Overture Technical Report Series
No. TR-005

December 2020

Guidelines for using VDM Combinatorial Testing Features

by

Nick Battle
Peter Gorm Larsen

Document history

Month	Year	Version	Version of Overture.exe	Comment
December	2010		3.0.2	Initial version

Contents

Chapter 1

Introduction

This manual is a complete guide the combinatorial testing of VDM models. It assumes the reader has no prior knowledge of combinatorial testing, but a working knowledge of VDM, in particular, the VDM++ and VDM-SL dialects.

1.1 What is Combinatorial Testing?

Creating a comprehensive set of tests for VDM specifications can be a time consuming process. To try to make the generation of test cases simpler, Overture provides a VDM language extension (for all dialects) called Combinatorial Testing.

In general, specifications are tested to verify that certain properties or behaviours are met, as specified by constraints in the specification and validation conjectures in the tests.

The simplest way to test a specification is to write ad-hoc tests, starting from a known system state and proceeding with a sequence of operation calls that should move to a new state or produce some particular result or error response. The problem with this kind of testing is that it can be very laborious to produce the number of tests needed to cover the complete system behaviour. It is also expensive to maintain a large test suite as the specification evolves.

The most complete way to test a specification is to produce a formal mathematical proof that it will never violate its constraints, and always meet its validation conjectures if presented with a legal sequence of operation calls. This provides the highest level of confidence in the correctness of a specification, but it can be unrealistic to produce a complete formal proof for complex specifications, even with tool support.

Model checking provides an approach to formal testing that is considerably better than ad-hoc testing but not as complete as formal proof. This approach uses a formal specification of the system properties desired, often written in a temporal calculus, and the model checker symbolically executes the specification searching for execution paths that violate the constraints. Since the execution is symbolic, extremely large state spaces can be searched (billions of cases is not uncommon), and failed cases can produce a "counter example" that demonstrates the failure. This is a very powerful technique, but in practice, realistic specifications often produce a state space explosion that is too great for model checkers.

Combinatorial testing is an approach that is far more powerful than ad-hoc testing, but not as complete as model checking. Tests are produced automatically from "traces" that are relatively simple to define. The approach allows specifications to be tested with perhaps millions of test cases, but cannot guarantee to catch every corner case in the way that a model checker can. Therefore the technique is useful for specifications that are too complex for model checking or formal proof.

A combinatorial trace is a pattern that describes the construction of argument values and the sequences of operation calls that will exercise the specification. A specification may contain several traces, each designed to test a particular aspect. Traces are automatically expanded into a (potentially large) number of tests, each of which is a particular sequence of operation calls and argument values. The execution of tests is performed automatically, starting each in a known state; a test is considered to pass if it does not violate the specification's constraints, or the test's validation conjectures. Individual failed tests can be executed in isolation to find out why they failed, which is similar to a model checker's counter example.

Chapter 2

Working with Traces

2.1 Basic Trace Constructs

Combinatorial tests are embedded within a VDM specification using a section called "traces". Typically, one or more traces are added to a separate class or module that is intended for testing rather than the main specification, though you can add traces to any class you wish. In this chapter, we will use the example classes below:

```
class Counter
instance variables
    total:int := 0;

operations
    public inc: () ==> int
    inc() == ( total := total + 1; return total; )

    public dec: () ==> int
    dec() == ( total := total - 1; return total; )

end Counter

class Tester
instance variables
    obj:Counter := new Counter();

traces
    T1: obj.inc();

end Tester
```

Notice that there are two classes, Counter and Tester. The Counter defines a simple class that increments and decrements a total state value that is initially zero. The Tester class creates an

instance of Counter and defines a single trace called T1. This is the simplest trace possible and indicates that the trace should expand to a single test that calls `obj.inc()`.

This trace can either be executed in Overture in the Combinatorial Testing perspective, or it can be executed from the command line using the `runtrace` command. The command line output is illustrated here for simplicity:

```
> runtrace Tester`T1
Generated 1 tests in 0.004 secs.
Test 1 = obj.inc()
Result = [1, PASSED]
Executed in 0.007 secs.
All tests passed
```

The first line of output indicates that one test has been generated from the trace. With more complex examples, this generation could expand to thousands or millions of tests, and consequently it may take a few seconds.

The next line of the output describes the test that was generated. Note that this is called "Test 1", and consists of a single call to `obj.inc()`.

The line below the test gives the result of executing that test. There is a single return value from the call to `obj.inc()`, 1, which is listed along with the word "PASSED" that indicates that there were no constraint violations in the test execution.

Lastly the time taken to execute all of the tests is given, and an indication of whether any tests failed.

The reason that this trace only expands to a single test is that the trace, when considered as a pattern, only matches a single operation call. But if we change the trace to the following:

```
traces
    T1: obj.inc() | obj.dec();
```

The trace is now saying that it would match either a call to `obj.inc()` or a call to `obj.dec()`. Therefore the test expansion produces the following:

```
> runtrace Tester`T1
Generated 2 tests
Test 1 = obj.inc()
Result = [1, PASSED]
Test 2 = obj.dec()
Result = [-1, PASSED]
Executed in 0.033 secs.
All tests passed
```

This time two tests are generated. The first calls `obj.inc()`, the second `obj.dec()`. Notice that the decrement test is completely separate from the increment test. It produces -1 as its

result, because the Counter object is re-created for each test. It does not decrement the counter back to zero after the first test incremented it.

If we want to test an increment followed by a decrement, that would be expressed using semi-colon separators:

```
traces
    T1: obj.inc(); obj.dec();
```

This produces the output:

```
Generated 1 tests
Test 1 = obj.inc(); obj.dec()
Result = [1, 0, PASSED]
Executed in 0.027 secs.
All tests passed
```

This generates a single test again, but you can see that the test involves two calls and that they return 1 and 0. So this time the second call is operating on the same object instance as the first.

If the increment and decrement operations are independent, it makes sense to test calls to them in either order, which would be expressed as:

```
traces
    T1: || ( obj.inc(), obj.dec() );
```

That produces the output:

```
Generated 2 tests
Test 1 = obj.inc(); obj.dec()
Result = [1, 0, PASSED]
Test 2 = obj.dec(); obj.inc()
Result = [-1, 0, PASSED]
Executed in 0.029 secs.
All tests passed
```

So now both orderings of the two calls are produced. This is because there are two orderings that match the pattern `|| ( ..., ...)`. This particular trace construct naturally expands to an arbitrary number of calls and produces a test for every permutation of the calls in brackets.

But what if some tests are a pair of calls and some are not? If we want to make a call optional, the `?` operator can be added to indicate that this will match tests where the call is made and where it is not. For example:

```
traces
    T1: || ( obj.inc(), obj.dec()? );
```

This means that the decrement call is optional and so although it is included in the orderings of the pair, it should also be absent in some cases. This example produces the following:

```
Generated 4 tests
Test 1 = obj.inc(); skip
Result = [1, (), PASSED]
Test 2 = obj.inc(); obj.dec()
Result = [1, 0, PASSED]
Test 3 = skip; obj.inc()
Result = [(), 1, PASSED]
Test 4 = obj.dec(); obj.inc()
Result = [-1, 0, PASSED]
Executed in 0.043 secs.
All tests passed
```

You see that the decrement call is sometimes present and sometimes replaced by `skip`, which indicates the absence of an optional call. Notice also that the `||` operator and the `?` operator work together to combine their effects.

Along the same lines as `?`, it is possible to add `*` and + operators to a call, which indicate that it should be called zero or more times, and one or more times. The maximum number of times is a tool preset value that defaults to 5, though it can be changed. So for example:

```
traces
    T1: obj.inc()*;
    T2: obj.dec()+;

> runtrace Tester`T1
Generated 6 tests
Test 1 = skip
Result = [(), PASSED]
Test 2 = obj.inc()
Result = [1, PASSED]
Test 3 = obj.inc(); obj.inc()
Result = [1, 2, PASSED]
Test 4 = obj.inc(); obj.inc(); obj.inc()
Result = [1, 2, 3, PASSED]
Test 5 = obj.inc(); obj.inc(); obj.inc(); obj.inc()
Result = [1, 2, 3, 4, PASSED]
Test 6 = obj.inc(); obj.inc(); obj.inc(); obj.inc(); obj.inc()
Result = [1, 2, 3, 4, 5, PASSED]
Executed in 0.046 secs.
All tests passed

> runtrace Tester`T2
Generated 5 tests
Test 1 = obj.dec()
Result = [-1, PASSED]
Test 2 = obj.dec(); obj.dec()
```

```
Result = [-1, -2, PASSED]
Test 3 = obj.dec(); obj.dec(); obj.dec()
Result = [-1, -2, -3, PASSED]
Test 4 = obj.dec(); obj.dec(); obj.dec(); obj.dec()
Result = [-1, -2, -3, -4, PASSED]
Test 5 = obj.dec(); obj.dec(); obj.dec(); obj.dec(); obj.dec()
Result = [-1, -2, -3, -4, -5, PASSED]
Executed in 0.042 secs.
All tests passed
```

The important difference between these two is that T1 includes an extra `skip` case, whereas T2 does not.

