Analysis Separation without Visitors

Nick Battle

Fujitsu UK (nick.battle@gmail.com)

Abstract. The design of VDMJ version 3 limits the tool's extensibility because the code for different analyses is tightly coupled with the classes that represent the AST. This situation was improved with Overture version 2 by using visitors. However, while this improves the tool's extensibility, the resulting code is slow to execute and complex to work with. This work presents an alternative approach, implemented in VDMJ version 4, which separates analyses from the AST and from each other, but which avoids the complexity of the visitor pattern while maintaining the performance advantage of VDMJ.

Keywords: design patterns, separation of concerns, code complexity

1 Introduction

VDMJ [3] is a command line tool written in Java, offering support for VDM [1] specifications. As well as parsing all dialects of the VDM specification language, the tool offers three analyses: type checking, model simulation (via an interpreter, with combinatorial testing [7]) and the generation of proof obligations. The source code of VDMJ is compact and efficient, but it is not easily extended, nor would it easily support independently written analysis plugins. This is because the supported analyses are tightly coupled with each other in the same Java classes.

The Overture [2] project has produced an Eclipse [4] based VDM tool. The Overture tool is based on VDMJ, but it has been re-structured internally to de-couple and separate the analyses. This extensibility comes at a cost, however. The internal structure of the Overture code is more complex than VDMJ, and as a result it is harder to develop and maintain. Overture's analyses are also slower to execute.

This work presents recent changes in VDMJ version 4 which add similar extensibility features to Overture, but without adding extra complexity or slowing the analyses. It does this by avoiding the visitor pattern [5], instead separating analyses by producing new data structures dynamically when each analysis is selected.

Section 2 describes the design of VDMJ version 3. Section 3 describes how Overture uses the visitor pattern to separate analyses. Section 4 introduces the alternative to the visitor pattern used in VDMJ version 4. Section 5 looks at the performance characteristics of the new pattern and Section 6 considers its drawbacks. Lastly, Section 7 considers related work, Section 8 possible future work and Section 9 adds a summary.

2 VDMJ Version 3

The design of VDMJ, up to and including version 3, is simple. The parser produces an Abstract Syntax Tree (AST) comprised of objects whose classes include the code of the analyses supported. The execution of those analyses then proceeds by direct recursion over the AST. This is a version of the interpreter pattern [5].

For example, in Figure 1, the parse of the expression "1 + a" produces a tree of three objects. The classes of those objects contain code for type checking (TC), interpretation (IN) and proof obligation generation (PO) for each node, as well as the linkage and primitive data from the parser (AST).

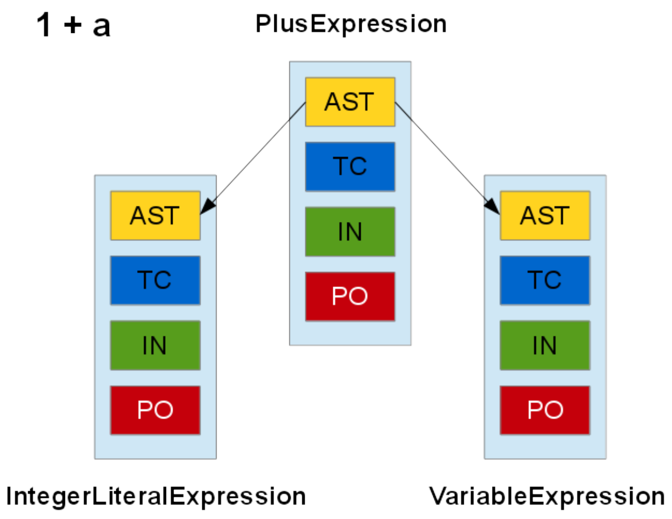

Fig. 1. Analyses in VDMJ version 3

To type check the expression, the TC method of the plus expression at the root of the tree is invoked. This recurses into the literal and variable expression TC methods, which return the types of those sub-expressions. The root then checks whether both subtypes are numeric (as required for the "+" operator) and finally deduces and returns the resulting type to its caller. The same recursive principle is used by the other analyses.

This approach is simple and has several advantages:

- Classes are small and only contain analysis code related to the one AST node concerned.
- Classes can form a hierarchy, with common code in abstract parent classes (eg. NumericBinaryExpression)
- Results from one analysis can be stored in the node and processed by other analyses of the same node.

– The execution of analyses is efficient, because all the required methods are local and can be called directly.

