# 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...
```

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.

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 configurable classpath. 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.
- 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, loaded from the same classpath, 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, 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. 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 four areas of VDMJ operation, 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.

# 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.
```

## 6.5. @OnFail

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)
```

## 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.

# 7. Creating New 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);
```

- Put the new classes and the necessary mapping files on the classpath when VDMJ is executed. This is easily done by putting them in a jar file, with the mapping file(s) at the top level. Use the property vdmj.annotations to set the classpath for the annotation classes, unless you use the standard annotations.ast package for them.
- Add @Name comments to your specification :-)

Note that VDMJ is issued with @Override, @Trace, @NoPOG, @Printf, @OnFail and @Warning annotations in separate "annotations" jars. These contains the classes and mapping file extracts 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: