBind(p.location, p);
      }
      catch (ParserException e)
      {
            reader.pop();
            e.adjustDepth(reader.getTokensRead());
            throw e.deeperThan(bindError) ? e : bindError;
      }
}
```

The top level of the parsers contain the recovery code to attempt to move the parse beyond a given syntax error. This is done by each top level case (eg. parsing a whole function definition) defining two lists of tokens: those which should be read up to, and those that should be read up to *and past* in the event of an error. Then a common recovery method (“report” in the Reader class) reads tokens up to one of those specified before continuing with the parse. The objective is, say, for a Statement reader to read tokens up to the end of the broken statement before continuing. In general this is not foolproof, and often syntax errors produce a short cascade of unrelated errors later in the specification.

2.3.1. Comments

There are some ugly parts to the parsing. One is concerned with equals definitions, which are defined as "def" <pattern bind>=<expression> "in" <expression>, but if the <pattern bind> is actually a set bind of the form “e in set S”, this parses as "s in set (S = <expression>)". There is some nifty footwork in the code to get round this (see the comments in readEqualsDefinition in the DefinitionReader).

Another ugly case is concerned with object call statements, which grammatically look like object apply designators as they end in ...<name>(args). So the parser reads an apply designator, then looks inside it to see whether the object being applied is a field designator or an identifier. The former is an object member invocation, the latter is a simple operation call.

The SyntaxReader base class provides a set of methods for reading and optionally advancing by one token. The differences between them are subtle (eg. advance and return the next token, or return the current token and advance), and I suspect the code could be cleaned up by reducing the number of options.

Recursive descent parsing always has difficulty with accurate error reporting and recovery. The method chosen is not perfect.

2.4. vdmj.ast

Class Summary	
ASTConverter	The AST tree converter.

The vdmj.ast package contains one class which is used to translate an Overture AST parse tree into the equivalent structure for VDMJ. The class is constructed with a filename and an IOmlDocument obtained from the OvertureParser. A convertDocument method converts the document tree into a list of ClassDefinitions.

2.4.1. Comments

This is a very large monolithic class, which should probably be broken up.

The Overture parser is compiled using Java 1.6, but the interface does not use generics – for example, the *get* methods to extract a list of classes from a specification is declared as a raw Vector, not a List<IOmlClass>. The converter assumes that the get methods will be converted to use generics one day, and assigns their return values to (likely) generic values, like List<IOml...>. To get this to compile, there is a class-wide annotation to suppress unchecked warnings. When the AST interface is upgraded, this can be removed and the compiler should point out any type inconsistencies between the assumed interface and the actual declarations (though since the code works, this is very likely to be right).

2.5. vdmj.types

Class Summary	
Type	The parent class of all static type checking types.
***Type	A *** type. There are 25 such classes.
BasicType	The parent of the basic types (numbers, booleans and characters).
NumericType	The parent of the numeric types (real, rat, int, nat, nat1)
InvariantType	A type which has an invariant function associated with it.
NamedType	A type with a name.
OptionalType	An optional type.
ParameterType	A type associated with a polymorphic parameter name.
UnionType	A union of types.
UnknownType	A type representing a parser error.
UnresolvedType	A type name identifier by the syntax analyser.
VoidType	A type indicating the absence of a type.
VoidReturnType	A type indicating that a return statement has returned “()”.
PatternListTypePair	A pattern list combined with a single type.
PatternTypePair	A pattern plus a type.
TypeList	A list of types.
TypeSet	A set of types.

The vdmj.types package contains value classes that represent the various static types that can be contained in a VDM specification. They are all sub-classes of an abstract Type class.

Most simple types have a class dedicated to them of the same name, for example IntegerType or QuoteType. Similarly, the composite types have classes, like RecordType, SetType, SeqType and MapType; these have fields that in turn indicate the types of their components.

All types have a method to “resolve” themselves. The process of type resolution occurs early in the type checking process (see below), and turns UnresolvedTypes, which are just the names of types from the syntax analysis, into the actual type of the corresponding definition. The core of this happens in the typeResolve method of UnresolvedType, though other types call typeResolve recursively for any types that they contain – for example, FunctionTypes must resolve their parameter types and return type. The type resolution mechanism contains a recursive defence to avoid types which reference themselves from blowing the stack.

All types have a method to “polymorph” themselves. This means that given an actual type parameter, they substitute that type for any ParameterTypes they contain to yield a new Type object. This is used during the type check and execution of polymorphic function instantiations, when the actual type parameters are known.

All types implement a number of “is” and “get” methods – for example, (boolean) isMap and (MapType) getMap. These are used during type checking to determine whether a type is suitable for use in (say) a map application context. For simple types, these methods return false for the “is” method, except for the type concerned, which returns “true”; and “this” from the “get” method. For more complex types, these methods support the situation where the static type checking cannot know the actual runtime type, but knows that it is one of several. In this case – the most obvious example being a UnionType – the “is” method will return true if any of the member types of the union would return true; and the “get” method will construct a new type representing the aspects of the applicable members of the union, all spliced together. The type checking can then proceed using the information of this single blend of possibilities, in the knowledge that the tests it is making could occur at runtime.

For example, if a type is a union of two records, each of which has a field called “label”, one of which is a “seq of char” and the other of which is a “nat1”, the getRecord method of the UnionType would return a new synthetic RecordType with a single field called “label”, with type “(seq of char) | nat1”. So the type checking of access to the label field would proceed as though that was its type, even though at runtime the type will be one or the other.

All types have a getAllValues method which is used during the evaluation of type binds. The method returns all the values for the type, though the only type which implements this is BooleanType; other types throw an exception if this is called.

UnknownTypes are used during error handling. Typically, an error will be encountered and reported, but rather than returning (say) the type of a sub-expression from type checking, an UnknownType is returned. This type has the property that all of its “is” methods return true, and its “get” methods will try to return a plausible Type. This means that subsequent type checking will not produce a cascade of errors as a single type checking fault deep in a specification winds its way out to the top level.

The VoidType is usually used to mean the absence of a type, so for example an operation which returns “()” would be represented by a VoidType. During the type checking of a sequence of statements in an operation, any statements which follow a “return” statement on an execution branch will be unreachable (a warning). So to distinguish this deliberate return of nothing from most statements which yield nothing, the VoidReturnType is used.

The TypeList and TypeSet classes implement lists and sets of Types, respectively. These are used in processing when a collection of types are encountered, and they must be turned into a single product type (TypeList) or a single union type (TypeSet). The ProductType and UnionType classes contain one TypeList and TypeSet, respectively.

2.5.1. Comments

There is no ReferenceType (compare with a ReferenceValue), yet several of the types do contain directly referenced types (like OptionalType and BracketType). It might be possible to simplify the hierarchy by adding one.

Do we call them products or tuples? I went for ProductTypes and TupleValues in the end.

2.6. vdmj.expressions

Class Summary	
Expression	The parent class of all VDM expressions.
***Expression	An expression of type ***. There are >100 of these.
BinaryExpression	The parent of all binary expressions.
NumericBinaryExpression	The parent of all numeric binary expressions (+, -, *, /)
BooleanBinaryExpression	The parent of all boolean binary expressions (and, or, <=>, =>)
UnaryExpression	The parent of all unary expressions.
ExpressionList	A list of Expressions.

The vdmj.expressions package contains value classes that represent the various types of expression that can be defined in a specification. They are all sub-classes of an abstract Expression class, which defines methods for an expression to be type checked and evaluated.

The typeCheck method is implemented by all expressions, and is passed an environment defining the variables and types in scope, together with a NameScope which identifies what sorts of names are accessible (eg. whether state values are in scope – they are for expressions in operations, but not in functions).

The typeCheck method returns a Type which indicates the result of evaluating the expression in the environment passed. So literal expressions simply return an appropriate type, like IntegerType. More complex expressions have to consider whether the definitions they contain affect the environment, whether those definitions have to be type checked, and whether the type check of any sub-expressions returns the expected result for the overall expression.

For example, consider a “forall” expression. This will contain a bind list of variables, and a predicate to evaluate for each (at runtime). The typeCheck method of ForAllExpression is as follows:

```
@Override
public Type typeCheck(
      Environment base, TypeList qualifiers, NameScope scope)
{
      Definition def = new MultiBindListDefinition(location, bindList);
      def.typeCheck(base, scope);
      Environment local = new FlatCheckedEnvironment(def, base);

      if (!predicate.typeCheck(
            local, null, scope).isType(BooleanType.class))
      {
            predicate.report("Predicate is not boolean");
      }

      local.unusedCheck();
      return new BooleanType(location);
}
```

A MultiBindListDefinition is a type of Definition (see below) which, when given the bind list for the forall expression, can expand the Environment passed in to make the names and types of the bind variables visible. Once created, the new definition is type checked to make sure the bindings contain no errors (for example, if the bind list contains a set bind, the expression representing the set must be a SetType).

A new FlatCheckedEnvironment is created to chain the new definitions onto the base Environment passed in, and this is used to type check the predicate of the forall expression. The return value of this is the Type of the predicate, which must be a boolean expression. The local environment is then checked to see whether all the names added to it by the bind list were actually used when type

checking the predicate; any unused variables generate a warning. Lastly, this method returns a boolean Type, since a forall expression returns a boolean.

The typeCheck method on Expression is also passed a TypeList. This is used when trying to resolve name overloading during function and operation application. The typeCheck method of ApplyExpression starts by generating a TypeList from the typeCheck of each of the argument expressions it has. That may generate (say) [int, int, bool]. That list is then passed to the typeCheck method for the root of the apply (the thing being applied). If this root is a VariableExpression or a FieldExpression, the name of the variable or field is qualified with the list of types passed in, and this is used to find an overloaded name of a function or operation definition that has parameters whose types are compatible with the arguments (note, compatible with, not identical to). If it turns out that the variable or field is actually a map (which has no qualifiers), the search is repeated for the name without type qualification.

The other important method on Expressions is the eval method. This is called to evaluate the expression given the runtime Context (the runtime equivalent of an Environment). The method returns a Value object, which can represent any value in VDM. The eval method for PlusExpression is as follows:

```
@Override
public Value eval(Context ctxt)
{
	breakpoint.check(location, ctxt);

	try
	{
		double lv = left.eval(ctxt).realValue(ctxt);
		double rv = right.eval(ctxt).realValue(ctxt);

		return NumericValue.valueOf(lv + rv, ctxt);
	}
	catch (ValueException e)
	{
		return abort(e);
	}
}
```

All Expressions and Statements contain a Breakpoint object, and the eval method of all expressions and statements call their breakpoint's check method at the start. This usually does nothing, unless a breakpoint is set at this location, in which case execution stops and calls debugger code.

The PlusExpression evaluates its left and right hand sides, and converts the resulting Value objects to raw Java doubles. The result is a new Value, created using the valueOf method of NumericValue, which will create the simplest type of NumericValue capable of holding the result of the addition – for example, this might be an IntegerValue (for -123) or a NaturalOneValue (for 123) or a RealValue (for 1.23).

Note that the code may throw a ValueException, and that this is caught within the eval method rather than being propagated. This can only occur in the conversion of the sub-expression results to doubles, or the construction of the result Value. ValueExceptions indicate problems while evaluating an expression, but they are caught and propagated using the abort method, which creates and throws a ContextException (a Java RuntimeException). This is done to distinguish between expected value errors – for example, trying to convert a Value to one of several types in a union, and failing before the right one is found – and serious errors, which should cause the system to halt.

The most complex evaluations are for functions and operations, but these are delegated to the corresponding FunctionValue and OperationValue classes. This is so that a function value can be separately created (for example via a lambda expression) and applied. Operations cannot be created like this, but having operation values too means that, in VDM++ or VDM-RT, an object becomes a map of names to values – whether those are instance variable values or member operations and functions. The eval method of the ApplyExpression to call a function is therefore just:


```
try
{
      Value object = root.eval(ctxt).deref();

      if (object instanceof FunctionValue)
      {
            ValueList argvals = new ValueList();

            for (Expression arg: args)
            {
                  argvals.add(arg.eval(ctxt));
            }

            FunctionValue fv = object.functionValue(ctxt);
            return fv.eval(argvals, ctxt);
      }
      else if (object instanceof OperationValue)
      ...
```

All Expressions implement a method called findExpression. This has a line number parameter, and all implementations are responsible for returning themselves if they start on the line number, or recursing into their sub-expressions if they have them. This is used to set breakpoints on specific lines of a specification.

2.6.1. Comments

The implementation of name overloading may be problematic. A LexNameToken is optionally qualified with a TypeList, and the equals method uses the TypeComparator (part of vdmj.typechecker) to make a compatible comparison of the two names' type lists. Care must be taken when comparing names, especially during the "bootstrap" phases when qualifiers are not yet available.

2.7. vdmj.statements

Class Summary	
Statement	The parent class of all statements.
***Statement	A statement of type ***. There are 40 or so of these.

The vdmj.statements package contains value classes that represent the various types of statement that can be defined in a specification. They are all sub-classes of an abstract Statement class, which defines methods for a statement to be type checked and executed.

Many of the principles for Statements are the same as those for Expressions covered above. Like Expressions, all Statements include a typeCheck method which is passed an Environment and a NameScope – though note that there is no need for a TypeList of qualifiers because an operation call can only be rooted on something that is already known by the statement (an operation name or an object designator), whereas function application in an expression has to evaluate an arbitrary expression to generate the root to which to apply the arguments.

Similarly, like Expressions, all Statements define an eval method which executes them, returning a Value – though statement executions usually return VoidValues.

Statements implement a method called exitCheck, which explores the statement tree (following blocks and branches from compound statements) looking for statements which can raise an exit status, like the ExitStatement itself. This is used in the type checking of statements that catch and process exits, like TrapStatements. The method returns a set of exit Types that can be thrown.

All Statements implement a method called findStatement. This has a line number parameter, and all implementations are responsible for returning themselves if they start on the line number, or recursing into their sub-statements if they have them. This is used to set breakpoints on specific lines of a

specification.

2.7.1. Comments

The exitCheck is not perfect. In particular, the check does not cross the boundary of an operation call from a statement block, nor can it follow an expression evaluation that involves operation calls. That would require code to work out which operation(s) are actually involved, and it would require recursive defence against operations which recurse. So exitCheck produces false negatives (indicates that operations can't exit, when in fact they can). I gather VDMTools is better, but not perfect either. It is known to produce false positives.

2.8. vdmj.patterns

Class Summary	
Pattern	The parent type of all patterns.
****Pattern	A pattern of type ****. There are 14 such pattern types.
Bind	The parent class of SetBind and TypeBind.
MultipleBind	The parent class of MultipleSetBind and MultipleTypeBind.
PatternBind	A pattern or a bind.
PatternList	A list of patterns.

The vdmj.patterns package includes value classes to represent the various patterns and binds in a specification. They are all subclasses of an abstract Pattern, Bind or MultipleBind class.

Patterns generate definitions, given a Type. For example, the pattern “[a,b,c]” will produce definitions for the three integer variables, given that it is of type “seq of int”. The process of definition generation is recursive over the tree of nested patterns that might be defined. So the getDefinitions(Type) method which all patterns implement, will recurse for those pattern types which are defined as containing sub-patterns. The leaves of the pattern tree are the simple pattern types: BooleanPattern, CharacterPattern, etc. At the leaves, it is only IdentifierPattern which produces variable definitions.

Similarly, patterns generate a list of name/value pairs given a Value. The value is matched against the “shape” of the pattern and its sub-patterns, and any IdentifierPatterns at the leaves are populated with the corresponding part of the original value. This getNamedValues method is called from definitions which include patterns (such as function parameter definitions) when the actual values are required.

Patterns can also be asked for a simple list of their variable names, which involves a depth search for IdentifierPatterns.

Patterns include a typeResolve method because ExpressionPatterns can contain referenced to UnresolvedTypes that need to be resolved early in the type check. Definitions which include Patterns recurse into the pattern's typeResolve from their own typeResolve methods.

A Bind, which is sub-classed by SetBind and TypeBind, comprises a Pattern and a set of Values (either an explicit set, or the set of all the Values that a Type can generate, in theory). These are used in quantified expressions, like “exists {a,b} in set S & a.type = b.type”. Here the set pattern {a,b} contains two identifier patterns that must (potentially) iterate through all the elements of S, taking the corresponding values. To drive the iteration, the Bind needs to generate all the possible Values, which are then given to the pattern to generate name/value pairs for each iteration. Therefore Bind has a method called getAllValues, which is implemented by both sub-classes (the TypeBind implementation calls the getAllValues method of the Type, which is an error for everything except BooleanType).

MultipleBind, which is sub-classed by MultipleSetBind and MultipleTypeBind, is very similar except that they comprise a list of patterns and a set or type. But they still have a getAllValues method which collects together all the possible values in the set.

Note that in VDMJ, the generation of all values from a set includes the permutations of all the

orderings of the values in that set. Internally, sets of Values are held in a ValueSet which is actually an ordered list (but with set semantics, with regard to no duplicates). Therefore the getAllValues methods for the various set binds call the permuteSets method of ValueSet (via the same method on SetValue). Note also that the original set is sorted before this process, which means that given the same set content, the order of processing in a bind (and therefore any looseness based on it) is consistent.

```
@Override
public ValueList getBindValues(Context ctxt)
{
	try
	{
		ValueList results = new ValueList();
		ValueSet elements = set.eval(ctxt).setValue(ctxt).sorted();

		for (Value e: elements)
		{
			e = e.deref();

			if (e instanceof SetValue)
			{
				SetValue sv = (SetValue)e;
				results.addAll(sv.permutedSets());
			}
			else
			{
				results.add(e);
			}
		}