However, if the objective is to produce an extensible tool, this approach is limiting because the code for the different analyses is tightly coupled in the same classes. If different groups want to work on new analyses independently, they would all have to modify the same class files. Supporting a dynamic "plugin" design for analyses would be all but impossible. These points were noted in [6].

3 Overture Version 2

Overture was originally based closely on VDMJ. But to address these extensibility concerns, Overture version 2 was re-designed to use the visitor pattern [5][6]. This pattern separates the data (AST) from the analysis of that data, as well as separating analyses from each other. A simplified view of the new design is illustrated in Figure 2. Note that the AST classes now have no analysis code, and all the code for each type of analysis is collected together into one "visitor" class in separate methods.

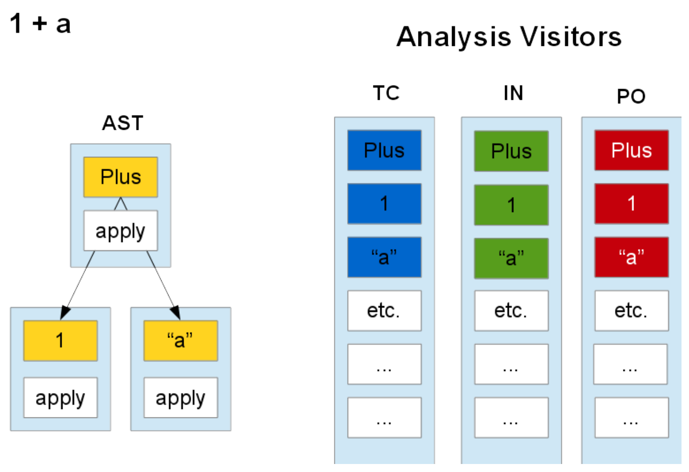

Fig. 2. Overture version 2 Analysis Separation with Visitors

This is a much more extensible design pattern, and allows analyses to be worked on independently, as well as allowing new analyses to be produced without changing the fundamental architecture. It would also support a dynamic plugin architecture in future. But when the visitor pattern is scaled up to support a complex grammar, some problems arise. For example:

– Class sizes can become very large. The VDM grammar has close to 300 node types and therefore naively, a comprehensive visitor would require 300 methods, amounting to several thousand lines of code.

– Visitor methods have no hierarchy, and so there is nowhere obvious to put common code. Private methods could be used, but that only makes the visitor classes even larger.
– Visitors are usually stateless, so there is nowhere obvious to store the node-specific results or working data of analyses.
– The solutions to the previous two points, together with the "double dispatch" semantics of the visitor pattern significantly slow down the execution of analyses.

To address the problem of large class sizes, Overture splits up the main visitors, firstly into large grammatical groups (definitions, expressions, statements etc). Secondly, large visitors are split into separate classes which extend each other. This produces a child class that acquires all the methods needed by inheritance, without including them all in one large source file. But the problem of relatively large source files remains (the largest is still 3751 lines).

To address the issue of where to put common code, Overture uses "assistants". These are usually stateless classes that contain a small group of methods that relate to common processing for a particular analysis, for a particular set of node types. This idea has merit, and keeps the common code away from the visitors (which are already large). But it can be difficult to find the right assistant for the right purpose in a given context. There are 66 assistants currently, and while they have sensible names, they are not related to the visitors and methods they serve in any obvious way (cf. common code in a class hierarchy).

The problem of where to hold analysis state is solved in one of two ways in Overture. Some information that is globally useful (for example the type information from the type checker) is stored in the AST nodes. Other information that is only used by analyses internally (for example, current breakpoint settings in the interpreter) is held by stateful assistants. This data is usually a map from AST nodes to a given type.

Storing type information in the AST may seem an obvious solution, but it implicitly creates a dependency between the type checker and the AST, and between the AST and the consumers of that type information. Consider what would happen if someone produced an alternative type checker with a different representation of types. The AST would have to change or be extended to include both types, and all type dependent analyses would have to change too.

Storing state information in a map is the only other reasonable alternative. But consider that (for example) the interpreter has to consult the current breakpoint setting for every single node that it encounters, and to do each check it has to obtain an assistant, and to do that it has to use an assistant factory. This is a significant performance overhead.

4 VDMJ Version 4

VDMJ version 4 uses an alternative approach to analysis separation that avoids many of the problems associated with the visitor pattern.

As with Overture, the AST is simplified to use objects whose classes only contain parser information. To perform an analysis (the first is usually type checking, but it need not be), the AST structure is used to create a *separate tree* that is exactly the same shape, but comprised of classes that only contain code for that particular analysis. The analysis then proceeds by recursive descent over the new tree.

When a subsequent analysis is performed that requires the results of the first analysis (for example, the interpreter uses the type checker output), the same tree creation procedure is performed, except transforming the type checker tree - along with its embedded type information - into a tree that is specialized for interpretation. The execution of the interpreter then proceeds by recursive descent over the new tree, as it did with VDMJ version 3. This is illustrated in Figure 3.

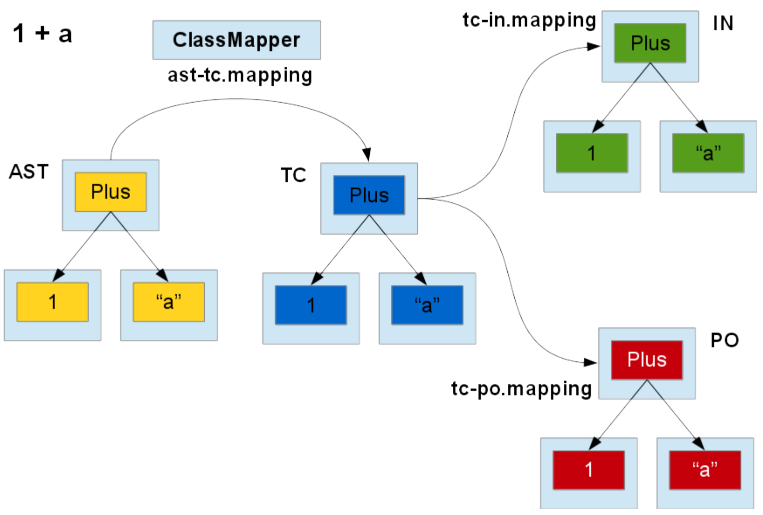

Fig. 3. VDMJ Analysis Separation by Tree Creation

The process of analysis tree creation is performed by a new component called the *ClassMapper*, which is driven by "mapping files" that describe how to create each object in the target tree, given an object in the source tree. For example, the part of the mapping file to create the TC classes from the AST classes used in the example (in the file `ast-tc.mapping`), would be as follows:

```
package com.fujitsu.vdmj.ast.expressions to
        com.fujitsu.vdmj.tc.expressions;

map ASTPlusExpression{left, op, right} to
    TCPlusExpression(left, op, right);
map ASTIntegerLiteralExpression{value} to
    TCIntegerLiteralExpression(value);
map ASTVariableExpression{location, name, original} to
    TCVariableExpression(location, name, original);
...
```

A mapping file consists of *package* lines, which define the source and target Java packages for the *map* lines that follow until the next package line. Map lines define a source class and fields within that class which should be passed to a corresponding target class constructor, with the field arguments identified. Alternatively, a line may be defined as *unmapped*, which means the target class is the same as the source class. A mapping file has a map or unmapped line for every source/target conversion that may be required, which is therefore around 350 lines.

The process of tree conversion starts at the tree root. The source class is identified in the mapping file, and the list of required fields read from the root object using Java reflection. These field values are then used to call the constructor of the target class identified. But *before* the constructor is called, the constructor arguments are themselves converted by the same process. This therefore duplicates the entire tree using the mapping. The mapper additionally keeps a cache of objects that are in the process of being converted and which have already been converted, so that cycles in the source tree can be correctly converted into the same cycle structures in the target tree.

The EBNF grammar of the mapping file is as follows:

```
mappings = { package | map | unmapped }
package = "package", pname, "to", pname, ";"
map = "map", source, "to", destination, ";"
unmapped = "unmapped", fqcn, ";"
source = cname, "{", [ fields ], "}"
destination = cname, "(", [ fields | "this" ], ")"
fields = fname, [ { ",", fname } ]

pname = <a Java package name>
cname = <a Java constructor name>
fname = <a Java field name>
fqcn  = <a Java fully qualified class name>
```

Mapping files are read on demand and cached. The Java reflection *Field* and *Constructor* classes for each map line are created and checked when the file is loaded, so that the process of creating the output tree can proceed as quickly as possible.

