VDMJ Annotations Guide

0. Table of Contents

1. Overview

Annotations were introduced in VDMJ version 4.3.0 as a means to allow a specifier to affect the tool’s behaviour without affecting the meaning of the specification. The idea is similar to the notion of annotations in Java, which can be used to affect the Java compiler, but do not alter the runtime behaviour of a program.

VDMJ provides some standard annotations, but the intent is that specifiers can create new annotations and add them to the VDMJ system easily.

2. Syntax

Annotations are added to a specification as comments, either block comments or one-line comments. This is so that other VDM tools will not be affected by the addition of annotations, and emphasises the idea that annotations do not alter the meaning of a specification.

An annotation must be present at the start of a comment, and has the following default syntax:

```
‘@’, identifier, [ ‘(‘, expression list, ‘)’ ]
```

So for example, an operation in a VDM++ class could be annotated as follows:

```
class A
operations
    -- @Override
    public add: nat * nat ==> nat
    add(a, b) == ...
```

Or the value of variables can be traced during execution as follows:

```
functions
    add: nat * nat +> nat
    add(a, b) ==
        /* @Trace(a, b) */ a + b;
```

3. Location

Annotations are always located next to another syntactic category, even if they do not affect the behaviour or meaning of that construct. In the examples above, the @Override annotation applies to the definition of the add operation, and the @Trace annotation applies to the expression “a”.

Specific annotations may limit where they can be applied (for example, @Override only makes sense for operations and functions in VDM++ specifications), but in general annotations can be applied to the following:

- To classes or modules.
- To definitions within a class or module.
- To expressions within a definition.
- To statements within an operation body.

In each case, the annotation must appear at the *start* of a comment, before the construct concerned. Multiple annotations can be applied to the same construct, and may be interleaved with other textual comments, but each annotation must appear in its own comment. Text *after* the annotation will be ignored, for example:

```
-- @Warning(5000) - ignore this unused warning for now...
```

Some annotations may want to include significant processing in their arguments. In this case, it is possible that the annotation itself may spread over several lines, and so a block comment has to be used. However, annotations in a block comment may not be visually clear because there cannot be “comment continuation” characters, like a “*” at the start of each line. To help with this, the parser is able to merge *adjacent* line comments into a block comment, but ignoring the leading “--”. This can be visually clearer, since each line of the block is explicitly marked. To enable this, the VDMJ property vdmj.parser.merge_comments must be set “true”. For example:

```
-- @OnFail("LogCategory names are not unique: %s",
--     { cats(a).name | a, b in set inds cats &
--         a <> b and cats(a).name = cats(b).name })
( card { c.name | c in seq cats } = len cats );
```

Note that an annotation in an expression effectively acts as an operator which has a very high binding precedence. So in the @Trace example above, the annotation binds to the “a” variable sub-expression, not “a + b”. This makes no difference with @Trace, but @NoPOG and @OnFail apply to a specific sub-expression and they should be used with bracketed expressions to make that clear. For example:

```
divide: nat * nat +> real
divide(p, q) ==
    /* @NoPOG */ (p/q);
```

Without the brackets around “(p/q)” you would get a warning, and the @NoPOG would only apply to the “p”, which would therefore still generate a PO for the division operator.

```
Warning 5030: Annotation is not followed by bracketed sub-expression
```

4. Tool Effects

Annotations can be used to affect the following aspects of VDMJ's operation:

- The parser (for example) to enable or disable new language features.
- The type checker (for example) to check for overrides or suppress warnings
- The interpreter (for example) to trace the execution path or variables' values
- The PO generator to (for example) skip obligations for an area of specification.
- User defined plugins (see section 7.2).

Note that none of these examples affect the meaning of the specification, only the operation of the tool. Although it would be possible to create an annotation to affect a specification's behaviour, this is strongly discouraged.

