Implementing Mutation Testing for VDM-SL in ViennaTalk

Tomohiro Oda[1]

Software Research Associates, Inc. (tomohiro@sra.co.jp)

Abstract. Executable subsets of formal specification languages allow software testing techniques to be applied earlier, during the specification phase. In the VDM family of languages, both unit testing and combinatorial testing have been actively studied, implemented as tools, and applied in practice. The effectiveness of these techniques depends on the quality of the test suite. Mutation testing is a widely used approach to assess this quality by measuring the ability of test cases to detect faults. This report presents a mutation testing framework implemented in ViennaTalk and demonstrates its application to both unit test cases and combinatorial test traces.

1 Introduction

Detecting defects in specifications is highly desirable, as it is well known that such defects can significantly impact the overall development cost and the quality of the final product. Formal specification techniques have been studied and applied to uncover these defects using mathematically grounded theories and tools. Testing techniques, alongside theorem proving, are effective means of detecting faults in formal specifications. The rich executable subsets of VDM-family specification languages make it possible to apply software testing techniques even before the implementation phase begins. Unit testing and combinatorial testing techniques have been implemented in development tools for the VDM family, including VDMTools, the Overture Tool, and ViennaTalk [2][5][6].

Evaluating the quality of tests remains an active area of research in the field of software testing. A single execution of a test case can only confirm the presence or absence of a failure along one particular execution path. Moreover, test execution incurs cost both in writing the test code and in running it. Therefore, managing the quality of a test suite is essential for the effective use of testing techniques.

Mutation testing is a widely studied approach for evaluating the quality of software tests [1]. Its core idea is to create faulty versions of a program called mutants by introducing small syntactic changes to the original program, and then measuring how many of these mutants are detected (or *killed*) by the test suite.

This report describes the mutation testing functionality implemented in ViennaTalk [4]. Section 2 provides a brief overview of mutation testing, and Section 3 explains its implementation in ViennaTalk. Section 4 presents case studies demonstrating the application of mutation testing to both unit tests and combinatorial test traces for the Sieve of Eratosthenes. Finally, Section 5 offers discussion, concluding remarks, and directions for future work.

The mutation testing functionality described in this report is implemented in the development version of ViennaTalk. Its source code is available at `https://github.com/tomooda/ViennaTalk/tree/dev`. Binary packages for macOS (Apple Silicon and Intel 64-bit) and 64-bit Windows can be downloaded from `https://viennatalk.org/builds/viennatalk/dev/`.

2 Overview of mulation testing in ViennaTalk

Passing a software test does not guarantee that the target system is free of defects. Mutation testing addresses this limitation by evaluating the quality of test code through the use of numerous error-prone variants of the original specification, known as mutants. These variants are generated by applying small syntactic modifications, called mutation operators, to the original specification.

In mutation testing, a large number of mutated specifications are created by systematically applying mutation operators to the original test target. The test suite, typically composed of unit test cases and named traces, is then executed against each mutant. If the test suite fails when run on a mutant, the defect introduced by the mutation has been successfully detected, and the mutant is considered *killed*. If the test suite passes, it has failed to detect the defect, and the mutant is considered *surviving*. The mutation score is defined as the proportion of killed mutants relative to the total number of mutants. A higher mutation score indicates a more effective test suite.

Mutation operators are the core of mutation testing. ViennaTalk currently implements two such operators, which are described in the remainder of this section.

2.1 Add-One Mutation

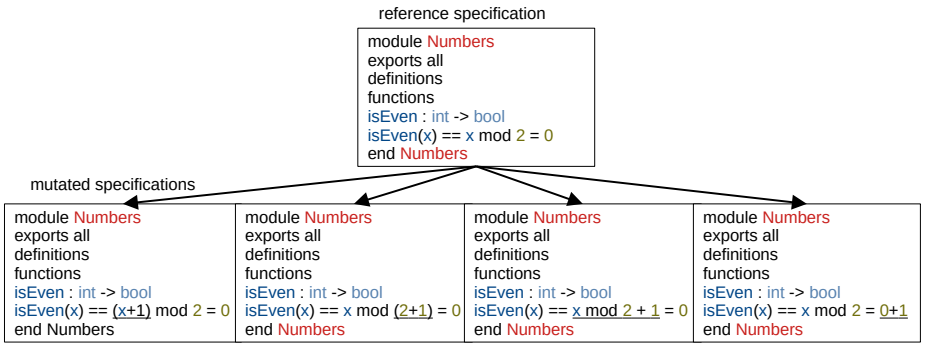

Fig. 1. An example application of the add-one mutation

The add-one mutation modifies numeric expressions in the specification. When this operator is applied, an expression of type `real` is selected and replaced with `exp + 1`, where `exp` is the original expression.

It is worth noting that in VDM-SL, the types `int`, `nat`, and `nat1` are subtypes of `real`. Mutation operators for other data types are planned for future development. The add-one mutation was chosen first because the `nat1` type is widely used in VDM-SL, particularly in contexts like sequence indexing.

An important feature of the add-one mutation is that it typically preserves static typing, except in cases where a type invariant is violated. Even when an invariant is broken, such mutations remain useful, as they allow the test code's ability to catch run-time invariant violations to be evaluated.

Mutants generated by the add-one operator are valid in both the interpreter and transpiler (code generator), though the proof obligation generator (POG) may still produce unprovable conditions. Ensuring that mutated specifications remain executable aligns with the central goal of mutation testing: to evaluate the test suite, not the specification itself.

2.2 Negate Mutation

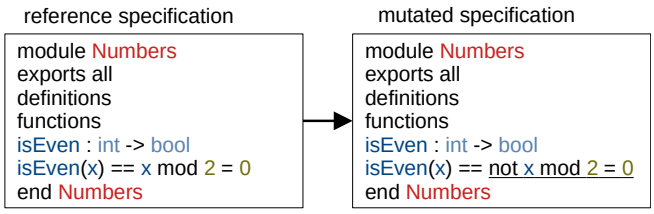

Fig. 2. An example application of the negate mutation

The second mutation operator implemented in ViennaTalk is the negate mutation. This operator targets Boolean expressions, replacing each expression `exp` of type `bool` with `not exp`.

While the add-one mutation introduces numeric discrepancies, the negate mutation is intended to alter control flow by affecting decisions such as conditional branches and guard expressions. Boolean expressions are prevalent in VDM specifications, and applying negation does not violate type rules. Thus, this operator provides a valuable means of testing the sensitivity of the specification to logical conditions.

3 Implementation in ViennaTalk

This section describes the implementation of mutation testing in ViennaTalk. Mutation testing requires two core components: a set of mutation operators, which generate a large number of mutated specifications, and a test runner, which executes tests on each mutated specification and collects the results. Each component is described in the subsections below.

Table 1. Abstract methods that subclasses of `ViennaMutation` should implement

method name	argument	return value	description
canMutate:	node	bool	answers whether the operator can be applied to the given AST node.
mutate:	node	node	mutate at the given AST node.
name	-	string	answers the name of the operator.

3.1 Mutation

Two mutation operators have been implemented in ViennaTalk as subclasses of the abstract class `ViennaMutation`. The primary responsibility of a mutation object is to generate mutated abstract syntax trees (ASTs) from a given reference specification AST. In general, a reference AST may contain multiple mutation points – locations where AST nodes can be replaced with alternative versions. As a result, applying a mutation operator may produce multiple mutated specifications.

The `ViennaMutation` class provides a method that takes a reference specification AST as input and enumerates all possible mutated ASTs. It also defines three abstract methods that must be implemented by each subclass, as summarized in Table 1.

3.2 Test Runner

Figure 3 shows a screenshot of ViennaTalk's specification browser [5]. The user interface for testing is located in the lower-left section of the window. The `Run` button executes all unit test cases and named traces, while the `Auto Run` button toggles automatic re-execution of tests upon each modification to the specification. The `Mutation Testing` button initiates mutation testing for the module selected in the top-left list of modules and opens the results window shown in Figure 4.

In Figure 4, the left pane displays a list of mutated specifications. Killed mutants are marked with green circles, and surviving mutants with red circles. Each entry also shows the mutation operator used and the name of the mutated definition. When a mutant is selected, the right pane displays the annotated source code. In the example shown, the argument `n` of the `createSieve` operation has been replaced with `n + 1` by the add-one mutation. Above the list, the mutation score is displayed, along with the number of killed mutants, the total number of mutants tested, and the total number of generated mutants. Mutation testing is executed in the background, allowing users to continue editing or navigating the specification while testing is in progress.

4 Examples

The Sieve of Eratosthenes is an algorithm for identifying prime numbers by iteratively eliminating multiples of known primes. A VDM-SL specification of this algorithm, including unit tests and combinatorial traces, is available at `https://github.com/tomooda/Eratosthenes/blob/main/Eratosthenes.vdmsl`, and is also attached to this report as an appendix for convenience.