This tree mapping approach has several advantages:

- Analysis separation has been achieved. Analysis code is separate from the AST and different analyses are separated from each other.
- The individual class files in the analyses are very small indeed – even smaller than VDMJ version 3 – and they are tightly focussed on a single purpose. The analysis code itself is virtually identical to VDMJ version 3, but split into one class per node per analysis.
- The AST classes are immutable, with fewer fields that VDMJ version 3. This means that their constructors are more efficient and the parser is up to 30% faster than before.
- Code complexity metrics are as expected. The average methods per class is much lower; the average inheritance depth goes up (more classes have the same depth); the average method lines of code decreases because of more simple constructors.

Afferent coupling and McCabe complexity decrease. Most metrics are roughly the same.

- The classes retain the same relative hierarchy as before, and so common code can reside in abstract parent classes.
- The output from an analysis or intermediate state resides within the objects of each analysis tree. If a subsequent analysis wants to use that information, the process of creating its tree will copy a reference to the information from the source tree. Note that the parser output AST is completely immutable.
- There are still dependencies between analyses, but those dependencies are made explicit via the mappings. Multiple analyses of the same kind, such as different type checkers, can co-exist.
- Since there are no visitor redirections, no assistants or assistant factories, and no state map-lookups, the execution of the analyses goes as fast as VDMJ version 3 - that is to say, faster than Overture version 2.

However, these advantages can only be realised once the cost of the analysis tree creation has been paid. Compared to creating assistants, or looking up nodes in a map, this cost may appear prohibitive. But in practice, Java is heavily optimized for object creation, and the overall cost of the tree creation is modest. For most specifications, analysis trees can be created in a small fraction of a second.

5 Performance

The cost of analysis tree creation comes in two parts: the mapping file must be read, and then the source tree must be processed to produce the target tree. The cost of reading the mapping file is fixed (though the actual performance in a multi-process environment depends on whether disk blocks are cached, which other processes are active and so on). The memory footprint is also fixed at a few hundred Kb. However, the cost of the target tree creation depends on the size of the source tree being processed.

Both the visitor pattern and the tree creation pattern incur a performance penalty. But one important difference with tree creation is that the penalty occurs “between” analyses, whereas the visitor pattern penalty is “during” analyses. So to determine the real impact of the tree creation overhead, one important consideration is whether an analysis is one-shot, or whether it is performed repeatedly. For example, a specification need only be type-checked once, and it need only have proof obligations generated once. But many simulations may be performed with the interpreter in a given session - with combinatorial tests, perhaps millions of executions. So in the case of the interpreter, the performance advantage of the VDMJ execution engine will almost certainly outweigh the cost of the production of the interpreter tree.

In practice, the cost of loading the mapping file is negligible. All of the mapping files are roughly the same size, and are read in less than 0.2 of a second. This cost is once-only per session per analysis, so if multiple specifications were processed, this cost would only be paid once. This is not noticeable. In a multi-project workspace like Overture, the cost could effectively be paid once at startup.

The cost of tree duplication is more variable, and depends on the analysis being created as well as the size of the tree. The ClassMapper can create new tree nodes at a rate of between 200K and 800K nodes per second. Note that 500K AST nodes is roughly equivalent to a 100,000 line specification. So this means that a 10,000 line specification could be prepared for (say) type checking in a fraction of a second. Given that most specifications are less than 10,000 lines, this means that the tree duplication times are usually not noticeable either.

To illustrate this, Figure 4 compares the performance in seconds of the VDMJ version 3 and 4, and Overture type checkers for a series of VDM-SL specifications, from 500 lines to 12,000 lines. The “shmem.vdmsl” specification was used (from the Overture examples), its module being copied to produce ever larger specifications. This specification was chosen because it includes a variety of functions, operations, expressions, statements and patterns.