		return results;
	}
	catch (ValueException ex)
	{
		abort(ex.getMessage(), ctxt);
		return null;
	}
}
```

2.8.1. Comments

Binds and MultipleBinds look like they have a lot in common and could probably be put together to avoid a small amount of code duplication.

2.9. vdmj.traces

Class Summary	
TraceDefinition	An abstract class representing a trace definition.
TraceDefinitionTerm	A class representing a sequence of trace definitions.
TraceLetBeStBinding	A class representing a let-be-st trace binding.
TraceLetDefBinding	A class representing a let-definition trace binding.
TraceRepeatDefinition	A class representing a repeated trace definition.
TraceCoreDefinition	Abstract of all core trace expressions.
TraceApplyExpression	A class representing a core trace apply expression.
TraceBracketedExpression	A class representing a core trace bracketed expression.

TraceNode	An abstract class representing an expansion node.
AlternativeTraceNode	An expansion node for alternatives.
RepeatTraceNode	An expansion node for repeats.
SequenceTraceNode	An expansion node for sequences.
StatementTraceNode	An expansion node (leaf) for statement applies.
TestSequence	A sequence of CallSequences
CallSequence	A sequence of CallObjectStatements.
Permutor	A utility to permute a set of values.
TraceVariable	A class containing a name/value/location tuple.
TraceVariableList	A list of TraceVariables
TraceVariableStatement	A statement wrapping a TraceVariable.

Enum Summary	
Verdict	A test outcome: PASSED, FAILED or INDETERMINATE.

The vdm.traces package includes classes to represent the definitions which can occur in a VDM++ or VDM-RT "traces" section, and their subsequent expansion and execution.

The subclasses of TraceDefinition are uncontroversial and follow the structure of the trace grammar closely. These classes have the usual typeCheck methods which permit the trace specifications to be checked as part of the overall specification type check phase.

In order to evaluate traces, they must first be expanded into all the possible execution paths represented by the definition. For example, a trace that is of the form "a;(b|c);d" would expand to "a;b;d" and "a;c;d". Similarly, "a{1,3}" would expand to "a", "a;a" and "a;a;a". Repeats, sequences and alternations multiply together to generate large numbers of tests very quickly – this is the whole point of combinatorial test specification. The TraceNode class and its subclasses represent the expanded traces, and are generated by "expand" methods on all TraceDefinitions. Strictly, these classes don't expand the traces, but produce a tree structure that is capable of expanding them. The getTests method of TreeNodes actually expands the tests, returning a TestSequence, which is a list of CallSequence (ie. a list of tests), and a CallSequence is a list of operation applies (of CallObjectStatements).

All TraceNodes include a field called variables, which may contain a TraceVariableList. These are populated by TraceLetBeStBinding and TraceLetDefBinding when a particular name/value pair is chosen in the expansion of a give test sequence. The getVariables method of TraceNode returns a CallSequence which is populated with any TraceVariableStatements for the expansion. When executed, these statements just add their named value to the local context, making it available to subsequent calls in the test.

A NamedTraceDefinition (see below) is created for each parsed trace definition, and this produces an operation in the class which, when executed, will expand the trace and execute all of the tests in turn, printing the results to the console. Between each test execution the system is initialized (the init method of the ClassInterpreter) and a new base object is created to run the test. The body of these operations are TraceStatements, which are constructed with reference to their NamedTraceDefinition. The eval method of this class first gets a TestSequence from the NamedTraceDefinition. Then it creates an Environment which is suitable to type check the statements in the tests – it is necessary to type check statements before their execution. Then for each CallSequence in the TestSequence, it initializes the ClassInterpreter, and calls its runtrace method, passing the CallSequence and the Environment. The runtrace method executes the sequence of statements in the test and returns a list of java.lang.Object, being either the return values from the test steps or error messages, and the last item will always be an instance of a Verdict object, indicating the test outcome.

Tests which have a FAILED verdict are "stemmed" and then remaining tests in the TestSequence which have the same stem are marked as "filtered" by this test – ie. there is no point in running them because their initial sequence of calls will fail at the same point, for the same reason. Before each

tests is executed, its filtered flag is tested, and such tests are not executed. Note that the stem check includes the value of TraceVariableStatements in the test as well as CallObjectStatements. This means that tests which are superficially the same, but which have different variable values will not match.

2.9.1. Comments

There is a minor quibble in the parsing of trace sections, in that the grammar prohibits the use of semi-colons between trace definitions (while permitting them to separate parts of a trace definition). VDMJ permits these separate semi-colons; the Overture parser currently does not.

I tried very hard to combine the expansion of the tests with the classes that represent the parsed trace definitions – this is just a tree after all. But I couldn't get it working properly, hence the solution where the definitions are expanded into a separate tree, which is then "walked" to generate the trace permutations.

The intention of the TraceVariable was to hold information about the location of a particular value from a set of (possibly) anonymous values, such as "let x in set {new A(1), new A(2), ...} in ..." Here, x will take one of the values from the set in each expansion, but it will always be called "x" and it is hard to distinguish cases when a given test fails. Unfortunately, this is very hard to achieve without giving (pure) Values a location. Currently, the location stored with TraceVariables is the location of the name of the variable. The name/value is included in the CallSequence (via the TraceVariableStatement), so debuggers can look at the raw value, which may be of some help.

2.10. vdmj.definitions

Class Summary	
Definition	The abstract parent of all definitions.
AccessSpecifier	A class to represent a [static] public/private/protected specifier.
AssignmentDefinition	A class to represent assignable variable definitions.
ClassDefinition	A class to represent a VDM++ or VDM-RT class definition.
SystemDefinition	A class to represent a VDM-RT system class definition.
CPUClassDefinition	A class to represent a VDM-RT CPU definition.
BUSClassDefinition	A class to represent a VDM-RT BUS definition.
ClassInvariantDefinition	A class to hold a class invariant definition.
EqualsDefinition	A class to hold an equals definition.
ExplicitFunctionDefinition	A class to hold an explicit function definition.
ExplicitOperationDefinition	A class to hold an explicit operation definition.
ExternalDefinition	A class to hold an external state definition.
ImplicitFunctionDefinition	A class to hold an implicit function definition.
ImplicitOperationDefinition	A class to hold an explicit operation definition.
ImportedDefinition	A class to hold an imported definition.
RenamedDefinition	A class to hold a renamed import definition.
InheritedDefinition	A class to hold an inherited definition in VDM++.
InstanceVariableDefinition	A class to hold and instance variable definition.
LocalDefinition	A class to hold a local variable definition.
MultiBindListDefinition	A class to hold a multiple bind list definition.
MutexSyncDefinition	A class to hold a mutex synchronization definition.

PerSyncDefinition	A class to hold a permission synchronization definition.
StateDefinition	A class to hold a module's state definition.
ThreadDefinition	A class to hold a thread definition.
TypeDefinition	A class to hold a type definition.
UntypedDefinition	A class to hold a definition of, as yet, an unknown type.
ValueDefinition	A class to hold a value definition.
NamedTraceDefinition	A class to hold a named trace definition.
ClassList	A class for holding a list of ClassDefinitions.
DefinitionList	A class to hold a list of Definitions.
DefinitionSet	A class to hold a set of Definitions with unique names.

The vdmj.definitions package contains classes representing the definitions in a specification. They are all sub-classes of an abstract Definition class.

All definitions have a few things in common (fields of the abstract class). They belong to a Pass, which guides the type checking; they have a location; they have a name, though this may be null if they contain sub-definitions; and they have a NameScope to define what sort of name(s) they define, which compliments the name scope used in type checking that searches for names of certain types.

Definitions define a typeResolve method which is used very early on to resolve the UnresolvedTypes that may have come through from the syntax analysis. For example, an ExplicitFunctionDefinition would typeResolve the Type of the function, and if there were pre or postconditions, these expressions would be typeResolved, and any parameter patterns would be typeResolved.

Definitions also define a typeCheck method, which is similar to the ones defined for Expression and Statement, except that there is no Type to return. For example, the ExplicitFunctionDefinition performs the following tasks in its typeCheck method:

- If there are any polymorphic type parameters for this function, check that the overall function type does not reference any type parameters except those named type parameters.
- For each type parameter, create a LocalDefinition of a ParameterType and add this to a local Environment.
- Check that the parameter patterns match the overall Type’s parameters, and iterate through curried sets of parameters, using the return value from the overall Type (and its return value and so on for subsequent sets of parameters). Remember the expected result.
- Extend the local Environment with definitions for all the variables of all the patterns from all of the curried parameter sets.
- Type check the definitions this produced in the base environment (this will just do type resolution, if necessary).
- Label the local Environment as static (VDM++) if the definition’s access specifier is static.
- If we are in VDM++ and the function is not static, add a “self” definition to the local Environment.
- If there is a precondition expression, type check the definition for it.
- If there is a post condition expression, type check the definition for that too.
- Type check the body expression of the function, remembering the actual type returned.
- If the actual return type is not assignable to the expected return type, raise an error.

- If the VDM++ accessibility of the expected return type is narrower than that of the definition itself, raise an error (eg. a public function cannot have a private return type).
- If the function is recursive and does not define a "measure" function, raise a warning, else if there is a measure defined, check that it exists and has the correct type.
- Check that the parameter variables have been referenced in the local Environment, else raise an unused parameter warning. (This is suppressed for pre and post conditions, which are permitted to not necessarily use their implicit parameters).
- Return.

This illustrates the principles that are used by all definitions' typeCheck methods.

Some definition types are only used to "wrap" others. For example, during module imports and exports, a definition may be imported and/or renamed. These methods just delegate their calls to the referenced definition that they wrap.

As with Patterns, definitions can yield their contained definitions or a list of names or name/value pairs that they define. This is the purpose of the getDefinitions, getVariableNames and getNameValuePairs methods. Simple local definitions only define one variable, but many definition types include a Pattern specifier that may define many variables.

The findName method is implemented by all definitions to return whether they define a name being sought by type checking. As above, for simple definitions, this just compares their name and scope with that being searched for, but for definitions that include patterns, all the names generated by the pattern must be considered.

The findType method is implemented by those definitions that define a type (TypeDefinition, StateDefinition and ClassDefinition).

Definitions also define findExpression and findStatement methods which recurse into their bodies in search of an expression or statement that starts on the given line.

The ClassDefinition class is slightly different from the others in that its main job is to contain the definitions in a class, though it does have the job of setting up the static and instance environment for new objects when a "new Object()" statement is executed. It is also responsible for stitching together the class hierarchy and arranging for symbols to be inherited so that type checking may be performed. The hierarchy is built during the generation of implicit definitions.

Note that class static data (eg. instance variables that are declared static) is held inside the ClassDefinition at runtime (in the public/privateStaticValues fields), whereas object instance data is held inside an ObjectValue (produced by the makeInstance method of ClassDefinition), though references to the static data is included in the object's member list, so that the runtime can find them. See section 2.14 for more information about runtime variable access.

The SystemDefinition class is a subclass of ClassDefinition, and adds VDM-RT specific processing for system classes. The implicit definition generation (see 2.12) checks whether the definitions in the system class meet the VDM-RT restrictions. An init method is called during system initialization and creates the necessary CPU and BUS objects from the system definition. Lastly, the newInstance method is overridden, since it is not legal to create instance of a VDM-RT system class.

The CPUClassDefinition and BUSClassDefinition classes represent VDM-RT CPU and BUS classes respectively. These are also subclasses of ClassDefinition, and override the newInstance method to perform special processing. Both classes create their operations (ie. create ExplicitOperationDefinitions for their definition list) by parsing a string literal representing the operations required. For example:

```
private static String defs =
      "operations " +
      "public BUS:(<FCFS>|<CSMACD>) * real * set of CPU ==> BUS " +
      "     BUS(policy, speed, cpus) == is not yet specified;";

private static DefinitionList operationDefs()
```



```
        throws ParserException, LexException
{
        LexTokenReader ltr = new LexTokenReader(defs, Dialect.VDM_PP);
        DefinitionReader dr = new DefinitionReader(ltr);
        dr.setCurrentModule("BUS");
        return dr.readDefinitions();
}
```

This technique permits the rest of the code to use these operations as normal. The *is not yet specified* processing intercepts the operation calls for CPU and BUS, calling back to the CPUClassDefinition and BUSClassDefinition classes to perform the actual processing.

2.10.1. Comments

There is a great deal of commonality between some definitions, especially implicit and explicit functions and operations. It might be possible to simplify the code by creating abstract bases for these.

2.11. vdmj.modules

Class Summary	
Export	The parent class of all export declarations.
Export***	A class for representing exports of a given type.
Import	The parent class of all import declarations.
Import***	A class for representing imports of a given type.
Module	A class holding all the details for one module.
ModuleList	A list of Modules.

The vdmj.modules package contains value classes that represent the modular structure of VDM-SL specifications, including their import and export declarations.

The only purpose of these classes is to represent the parsed module structures from the specification, and to generate/find the list of exported and imported definitions that extend the scope of what is visible from a single module.

The ModuleList class is important because it contains the "initialize" method which is used to set the initial state of all modules when the interpreter is started.

2.12. vdmj.typechecker

Class Summary	
TypeChecker	The abstract root of all type checker classes.
ClassTypeChecker	A class to coordinate all class type checking processing.
ModuleTypeChecker	A class to coordinate all module type checking processing.
Environment	The parent class of all type checking environments.
FlatEnvironment	Define the type checking environment for a list of local definitions.
FlatCheckedEnvironment	Define the type checking environment for a list of local definitions, including a check for duplicates and name hiding.
ModuleEnvironment	Define the type checking environment for a modular specification.

PrivateClassEnvironment	Define the type checking environment for a class as observed from inside.
PublicClassEnvironment	Define the type checking environment for a set of classes, as observed from the outside.
TypeComparator	A class for static type checking comparisons.

Enum Summary	
NameScope	An enum to represent name scoping.
Pass	An enum to indicate which type checking pass a definition belongs to.

The vdmj.typechecker package includes classes which organize the type checking of VDM-SL, VDM-RT and VDM++ specifications. Most of the actual type checking is performed by the definition, expression and statement classes that collectively describe the specification (above), but the typechecker package defines the supporting classes to invoke the type checking methods of the other objects, and to represent the static environment in which type checking is performed.

Type checking is different for VDM-SL, VDM++, though the checking of the basic statements and expressions is very similar, and VDM-RT is really an extension of VDM++. Therefore there is one common abstract TypeChecker class with two subclasses: ModuleTypeChecker and ClassTypeChecker. The subclasses are constructed with a list of modules or classes – which is the overall result of a successful syntax analysis – and they implement a single abstract method from their parent, called typeCheck. The method takes no arguments and returns no result. Any errors or warnings raised during type checking are recorded by the parent class (see VDMMessage).

The sequence of events is slightly different for the type check of modules and classes, but they follow the same principles. For modules, the sequence is:

- Check for duplicate module names in the list passed
- For each module, generate its definitions' implicit definitions (like pre and post functions)
- For each module, check the export definitions exist and are of the declared type, and make a list of exported definitions for the module.
- For each module, go through the import definitions and resolve against the exports.
- Create a list of all definitions from all modules (including their imports), create an Environment that contains them all, and attempt to perform type resolution on them – ie. find the type definition for every named type.
- In the pass order: [types, values, definitions], for each module, create a ModuleEnvironment representing the visible definitions, and type check the definitions of the given pass. This calls the typeCheck method on the definitions, which calls the similar method on the definitions subparts, if any.
- Report any discrepancies between the final checked types of the modules' definitions and their explicit imported types.
- Any definition names that have not been referenced or exported produce "unused" warnings.

There are a couple of important points to note:

Firstly, the syntax analysis does not understand anything about the relationship of a type name in a declaration to its type definition. All type names come through from the syntax phase as UnresolvedTypes, which simply have a name. So a very early phase of the type checking must find the corresponding type definition, in order to understand the structure of the type and what is/is not a legal type manipulation. This is done on a global basis, even though not all types are in scope for a

module (all type names are fully qualified with a module name, so there is no ambiguity). A subsequent pass, which uses just those definitions that are visible, will subsequently spot any scope problems.

Secondly, “environments” are used to support the type checking. An environment (a subclass of the abstract Environment class) is essentially a list of names and corresponding definitions that are in scope at any point. The different subclasses allow the different scope rules to be followed, so for example a module’s type checking will start with a ModuleEnvironment that references a single module definition, and understands the rule about resolving names from its imported definitions and its own definitions. Different environments are then chained together, so when the module environment is passed to (say) a function definition, the type checking creates a new environment with local definitions for the names that are generated by the parameter patterns. Since the parameter names are in scope for the body of the function, the chain of two environments is passed to the type checking of the body expression. That may in turn involve “let” expressions that define further local variables that are chained onto the environment before their body is type checked, and so on. As the chain unwinds (as typeCheck methods return), each environment in the chain is checked to see whether all of its definitions were referenced; any that were not referenced generate “unused” warnings.

The two most important methods on an Environment subclass are findName and findType, which lookup definitions by name (using the same methods on the Definition classes they reference). There is also a name scope parameter passed to findName to indicate what sorts of names are in scope – for example, functions “see” value and parameter names, operations see these names and state variables, and the post conditions of operations see names, state and “old” names. The name scope (mask) is passed around the tree of typeCheck invocations as the names that are visible in a given context is generally only known by the caller – eg. an expression does not know that it is part of an operation, so it must be told from the outside that state variables are in scope.

Very similar principles are followed by the ClassTypeChecker, which performs the following actions:

- Make sure there are no duplicate class definitions.
- For all classes and their definitions, generate the implicit definitions. This includes the construction of the class type hierarchy and the implicit local names for access to inherited definitions. VDM-RT specifications limit what can be done in system classes here.
- Create a public class environment that can see all public class definitions.
- For each class, chain a private class environment to the public environment, and perform type resolution on the definitions in the class.
- For each class, check for overloading and overriding of its definitions.
- In the pass order: [types, values, definitions], for each class, create a private class environment, and type check the definitions of the given pass.
- Check for any definition names that have not been referenced or exported, and produce "unused" warnings.

The use of public and private class environments to control the resolution of names is exactly analogous to the module case, though the class versions use the static/public/protected/private definition modifiers to decide on visibility.

The TypeComparator is used at the heart of type checking. The "compatible" method is used to decide whether two Types are assignment compatible (with "possible" semantics). This involves finding the “underlying” type (eg. removing the names of types) and making flexible recursive comparisons where unions are involved. It also has to have recursive defence, since type structures may reference themselves. Two optional types are always considered compatible (as they could both be nil). Compound types that involve more than one subtype (sets, sequences, maps, functions and operations, records and classes) are unpicked and their subtypes recursively compared. Finally, a Type.equals comparison is made between simple types.

The TypeComparator is also involved in proof obligation generation for subtype testing (see below.

This is effectively a "definite semantics" check). The "isSubType" method takes two types and returns a boolean indicating whether the first is a subtype of the second – for example, a nat1 is a subtype of a real (all nat1 values are real values), but not the other way round. With union types, a simple type is a subtype of a union if it is a subtype of any of the union members; a union type is a subtype of another union if every member of the first union is a subtype of the second union.

2.12.1. Comments

I had a lot of trouble getting the order of the initialisation and type checking right to be able to deal with all the specifications in the test suite. I'm not 100% sure that there aren't still obscure orderings of declarations that will defeat it.

The TypeComparator is currently a static class, which means that its methods need to be synchronized to work with VDM++ threads (it has state for recursion defence). This isn't really necessary, but there are quite a few places where the code would have to create a new TypeComparator instance if this is changed.

Type checking error messages are produced by static methods on the TypeChecker class, and the error count is held statically there too. This is because it is otherwise difficult to find the type checking object instance deep in a typeCheck call chain. This is in contrast to the syntax analysers, where a Reader keeps track of the errors for itself (plus any from the Readers it creates).

2.13. vdmj.pog

Class Summary	
ProofObligation	The abstract root of all proof obligations.
***Obligation	A particular type of proof obligation.
ProofObligationList	A list of proof obligations.
POContext	The abstract root of all obligation contexts.
PO***Context	A particular type of obligation context.
POContextStack	A stack of obligation contexts.

Enum Summary	
POType	An enumeration of the various proof obligation types.

The vdmj.pog package defines classes that support the generation of proof obligations. Most of the actual obligation generation is performed by Definitions, Expressions and Statements, which include a getProofObligations method that returns a ProofObligationList.

A typical proof obligation contains a stack of nested "contexts" in which a particular obligation must be determined, plus a specific obligation to check. Therefore there are two distinct class hierarchies in the pog package: the POContext hierarchy, and the ProofObligation hierarchy.

For example, if "m" is a map of int to int, a simple function like:

```
f: int -> int
      f(i) == if i < 10 then m(i) + 1 else m(i) - 1;
```

would generate two proof obligations, requiring that the two m(i) map applications are correct in the "then" and "else" branches, respectively:

```
A`f(int): map apply obligation in 'A' (test.vpp) at line 15:32
```



```
(forall i:int &
  ((i < 10) =>
    i in set dom m))

A`f(int): map apply obligation in 'A' (test.vpp) at line 15:46
(forall i:int &
  (not (i < 10) =>
    i in set dom m))
```

Notice that both obligations contain an outermost "forall" that represents the possible function parameter values, and that the "if" test value is determined to be true or false in order to check the map application in the two branches. These two form the "context" of the proof obligation; the last line in both POs is the actual obligation, which tests that the argument is within the domain of the map.

The outer forall context is represented by a POFunctionDefinitionContext object, and the if/else contexts are represented by POImpliesContext and PONotImpliesContext objects, respectively. The proof obligation itself is a MapApplyObligation. The two contexts form a stack for each obligation, and these are held by a POContextStack, which extends Stack<POContext>.

To generate proof obligations for an entire specification, an empty POContextStack is created and the getProofObligations method of each definition is called, passing the stack. Depending on the definition type, the implementation may add to the context (eg. a function definition would push a POFunctionDefinitionContext) before calling getProofObligations for their inner expression(s) or statement(s). In the example above, the getProofObligation method of the IfExpression that comprises the function body would be called. That in turn would push an POImpliesContext before generating obligations in its "then" sub-expression, **popping** the stack, and pushing a PONotImpliesContext before generating obligations for everything in the else-if list, and final else. Lastly it would pop the stack back to the state it found it in before returning a list of all the generated POs.

```
public ProofObligationList getProofObligations(POContextStack ctxt)
{
	ProofObligationList obligations = ifExp.getProofObligations(ctxt);

	ctxt.push(new POImpliesContext(ifExp));
	obligations.addAll(thenExp.getProofObligations(ctxt));
	ctxt.pop();

	ctxt.push(new PONotImpliesContext(ifExp));	// not (ifExp) =>

	for (ElseIfExpression exp: elseList)
	{
		obligations.addAll(exp.getProofObligations(ctxt));
		ctxt.push(new PONotImpliesContext(exp.elseIfExp));
	}

	obligations.addAll(elseExp.getProofObligations(ctxt));

	for (int i=0; i<elseList.size(); i++)
	{
		ctxt.pop();
	}

	ctxt.pop();

	return obligations;
}
```

Notice that the IfExpression does not actually generate any proof obligations itself; it only sets up context so that obligations generated from its sub-expressions will be correct. The ApplyExpression actually generates the proof obligations in this example:

```
public ProofObligationList getProofObligations(POContextStack ctxt)
{
```



```
        ProofObligationList obligations = new ProofObligationList();

        if (type.isMap())
        {
              MapType m = type.getMap();
              obligations.add(
                    new MapApplyObligation(root, args.get(0), ctxt));

              Type atype = argtypes.get(0);

              if (!TypeComparator.isSubType(atype, m.from))
              {
                    obligations.add(new SubTypeObligation(
                                args.get(0), m.from, atype, ctxt));
              }
        }
        ...
```

Here, the apply expression is first tested for whether it is a map application (it could be a function or operation application), and if so, a new MapApplyObligation is created, which is passed the context. Similarly, if the apply argument type is not a subtype of the domain of the map, a SubTypeObligation is also generated.

The constructor of the ProofObligations generated use the context passed to generate the "value" of the obligation (ie. the string form of its expression):