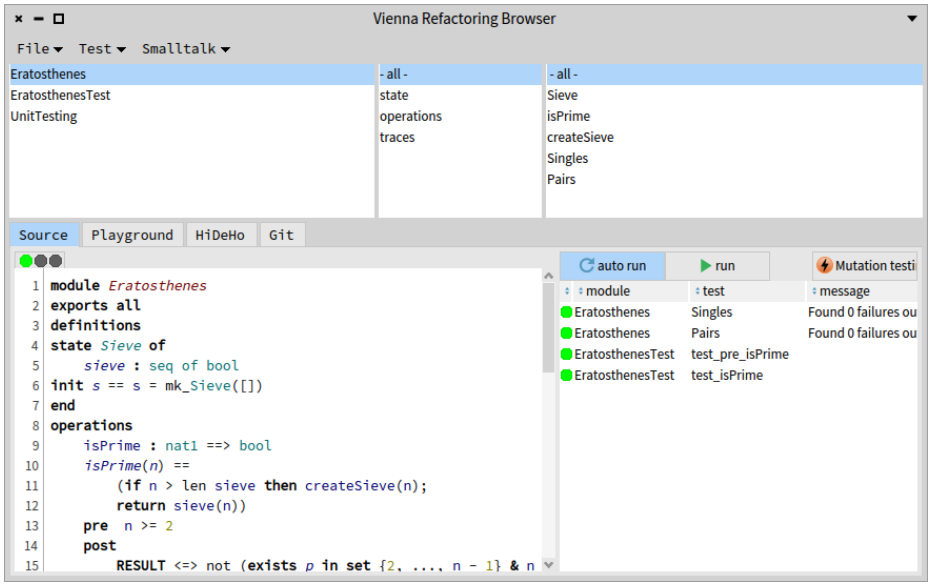

Fig. 3. A screenshot of testing UI

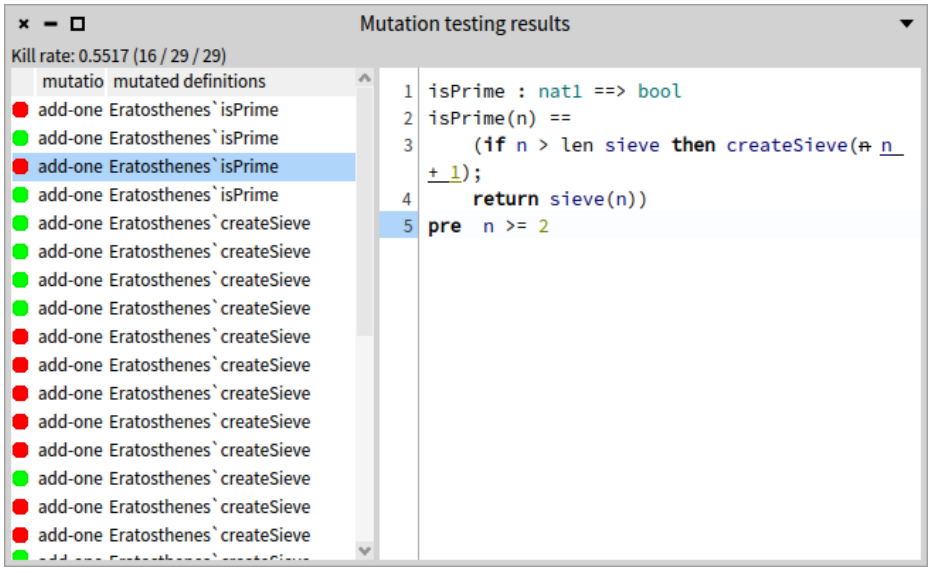

Fig. 4. A screendump of mutation testing results

The `Eratosthenes` module defines two operations, `isPrime` and `createSieve`, which access a shared state variable `sieve`. The `isPrime` operation takes a natural number as input and returns whether it is a prime number. It relies on the state variable `sieve`, a sequence of booleans where the `n`-th element indicates whether `n` is prime. If the sieve is not long enough to answer a query, the operation `createSieve` is invoked to extend it accordingly. The precondition of `isPrime` requires the input argument to be greater than or equal to 2. The postcondition specifies that the result must be logically equivalent to: `not (exists p in set {2, ..., n - 1} & n mod p = 0)`

The module also includes two named traces for combinatorial testing. The `Singles` trace verifies that `isPrime(n)` works correctly for all `n` in `2,...,50`. The `Pairs` trace verifies the correct management of the `sieve` length by checking all ordered pairs `(n, m)` where `n, m ∈ 2,...,50`—i.e., it executes `isPrime(n)` followed by `isPrime(m)` to simulate reuse of an extended sieve.

