Using JML-based Code Generation to Enhance Test Automation for VDM Models

Peter W. V. Tran-Jørgensen[1], Peter Gorm Larsen[1], and Nick Battle[2]

[1] Aarhus University, Department of Engineering, Finlandsgade 22, 8200, Denmark
{pvj,pgl}@eng.au.dk
[2] Fujitsu Services, Lovelace Road, Bracknell, Berkshire. RG12 8SN,UK
nick.battle@gmail.com

Abstract. The Vienna Development Method (VDM) uses contract-based elements such as invariants to constrain data, and pre- and post conditions to specify intended behaviour. Data and behaviour can be validated against these contract-based elements using test automation for VDM. This technique uses traces – a kind of pattern – to specify test sets, which can be executed against the VDM model. In this paper we demonstrate how traces can be code generated to Java and used to test a version of the VDM model, implemented as a Java Modeling Language (JML) annotated Java program. This approach has potential to allow a larger number of tests to be executed since the tests are run as compiled code rather than using a VDM interpreter. To study the performance of code generated traces, execution times are compared to those obtained using different VDM interpreters.

1 Introduction

Inspired by TOBIAS [22, 23], the Vienna Development Method (VDM) [10, 14] has been enhanced with combinatorial testing – a test automation feature that uses a pattern-based notation to describe and execute test sets. In VDM, combinatorial testing is used to validate data and behaviour against invariants and pre- and post conditions [18, 19]. The tests are generated from a pattern, referred to as a *trace*, which describes a finite subset of all possible executions of a VDM model. Combinatorial testing for VDM is supported for an executable subset of VDM by the Overture [20] and the VDMJ [1] interpreters.[3] However, execution of large combinatorial test sets can be slow and demanding in terms of memory resources. To improve on this situation, this paper presents a technique that allows traces to be executed as compiled code.

Using Overture's [17, 5, 25] VDM-to-Java code generator [15, 28] a VDM-SL model can be translated to a Java program, where the contract-based VDM elements are described using Java Modeling Language (JML) constraints [21]. In this paper we extend this code generator to support traces.[4] When a code generated trace is executed using a

[3] We note that although VDMJ used to be the core of the Overture tool, Overture has been redesigned to promote its extensibility [6, 7]. The Overture interpreter and VDMJ are therefore different interpreters, although the former is inspired by the latter.

[4] The trace code generator is included in the Overture tool.

JML tool, the trace tests are expanded and run against the code generated version of the VDM specification. The verdict of each test is determined by checking the generated code against the JML constraints. There are two benefits to this approach. First, it has potential to support faster execution of larger test sets since the tests are executed as compiled code rather than using a VDM interpreter. Secondly, it allows the trace to be used to validate the generated code against the properties described by the VDM model. In our work we use the OpenJML [3] runtime assertion checker to execute the code generated trace. The reason for this is that OpenJML is the only tool that we are aware of that currently supports Java 7 as well as the subset of JML produced by Overture's Java code generator.

We have used code generated traces to analyse properties of an algorithm used to obfuscate Financial Accounting District (FAD) codes, which are used to identify branches of a retailer. The algorithm was modelled in VDM and validated using combinatorial testing. One of the interesting aspects of this case study is that it involves the generation and execution of one million tests, which code generated traces enabled us to execute. However, as we shall see, very recent advances in trace expansion techniques allows traces to be executed much more efficiently. Currently VDMJ is the only VDM interpreter that supports this expansion algorithm. To study the performance of code generated traces we compare the execution times to those obtained using Overture and VDMJ.

Combinatorial testing has been researched for several years [24] with most of the work centred around tools that support combinatorial testing for different programming languages [29]. At the level of specification languages it is worth comparing our work to [9]. In their work, Dick et al. use a finite-state automaton approach to support test sequencing of implicit-style VDM models. However, since their technique uses symbolic values, one needs to provide the means to select concrete ones.

Similar to VDM, TOBIAS supports test case generation from a pattern that describes a test set. The test cases generated by TOBIAS can be output as sequences of VDM operation calls or as JUnit [16] test cases. In particular, the latter can be used to test a Java implementation against a JML specification. Although our work currently only supports Java and JML, it also enables code generation of the VDM specification in addition to the tests. Our technique may, however, be adopted to use other Design-by-Contract (DbC) implementation technologies (section 6).

The test automation technique presented in this paper, is also comparable to what can be achieved using SAT solvers [2] since both techniques conduct an analysis for a finite collection of cases. However, while SAT solvers are limited by the number of values for each domain, our approach is guided by traces for the finite number of combinations to be analysed. Similar to our approach the Spin model checker translates a Promela model into an optimised C-program that performs exhaustive exploration of the state space [12]. In order to avoid the state explosion problem that is inherent to model checking, one needs to limit the size of the state space subject to exploration. In VDM the state space is limited by the desired paths undertaken, which is expressed using a trace.

The structure of the paper is as follows: after this introduction, combinatorial testing for VDM is described in section 2. Next, our approach to code generating traces is

presented in section 3. Afterwards, the performance of code generated traces is studied using an industrial case study in section 4. This is followed by a discussion of the performance results in section 5. Finally, this paper concludes and outlines future work in section 6.

2 Background

2.1 VDM

VDM is one of the longest established formal methods for the development of computer-based systems. In VDM data are defined by means of types built using constructors that define records and collections such as sets, sequences and mappings from basic values such as booleans, characters and numbers. Data types can be further constrained with type invariants and the overall system state can similarly have a state invariant defined. Functionality is defined in terms of functions and operations over data types. These can be defined implicitly by pre conditions and post conditions that characterise their behaviour, or explicitly by means of specific algorithms. Function and operation arguments are passed by value, which avoids the complexity of value aliasing.

Collectively, the state/type invariants and pre- and post conditions define "contracts" that the specification must meet. A specification implicitly defines functions that represent each of its constraints. For example, a pre condition defines a total function which has the same parameters as the function (or operation) that it guards and a boolean result; the body of the function is the pre condition expression. Similarly, a type definition with an invariant implicitly defines a total function with parameters that match its value constructor and a boolean result; the body of the function is the invariant expression.

As an informative annex to the VDM-SL ISO standard [13], a module-based extension has been added to the language. This extension enables structuring of a VDM-SL model into a collection of modules with capabilities to import and export definitions to/from other modules. A module may define a single state component which can be constrained by an invariant.

2.2 Combinatorial testing of VDM models

Combinatorial tests are a form of animation that allow large numbers of tests to be generated using patterns. A combinatorial test generator expands these patterns, or traces, by considering every possible combination of values that would match the pattern. Then for each combination of values, the system is reset and an animation performed. It is not unusual to generate hundreds of thousands of tests this way, which therefore explores far more of the possible system states than ad-hoc animation testing.

Traces are closely related to normal VDM-SL expressions, but expand "looseness" to consider all possible results. For example, in a normal specification the expression `let a in set S be st p(a)` selects a value from the set `S` such that `p` is true. If there are many such values in `S`, VDM does not define which one is selected, only that the one selected meets `p`. In a trace, the same expression generates a new test, with a new binding for `a`, for every value in `S` that meets `p(a)`. Similarly, the non-deterministic statement `||(op1(), op2(), op3())` usually means that the three

operations are called sequentially, but in an undefined order. When this appears in a trace, it generates one test for every possible ordering of the calls.

Futhermore, when a trace contains two or more clauses that would expand to multiple tests, a test is generated for every *combination* of the expansions. For example, the trace in listing 1.1 calls the operation `op` with every possible combination of a value from the set `A` and a value from the set `B`.

```
let a in set A in
  let b in set B in
    op(a, b);
```

Listing 1.1. Example of trace in VDM.

At the end of the expansion, every generated test is an ordered sequence of variable assignments and operation or function calls. So if `A`=$\{$`1`$\}$ and `B`=$\{$`7,14`$\}$, the example above would expand to `a=1; b=7; op(a,b)` and `a=1; b=14; op(a,b)`. Each test will then execute successfully if the sequence of operation calls do not violate any constraints, either in the creation of the variable values or in the execution of the operations. In this context, a constraint violation is a condition that leads to a runtime-error such as an invariant violation or division by zero. A test which does not violate any constraints is considered a `PASS`. A test which breaks a constraint is normally considered a `FAIL`, but tests which violate a constraint directly when an operation is called from the test is considered `INCONCLUSIVE`. The reason is that it is possible that the specification is correct but the test generation is at fault. For example, the generation may produce test cases that violate the outermost operation pre conditions.

2.3 Code generation for VDM-SL

The work presented in this paper is implemented as an extension of Overture's VDM-to-Java code generator. This code generator represents each VDM-SL module using a **`final`** Java class that has a **`private`** constructor to protect against instantiation and subclassing. The module class uses a **`static`** field to represent the state component (if defined) and functions and operations are translated to **`static`** Java methods that are added to the module class.

The generated Java code is supported by a small runtime library, which includes Java implementations of some of the VDM types and operators. For example, sets, sequences and maps are implemented using the `VDMSet`, `VDMSeq` and `VDMMap` classes, which are extensions of standard Java collections.

Overture's Java code generator can translate pre- and post conditions and invariants of VDM-SL specifications to JML annotations that are added to the generated Java code [28]. JML is a DbC specification language, used for specification of Java classes and interfaces. To demonstrate how the code generator uses JML, consider a code generated method `f`, with input parameters $\texttt{i}_1, \ldots, \texttt{i}_n$, derived from a user-defined VDM-SL function. Then `f` has a code generated **`static`** pre condition method `pre_f` with the same parameters as `f`. To indicate that `pre_f` has no write-effects it is

marked with the JML **pure** modifier. In addition, `f` is annotated with the **requires** `pre_f(i1,...,in)` annotation to make `pre_f` a pre condition of `f`. The generated Java program can be validated against the JML annotations in order to ensure that the generated code meets the properties described by the contracts of the VDM specification.

The Java code generator also produces JML checks to ensure the consistency of VDM types when they are translated to Java. This is achieved using a function `Is(v,T)` which checks that a Java value or object reference `v` is consistent with the VDM type `T` that it originates from. For example, consider a Java value or object reference `v` which represents a VDM identifier that is either a natural number or a boolean value, i.e. it is of type **bool|nat**. In the generated Java code `Is(v,`**bool|nat**`)` is a JML predicate that is used to test whether `v` is either **true**, **false** or some positive integer. For this simple example the JML annotation checks that `Utils.is_nat(v) || Utils.is_bool(v)` is true. More complex types, such as user-defined types constrained by invariants or collection-based types, generate more complicated JML checks. The full definition of `Is(v,T)` and a description of how this function is used by the Java code generator, is described in detail in [28]. Some of the checks generated by `Is(v,T)` uses functionality of the Java code generator's runtime library, including a small extension of it called `V2J`. For example, `V2J` has functionality to check if a Java object represents an injective mapping or a non-empty sequence, which is used to check type constraints in the generated code.

The trace code generator uses the JML annotations to detect contract and type violations in the generated code. This is used to determine the verdicts of the trace tests. For example, if one of the tests violates a JML post condition then this test is considered a `FAIL`. As another example, a test that passes arguments to an operation that do not match the operation signature – with respect to the VDM-SL specification – is considered `INCONCLUSIVE`. This is detected using the JML predicates produced by `Is(v,T)`.

3 Code Generating Traces

Internally Overture represents a trace as an Abstract Syntax Tree (AST) composed of nodes that correspond to the different kinds of trace constructs (e.g. a **let** binding or a non-deterministic statement). This AST represents a pattern that can be expanded into a test set. The code generator takes a similar approach to representing traces by constructing the trace AST using trace constructs or nodes available via the Java code generator's runtime library: the `Alternative` trace node is used to represent the tests produced by the `|` trace operator or the **let be st** bindings. The **let** binding only defines trace variables. The `Concurrent` trace node represents the `||` trace operator and expands to all possible orderings of the tests of its child nodes. The `Repeat` trace node is used to repeat tests a specified number of times according to some repetition pattern, e.g. `op(x){1,2}`. The `Sequence` trace node expands to the sequencing of the tests of its child nodes. The `Statement` trace node expands to a single test, which is the invocation of a `Call` statement. The `Call` statement nodes constitute the leaves of the trace AST.

In order to construct a code generated version of the trace, the code generator produces Java code that when executed builds a trace AST, composed of nodes from the Java code generator's runtime library. To demonstrate the process of code generating a trace AST, consider the VDM-SL trace in listing 1.2. In this listing, the non-deterministic choice between `fun(x)` and `op1(x)` produces two tests: `fun(x); op1(x)` and `op1(x); fun(x)`. The repetition of `op2(x)` further produces two tests: `op2(x)` and `op2(x); op2(x)`. Since the repeated and concurrent trace operators are grouped as alternatives, and the tests are expanded for all bindings for `x`, the total number of tests accumulates to eight. These tests are shown in listing 1.3.

```
let x in set {1,2} in (
  ||(fun(x),op1(x)) | op2(x){1,2}
)
```

Listing 1.2. Example of a trace specified using VDM-SL.

```
x = 1; fun(x); op1(x);   /* Test 1 */
x = 1; op1(x); fun(x);   /* Test 2 */
x = 1; op2(x);           /* Test 3 */
x = 1; op2(x); op2(x);   /* Test 4 */
x = 2; fun(x); op1(x);   /* Test 5 */
...
x = 2; op2(x); op2(x);   /* Test 8 */
```

Listing 1.3. The tests generated from the trace in listing 1.2.

At runtime the code generated version of a trace is represented as an object tree composed of the different trace nodes and trace variables used during the expansion of the trace. For the trace in listing 1.2, the trace AST is visualised using the Unified Modeling Language (UML) object diagram in fig. 1.

Expansion and execution of the trace is handled entirely by the runtime library, and performed using the `ExecTests` method in the `TraceNode` class. The execution of the trace tests is shown using a UML sequence diagram in fig. 2. In this figure `ast` represents the trace AST, `module` is the code generated version of the module enclosing the trace, `testAcc` is used to record the test results, and finally `store` is used to manage system states between the different test runs. As described in section 2, the system is reset between each test, i.e. the tests are executed independently. The code generator uses the `store` to achieve this when executing the code generated version of the trace. The test accumulator (`testAcc` in fig. 2) receives information about each test that has been executed via the `registerTest` method. This method call has been omitted from fig. 2 to keep the figure simple. A test accumulator is implemented as a strategy [11], i.e. one test accumulator may print the test results directly to the console, another test accumulator may write the results to the file system and so on.

As a first step of the test expansion and execution, the module class enclosing the code generated trace is registered in the `store`. In case there are other module classes

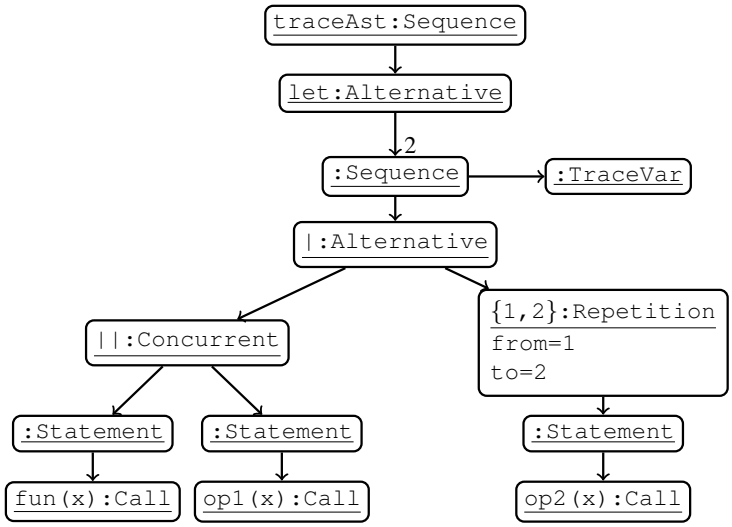

Fig. 1. Runtime representation of a code generated trace, shown using a UML object diagram.

they are also registered in the `store`, since they might also have their state changed during test execution. Subsequently the tests are derived by invoking the `getTests` method on the root of the `ast`, which returns a `TestSequence` that contains the generated tests. Next, each test is executed and the `store` is reset to restore the system state. This process continues until there are no more tests to be executed.

The `Call` statement is an abstract class defined by the runtime library. When a trace is code generated, the code produced implements and instantiates the `Call` statements as anonymous classes. As shown in fig. 3, the `Call` statement defines three methods: The `isTypeCorrect` method is used to determine whether the input to the `Call` statement matches the types of the formal parameters of the function or operation that the `Call` statement originates from. If this method returns **`false`** the current test is considered `INCONCLUSIVE`. Since the implementation of the `isTypeCorrect` method depends on the signature of the function or operation that the `Call` statement originates from, this method is implemented by the Java code generator when the trace is code generated. The `isTypeCorrect` method returns **`true`** by default, which corresponds to the situation where the input to the `Call` statement can be guaranteed, using static analysis, to be type correct. In this particular case the code generator does not have to implement the `isTypeCorrect` method. To illustrate, the construction and implementation of the `Call` statement object used to represent `op2` is shown in listing 1.4. Note that the code generated version of the argument `x` is accessed from a scope enclosing the `Call` methods.

For a `Call` statement to be type correct the input arguments $a_1, \ldots, a_n$ must match the types of the formal parameters, $T_1, \ldots, T_n$, of the function or operation that the `Call` statement originates from. Using the function `Is(v,T)`, described in section 2, we require that $\texttt{Is}(a_1, T_1)$ `&&` $\ldots$ `&&` $\texttt{Is}(a_n, T_n)$ holds in order for

the `Call` statement to be considered type correct. This check is code generated to a JML assertion, which can be configured to produce an `AssertionError` that signals that the `Call` statement is not type correct for the given arguments. As indicated by the generated JML check in listing 1.4, it is assumed that the formal parameter of `op2` is of type **`nat`**.

If a test is considered type correct the runtime library will check that the pre condition of the `Call` statement is met, which is checked using the `meetsPreCond` method. The `meetsPreCond` method returns **`true`** by default, which corresponds to the situation where no pre condition is defined. If either the `isTypeCorrect` or `meetsPreCond` method yield false the test is `INCONCLUSIVE`. Otherwise the runtime library proceeds by calling the `execute` method, which evaluates the `Call`

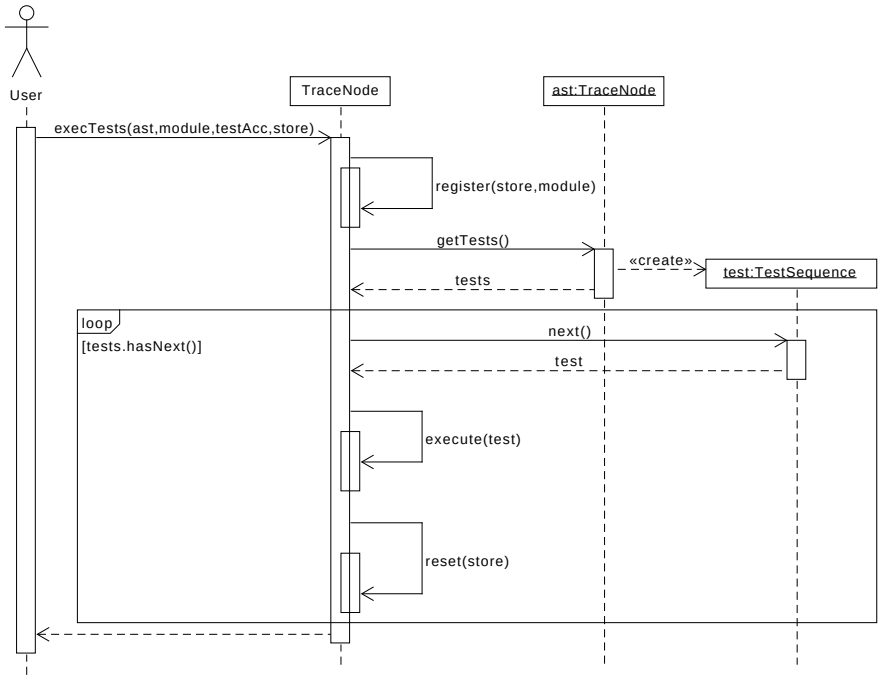

Fig. 2. Execution of a code generated trace, shown using a UML sequence diagram.

Call
`isTypeCorrect() : Boolean` `meetsPreCond() : Boolean` *`execute() : Object`*

Fig. 3. The interface of the `Call` statement node.

```
Call callStm_3 = new Call() {
   public Boolean isTypeCorrect() {
     try {
       //@ assert Utils.is_nat(x);
     } catch (AssertionError e) {
       return false;
     }
     return true;
   }
   public Boolean meetsPreCond() {
     return pre_op2(x);
   }
   public Object execute() {
     return op2(x);
   }
   public String toString() {
     return "op2(" + Utils.toString(x) + ")";
   }
 };
```

Listing 1.4. Implementation of a `Call` statement.

statement and returns the result to the runtime library. This method is abstract and implemented by the code generator when the trace is code generated (see listing 1.4).

4 Case study

4.1 FAD codes

A FAD code is a six digit number, used to identify branches of a retailer. The customer asked us to consider a scenario where FAD codes were obfuscated such that codes were still six digits, still unique per branch, and the entire 0-999999 value range was still available. This is equivalent to creating a permutation on the list of all possible FAD codes. But the obfuscation of a FAD code also needed to be a lightweight calculation, rather than a lookup in a large table, for example.

The design team thought that a permutation could be defined using an injective map of the 0-9 digits onto themselves, but with no digit mapping to itself. If that map was then used to transform the individual digits of a FAD code, then it was believed that the overall set of FAD codes and their transformations would itself form an injective map, defining a permutation. This is intuitively true, but it was not considered “obvious”. To investigate whether it did meet the requirements, a VDM model was created that defined FAD codes and the injective digit map. It was then stated that if the map was applied to every possible FAD code, then the set of obfuscated FAD codes would meet the requirements. Relevant excerpts from the VDM model are shown in listing 1.5.

```
values
  SIZE = 6; -- FAD code size
  MAX = 10 ** SIZE - 1; -- The highest FAD code
  DM1 : DigitMap = -- Arbitrary digit mapping
   { 1 |-> 9, 2 |-> 8, 3 |-> 7, 4 |-> 6, 5 |-> 0,
    6 |-> 4, 7 |-> 3, 8 |-> 2, 9 |-> 1, 0 |-> 5 };
types
  DigitMap = inmap nat to nat
  inv m ==
    let digits = {0, ..., 9} in
      dom m = digits and rng m = digits
      and forall c in set dom m & m(c) <> c;

  FAD = nat
  inv f == f <= MAX
functions
  convert: FAD * DigitMap -> FAD
  convert(fad, dm) ==
    let digits = digitsOf(fad) in
      valOf([ dm(digits(i)) | i in set inds digits ])
  post RESULT <> fad;
traces
  AllDifferent:
    let fad in set {0, ..., MAX} in
      convert(fad, DM1);
```

Listing 1.5. Excerpts from the FAD code VDM model

The trace in listing 1.5 has no combination of cases, but still generates one million test cases when `SIZE` is set to six. The validity of each test is checked by the fact that the digit map `DM1` must meet its constraints, and when applied to every FAD code that map must produce an obfuscated result which meets the `convert` post conditions – that it is a different value.

One could write a trace which applies every possible injective map to the entire set of FAD codes to test that every case produces a permutation. However, with one million FAD codes per map, this would be intractable. This illustrates the point that although trace expansion is useful, combinatorial explosions can easily limit the size of the traces that can be executed.

4.2 Performance results

To analyse the performance gained by using code generated traces, the trace in listing 1.5 was executed using different VDM tools, for FAD codes consisting of up to six digits. These VDM tools exhibit different performance characteristics in terms of execution time and memory consumption due to the way they expand the trace. The memory consumption is an indication of how well a tool scales for large test collections. For example, when the memory consumption of a tool approaches the maximum

amount of memory available, the execution time will increase significantly. In the worst case the tools run out of memory and crash.

Each tool was run twice for each FAD code size. The first run was used to measure execution time. The second run was used to confirm that the tool did not suffer from memory starvation, which would yield a misleading execution time. Checking the memory consumption was performed using a separate run to avoid affecting the execution time. The execution times are those reported by the tools. The memory consumption was analysed using the `/usr/bin/time` tool available on Ubuntu Gnome 15.10 (wily). This tool will report the maximum resident set size for a process, i.e. the maximum amount of memory allocated by the process that is stored in RAM during execution. This is not an accurate measure of the actual memory consumption but it gives an idea of whether the VDM tools are suffering from memory starvation.

All the performance measurements were performed on a Fujitsu LIFEBOOK U772 laptop with a 1.7GHz Intel Core i5 processor and 8Gb of memory running a Linux OS (Ubuntu Gnome). The VDM tools were executed on a 64-bit Java 7 virtual machine with a maximum heap size of 5Gb.

The execution times for FAD codes of sizes one through six are shown in table 1 and visualised using a logarithmic data plot in fig. 4. For the scenario that did not complete, due to the VDM tool running out of memory during trace expansion, the result is specified as "failed". For FAD codes of size six, the maximum resident set sizes for VDMJ was measured to 2.17 Gb, for code generated traces it was 2.31 Gb, whereas no measurement was made for Overture because this tool crashed. Based on the maximum resident set sizes it was confirmed that the tools did not suffer from memory starvation during the trace expansion and execution.

Table 1. Execution times for different VDM tools and FAD code sizes.

Size	VDMJ-3.1.1 [ms]	Overture-2.3.2 extension [ms]	Code Generated [ms]
1	46	124	211
2	465	621	633
3	2,139	3,288	3,217
4	8,692	9,068	29,032
5	35,610	57,999	279,401
6	379,635	failed	2,953,318

5 Discussion

This section discusses the performance results presented in section 4.2. As illustrated using the plot in fig. 4, execution times increase exponentially as more digits are added to a FAD code. This is expected since adding more digits cause an exponential increase in the number of tests generated.

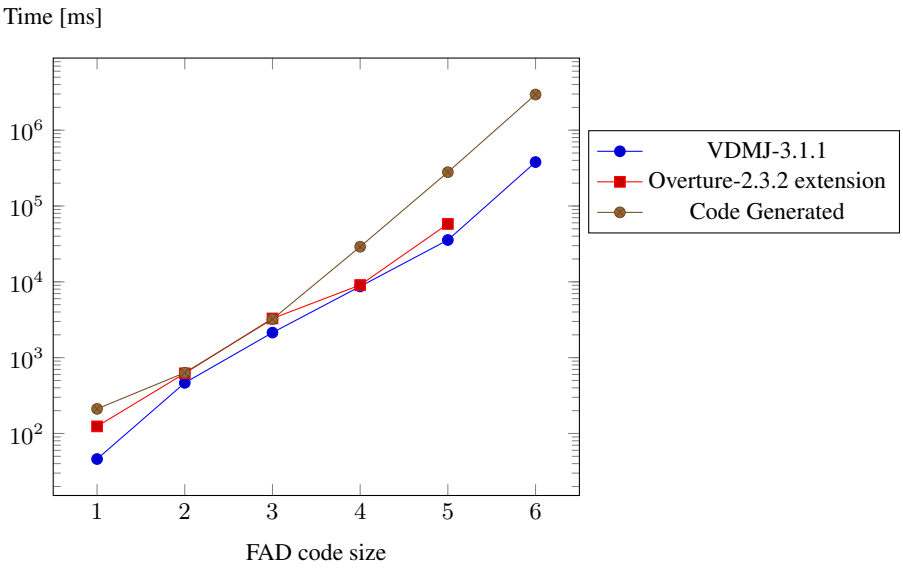

Fig. 4. The execution times in table 1 visualised using a logarithmic data plot.