5. Loading and Checking

A global flag can be set by an "-annotations" command line argument, or the "set" command. This flag controls whether the comments in a specification are parsed for annotations. It defaults to false, so unless the command line switch or set command is used, no annotation processing will be performed. If the set command is used from within VDMJ, the specification must be reloaded to parse the comments:

```
> set
Preconditions are enabled
Postconditions are enabled
Invariants are enabled
Dynamic type checks are enabled
Pre/post/inv exceptions are disabled
Measure checks are enabled
Annotations are disabled
> set annotations on
Specification must now be re-parsed (reload)
> reload
> ...
```

When annotations are enabled, comments are processed as follows by the parser:

- All comments that precede class/modules, definitions, expressions and statements are collected by the lexical analyser and added to the corresponding AST node by the parser.
- The parser checks the comments in an annotated node, looking for those that start with @<Name>.
- Each comment that looks like an annotation is used to attempt to load a Java class called AST<Name>Annotation from a resource file called vdmj.annotations. If the class cannot be found, the comment containing the annotation is silently assumed to coincidentally contain something that is a valid annotation syntax, but which is not actually an annotation - like using @NickBattle to refer to a person by their Twitter handle.
- For the annotation classes that load successfully, the parser instantiates each annotation, and calls it to parse the rest of the comment. By default, this will parse an optional list of expressions between brackets, but each annotation can override the "parse" method to handle its own argument syntax. If there are parse errors, the annotation is silently ignored, as it is assumed to be a comment. This behaviour can make it hard to find syntax errors in annotations! So there is a boolean VDMJ setting called “vdmj.annotations.debug” which will raise all parse errors as warnings, if set to true.
- An "astBefore" method on the annotations is called, passing the SyntaxReader that is currently processing the specification. This allows the annotation to affect the parse of the syntactic element that follows the annotation.

- The parser parses the element following the comments using the SyntaxReader.
- The parser calls an "astAfter" method on the annotations after the parse of the element, passing the SyntaxReader and the parsed AST node, to allow the annotation to affect the result of the parse or undo any changes it made to the SyntaxReader.
- The parse then continues as normal.

Note that so far there has been no checking of the annotation itself, other than its syntax.

If VDMJ correctly parses an entire specification, it next performs type checking. This is done by converting the tree of AST objects into an equivalent tree of TC objects. This includes annotations that are attached to AST nodes - AST<Name>Annotation objects are converted to TC<Name>Annotation objects, defined via the ast-tc.mappings resource, which contain code to type check the annotation itself as well as code which may affect the type check of the annotated element.

Type checking proceeds as follows:

- When the TC tree is created, AST<Name>Annotation objects from the parse are converted to TC<Name>Annotation objects.
- When the type checker starts, it calls a static "doInit" method on all loaded TC annotations. This allows them to reset or set up any persistent data that they require.
- Before the type check of an annotated element, the type checker calls the "tcBefore" method of annotations attached to the node, passing the TC node of the element and the Environment list.
- The type check of the annotated element then proceeds as normal.
- After the type check, the "tcAfter" method of the annotations is called, passing the TC node and Environment as before, but also passing the TCType of the checked node.
- When the type checker completes, it calls a “doClose” method on all loaded TC annotations. This allows them to summarise and process any information they collected.

The annotation typically uses the "tcBefore" method to type check its arguments (if necessary) and check anything it needs to check about the annotated TC node. For example, the @Trace annotation checks that its arguments are simple variable identifiers that are in scope; and the @Override annotation checks that there are no arguments, that the dialect is VDM++, that the definition annotated is an operation or function and that there is a superclass that has a definition which is being overridden by the annotated element.

After the type checking phase, if there are no errors, VDMJ will normally create a tree for the interpreter: TC classes are converted to IN classes (via the tc-in.mappings resource), and this includes annotations. Annotations which apply to classes/modules and definitions do not affect the interpreter, but those that apply to statements and expressions do (since these elements are "executed").

Execution proceeds as follows:

- When the IN tree is created, TC<Name>Annotation objects are converted to IN<Name>Annotation objects.
- When the interpreter is initialised, it calls a static "doInit" method in all loaded IN annotations. This allows them to reset or set up any persistent data that they require.
- When an annotated INStatement or INExpression is executed, the evaluation first calls the "inBefore" method of the annotations, passing the statement or expression and the runtime Context stack.
- The statement or expression is then evaluated as normal.

- The evaluation then calls an "inAfter" method on the annotations, passing the statement or expression, the runtime Context and the Value from the execution. The annotation cannot affect the return value.
- Finally the return value is returned as usual and the overall evaluation proceeds as normal.

The inBefore and inAfter methods allow annotations that affect the interpreter to either intervene before the annotated element is evaluated or to look at the result after its execution (or both).

If PO generation is required, the TC tree is used to generate a tree of PO objects, including PO<Name>Annotation classes, defined in the tc-po.mappings resource. Proof obligation generation then proceeds as follows:

- When the PO tree is created, TC<Name>Annotation objects are converted to PO<Name>Annotation objects.
- When PO generation starts, it calls a static "doInit" method in all loaded PO annotations. This allows them to reset or set up any persistent data that they require.
- Before any annotated class/module, definition, statement or expression is processed by the PO generator, the "poBefore" method of the annotations is called, being passed the POContextStack and the PO node concerned.
- PO generation of the PO node then proceeds as normal.
- After the PO generation, the "poAfter" method of the annotations is called, passing the POContextStack, the PO node and the ProofObligationList generated by the node. These can be modified by the "poAfter" method - for example, the @NoPOG annotation clears the obligation list.
- When the PO generation completes, it calls a “doClose” method on all loaded PO annotations. This allows them to summarise and process any information they collected.

Note that the same annotation (that is, one @Name comment in the source) can affect all VDMJ plugins (including user defined plugins, see 7.2), though to do so it needs to define code in the AST, TC, IN and PO trees. If an annotation affects the type checker, but not (say) the interpreter, the tc-in.mapping for the annotation should map TC<Name>Annotation to INNullAnnotation, which does nothing. The same principle applies to other unused analysis mappings; all annotations must define AST<Name>Annotation, and include the full classname in the vdmj.annotations resource.

6. Standard Annotations

VDMJ includes some standard annotations. They are provided in separate jar files which needs to be on the classpath when VDMJ is started. If its jar is not on the classpath, annotations from that jar will be silently ignored.

The standard annotations perform the following processing:

6.1. @Override

This is very similar to the Java @Override annotation, which is used to verify that a Java method overrides a superclass method, raising an error if not. In VDMJ, the override applies to operations or functions in a VDM++ class.

The typecheck of the annotation (in the TCOverrideAnnotation "tcBefore" methods) verifies that the dialect is VDM++, that the annotation has no arguments, and that it is applied to either an operation or a function definition. Lastly, if there is no "inherited" definition that this definition overrides, an error is raised.

```
Error 3363: Definition does not @Override superclass in 'A' (test.vpp) at line 3:9
```

The annotation has no effect on the interpreter or the PO generator.

6.2. @Trace

The @Trace annotation is intended to trace the flow of control in the interpreter, either to note that a particular point in the execution has been reached, or to log the values of variables at that location.

The typechecker checks (in the TCTraceAnnotation "tcBefore" methods) that the annotation is applied to a statement or an expression only. The check also makes sure that any arguments supplied are simple variable names and refer to variables that are in scope.

When the interpreter is running, the INTraceAnnotation "inBefore" method either just prints out the current location to stderr, or it prints the location and "<name> = <value>" for each argument (ie. each variable name traced). For example, the specification in section 2 produces the following:

```
> p add(1,2)
Trace: in 'A' (test.vdm) at line 9:13, A`a = 1
Trace: in 'A' (test.vdm) at line 9:13, A`b = 2
= 3
Executed in 0.007 secs.
```

The annotation has no effect on the execution values or the PO generator.

6.3. @NoPOG

The @NoPOG annotation is intended to suppress PO generation over a region of the specification. It can be applied to a class/module, a definition, a statement or an expression.

The typechecker (the TCNoPOGAnnotation "tcBefore" methods) just checks that the annotation has no arguments passed.

```
Error 3361: @NoPOG has no arguments in 'A' (test.vdm) at line 9:13
```

The PO generator (the PONoPOGAnnotation "poAfter" methods) clear the list of proof obligations generated for the annotated element (class/module, definition, statement or expression).

The annotation has no effect on the interpreter.

6.4. @Printf

The @Printf annotation is almost identical to the IO`printf library operation, except that as an annotation it can be used in functions as well as operations, and the arguments do not have to be presented as a sequence literal.

The typecheck (the TCPrintfAnnotation "tcBefore" methods) checks that the annotation has a string as its first argument.

Execution of the annotation (the INPrintfAnnotation "inBefore" methods) evaluate the arguments and then pass the Values generated to System.out.printf.

Note that, as with IO`printf, the format string can only contain %s converters, even if the values being printed are numeric. But you are able to use argument numbers and field widths. For example:

```
f: nat * nat -> nat
f(a, b) ==
        -- @Printf("b=[%2$5s], a=[%1$-5s]\n", a, b)
        a + b;

> p f(123, 456)
b=[  456], a=[123  ]
= 579
Executed in 0.003 secs.
```

You can also use the alternative format ‘#’ flag in conversions, like “%#5s”, which affects the output of some VDM types. Specifically:

- Chars are usually printed with quotes, like ‘x’, but the alternative prints them without.
- Strings are usually printed in “double quotes”, but the alternative prints them without.
- Empty sequences are usually [ ], but the alternative prints a blank string.
- Sequences and sets are usually printed as {[values]}, the alternative prints CSV values.
- Tokens are usually printed as mk_token(value), but the alternative just prints the value.
- Tuples are usually printed as mk_(values), but the alternative just prints the CSV values.
- Quote types are usually prints as <TYPE>, but the alternative just prints TYPE.

6.5. @OnFail and @DocLink

The @OnFail annotation is virtually the same as the @Printf annotation, except that it can only be used to annotate boolean expressions – and the expression that follows should be bracketed to make it clear. The annotation will only print the output if the evaluation of the expression is false. This is very useful to add at various points in complex boolean expressions, such as large pre/postconditions or type invariants. For example:

```
inv mk_R(p, q) ==
        -- @OnFail("p=%s, should be <10", p)
        (p < 10)

        -- @OnFail("p=%s, should be in PSET", p)
        and (p in set PSET)

        -- @OnFail("q=%s, should be >10", q)
        and (q > 10)

        -- @OnFail("q=%s, should be in QSET", q)
        and (q in set QSET);
```

If this type invariant is violated, the error message indicates that the error is where the invalid value is generated (the console, here), rather than where in the invariant that the boolean expression fails. The @OnFail annotations catch the failing sub-clause and print a helpful message:

```
> p mk_R(10,2)
p=10, should be <10 in 'DEFAULT' (test.vdm) at line 10:13
Error 4079: Type invariant violated by mk_R arguments in 'DEFAULT' (console)

> p mk_R(5,10)
p=5, should be in PSET in 'DEFAULT' (test.vdm) at line 13:13
Error 4079: Type invariant violated by mk_R arguments in 'DEFAULT' (console)

> p mk_R(1,2)
q=2, should be >10 in 'DEFAULT' (test.vdm) at line 16:13
Error 4079: Type invariant violated by mk_R arguments in 'DEFAULT' (console)

> p mk_R(1,15)
q=15, should be in QSET in 'DEFAULT' (test.vdm) at line 19:13
Error 4079: Type invariant violated by mk_R arguments in 'DEFAULT' (console)
```

You can optionally include an error number as the first argument of an @OnFail annotation, which is then printed out ahead of the message. This is useful for being able to distinguish @OnFail occurrences which have the same message.

The message body can contain the string “%NAME”, which will be substituted for the name of the enclosing VDM definition. Additionally, if the last character of the message is “$”, the location of the annotation in the VDM source is substituted. For example:

```
functions
        compare: nat * nat -> bool
        compare(a, b) ==
                -- @OnFail(1234, "Function=%NAME: Oops %s < %s $", a, b)
                (a < b);

> p compare(2, 1)
1234: Function=compare: Oops 2 < 1 in 'DEFAULT' (test.vdm) at line 4:13
= false
Executed in 0.005 secs.
```

The @DocLink annotation extends the functionality of @OnFail, allowing a specification to trace the “specification path” to the @OnFail error. @DocLink annotations can be added to class/modules, definitions, expressions and statements. As the evaluation proceeds, @DocLink annotated elements are entered and exited, which maintains a stack of locations. Then, when an @OnFail is triggered, the @DocLink path for the location is printed as well as the error message.

For example, a module might define a set of functions for validating something that is associated with a particular Chapter in a standard. That Chapter reference would be marked with an @DocLink at the top of the module. Individual functions might check Sections of that Chapter, and so the functions would have @DocLinks naming their Section. Lastly, if/then/else choices within the function might correspond to specific bullet items in the Section, which would also have @DocLinks. Then finally, when an @OnFail is issued, the Chapter, section and item references for that location would be printed too.