Additionally, the `EratosthenesTest` module defines two unit test cases. The `test_pre_isPrime` test case verifies that the operation correctly rejects `1` as input and accepts `2` and `4`, regardless of whether the current `sieve` is shorter or longer than the input. The `test_isPrime` test case validates a scenario in which the input argument increases, remains the same, and then decreases relative to previous inputs.

Mutation testing was conducted using both the add-one and negate mutation operators under various conditions related to preconditions and postconditions. One experimental condition determined whether the mutation operators were allowed to mutate expressions inside pre and pos tconditions. In general, mutation testing in programming languages does not alter the test code itself. Since postconditions in VDM-SL often correspond to assertions in program tests, we performed unit and combinatorial testing without mutating postconditions to simulate this conventional setting.

However, pre and post conditions are central to formal model-based specifications. Defects in these conditions are considered defects in the specification itself. Therefore, to properly assess the ability of test cases and traces to detect such faults, additional experiments were conducted including mutations in pre and pos tconditions. Furthermore, as a baseline, we conducted mutation testing with postconditions entirely removed from the specification, noting that mutated specifications may fail during animation due to runtime errors, not just assertion violations.

Table 2. The summarized result of mutation testing to unit testing and combinatorial testing

test condition	killed	surviving	mutation score
unit testing without mutating pre/post conditions	20	9	0.6897
unit testing including mutated pre/post conditions	55	14	0.7971
unit testing without post conditions	16	13	0.5517
combinatorial testing without mutating pre/post conditions	25	4	0.8621
combinatorial testing including mutated pre/post conditions	60	9	0.8696
combinatorial testing without post conditions	7	22	0.2414

The results are summarized in Table 2. With postconditions, combinatorial testing outperformed unit testing, demonstrating its superior ability to detect specification-level faults. Without postconditions, unit testing yielded better scores than combinatorial testing, as it retained instance-based internal assertions within the test operations. In contrast, combinatorial tests relied solely on assertions in the target specification, lacking additional instance-based assertion checks, which led to a lower mutation score. Please note that the lower mutation score of the combinatorial testing without postconditions does NOT mean a limitation of combinatorial testing but of specifications with loose assertions.

5 Discussions and Concluding Remarks

Two mutation operators, the add-one operator and the negate operator, have been implemented in ViennaTalk. Extending the set of mutation operators to handle other types of expressions and statements, beyond just `real` and `bool`, remains an important direction for future work.

The current implementation allows expressions within pre- and postconditions to be mutated. However, further applications of mutation testing to formal specifications are needed to clarify how assertions (i.e., preconditions and postconditions) should be treated: whether they should be protected as part of the test suite, or mutated to evaluate the robustness of the specification itself.

In general, mutation testing is costly because it requires executing the entire test suite for each mutant. This cost becomes more significant in the context of combinatorial testing, which generates many input combinations. In the case study of the Sieve of Eratosthenes, 2,450 test cases were run for each mutant in the combinatorial trace, alongside two unit test cases. With 69 mutants generated (including those mutated in postconditions), this leads to a worst-case estimate of 169,188 test executions. It is important to note, however, that many mutants were killed early in the test process leaving many test cases unexecuted. The number of mutants and total test executions will further increase with the introduction of additional mutation operators and more comprehensive test suites.

Despite this computational cost, performance remained practical in the evaluation. On an Apple M2 Pro processor, ViennaTalk completed the full mutation testing process, including postcondition mutations, in approximately 44 seconds. This efficiency is due to ViennaTalk's transpilation approach, which translates VDM specifications into native Smalltalk classes [3]. Each mutant is executed as compiled Smalltalk code, benefiting from the just-in-time (JIT) compilation capabilities of the underlying Smalltalk virtual machine. This makes it possible to handle a large number of repeated executions per mutant efficiently. Mutation testing is performed in the background, allowing users to continue working without interruption.

A key distinction between VDM specifications and traditional programming languages lies in where assertions are written: VDM embeds them within the specification, while programming languages often place them in test cases. This raises questions about how best to interpret mutation results in the context of formal methods. For example, Table 2 raises the question: Can mutation testing on combinatorial testing be used

as an indicator of loose assertions? Further research is needed to investigate how mutation testing can guide the improvement of test cases, traces, and embedded assertions in formal specifications.

Acknowledgements

The author thanks anonymous reviewers for their valuable comments.