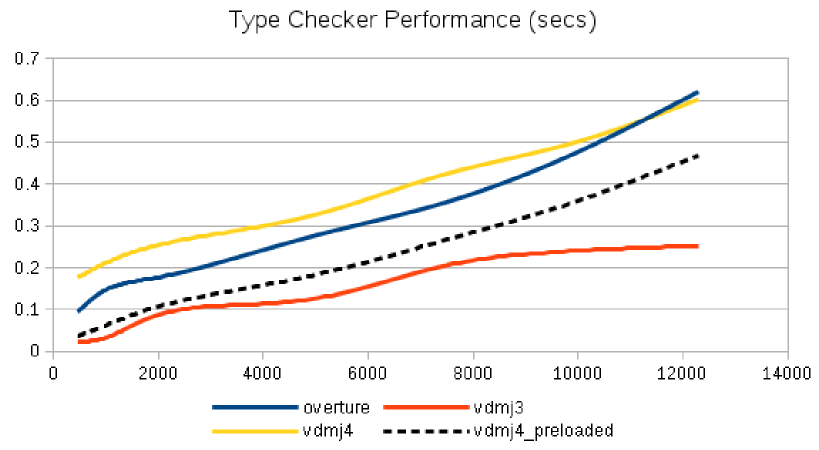

Fig. 4. Type checking performance, lines of specification per second

The figure shows the average performance of 20 consecutive executions[1]. The collection of this data was difficult, since the small differences between the performance of the various tools is comparable to the natural fluctuations that are experienced using Java on a multi-process operating system. Hence the use of averaging over multiple runs.

The execution time of VDMJ version 4 is consistently slower than VDMJ version 3, which is mostly the delay for reading the `ast-tc.mapping` file. It is also possible to see a divergence of the two VDMJ lines. This is caused by the extra delay of the tree creation, which adds between 0.03 seconds for a 500 line specification, and about 0.17 seconds for a 12,000 line specification. As a result, compared to the Overture

[1] All measurements were made on 32-bit Windows 7 running on an i7 laptop, using Java 8 with 1Gb of heap.

2.4.8 performance, VDMJ version 4 is slightly slower below about 11,000 lines in this example (the difference is less than 0.1 secs).

However, if the time to load the mapping file is removed from the VDMJ version 4 times, just leaving the tree creation time (the dotted line), then both VDMJ versions perform better than Overture, and version 4 performance is a few hundredths of a second slower than version 3 below 2000 lines. This may be a fairer comparison, since the mapping file will only be loaded once. Overture performs a type check every time a specification file is saved in the editor; at worst, the cost of reading the mapping file would be incurred once on the first save.

Figure 5 compares VDMJ and Overture type checker performance for considerably larger specifications. Here the fixed delay to load the mapping file is outweighed by the cost of the tree copying. As shown, for an 80,000 line specification, the divergence between VDMJ versions 3 and 4 is just over two seconds. But even with this extra cost, the overall performance advantage of VDMJ type checking means that it is still faster than Overture.

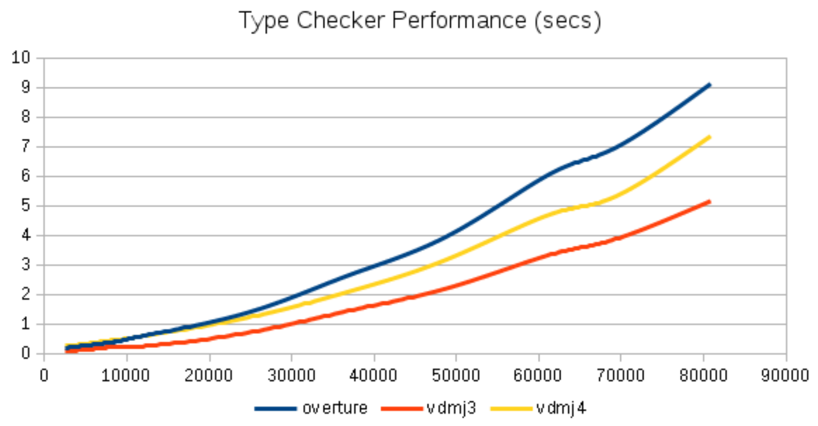

Fig. 5. Type checking performance, lines of specification per second

These figures just show the performance of the type checker tree creation, but the costs of the interpreter and proof obligation tree creations are similar. So for example, a 10,000 line specification might incur a 0.25 second delay, once, before the interpreter could be used. But thereafter the execution of the interpreter would run at "VDMJ speeds" no matter how many expressions were evaluated, which is usually faster than Overture.

6 Drawbacks

One significant drawback of the tree creation method is that it is not a recognised design pattern. That is always the case for novel solutions, but it would be wise to carefully consider possible problems, and the advantages of visitors, before adopting this new pattern exclusively.