Overture did not manage to run the one million tests generated for six digit FAD codes because the tool ran out of memory. The code generated trace, on the other hand, completed these tests in over $2,900$ s, or 49.22 minutes. What is surprising about the results obtained using these two tools is that Overture expands and executes the tests significantly faster than the code generated trace, for FAD code sizes smaller than six. Therefore, the only performance gain of the code generated traces is the reduction in memory needed to expand and execute the trace. This is surprising since traces that are run as compiled code are, by intuition, expected to run faster. Comparing the execution time of a code generated trace to that obtained using Overture provides a good indication of the performance that can be gained, since these tools use the same algorithm to expand the trace.

VDMJ completes all one million tests in over 379 s, or 6.33 minutes and is the fastest tool to execute the tests. Compared to Overture, VDMJ manages to complete the tests because it uses a more memory efficient algorithm to expand the trace. Older releases of VDMJ use an algorithm similar to that of Overture. However, very recently VDMJ was released with a new expansion algorithm that addresses some of the performance issues with the old expansion algorithm.

To study the performance overhead posed by OpenJML, we first tried to compile and execute the code generated trace as a normal Java program – without using OpenJML. This is similar to running the code generated trace without checking the generated JML annotations. When this is done, even with the poor expansion algorithm, the one million tests are expanded and executed in 33.94 seconds. If the expansion algorithm is changed to that used by VDMJ, this will most likely be significantly lower. The reason

for this is that VDMJ seems to scale better than Overture for larger test sets, as indicated by the execution times in table 1.

To further analyse the overhead of using OpenJML we tried to remove the JML annotations from the generated code and compile and execute the tests using the OpenJML runtime assertion checker. The point of this is to eliminate the overhead directly related to checking the JML constraints and focus solely on the overhead posed by OpenJML. Although this reduces the number of extra checks that are performed, OpenJML still guards against variables and fields that hold the value `null`, as this is not allowed by default. When the JML annotations are removed, OpenJML expands and executes the one million tests in 11.15 minutes. Ideally, the execution time should be close to that obtained by running the code generated trace without the OpenJML runtime assertion checker (33.94 seconds). This is an indication that OpenJML has a significant influence on the rather disappointing performance results obtained using code generated traces.

To address the performance issues we believe that two things must be done. First, the expansion algorithm must be updated to that used by VDMJ, which is available as open-source. Secondly, we plan to look into other DbC technologies that can be used to support our work (section 6).

6 Conclusion and future plans

In this paper we have shown how VDM traces can be code generated and used to test the system realisation, or some part of it. Code generated traces have potential to allow a larger number of tests to be executed since they are run as compiled code rather than using a VDM interpreter. Our work is implemented as an extension of Overture's Java code generator, which translates a VDM-SL model to a Java program annotated with JML derived from the VDM constraints. When the code generated trace is expanded and executed, using a JML tool, the code generated version of the VDM specification is validated against the JML annotations.

In the FAD code case study, code generated traces allow more memory efficient expansion and execution of traces, compared to using the Overture interpreter. However, we still regard our current performance results as disappointing. Especially because the FAD code trace executes much faster with VDMJ compared to the code generated version of the trace. In order for code generated traces to execute faster than traces interpreted using VDMJ, we believe that two things must be addressed. First, the algorithm used for the expansion must be improved, for example, by using that of VDMJ. Secondly, there are also indications that the performance issues are directly related to the use of OpenJML.

Looking forward, we plan to investigate other technologies that can support our work and help us achieve better performance results. One way to ensure that the contracts and type constraints, as specified in VDM, hold across the translation, is by adding extra Java checks to the generated code – without using a particular DbC technology. Although this allows us to control exactly when these checks are triggered, the separation between specification and code becomes less clean. One DbC technology that is worth investigating, as an alternative to OpenJML, is the .NET-based technology, Microsoft Code Contracts [27, chapter 15]. Compared to JML, which uses a dedicated

syntax for program specification, Code Contracts provides its features via libraries to support all languages within the .NET framework. JML and Code Contracts share many of the same concepts although their semantics sometimes differ. Changing our work to use another DbC technology therefore requires new rules for representing VDM constraints in the generated code. However, due to the similarities between Java and C#, we expect the approach used to code generate traces to be readily reusable. In a C++ context another DbC technology that is worth investigating is the Contract++ library [4], which has been accepted into Boost [26]. On a longer term, contracts may also be a native feature of C++17 [8].

Acknowledgements We would like to thank the anonymous referees for valuable input on this work.