```
public MapApplyObligation(
      Expression root, Expression arg, POContextStack ctxt)
{
      super(root.location, POType.MAP_APPLY, ctxt);
      value = ctxt.getObligation(arg + " in set dom " + root);
}
```

The getObligation method on the context stack will generate a string composed of the contexts in the stack, plus the string passed in for the obligation. It is also responsible for the indentation and bracketing of the expressions. All ProofObligation subclasses are similar to the example above, though the more complex ones take a lot of effort to generate the string of the obligation. The most complex obligation is SubTypeObligation, which has a recursive private method to generate all the subtype tests that are required for the "structure" of the type being considered.

The process of proof obligation generation is exactly analogous with operation definitions which include statements – though often, statement obligations are generated without any context since in general it is too difficult to determine the scope of side effects generated by a specification.

2.13.1. Comments

The proof obligations generated are based on those in the CSK POG test suite. It is possible that there are other obligation types that should be added, especially in the case of VDM++.

ProofObligation subclasses generate the entire string of the obligation, including the context in which it is generated. This seemed better than keeping the context objects with the obligation, but it does mean that the structure of the PO is lost in the flat string.

The generation of expression strings depends on the accuracy of the toString methods for Expressions. These have been tidied up, but unfortunately preserving the precedence of the original operators means that many expression strings end up being excessively bracketed.

2.14. vdmj.runtime

Class Summary	
Interpreter	An abstract VDM interpreter.
ModuleInterpreter	The VDM-SL module interpreter.
ClassInterpreter	The VDM++ and VDM-RT interpreter.
Context	A class to hold runtime name/value context information.
RootContext	An abstract context for the root of a function or operation call.
ObjectContext	A root context for object member invocations.
StateContext	A root context for non-object member invocations.
ClassContext	A root context for static member invocations.
Breakpoint	The concrete root of the breakpoint hierarchy.
Stoppoint	A breakpoint where execution must stop.
Tracepoint	A breakpoint where something is displayed, but execution continues.
SourceFile	A class to hold a source file for source debug output.
ThreadState	A class to hold runtime information for a VDM thread.
VDMThread	A class representing a VDM++ thread.
VDMThreadSet	A class containing all active VDM++ threads.
AsyncThread	A class representing a VDM-RT thread.
ControlQueue	A class enabling two AsyncThreads to synchronize operations.
BUSPolicy	An enumeration for supported BUS policies.
CPUPolicy	An enumeration for supported CPU scheduling policies.
CPUThread	A pair, comprising a Java Thread and a CPUValue.
Holder<T>	A container for updating type T, protected by a ControlQueue.
MessagePacket	The parent class of all BUS message types.
MessageRequest	A request BUS message.
MessageResponse	A response BUS message.
RunState	An enumeration with VDM-RT thread run states.
SchedulingPolicy	The abstract parent of CPU scheduling policies
FPPolicy	The Fixed-priority CPU scheduling policy.
FCFSPolicy	The First-come First-served CPU scheduling policy.
SystemClock	The coordinated time source for VDM-RT.
ThreadObjectMap	A map of Java Thread IDs to VDMJ objects.

Exception Summary	
ContextException	A fatal interpreter error, including a “stack” context to dump.
DebuggerException	An exception used to stop the interpreter cleanly from the debugger.
ExitException	An exception used to implement exit statements.
PatternMatchException	A non-fatal exception indicating a pattern match has failed.
StopException	An exception passed between threads to cause them to abort.
ValueException	A non-fatal exception concerning an evaluation.

RTException	An exception used to stop VDM-RT threads at the end of execution.

The vdmj.runtime package defines the abstract interpreter, and its subclasses to interpret VDM-SL, VDM++ and VDM-RT specifications. It includes classes to represent the runtime execution context, exceptions which can be generated at runtime, and the debugging and thread control classes.

At its simplest, an interpreter has to create a runtime environment that represents the globally visible name/values in a specification, then evaluate a function or operation indicated by the user, returning the result.

The runtime name/value pair environment is held by the Context class and its sub-classes. A Context extends a HashMap<LexNameToken, Value>, so it is capable of storing and retrieving named values. In the same way that Environment objects were chained together during type checking, Context objects can be chained together as evaluation creates new named values, and those names later disappear from scope.

Contexts allow named values to be retrieved by searching the present context, and then subsequent chained contexts. Hence a context for global variables might be chained with one for a function's parameters, and a further one defining variables in a "let" expression. Then a lookup of a variable would search this chain in reverse order.

In the case of functions or operations calling other functions or operations, the name searching should not proceed down the context chain beyond the nearest function or operation point – ie. if func1 calls func2, func1's variables are not in scope in func2. But global variables are visible in this case. Therefore a sub-class of Context, called RootContext is used to represent points in the context chain where the search should "jump" down to the global level. In fact there are three sub-classes of RootContext, one of which, ObjectContext, is specialized to hold an object value for "self" (which is in scope at the start of object member calls), another, ClassContext refers to a ClassDefinition for static invocations, and the third, StateContext, is able to have a module's state data attached (the actual state that is visible changes as the operation call stack jumps from module to module, so this must be held in the context chain somewhere).

To create the global environment, the interpreter asks the default module or the ClassList passed to create the name/values from their definitions. The ClassList initialize method will dump all class' public static values into one global public static Context object; while the ModuleList initialize method will let each module initialize itself. The ClassInterpreter will then take the global public static scope and add the private static scope of the default class before starting. The ModuleInterpreter takes the initial context from the default module before starting.

The code for the two interpreters' execute methods are similar therefore. Here is ClassInterpreter's:

```
@Override
public Value execute(String line, DBGPReader dbgp) throws Exception
{
	Expression expr = parseExpression(line, getDefaultName());
	Environment env = getGlobalEnvironment();
	Environment created =
		new FlatCheckedEnvironment(createdDefinitions.asList(), env);
	typeCheck(expr, created);

	return execute(expr, dbgp);
}

private Value execute(Expression expr, DBGPReader dbgp)
{
	mainContext = new StateContext(
		defaultClass.name.location, "global static scope");

	mainContext.putAll(initialContext);
	mainContext.putAll(createdValues);
	mainContext.setThreadState(dbgp);
	clearBreakpointHits();

	CPUValue.resetAll();
```



```
        BUSValue.resetAll();
        Value rv = expr.eval(mainContext);
        VDMThreadSet.abortAll();
        CPUValue.abortAll();
        TransactionValue.commitAll();

        return rv;
}
```

Note that the public method is given a string expression to evaluate, so first it parses and type checks the expression given. The private method contains the important bit: a new StateContext is created, the global public static content is added, any created values are added, the threads system is initialized, the breakpoint hit counts are cleared, and lastly the expression passed is evaluated in the context constructed – CPU/BUS resets and thread aborts being wrapped around the execution. The return value is the result of the execute method.

Breakpoints are objects that exist in all expressions and statements – ie. inside anything that is directly executed. Breakpoint is the main class, with two sub-classes: Stoppoint and Tracepoint. All breakpoints have a “check” method which is called at the start of every expression and statement execution (and so must be fast!). The method is responsible for deciding whether to stop, and also recording that the statement or expression was reached for code coverage. In the case of a Stoppoint, the code must stop if it has no “test” expression, or the test evaluates to true (this is a user-inserted breakpoint); a Tracepoint will not stop, but it must evaluate its “show” expression and display it; and a regular Breakpoint will only stop if we are single-stepping. If the breakpoint decides to stop, it instantiates a DebuggerReader class, which is responsible for interacting with the user via the command line (alternatively, it interacts with the current DBGPReader object – see 2.18). As users set and clear breakpoints, the affected expression or statement is found (using findExpression) and the breakpoint member of the Expression or Statement object affected is changed by DebuggerReader. The interpreter manages the list of current breakpoints set, and the DebuggerReader has a reference to the interpreter. This also enables the DebuggerReader to evaluate “ad hoc” expressions when a stop point is reached. The SourceFile class enables selected source files to be read and lines displayed for clearer single stepping and breakpoint management, as well as displaying test coverage output.

The DebuggerReader’s run method is internally synchronized on the DebuggerReader class. This is to prevent more than one thread (in VDM++) from gaining control of the console. It unfortunately means that single stepping can have confusing results if many threads have actually stopped and are waiting for access to the console. See the User Guide for more information about this.

A StartStatement creates new VDMThread classes in VDM++ (which extend java.lang.Thread) to execute the threads started. These create new ObjectContexts to form the root of the thread based on the object instance started, and otherwise perform a normal execution of the “thread” statements defined in the class, via a synthetic ThreadStatement. VDMThreads add themselves to the set managed by VDMThreadSet, and this also provides methods to suspend, resume or abort all threads in one go, which is used by the command line debugger.

Context objects include a ThreadState field, a reference to which is copied as Contexts are chained together, and the initial value of which is set when execution of a thread starts. The ThreadState tells the breakpoint system whether the current thread is suspended or single-stepping etc., and is used in the check method, and is manipulated by the DebuggerReader.

VDM++ thread scheduling is left entirely to the Java thread scheduler. All threads have the same priority.

The creation and scheduling of threads in VDM-RT is completely different to that in VDM++. This is because thread scheduling, the passage of time, and CPU/BUS resource allocation is directly relevant to the model.

The class used at the centre of VDM-RT thread execution is AsyncThread. This is similar to VDMThread used in VDM++, but it allows for the threading of arbitrary operations (eg. threads are created implicitly when an "async" operation is called in VDM-RT, as well as for explicit thread starts). AsyncThread is also responsible for creating a regular sequence of thread executions for a "periodic" thread. This is because VDM-RT requires periodic threads to execute their statement at the given times, regardless of whether the previous execution has completed. So periodic AsyncThreads start

another AsyncThread before they execute themselves; and all periodic AsyncThreads, except the first, wait until the given time before executing.

AsyncThreads can either call a parameterless operation, or they can call a specific operation with arguments passed over a BUS, sending back any replies to a waiting sender. In the case of a BUS invocation, the BUSValue concerned is passed the request message, and this creates the AsyncThread, passing the arguments from the client. In the case of local async operations (ie. within one CPU, but asynchronous) the AsyncThread is created directly by the OperationValue being executed (see the asyncEval method).

A ControlQueue is used for coordinating the operation of AsyncThreads. They comprise a list of waiting CPU/Thread pairs, plus methods to join/leave the queue and send/receive signals while joined. A signal will wakeup one waiter from the queue, changing its run state to RUNNING from WAITING. ControlQueues are used in combination with a Holder<T> to wait for responses to asynchronous operation requests over a BUS. The client sends the operation request through the BUS, and waits on a Holder's ControlQueue for a MessageResponse to be returned. The BUSValue's reply method puts the MessageReply into the Holder, which signals the ControlQueue to wake up the client, which then retrieves the result.

The actual scheduling of AsyncThreads, in particular the exclusive scheduling on a CPU, is handled by the CPUValue class, which contains a SchedulingPolicy subclass (depending on the CPU declaration). Each CPUValue only ever allows one Java thread to execute at a time, giving each a variable timeslice before using the policy object to select another thread. This is covered in 2.15.

The handling of discrete time by VDM-RT is also one of its defining characteristics. This is handled in part by the SystemClock class, and in part by the duration method of CPUValue. When a scheduled thread makes a duration call (either because of a duration or cycles statement, or because an implicitly timed event occurs, like passing data through a BUS or swapping threads in and out) that thread is blocked and no other thread can run on that CPU. But the system cannot advance system-wide time by the amount requested until all the CPUs in the system are either idle, or their active threads are also waiting at duration methods. This inter-CPU coordination is managed by the SystemClock class. Its timestep method is called by duration, and refers to a bit mask of all CPUs to see which are idle – time can only move when CPUs are idle or at a duration point. It also keeps track of the minimum timestep requested for all CPUs, as this is the amount by which time can be moved. It also watches for the special "duration(0)" case, which causes variable update transactions to be committed for the thread in question (see 2.15).

2.14.1. Comments

The initialization of the VDM++ and VDM-RT runtime system is quite complicated, and should probably be cleaned up.

Here are a few points of general interest:

- The runtime system is initialized from the ClassList or ModuleList classes' initialize method.
- VDM++ and VDM-RT have to initialize the statics for each class first. This is done in two steps via methods on the ClassDefinitions. The first, setStatics creates the functions, operations and types – these do not have initializers; the second setStaticValues covers instance variables and values, which can make calls to other static methods in their initializers.
- Value initializers can make forward references to other values which are not yet initialized. This causes characteristic exceptions, which are caught (ignored) and produce a second pass of the setStaticValues initialization – up to a limit (currently 3). This ought to be done by working out the reference graph for the initializers.
- The static data thus initialized is written to a Context passed in (which becomes the global static environment). Static name value pairs are also written to maps inside the ClassDefinition – static data resides "in" the ClassDefinition at runtime.

And regarding object creation:

- The construction of an object (an ObjectValue) creates the superclass objects first, then for inherited fields, it adds a "locally named" **reference** to the superclass value. This is so that they can be explicitly referred to with a local name like C`x at runtime.
- Lastly, local definitions are used to override anything inherited, and the set of members are passed, together with the superclass objects, to the ObjectValue constructor.
- If the class defines an operation with the same name as the class (a constructor), it is called.

At runtime, an object's "self" values are accessed via an ObjectContext which refers to the self ObjectValue. The check method of the context first searches the local contents for the name, then uses the "get" method of the ObjectValue (and lastly uses globals, if the name is not in self). The ObjectValue's get method is told whether the name is explicit or implicit (ie. whether it is qualified with a class name and a backtick), and uses this to either create a local name and search each object in the contained hierarchy, or uses the explicit name to search the hierarchy. If a static function or operation is called, there is no self, only static values. These are managed via a ClassContext, which refers to a ClassDefinition rather than an ObjectValue. The ClassDefinition's "get" method is conceptually the same as the ObjectValue's except it searches the static values of the class hierarchy.

The ControlQueue mechanism for coordinating threads and message passing is tidy, and it would be nice if the scheduling of AsyncThreads on a CPU could be handled similarly – this is just a matter of controlling multiple waiters for a resource, which is what CPU thread scheduling does. At the moment this is done directly by the CPUValue class (see 2.15)

2.15. vdmj.values

Interface Summary	
ValueListener	Implemented by classes that watch for changes to a Value.

Class Summary	
Value	The parent of all runtime values.
****Value	A value of type ****. There are about 30 of these.
NumericValue	The root of the numeric value types.
ReferenceValue	The root of the value classes which reference another value.
InvariantValue	A ReferenceValue which includes an invariant function.
UpdatableValue	A ReferenceValue in which the referenced value can be changed.
NameValuePair	A class to hold a name and a runtime value pair.
NameValuePairList	A list of name/value pairs.
NameValuePairMap	A map of name/values.
Quantifier	A class representing a quantifier.
QuantifierSet	A class representing a set of quantifiers.
State	A class for holding a module's state data.
ValueList	A sequential list of values.
ValueMap	A map of value/values.
ValueSet	A set of values.
BUSValue	A VDM-RT interconnecting BUS.
CPUValue	A VDM-RT CPU.
TransactionValue	A VDM-RT thread-updated value.

The vdmj.values class contains a set of value classes to represent runtime values in the interpretation of a specification. They are all subclasses of the abstract Value class.

All Values implement a set of standard methods to allow them to work correctly with the Java collections framework: equals, hashCode, toString, clone.

The convertTo method takes a Type argument and is responsible for the dynamic type conversion of values from one type to another, where possible. This tests the -dtc settings flag, returning the value unchanged if the flag is set. The method is used when types are “enforced” in a specification, such as when values are passed as typed arguments, or when values are returned from a typed function or operation. The method is specialized in all the Value subclasses that generally know how to convert themselves into a small number of related types (eg. conversions between subclasses of NumericValue). If a value cannot convert itself, it delegates to its superclass. The convertTo at the root of the Value hierarchy deals with common complex cases: converting to a union (iteratively trying to convert to each type); converting to a parameter type (looking up the parameter’s name to get its actual type); converting to optional types (convert to the underlying type); and converting to a named type (convert to the underlying type, then wrap with an InvariantValue to enforce any type invariant).

A ReferenceValue is an abstract class that contains a value, and delegates all operations to that contained value. The concrete subclasses are InvariantValue, which includes an invariant FunctionValue as well as the value; and UpdatableValue, which has a set method capable of changing the value referenced (and all its methods are synchronized to allow for safe access to the changed value from other threads). When an UpdatableValue is created, it is passed a ValueListener (optionally), which is called back when the set method is called. This is used to invoke class or state invariant functions when values are changed.

Most value classes are immutable, in the sense that they are constructed with an underlying value that is held in a (public) final field of the class, and never changes. The only exception to this is the UpdatableValue, whose referenced value can be modified by the set method – note this changes the immutable value referenced; it does not change the value of the referenced object.

UpdatableValues are used for “state” values (instance variables, module state and “dcl” values that can be changed in an assignment). They are often initialized with structured immutable values, so values implement a method called "getUpdatable" to create UpdatableValue versions of themselves and their sub-values.

A Value can be held together with a LexNameToken in a NameValuePair, and such pairs can be collected into a list or a map for convenience. For example, the members of an ObjectValue are held in a NameValuePairMap.

Internally, many value classes have basic Java types at their heart. The basic values underlying SetValues, SeqValues and MapValues are ValueSets, ValueLists, and ValueMaps respectively. The basic value of a Value can be extracted using one of several methods, like realValue or functionValue. If the Value concerned cannot be converted to the basic type sought, a ValueException is thrown. The basic underlying values are used in the eval method of expressions to (say) actually perform arithmetic.

VDM-RT processing uses three special values to support the passing of messages over buses between CPUs, and the update of shared variables by threads:

A CPUValue represents a CPU declaration in a VDM-RT system class. Static data in the class allow some operations to be performed over all CPUs declared in the system, while constructors allow individual CPUs to be created with particular scheduling policies for their threads. A CPUValue maintains a map of Java thread IDs to objects (ObjectValues) which are currently executing operations of those objects on the current CPU (though they may be idle). Values are added to this map when a thread is allocated to a CPU, which happens when an AsyncThread is created; the map is cleaned up when the thread dies.

When a thread is executing normally, the Breakpoint check method is called at the start of every expression or statement. In VDM-RT, this checks a timeslice counter, and if it has expired, the reschedule method of the thread's CPUValue is called. This gives up control to any other thread which may be RUNNABLE on the CPU, suspending itself (waiting on the CPUValue object, in Java terms). When this current thread is again runnable, the reschedule method returns the new timeslice to the Breakpoint code and execution resumes.

The yield method of CPUValue is called from reschedule, as well as other places. It consults the SchedulePolicy object associated with the CPU, asking for the identity of the next thread to execute. If there are no threads ready, the CPU is idle (and the SystemClock is updated to that effect). When something subsequently happens to wake up the CPU (for example, a reply arrives for an inter-CPU request) the thread concerned is moved to a RUNNABLE state by calling the CPUValue's setState method (which informs the policy of the change), followed by the wakeUp method to update the best thread to run, and notify the waiting threads. The best thread is then allowed to continue (and update the SystemClock that the CPU is no longer idle). All CPUValue methods which produce a loggable action use the RTLogger class (see 2.17) to send a VDM-RT log message to the appropriate place.

A BUSValue similarly represents a BUS connection in VDM-RT. A BUS comprises a link between a set of CPUs. Access to the BUS is exclusive for the duration of a transmission of any message (though not for the duration of a complete request/reply). This exclusivity is managed by the BUSValue, using a ControlQueue (see 2.14).

A BUSValue has two important methods: transmit and reply. The transmit method is used to send a message, and causes a new AsyncThread to be created and associated with the target CPU, as well as introducing a delay for the transmission, depending on the size of the data passed (the algorithm takes the length of the toString of the data Value as the duration). The reply method puts a MessageReply into a Holder passed from the caller with the request, finally changing the state of the caller's thread to RUNNABLE and waking up the CPU (in case it is idle – though it may not be). As with CPUValues, loggable actions to do with message passing are logged using the RTLogger class.

In VDM-RT, when multiple threads access a shared variable – eg. a piece of state information in an object – updates to the variable value are only visible *outside* the current thread after the next timestep (while changes are obviously visible inside the modifying thread). A value cannot be modified by two threads concurrently in VDM-RT (this is a runtime error). This processing is managed by a TransactionValue class. This is a subclass of UpdatableValue, where the set method is overridden to write the new value to a local field rather than to the referenced value from the parent. The various retrieve methods (intValue, booleanValue etc) are overridden to use the new value for the thread that changed it, and the parent value otherwise. An additional commit method actually sets the parent value to the new value. The commit method is called during a time step, making the per-thread value visible to other threads.

2.15.1. Comments

It seemed sensible to have all Values contain some sort of primitive value, including sets, sequences and maps, rather than having the Value classes extend the Java collection framework directly. That allows the Value types to be in a "VDM" hierarchy, but it leads to the confusing situation where a MapValue contains a ValueMap – the former extends Value, the latter extends HashMap<Value,Value>. The ValueXXX classes should only be used internally – ie. they should never be returned from an eval method (they can't be, as they're not Value subtypes).

There is some more discussion about VDM-RT and time handling in section 2.14.

2.16. vdmj.commands

Class Summary	
CommandReader	A class to read and perform commands from standard input.
ClassCommandReader	A class to read and perform class related commands from standard input.
ModuleCommandReader	A class to read and perform module related commands from standard input.
DebuggerReader	A class to read and perform debugging commands from standard input.

The vdmj.commands package contains the command line readers that implement the interactive actions of the suite. This should be the only package that interacts with the user terminal.

A subclass of CommandReader is instantiated to provide the main command prompt of the interpreter: either ClassCommandReader or ModuleCommandReader. These classes can set or clear breakpoints, but they do not permit commands like “step” and “next”, as these are only legal from a breakpoint context. The (few) differences between the two classes concern the commands that are legal for each – eg. "classes" and "modules" – while the truly common code is in the parent.

At a breakpoint, a DebuggerReader is used to permit the extra debugging commands allowed from this context, and prevent the ones that aren’t (like initializing the environment on the fly). The one DebuggerReader is used to handle debugging sessions for VDM-SL and VDM++.

2.16.1. Comments

The idea here is to isolate the console interaction from the rest of the program so that the core interpreter could be included in (say) an Eclipse plugin. In practice, both Tracepoints and Stoppoints have to write to the console as well. This area needs some work to completely separate the implementation from the console control, so that breakpoints could interact with (say) Eclipse.

The CommandReader system works well for simple specifications, but multi-threaded debugging, especially in VDM-RT, is awkward. The *VDMJ Client*, which uses the DBGP protocol to a separate VDMJ process, is better for such debugging.

2.17. vdmj.messages

Class Summary	
VDMMessage	The root of all reported messages.
VDMError	A VDM error message.
VDMWarning	A VDM warning message.
Console	A class to provide System.in/out with a charset encoded wrapper.
Redirector	An abstract class to redirect output.
StdoutRedirector	A class to redirect standard output to a debugger.
StderrRedirector	A class to redirect standard error to a debugger.
RTLogger	A VDM-RT class for logging thread, CPU and BUS activity.

Exception Summary	
MessageException	An exception to carry a textural error message.
NumberedException	An exception to carry a number and text information.
LocatedException	An exception to carry number, text and location information.

The vdmj.messages package contains classes and exceptions to hold and display error and warning messages. Lists of VDMMessage values are returned from the TypeChecker and Reader classes. Similarly, internal exceptions that carry message numbers and position information are all subclasses of NumberedException.

The Console class manages output to stdout/stderr, in particular managing the character set to be used. The Redirector and its concrete subclasses are used to redirect or copy stdout and stderr to a debugger (see 2.18).

The RTLogger is used by VDM-RT specifications to write loggable events – such that the timing diagram can be analysed using *showtrace*. The class caches events in memory, occasionally flushing

them out to a given PrintStream. By default this is the Console. It is also possible to turn logging off (the default state), which causes log events to be discarded.

2.18. vdmj.debug

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.

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.debug 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 2.1). The command line arguments are as follows:

```
-h <host> -p <port> -k <ide key> <-vdmsl | -vdmpp | -vdmrt>
-e <expression> {<filenames>}
```

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 expression 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:/...").

Depending on the dialect, DBGPReader creates an instance of VDMPP, VDMRT or VDMSL (see 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 VDMPP or VDMSL 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 could 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 or VDM++.

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 2.14). But when the process is being remotely debugged, the breakpoint will find the DBGPReader object associated with its thread context, and call its "stopped" method. This sets the state of the debugger 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 debugger 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). 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 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 main thread may still be running – if the IDE chooses to stop all threads at a breakpoint, it must do so by interrupting the other threads by sending commands on their connections, though to do so, the DBGPReader must be able to handle asynchronous commands (receiving commands while it is running, not just at breakpoints). This functionality is not yet available.

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 2.17), 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.

2.18.1. 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 simply 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 "interrupt" method of VDM threads.

If you attempt to evaluate something at a breakpoint which causes another breakpoint to be hit, the system will trap back into the DBGPReader again (ie, three re-entrant calls to "run"). This should behave as you expect (ie. continue will take you back out to the first breakpoint), but whether it does what the Eclipse IDE expects remains to be seen. Ideally, breakpoints need to be disabled while stopped at a breakpoint, though there may be issues with multiple threads which can still be running.

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 send 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.

2.19. vdmj.util

Class Summary	
Utils	A utility class.
IO	The native IO library
VDMUtils	The native VDMUtils library
MATH	The native MATH library
Base64	A class to encode/decode base64 strings.

Utils is a small utility class. It just defines a few methods for printing out a comma separated list of

arbitrary types held in a Java List<T>.

The IO, VDMUtils and MATH classes implement the native interceptors for the standard CSK library routines. These are “not yet implemented” in the VDM source code, but the VDMJ classes for “not yet implemented” will intercept various reserved function/operation names and call these class' static methods instead.

Base64 does what you would expect. It is used by the DBGp protocol.

3. Thread Scheduling

The VDM-SL dialect does not have the notion of threads, and all functions and operations are evaluated by the interpreter in a single thread of control. However, the VDM++ and VDM-RT dialects do define threads, and suitably written specifications may evaluate functions or operations in parallel, with language mechanisms being provided for coordinating threads. This is mentioned in section 2.14, but this section takes a closer look at how threads are implemented, and in particular how they are scheduled and coordinated in the different dialects.

3.1. Java Thread Scheduling

VDMJ implements VDM threads using Java threads. The initial evaluation is executed in the "main" Java thread, and the execution of start statements, or certain asynchronous operations in VDM-RT, causes new Java threads to be created.

The Java language does not define the operation of the underlying JVM thread scheduler. Instead, Java places various constraints on the ordering of events that must occur between threads, in the light of synchronization primitives that control access to shared resources. This means that although a given Java program will have a well defined partial ordering of some of its operations, the thread scheduler has a great deal of freedom on how to execute threads that are not using synchronized access to variables or methods. In particular, there is no way to control which thread gets control of which (real) CPU in the system, or for how long, though this can be influenced by setting a thread priority.

Building on this, VDMJ has to use Java synchronization primitives to enforce the semantics of the various VDM language features to control concurrency (permission guards and mutexes). But as the thread scheduling is still delegated to the JVM, the unpredictability of VDM thread execution remains. VDMJ sets all threads to the same default priority.

VDM++ does not define thread scheduling either, so the model of VDM threading implemented by VDMJ is valid (assuming it has no bugs). The situation is more complicated with VDM-RT, as this places far more constraints on thread activity, but the language still does not define the thread scheduling and VDMJ implements a valid VDM-RT model using Java threads.

3.2. VDM++ Thread Scheduling

As explained above, VDM++ threads are implemented as Java threads, the scheduling of which is up to the JVM, given the synchronization constraints in the Java program.

A VDM++ thread is implemented by the class VDMThread. This extends Java's java.lang.Thread, and so its "run" method is executed in a new thread when its "start" method is called.

The VDMThread constructor is passed an ObjectValue, being the VDM object that the new thread belongs to, and the Context from the thread which created the new thread – this is used to find out whether the creating thread is stepping in a debugger, in which case the new thread is started in a stepping state too. A new ObjectContext is created for the new thread, linking to the global context. The operation to execute is always called "thread" (as created by ThreadDefinitions), and is held in the ObjectValue.

The Java thread's run method then just executes the "thread" operation, with no arguments, in the newly created object context. If the thread is defined to be periodic, the ThreadDefinition will have generated a thread operation that comprises a PeriodicStatement. This is created with the name of the operation to call periodically, together with an interval. The execution of the statement therefore calls the operation periodically, sleeping for the remainder of the period after execution, though it does not do anything special for executions which take longer than the period.

If multiple VDM++ threads are created like this, in the absence of guards and mutexes they will run independently with very little influence on each other. There is a small amount of Java synchronization, for example on UpdatableValues in the "set" method, to make sure that changes made to the same variable by multiple threads are not corrupted (one will succeed, but which is undefined).

Some VDM specifications may be written with independent threads like this, but more usually guards and mutexes are specified. When a VDM++ thread calls an operation with a guard or mutex (a mutex is just a guard in disguise), then a synchronized test of the guard has to be performed before the thread may continue into the body of the operation. An operation with a guard has an OperationValue with an Expression field called "guard" set to the guard expression.

The important point is that the guard evaluation, and the subsequent increment of the #act counter and progress into the operation are atomic. Multiple threads may want to call the operation, and in the case of a mutex guard, only one is allowed to proceed. Therefore the calculation of the guard, and the increment of the #act counter are made within a synchronized block, locking the ObjectValue to which the operation belongs (ie. "self"). Here's a simplified version:

```
private void guardPP(Context ctxt) throws ValueException
{
	synchronized (self)	// So that test and act() are atomic
	{
		while (!guard.eval(ctxt).boolValue(ctxt))
		{
			try
			{
				self.wait(DEADLOCK_DELAY_MS);
			}
			catch (InterruptedException e)
			{
				Breakpoint.handleInterrupt(...);
			}
		}

		act();
	}
}
```

So while the guard evaluates to false (we can't proceed) we do a Java "wait" on the self object, which releases the synchronization lock. When the guard evaluates to true, we increment the #act counter while still holding the lock. Returning from this method then enters the operation code. Whenever the code passes an "act()", "req()" or "fin()" call, it makes a Java "notifyAll" of self, waking up all threads who may be interested in the changed circumstances in order to re-evaluate the guard.

The Java wait terminates after a small timeout even without a notify, re-acquiring the lock and re-checking the guard. This is done for two reasons: deadlock is detected this way; and guard expressions may depend on "external" variables, the updates of which do not cause Java notifyAll signals.

There are no threads in the system other than the main thread and VDM threads created by the specification, so deadlock has to be detected by the threads themselves. They can't do this if they are all locked (ie. deadlock has occurred), so they periodically come out of the wait state, and in the full version of the code, counters are checked to see whether there has been any successful guard activity *anywhere*. If there is no req/act/fin activity for long enough (ie. by any operation in any class), then deadlock is assumed. This is poor, since although the deadlock delay is configurable, it is not the same as knowing that all threads are blocked. It also causes a pause of a few seconds before a genuine deadlock is detected.

If a guard depends on the value of an instance variable or static variable in another class, updates to that variable's value really ought to signal (notifyAll) the self objects which may have operation calls blocked on guards that wish to see the change. But VDMJ does not do this level of dependency analysis. Instead, the periodic re-test of the guard is used to catch such cases. This is poor, since it keeps the blocked threads busier than they could be, and they are not as responsive in the event of a change which would release them.

3.3. VDM-RT Thread Scheduling

VDM-RT has the same basic thread constraints as VDM++ regarding guards and mutexes, but in addition there is the concept if a CPU, where a given VDM thread is allocated to a CPU and can only run "on" that CPU – ie. can only run when there are no other threads running on that CPU. The scheduling of threads on CPUs is not defined by VDM-RT, though it is possible to select from a number of (undefined) scheduling policies using quite types, such as <FP> (fixed priority) and <FCFS> (first come first served).

VDM-RT threads are modelled as Java threads in VDMJ. Therefore the extra constraints needed have to be implemented in terms of Java synchronization between the objects concerned. All threads in a VDM-RT specification are implemented by the AsyncThread class, which extends java.lang.Thread. This is similar to the VDMThread class used by VDM++, but is more sophisticated to allow the more complex models required by VDM-RT.

VDMJ interprets the VDM-RT real-time model from [8] to mean that threads on separate CPUs are independent, in the same way that VDM++ threads are independent. The difficulty comes in implementing a CPU-based scheduling policy using synchronization. Here, as well as following the usual guards, a thread allocated to a CPU cannot execute if another thread is executing on the same CPU. This suggests that the scheduling should be performed by the CPUValue object, and it should delegate scheduling policy to a policy object.

A CPUValue is a subclass of ObjectValue. This is because VDM-RT declares CPUs as though they are objects of type CPU. When an application object is deployed to a CPU, the ObjectValue representing it is associated with a CPUValue.

When expressions and statements are evaluated in VDMJ, they regularly enter the "check" method of Breakpoint in order to decide whether to break the execution. This mechanism is also used in a VDM-RT evaluation to timeslice execution within a CPU:

```
if (Settings.dialect == Dialect.VDM_RT)
{
      state.reschedule();
}
```

The state is a ThreadState object, obtained from the runtime Context:

```
public void reschedule()
{
      if (CPUValue.stopping)
      {
            throw new RTException("CPU Stopping");
      }

      if (!atomic && --timesliceLeft < 0)
      {
            timesliceLeft = CPU.reschedule();
      }
}
```

That calls into the CPU on which the thread is executing. The reschedule method will not return again until it is this thread's turn to execute on the CPU, and since all threads allocated to that CPU will call into its reschedule, that method effectively enforces the unique scheduling of threads on each CPU – while Java takes care of the independent execution of threads across CPUs.

The CPU reschedule method just calls a private method called "yield" (which is called from several other places when a thread has to give up control of the CPU, for example when it is waiting for an asynchronous call to return), before returning a new timeslice, produced by the scheduling policy. Most of the Java synchronization occurs in the yield method. Here is an outline:


```
private void yield()
{
	Thread current = Thread.currentThread();
	policy.reschedule();

	synchronized (this)
	{
		runningThread = policy.getThread();

		if (runningThread != current)
		{
			ObjectValue object = objects.get(current);
			log "ThreadSwapOut -> ...
		}
	}

	if (runningThread != current)
	{
		synchronized (this)
		{
			notifyAll();
			boolean idle = (runningThread == null);

			if (idle)
			{
				SystemClock.cpuRunning(cpuNumber, false);
			}

			while (runningThread != current)
			{
				try
				{
					wait();
				}
				catch (InterruptedException e)
				{
					throw new RTException("stopped");
				}
			}

			if (idle)   // ie. was idle
			{
				SystemClock.cpuRunning(cpuNumber, true);
			}
		}

		duration(SWAPIN_DURATION);

		log "ThreadSwapIn -> ...
	}
}
```

The first call to policy.reschedule() causes the policy delegate to work out which thread should currently be running, which is returned by the getThread method and assigned to runningThread. It is *critical* that the runningThread value is updated under the synchronization lock on the CPU, since this will control which thread is allowed to proceed for the next time slice. It is perfectly possible that the thread which should be running is the one that is already running (especially if there's only one!), so if this test succeeds, no further changes are necessary and the method can return. But if there needs to be a thread-swap, we first log the current thread swap *out*, then throw a notifyAll at the CPU before waiting in a loop until our thread identity is runningThread again. All the other threads on the CPU that have called yield will be blocked in the wait call. They will awake and take the CPU lock, checking whether they are the one that should proceed. The new runningThread will be the one thread to succeed, and raise a log regarding its swap-in on the way out of the method.

If the CPU actually had no threads running (they were all waiting for something else) then the runningThread is set to null by getThread. The CPU is *idle* in this case. When an idle CPU becomes busy, a note of this fact is made in the SystemClock class; similarly, when a busy CPU becomes idle as a result of a yield, this is also noted in the SystemClock class.

Lastly, before a yield continues, a duration call is made to move time forward for the duration of a thread swap (two units). This is also a method on the CPU, but it interacts with the SystemClock class to actually move time:

```
public void duration(long step)     // NB. Not synchronized
{
      long end = SystemClock.getWallTime() + step;
      SystemClock.cpuRunning(cpuNumber, false);

      do
      {
            SystemClock.timeStep(cpuNumber, step);
            step = end - SystemClock.getWallTime();
      }
      while (step > 0);

      SystemClock.cpuRunning(cpuNumber, true);
}
```

The step passed is the amount the caller would like time to move forward by. In anticipation of having to interlock with other CPU threads to negotiate a time step, the current CPU is marked as idle – we are the current thread, and no other thread can be scheduled on the CPU until we have called reschedule, so this means that the CPU is genuinely idle until the time step has finished. The SystemClock.timestep method tries to move time by the step requested, but it may actually only move time by a smaller amount – being the smallest of all those wishing to move time at the moment. The while loop keeps the CPU idle and asks for a smaller time step until the original step is made and the CPU can be freed again.

Looking at the SystemClock class, the blocking and step negotiation is done in the timeStep method, which is static synchronized (ie. a global lock). This is simplified as follows:

```
public static synchronized void timeStep(int cpu, long step)
{
      if (step < minStepTime)
      {
            minStepTime = step;
      }

      if (runningCPUs.cardinality() == 0)
      {
            wallTime += minStepTime;

            minStepTime = Long.MAX_VALUE;
            runningCPUs.clear();
            unblock();
      }
      else
      {
            block();
      }
}
```

The interesting case is when all the CPUs are idle (running CPUs set is empty – this is manipulated by the cpuRunning calls made earlier). In this case it is safe to move time by the minimum step of all the requesters made since the last time step. The block and unblock methods just wait and notify on the class object. So notice that it is the last thread of the last busy CPU which enters this code which moves time, unblocking all the others who wake up and discover whether their request for a certain

step has been granted entirely or whether they have to wait a bit longer in the loop inside duration.

The real code for timeStep includes some extra cases which commit transactional variables – value changes which are not visible outside the thread which changed them until after a time step (or a step 0). They are not shown above for simplicity.

The above describes the key features of CPU synchronized thread execution, and time movement. The reschedule described (for a timeslice) called from ThreadState is only one place where a thread may yield. The other most significant place is when messages are sent via a BUS.

Busses connect CPUs, so internally there is a two-dimensional array of BUSValues (again, extending ObjectValue because BUS is a class), indexed by a pair of CPU numbers. When an operation call is made in VDM-RT, the exec method checks whether the CPU of the caller (from the Context) matches that of the deployment of the operation (from the self pointer of that operation). If the operation is on another CPU or if the operation is an "async" operation, then an asyncEval is made, rather than a localEval (the latter is always called for simple code like VDM-SL and VDM++).

Simplified, asyncEval is as follows:

```
private Value asyncEval(ValueList argValues, Context ctxt)
      throws ValueException
{
      CPUValue from = ctxt.threadState.CPU;
      CPUValue to = self.getCPU();

      log "OpRequest -> ...

      if (from != to)         // Remote CPU call
      {
            BUSValue bus = BUSClassDefinition.findBUS(from, to);

            if (isAsync)
            {
                  MessageRequest request = new MessageRequest(
                        bus, from, to, self, this, argValues,
                        null, stepping);

                  bus.transmit(request);
                  return new VoidValue();
            }
            else
            {
                  Holder<MessageResponse> result = ...;

                  MessageRequest request = new MessageRequest(
                        bus, from, to, self, this, argValues,
                        result, stepping);

                  bus.transmit(request);
                  MessageResponse reply = result.get(from);
                  return reply.getValue();
            }
      }
      else
      {
            MessageRequest request = new MessageRequest(
                  null, from, to, self, this, argValues, null,
                  stepping);

            new AsyncThread(request).start();
            return new VoidValue();
      }
}
```

So we're asynchronous, which either means the call is local asynchronous (and we don't wait for a

reply), or we're talking to another CPU, so we send a message and either wait for a reply or not depending on whether the remote operation is "async". A MessageRequest contains enough information to dispatch the operation in an AsyncThread, either locally, or via a bus.

If we're talking to another CPU, the BUSClassDefinition is used to lookup the right bus to use for the two CPUs concerned (error handling is not shown). A MessageRequest is then sent to the bus' transmit method, which will spawn a new thread on the target CPU, having moved time by an appropriate amount based on the size of the message and the capacity of the bus. If the remote call is synchronous, we wait for a reply which is sent back via the same bus, and "arrives" in the reply value. A Holder<T> is something which can be updated and signalled using a ControlQueue (borrowed from MASCOT, a Holder is really a pool in MASCOT terms). A ControlQueue manages a list of CPU/thread waiters, and when it needs to block the waiter it calls the CPU's yield method, as described above.

The transmit method of a bus also uses ControlQueues:

```
public void transmit(MessageRequest request)
{
	cq.join(request.from);

	log "MessageRequest -> ...
	log "MessageActivate -> ...

	if (request.bus.busNumber > 0)
	{
		long pause = how long to wait
		request.from.duration(pause);
	}

	log "MessageCompleted -> ...

	AsyncThread thread = new AsyncThread(request);
	thread.start();

	cq.leave();
}
```

Notice that the control queue keeps everyone else off the bus in question until the transfer has completed. There is also a duration call made (on the source CPU) between the activation and the completion. The end of the process creates a new AsyncThread, passing the request which has the operation and argument values to use.

The new AsyncThread executes its run method when Java schedules it. This is simplified as follows:

```
void run()
{
	cpu.startThread();	// Wait for swap in

	if (periodic)
	{
		if (first)
		{
			if (offset > 0 || jitter > 0)
			{
				long noise = random within jitter;
				cpu.duration(offset + noise);
			}
		}
		else
		{
			cpu.waitUntil(expected);
		}
```



```
                new AsyncThread(
                      self, operation, new ValueList(), period, jitter,
                      delay, offset, nextTime(), false).start();
        }

        try
        {
                Context ctxt = new ObjectContext...
                Value rv = operation.localEval(args, ctxt);
                response = new MessageResponse(rv, request);
        }
        catch (ValueException e)
        {
                response = new MessageResponse(e, request);
        }

        if (request.replyTo != null)
        {
                request.bus.reply(response);
        }

        cpu.removeThread();     // Log the destruction of the thread
}
```

The CPU cannot actually start running the thread (in VDM terms) until it is scheduled, so the first thing that a new thread does is call the CPU startThread method. This logs the thread creation, and arranges for it to be scheduled (it yields until it is scheduled). Then, if this is a periodic thread, we have to arrange for the thread at the *next period* to be created – this is so that the new thread will run on time, even if we have not finished doing what we do. If this is the first time we've run, we calculate when we start, based on the offset and jitter passed, otherwise we wait until the expected time passed. Initially, the expected time will be "now", but the expected value passed to the second and all subsequent periodics is calculated by nextTime() which accounts for the period, jitter and delays. After the next thread has been created, we can proceed with the execution of the current thread's operation.

Execution of the operation is very normal. An ObjectContext is created, linked to the global root, the operations localEval method is called (we know it is local this time), and a MessageResponse is created from the result. If a replyTo value is passed (a Holder<MessageResponse>), the response should be sent back to the caller. This is done via the same bus as the request, using the reply method. Then the thread dies, so it is removed from the CPU, which logs the destruction.

The reply method of the bus is similar in principle to the transmit method, as follows:

```
        public void reply(MessageResponse response)
        {
                cq.join(response.from);

                log "ReplyRequest -> ...
                log "MessageActivate -> ...

                if (response.bus.busNumber > 0)
                {
                        long pause = how long to wait
                        response.from.duration(pause);
                }

                log "MessageCompleted -> ...

                response.replyTo.set(response);

                cq.leave();
        }
```


Note that the "delivery" of the response is the call to the set method of the replyTo holder. This will wake up the thread that is waiting for the response in the requesting CPU, by sending a "stim" signal through the ControlQueue within the holder. Once again, the bus is protected by its own ControlQueue to prevent it being used for other messages while the reply is in transit.