In come cases, the visitor pattern may make code simpler than direct recursion. For example, although the "main visitors" of the analyses are large and complex, there are also many "small visitors" that implement what used to be recursive processes on the tree. Note that one visitor can only do one job because it implements one method for each node type. So a process that (say) matches a type against a pattern to derive a set of bound variable definitions must be performed by a separate visitor to the main type check visitor. There are many of these "small visitors" in Overture (over 120). The nice thing about them is that they encapsulate all of the processing for the (few) node types that are relevant in one small class. The equivalent code in VDMJ is spread over many classes, which can be harder to work with. This point was made in [6]. In principle, such small visitors could be re-used by different analyses; this would not be possible with tree creation.

Note that tree creation does not mean that the trees concerned cannot also support visitors. A hybrid solution is perfectly possible, using tree creation for the main analyses (with all the advantages listed above), but using visitors for the small recursive processes that the main analyses use.

The current VDMJ tree structures do not include links to ancestors, only to children. This is is not a fundamental issue with the VDMJ approach – such links could be added – but it is an advantage of automatically generated AST systems, such as that in Overture. This point was also made in [6].

Tree duplication occupies more memory than a single AST. But for the most part, the extra memory (over the data that was present in the VDMJ version 3 design) is comprised of object reference linkages between the new tree nodes. Even for large specifications, this overhead only amounts to a few Megabytes. In principle, analysis trees can be deleted once the analysis has been performed (for example, the AST tree is deleted in VDMJ version 4 once the type checker tree has been created from it, since the type checker tree has all the information).

7 Related Work

The Overture project's use of the visitor pattern is covered in detail in [6]. The visitor and interpreter patterns are described in [5].

The author is unaware of any detailed work discussing the weaknesses of the visitor pattern when used at scale, as raised here. Similarly, the author is not aware of any use of tree copying as a design pattern that may replace the visitor pattern.

8 Future Work

The weaknesses in the visitor pattern at scale, highlighted in section 3, may be due to Overture's implementation choices rather than because of fundamental problems with the visitor pattern. It may be that other tool implementations have solved visitor scaling problems in better ways, which should be investigated. It may also be the case that better organization of the Overture visitor code would mitigate many of the maintenance problems experienced by its developers.

The performance analysis in this work only used VDM-SL specifications. It is possible that other dialects perform differently, though this is unlikely since the majority of the AST is the same. However, measurements would give a more complete picture of the performance of the ClassMapper.

Although the analysis separation presented here would support a modular "plugin" architecture, this is not currently implemented by VDMJ. An analysis plugin would have to include its own mapping file, as well as an indication of the source analysis tree that it should be derived from. Plugins would then fit together in a dependency tree, not unlike the plugin architecture of the Apache Maven[8] build system.

The creation of new analysis classes is currently a manual process, as is the creation of new mapping files. This would benefit greatly from tool support, perhaps creating skeleton classes and a mapping file for a new analysis based on the class structure and mapping of the analysis from which it is derived.

Currently, the mapping process can only create a new analysis tree from one source tree. But it is conceivable that a new analysis would depend on information from two or more source analysis trees, as long as they were of the same shape. This would require more work in the ClassMapper.

Visitors are an elegant solution for small recursive processes that only involve a few node types. An overall improvement may be achieved by converting these processes into visitors, which would reduce the size of the analysis classes even more.

9 Conclusion

A method of analysis separation has been presented which does not use the traditional visitor pattern. The new method - separation by analysis tree creation - retains many of the advantages of tightly coupled code without the problems of the coupling. The performance penalty of tree creation has been shown to be small compared to the overall analysis cost.

The practicality of the solution has been demonstrated by its inclusion in VDMJ version 4.

References

1. International Organization for Standardization, Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language International Standard ISO/IEC 13817-1, December 1996.
2. P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef, "The Overture Initiative Integrating Tools for VDM". ACM Software Engineering Notes, vol. 35, no. 1, January 2010.
3. "VDMJ User Guide". Technical report, Fujitsu Services Ltd., UK, 2009. `https://github.com/nickbattle/vdmj`.

4. The Eclipse Integrated Development Environment, `http://www.eclipse.org/`.
5. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, “Design patterns, Elements of Object Oriented Software”, Addison Wesley, 1995.
6. “Migrating to an Extensible Architecture for Abstract Syntax Trees”, Luis Diogo Couto, Peter W. V. Tran-Jørgensen, Joey W. Coleman, and Kenneth Lausdahl
7. Peter Gorm Larsen, Kenneth Lausdahl, Nick Battle, “Combinatorial Testing for VDM”, Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods, Pisa, September 2010.
8. Apache Maven, `https://maven.apache.org/index.html`.