@DocLink arguments are string literals, typically URLs or Chapter names etc.

```
Mass = [mass]
inv m == m <> nil => allOf
([
        -- @DocLink("http://wiki.ros.org/urdf/XML/mass")
        -- @OnFail("Mass value %s must be positive at %#s", m.value, m.$loc)
        ( m.value >= 0 )
]);
```

6.6. @Warning

The @Warning annotation is used to acknowledge and suppress specific warnings at particular locations in a specification. It is passed a comma separated list of warning numbers to suppress and has this effect for the scope of the element that is annotated. For example, a warning suppression applied to a function definition will suppress all occurrences of the listed warnings in the body of the function, but not elsewhere.

In some cases, a warning is only generated when the entire specification is considered, for example unused definitions or a mutual recursion warning between functions. In this case, the @Warning can either be applied to the containing module or class, or if the suppression needs to be more limited, it can be added on the line before the occurrence of the warning. For example:

```
types
	-- @Warning(5000) T is unused until phase 2
	T = nat;

functions
	-- @Warning(5013) ignore this mutual recursion
	f: nat -> nat
	f(a) == g(a-1);

	-- Note, no annotation here
	g: nat -> nat
	g(a) == f(a-1);
```

Without annotations, that would produce the following warnings:

```
Warning 5013: Mutually recursive cycle has no measure in 'DEFAULT' (test.vdm) at line 9:5
Cycle: [g, f, g]
Warning 5013: Mutually recursive cycle has no measure in 'DEFAULT' (test.vdm) at line 5:5
Cycle: [f, g, f]
Warning 5000: Definition 'T' not used in 'DEFAULT' (test.vdm) at line 2:5
```

With the annotations shown, it would produce the one remaining warning that is not suppressed:

```
Warning 5013: Mutually recursive cycle has no measure in 'DEFAULT' (test.vdm) at line 11:5
Cycle: [g, f, g]
```

Note that, as in this example, it might be sensible to extend the annotation comment to say why a particular warning is being suppressed.

6.7. @Separate, @SepRequire and @DeadlineMet

These three annotations are provided to support validation conjectures in VDM-RT [1]. The functionality of each of the three is described in section 4 of the paper. They work with the *RTValidator* class in VDMJ to process VDM-RT log files, looking for violations of conjectures that they define.

For example, the @Separate annotation requires that two events (such as the #req or #fin of an operation) are separated by a minimum time period. If that condition is violated in the log file being processed, a record is written to a *violations* file by *RTValidator.*

These annotations implement an abstract *ConjectureProcessor* interface, which the *RTValidator* uses. This allows more conjectures to be written in a similar manner.

The type checking of the annotations requires them to be added to the constructor of the RT *system* class definition, and the annotation arguments are checked as well.

6.8. @TypeParam

This annotation is intended to narrow the types of polymorphic type parameters. VDM allows the definition of polymorphic functions, that include type parameters. But the types themselves are completely undefined – you cannot, say, directly specify that a parameter is a sequence of something or a set of something. For example:

[1] Validation Support for Distributed Real-Time Embedded Systems in VDM++, John S. Fitzgerald, Peter Gorm Larsen, Simon Tjell, Marcel Verhoef

```
-- @TypeParam @T = seq of ?;
f[@T]: @T -> nat
f(s) == len s;
```

Without the @TypeParam, this function produces a type checking error, since there is nothing to say that the @T type is a sequence with a length. By adding the annotation, the type checker knows that the type parameter is a sequence, but of an undefined type.

This also allows callers of the function to be checked. For example, the following call to the function above would be legal without the @TypeParam, but is caught now that we know it must use a sequence of something:

```
g: nat -> nat
g(a) == f[nat](a);

Error 3061: Inappropriate type for parameter 1 (test.vdm) at line 7:15
Expect: seq of (?)
Actual: nat
```

6.9. @LoopInvariant, @LoopGhost and @LoopMeasure

These annotations can be used to improve the runtime validation and generation of proof obligations for specifications that contain operations with loop statements. The annotations must be applied directly before a loop (either a *while* statement, or one of the three *for-loop* statements).

