VDMJ Annotations Guide

0.   Table of Contents

1.   Overview

Annotations were introduced in VDMJ version 4.2.1 as a means to allow a specifier to affect the tool's behaviour without affecting the meaning of the specification. The idea is very 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 regular 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 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 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+b.

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 in a comment, by itself, 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.

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 Settings boolean field called "annotations" is set by an "-annotations" command line argument. This field can be checked by any annotation processing (and skipped if not set). It is currently only used by the interpreter.

Annotations 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 @<identifier>, optionally followed by an expression list in brackets. If there are any parse errors at this stage, the comment is assumed to merely resemble an annotation and is silently ignored.

- Each annotation that parses correctly is then used to attempt to load a Java class called AST<Name>Annotation from a configurable package. If the class cannot be found, the comment containing the annotation is sliently 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 a "before" method, 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.

- The parser calls an "after" 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, where AST nodes are converted into TC nodes which contain the code to perform type checking. This process is the same with annotations that are attached to AST nodes - AST<Name>Annotation objects are converted to TC<Name>Annotation objects, loaded from the same root package, 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 method which calls a static doInit method in 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 "before" method of annotations attached to the node, passing the TC node of the element and the Environment list. The annotation uses this to type check itself (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.

- The type check of the element then procedes as normal.

- After the type check, the "after" method of the annotations is called, passing the TC node and Environment as before, but also passing the TCType of the checked node.

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 method which 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 "before" 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 "after" 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 before and after 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 requird, the TC tree is used to generate a tree of PO objects, including PO<Name>Annnotation 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 method which 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 definition, statement or expression is processed by the PO generator, the "before" 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 "after" method of the annotations is called, passing the POContextStack, the PO node and the ProofObligationList generated by the node. This list can be modified by the "after" method - for example, the @NoPOG annotation clears it.

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.   Writing New Annotations

Adding new 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 "before" and "after" 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.

- Create the necessary 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.

- Add @Name comments to your specification :-)

Note that VDMJ is issued with @Override, @Trace and @NoPOG annotations in a separate "annotations" jar. This 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.