References

1. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), 649–678 (2011)
2. 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 (2010), http://dx.doi.org/10.1109/SEFM.2010.32, ISBN 978-0-7695-4153-2
3. Oda, T., Araki, K., Larsen, P.G.: Automated VDM-SL to Smalltalk Code Generators for Exploratory Modeling. In: Larsen, P.G., Plat, N., Battle, N. (eds.) The 14th Overture Workshop: Towards Analytical Tool Chains. pp. 48–62. Aarhus University, Department of Engineering, Aarhus University, Department of Engineering, Cyprus (2016)
4. Oda, T., Araki, K., Larsen, P.G.: A formal modeling tool for exploratory modeling in software development. IEICE Transactions on Information and Systems 100(6), 1210–1217 (2017)
5. Oda, T., Araki, K., Sahara, S., Chang, H.M., Gorm, P.: Refactoring for exploratory specification in vdm-sl. Proceedings of the 19th Overture Workshop pp. 21–35 (2021)
6. Tran-Jørgensen, P.W.V., Nilsson, R., Lausdahl, K.: Enhancing Testing of VDM-SL Models. In: Pierce, K., Verhoef, M. (eds.) The 16th Overture Workshop. pp. 7–22. Newcastle University, School of Computing, Oxford (2018)

Appendix: The source specification of the Sieve of Eratosthenes

```
1  module Eratosthenes
2  exports all
3  definitions
4  state Sieve of
5    sieve : seq of bool
6  init s == s = mk_Sieve([])
7  end
8  operations
9    isPrime : nat1 ==> bool
10   isPrime(n) ==
11     (if n > len sieve then createSieve(n);
12     return sieve(n))
13   pre  n >= 2
14   post
15     RESULT <=>
16     not (exists p in set {2, ..., n - 1} & n mod p = 0);
17
18   createSieve : nat1 ==> ()
19   createSieve(size) ==
20     (dcl
21       newSieve:seq of bool := [true | i in set {1, ..., size}],
22       n:nat1 := 2;
23     newSieve(1) := false;
24     while n  * n <= size
25     do
26       (if newSieve(n) then
27         for m = n  * 2 to  size by  n do
28           newSieve(m) := false;
29       n := n + 1));
30     sieve := newSieve)
31   pre  size >= 2
32   post
33     len sieve = size
34     and (forall n in set {2, ..., size} &
35         sieve(n) <=>
36         not (exists p in set {2, ..., n - 1} & n mod p = 0));
37 traces
38   Singles:
39     let n in set {2, ..., 50} in isPrime(n);
40
41   Pairs:
42     let n1, n2 in set {2, ..., 50}
43     in
44       (isPrime(n1);
45       isPrime(n2));
46 end Eratosthenes
47
```

```
module EratosthenesTest
imports
  from Eratosthenes all,
  from UnitTesting
    operations assert: bool * seq of char ==> () renamed assert;
exports all
definitions
values
  no_sieve : Eratosthenes`Sieve = mk_Eratosthenes`Sieve([]);
  sieve : Eratosthenes`Sieve =
    mk_Eratosthenes`Sieve([false,true,true,false,true,false]);
operations
  test_pre_isPrime : () ==> ()
  test_pre_isPrime() ==
    (assert(not Eratosthenes`pre_isPrime(1, no_sieve),
       "should not give 1");
    assert(not Eratosthenes`pre_isPrime(1, sieve),
      "should not give 1");
    assert(Eratosthenes`pre_isPrime(2, no_sieve),
      "2 without sieve is OK");
    assert(Eratosthenes`pre_isPrime(2, sieve),
      "2 with sieve is OK");
    assert(Eratosthenes`pre_isPrime(4, no_sieve),
      "4 without sieve is OK");
    assert(Eratosthenes`pre_isPrime(4, sieve),
      "4 with sieve is OK");
    skip);

  test_isPrime : () ==> ()
  test_isPrime() ==
    (assert(Eratosthenes`isPrime(2), "2 is prime");
    assert(Eratosthenes`isPrime(3), "3 is prime");
    assert(Eratosthenes`isPrime(5), "5 is prime");
    assert(Eratosthenes`isPrime(5), "5 is prime");
    assert(not Eratosthenes`isPrime(6), "6 is not prime");
    assert(Eratosthenes`isPrime(5), "5 is prime"));
end EratosthenesTest

module UnitTesting
exports all
definitions
types
  AssertFailure :: msg : seq of char;
  AssertEqualsFailure :: actual:? expected:? msg:seq of char;
operations
  assert : bool * seq of char ==> ()
  assert(b, msg) == if not b then exit mk_AssertFailure(msg);
end UnitTesting
```