@LoopInvariant takes a single expression argument, though multiple annotations may be used, which is the same as a logical "and" of the expressions given. Each expression must be true before, during and after the loop evaluation. If applied to a *for-loop*, at least one @LoopInvariant must not reason about any of the loop-defined variables (eg. the "x" in "for x = 1 to 10"). These invariants can then be applied outside the loop body, when the variable is not in scope.

@LoopGhost can be used with *for-all* and *for-pattern* loops, to give the name of the ghost variable that is used in POs. By default it is called "GHOST$", but with nested loops this can cause problems and so an explicit name can be given. Only one @LoopGhost can be used on each loop. The ghost variable can be reasoned about in the @LoopInvariant.

@LoopMeasure can only be used for *while* loops. It is similar to the idea of measures in recursive functions, and specifies an expression whose value is a natural number and which decreases towards zero as the loop progresses. The annotation maintains an "invisible" variable called "*loop_measure_#*", where # is the line number of the annotation. It is visible while debugging.

For example:

```
op: nat ==> ()
op(a) ==
(
        dcl cnt:nat := a;
        dcl sv:nat  := 0;

        -- @LoopInvariant(cnt + sv = a)
        -- @LoopMeasure(cnt)
        while cnt > 0 do
        (
                sv  := sv + 1;
                cnt := cnt - 1;
        )
);
```

Loop invariants and measures are checked at runtime by the interpreter, as well as producing their own proof obligations. Breakpoints can be set on the annotation comment lines, and single-stepping will jump to the annotation lines when the values are being checked.

6.10. @TypeBind

This annotation can only be applied to *forall* and *exists* expressions that use type binds. The annotation provides an explicit list of values to use in the calculation of a multiple type bind within the annotated expression. This enables infinite type binds to be partially evaluated, but note that the result may be *undefined* since the bind process does not test every possible value.

The syntax of the annotation is *@TypeBind <multiple type bind> = <set expression>;*

Note that whole multiple type bind must be given. So in the example below, you cannot provide a value set for x without also affecting y.

```
f: nat -> bool
f(a) ==
      -- @TypeBind x,y:nat = {1,2,3};
      forall x,y:nat & x + y > a;

> p f(0)
= undefined
Executed in 0.036 secs.

> p f(2)
= false
Executed in 0.002 secs.
```

In this example, the function tries to check whether there is a sum of two natural numbers that exceeds a given number passed in. This is obviously *true*, but when only the values {1,2,3} are tried for f(0), the result has to be *undefined*, since although every combination of x,y is >0, further values have not been tried and may (would!) falsify the expression. However, when f(2) is tried, x=1, y=1 will fail the test and so the *forall* is a definite *false*.

The results are similar for *exists* expressions, but inverted.

These partial evaluations are similar to the ones used by the QuickCheck tool. In that case, *undefined* results are interpreted as a MAYBE result, when checking proof obligations.

6.11. @MaximalTypes

This annotation temporarily sets the “vdmj.parser.maximal_types” property for a small section of the specification. This is useful if you don’t want to set the option globally, or you are distributing a specification for others to use and you don’t want to burden them with setting the property.

It can be applied at any level (class, module, definition, expression or statement) and has no arguments. The previous setting of the property is saved and restored, so you can nest annotations if you need to (eg. both on a function and on an expression within it, which would make no difference).

For example:

```
types
      R ::
            a : nat
            b : nat
      inv r == r.a < r.b;

functions
      -- @MaximalTypes
      f: nat * nat -> R!    -- This gives an error without the annotation
      f(a, b) ==
            mk_R!(a, b);    -- This is allowed to break the invariant
```

6.12. @SetProperty

VDMJ's operation can be influenced in various ways using Java properties, as explained in the User Guide. But it can be inconvenient to set properties, particularly if different models require different settings. Therefore the @SetProperty annotation allows VDMJ properties to be set from within the specification that needs to be affected.

The annotation takes two arguments: the first is a string name of the property and the second is its value, which may be an integer literal, a boolean literal or a string literal. For example:

```
-- @SetProperty("vdmj.parser.comment_nesting", 2)
```

The annotation has effect when it is *parsed*, which means that if the option affects parsing (as the example above does) it should appear near the start of the specification file concerned. The annotation has to precede some VDM definition (ie. it annotates something), but it does not matter which one. In multi-file specifications, the change affects files which are parsed *after* the one in which the annotation occurs, because a property setting is global.