References

1. Battle, N.: VDMJ website. `https://github.com/nickbattle/vdmj` (2016)
2. Brummayer, R., Lonsing, F., Biere, A.: Automated Testing and Debugging of SAT and QBF Solvers. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing – SAT 2010. pp. 44–57. Springer, Edinburgh, UK (Jul 2010)
3. Cok, D.: OpenJML: JML for Java 7 by Extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G., Joshi, R. (eds.) NASA Formal Methods, Lecture Notes in Computer Science, vol. 6617, pp. 472–479. Springer Berlin Heidelberg (2011), `http://dx.doi.org/10.1007/978-3-642-20398-5_35`
4. Contract++ website. `https://sourceforge.net/projects/contractpp/` (2016)
5. Couto, L.D., Larsen, P.G., Hasanagić, M., Kanakis, G., Lausdahl, K., Tran-Jørgensen, P.W.V.: Towards Enabling Overture as a Platform for Formal Notation IDEs. In: Proceedings of the 2nd Workshop on Formal-IDE (F-IDE) (Jun 2015)
6. Couto, L.D., Tran-Jørgensen, P.W.V., Coleman, J.W., Lausdahl, K.: Migrating to an Extensible Architecture for Abstract Syntax Trees. In: Proceedings of the 12th Working IEEE / IFIP Conference on Software Architecture (May 2015)
7. Couto, L.D., Tran-Jørgensen, P.W.V., Lausdahl, K.: Principles for Reuse in Formal Language Tools. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC) (Apr 2016)
8. Thoughts on C++17. `http://www.infoq.com/news/2015/04/stroustrup-cpp17-interview` (2016)
9. Dick, J., Faivre, A.: Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In: Woodcock, J.C.P., Larsen, P.G. (eds.) FME'93: Industrial-Strength Formal Methods. pp. 268–284. Formal Methods Europe, Springer-Verlag (Apr 1993)
10. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edn. (2009), ISBN 0-521-62348-0
11. Gamma, E., Helm, R., Johnson, R., Vlissides, R.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company (1995)
12. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5) (May 1997)

13. ISO: Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language (Dec 1996)
14. Jones, C.B.: Scientific Decisions which Characterize VDM. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) FM'99 - Formal Methods. pp. 28–47. Springer-Verlag (1999)
15. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A Code Generation Platform for VDM. In: Battle, N., Fitzgerald, J. (eds.) Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, UK, Technical Report CS-TR-1446 (Jan 2015)
16. The JUnit website. http://www.junit.org (2016)
17. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes 35(1), 1–6 (Jan 2010), http://doi.acm.org/10.1145/1668862.1668864
18. Larsen, P.G., Lausdahl, K., Battle, N.: Combinatorial Testing for VDM. In: Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods. pp. 278–285. SEFM '10, IEEE Computer Society, Washington, DC, USA (Sep 2010), http://dx.doi.org/10.1109/SEFM.2010.32, ISBN 978-0-7695-4153-2
19. Larsen, P.G., Lausdahl, K., Battle, N., Fitzgerald, J., Wolff, S., Sahara, S., Verhoef, M., Tran-Jørgensen, P.W.V., Oda, T.: VDM-10 Language Manual. Tech. Rep. TR-001, The Overture Initiative, www.overturetool.org (Apr 2013)
20. Lausdahl, K., Larsen, P.G., Battle, N.: A Deterministic Interpreter Simulating A Distributed real time system using VDM. In: Qin, S., Qiu, Z. (eds.) Proceedings of the 13th international conference on Formal methods and software engineering. Lecture Notes in Computer Science, vol. 6991, pp. 179–194. Springer-Verlag, Berlin, Heidelberg (Oct 2011), http://dl.acm.org/citation.cfm?id=2075089.2075107, ISBN 978-3-642-24558-9
21. Leavens, G.T., Cheon, Y.: Design by Contract with JML (2005), ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf, draft, available from jmlspecs.org.
22. Ledru, Y., du Bousquet, L., Maury, O., Bontron, P.: Filtering TOBIAS Combinatorial Test Suites. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. pp. 281–294. LNCS 2984, Springer-Verlag Berlin Heidelberg (2004)
23. Ledru, Y., Dadeau, F., du Bousquet, L., Ville, S., Rose, E.: Mastering Combinatorial Explosion with the Tobias-2 Test Generator. In: ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. pp. 535–536. ACM, New York, NY, USA (2007)
24. Nie, C., Leung, H.: A Survey of Combinatorial Testing. ACM Comput. Surv. 43(2) (Feb 2011), http://doi.acm.org/10.1145/1883612.1883618
25. The Overture tool website. http://www.overturetool.org/ (2016)
26. Schling, B.: The Boost C++ Libraries. XML Press (2011)
27. Skeet, J.: C# in Depth, Second Edition. Manning Publications (2010)
28. Tran-Jørgensen, P.W.V., Larsen, P.G., Leavens, G.T.: Automated translation of VDM to JML annotated Java (Jan 2016, submitted to the International Journal on Software Tools for Technology Transfer (STTT))
29. Yu, L., Lei, Y., Kacker, R.N., Kuhn, D.R.: ACTS: A combinatorial test generation tool. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. pp. 370–375 (Mar 2013)