The annotation arguments are checked. As well as ensuring that there are two argument literals, the check also makes sure that the named VDMJ property exists, and that its type matches the type of the second argument.

Care should be taken when VDMJ property settings are coming from multiple sources, such as a "vdmj.properties" resource file, -D JVM options, and @SetProperty annotations. It is even more complex with VDM-VSCode which allows individual launches to set properties. The advice is to stick to one method at a time.

7. Creating Annotations

7.1. User Defined Annotations

Creating new user annotations is a matter of doing the following:

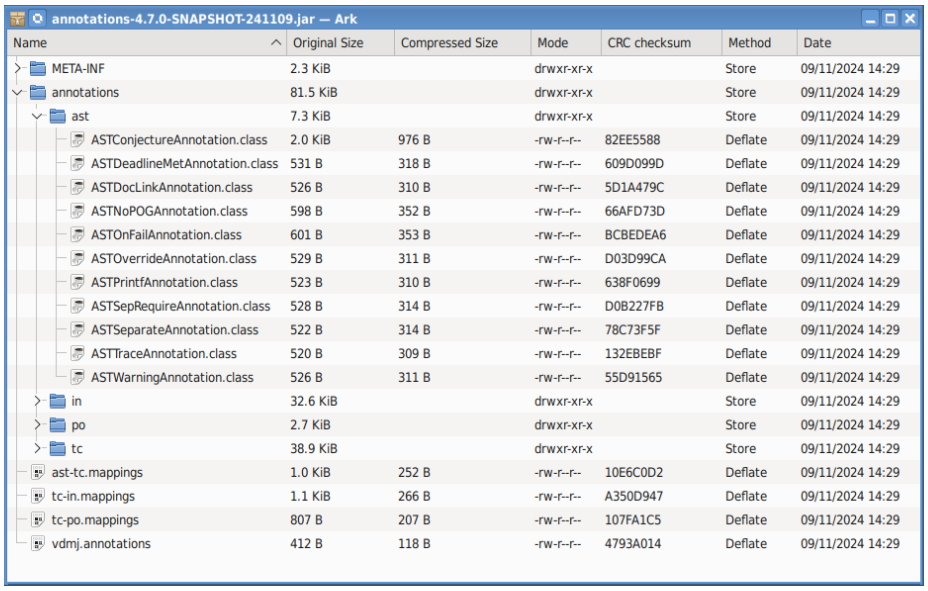
- Write a new AST<Name>Annotation class that extends ASTAnnotation. Annotations that don't affect the parse do not have to implement any methods; there are "astBefore" and "astAfter" methods that will be called during the parse, if required (see above).
- Write TC, IN and/or PO<Name>Annotation classes that extend TCAnnotation etc. and add the checking and functionality that you require in the before/after methods.
- Create the mapping file lines that map the new classes for AST-TC, TC-IN and TC-PO. For example to add two new annotations called @Notice and @Classic, produce an ast-tc.mappings file like this:

```
package annotations.ast to annotations.tc;

map ASTNoticeAnnotation{name, args} to TCNoticeAnnotation(name, args);
map ASTClassicAnnotation{name, args} to TCClassicAnnotation(name, args);
```

- Create a new resource file (or add to an existing one) called vdmj.annotations. This should contain the fully qualified classname(s) of your AST annotation(s), one per line.
- Put the new classes and the necessary resource files on the classpath when VDMJ is executed. This is easily done by putting them in a jar file, with the mapping file(s) and vdmj.annotations at the top level. Use the property vdmj.mapping.search_path to set the location of the mapping files if they are not at the top level.
- Add @Name comments to your specification :-)

Note that VDMJ is issued with @Override, @Trace, @NoPOG, @Printf, @OnFail and @Warning annotations in a separate "annotations" jar. This contains the classes and resource files required for the standard annotations. The source (in GitHub) may be a useful resource for producing new annotations. The layout of the main annotation.jar is as follows:



Name	Original Size	Compressed Size	Mode	CRC checksum	Method	Date
META-INF	2.3 KiB		drwxr-xr-x		Store	09/11/2024 14:29
annotations	81.5 KiB		drwxr-xr-x		Store	09/11/2024 14:29
ast	7.3 KiB		drwxr-xr-x		Store	09/11/2024 14:29
ASTConjectureAnnotation.class	2.0 KiB	976 B	-rw-r--r--	82EE5588	Deflate	09/11/2024 14:29
ASTDeadlineMetAnnotation.class	531 B	318 B	-rw-r--r--	609D099D	Deflate	09/11/2024 14:29
ASTDocLinkAnnotation.class	526 B	310 B	-rw-r--r--	5D1A479C	Deflate	09/11/2024 14:29
ASTNoPOGAnnotation.class	598 B	352 B	-rw-r--r--	66AFD73D	Deflate	09/11/2024 14:29
ASTOnFailAnnotation.class	601 B	353 B	-rw-r--r--	BCBEDEA6	Deflate	09/11/2024 14:29
ASTOverrideAnnotation.class	529 B	311 B	-rw-r--r--	D03D99CA	Deflate	09/11/2024 14:29
ASTPrintfAnnotation.class	523 B	310 B	-rw-r--r--	638F0699	Deflate	09/11/2024 14:29
ASTSepRequireAnnotation.class	528 B	314 B	-rw-r--r--	D0B227FB	Deflate	09/11/2024 14:29
ASTSeparateAnnotation.class	522 B	314 B	-rw-r--r--	78C73F5F	Deflate	09/11/2024 14:29
ASTTraceAnnotation.class	520 B	309 B	-rw-r--r--	132EBEBF	Deflate	09/11/2024 14:29
ASTWarningAnnotation.class	526 B	311 B	-rw-r--r--	55D91565	Deflate	09/11/2024 14:29
in	32.6 KiB		drwxr-xr-x		Store	09/11/2024 14:29
po	2.7 KiB		drwxr-xr-x		Store	09/11/2024 14:29
tc	38.9 KiB		drwxr-xr-x		Store	09/11/2024 14:29
ast-tc.mappings	1.0 KiB	252 B	-rw-r--r--	10E6C0D2	Deflate	09/11/2024 14:29
tc-in.mappings	1.1 KiB	266 B	-rw-r--r--	A350D947	Deflate	09/11/2024 14:29
tc-po.mappings	807 B	207 B	-rw-r--r--	107FA1C5	Deflate	09/11/2024 14:29
vdmj.annotations	412 B	118 B	-rw-r--r--	4793A014	Deflate	09/11/2024 14:29

7.2. Annotations in User Plugins

Section 5 describes the process of loading and checking annotations in the standard analysis plugins. But annotations can also be used by new plugins which provide their own analysis.

For example, if the user was to provide a new “EX” analysis plugin that provides some advanced type checking, that might have its own annotations and it may want to perform its own actions when it sees existing annotations, like @Warning. Let us assume it is derived from the “TC” plugin, and so defines a tc-ex.mapping file.

Typically, the following steps are required:

- The EX plugin should define an EXAnnotation abstract class which all EX annotations will extend. The form of this class is very likely to be similar to the existing classes for TC, IN and PO. But note that this class defines the before/after methods that EX annotations will use as the analysis proceeds. The exact form of these depend on what the plugin does, but typically they pass information about the part of the specification relevant to the annotation. They don’t have to use the before/after pattern, but this is often a useful way to do it.
- As with other annotations in Section 5, there must be a mapping file to convert annotations into the “EX” package. So if the @Warning annotation is required, there must be a mapping from TCWarningAnnotation to EXWarningAnnotation. If you have a new annotation that is only used by the EX plugin, it must still provide the intermediate classes to convert (say) @Ex from ASTExAnnotation via TCExAnnotation to EXExAnnotation, even though these classes are defined within the EX plugin.
- You will also need an EXAnnotationList class to receive a list of annotations from the equivalent TCAnnotationList via the tc-ex.mapping. This should also provide convenience methods to call (say) the exBefore method for all the annotations in the list.
- The EX plugin is responsible for calling methods of the EXAnnotation at the correct points in its processing, but typically there is some sort of init call that would be called while handling either the CheckPrepareEvent or CheckTypeEvent. Thereafter, as the plugin processes the specification, it should call the before/after methods defined in EXAnnotation. This either happens via EXAnnotationExpression and EXAnnotatedStatement, or if the annotation can be applied to top level classes/modules and definitions, the EXAnnotationList is called directly as these are processed by the plugin. The “TC” plugin is a good example of how and where to do this.