Lastly, it is possible to indicate a specific number of repetitions of a call or a range of repetitions. For example:

```
traces
    T1: obj.inc(){3};
    T2: obj.dec(){2, 4};

> runtrace Tester`T1
Generated 1 tests
Test 1 = obj.inc(); obj.inc(); obj.inc()
Result = [1, 2, 3, PASSED]
Executed in 0.026 secs.
All tests passed

> runtrace Tester`T2
Generated 3 tests
Test 1 = obj.dec(); obj.dec()
Result = [-1, -2, PASSED]
Test 2 = obj.dec(); obj.dec(); obj.dec()
Result = [-1, -2, -3, PASSED]
Test 3 = obj.dec(); obj.dec(); obj.dec(); obj.dec()
Result = [-1, -2, -3, -4, PASSED]
Executed in 0.01 secs.
All tests passed
```

The T1 trace now produces a single test with precisely three repetitions, while the T2 trace gives three tests with 2, 3 and 4 repetitions respectively.

If you combine a `||` operator with a repetition, the result is to repeat all of the possibilities of the permutation with the given number of repetitions. For example:

```
traces
    T1: || ( obj.inc(), obj.dec() ) {2};
```

```
> runtrace Tester`T1
Generated 4 tests
Test 1 = obj.inc(); obj.dec(); obj.inc(); obj.dec()
Result = [1, 0, 1, 0, PASSED]
Test 2 = obj.dec(); obj.inc(); obj.inc(); obj.dec()
Result = [-1, 0, 1, 0, PASSED]
Test 3 = obj.inc(); obj.dec(); obj.dec(); obj.inc()
Result = [1, 0, -1, 0, PASSED]
Test 4 = obj.dec(); obj.inc(); obj.dec(); obj.inc()
Result = [-1, 0, -1, 0, PASSED]
Executed in 0.038 secs.
All tests passed
```

Here, the `||` operator produces (inc, dec) and (dec, inc); then the repetition doubles this, but it doubles every combination of the two rather than simply repeating each one twice.

2.2 Using Variables

So far, the trace examples have called operations that do not include any arguments. Arguments can be passed as literals, but traces also provide the means to define variables that can change value as tests are generated from a trace.

If we overload the example increment and decrement operations with versions that take an integer parameter, by which to change the counter, we can write traces like this:

```
...
    public inc: int ==> int
    inc(i) == ( total := total + i; return total; );

    public dec: int ==> int
    dec(i) == ( total := total - i; return total; )

traces
    T1:
        let a in set {1, ..., 10} be st a mod 2 = 0 in
            obj.inc(a);
```

```
> runtrace Tester`T1
Generated 5 tests
Test 1 = obj.inc(2)
Result = [2, PASSED]
Test 2 = obj.inc(4)
Result = [4, PASSED]
Test 3 = obj.inc(6)
Result = [6, PASSED]
```

```
Test 4 = obj.inc(8)
Result = [8, PASSED]
Test 5 = obj.inc(10)
Result = [10, PASSED]
Executed in 0.04 secs.
All tests passed
```

In a standard VDM specification, the `let...be st` expression would choose an arbitrary element from the set that meets the `st` clause. But in a trace context, this looseness is used as a pattern that expands to a test covering each possible set value that would match. Notice that the tests list the actual value of the argument passed, rather than the symbolic name, "a".

A trace can include multiple `let` clauses, but if these are nested, then the trace expands to the *combination* of the variables. For example:

```
traces
    T1:
        let a in set {1, 2, 3} in
            let b in set {4, 5, 6} in
                ( obj.inc(a); obj.dec(b) );
```

```
> runtrace Tester`T1
Generated 9 tests
Test 1 = obj.inc(1); obj.dec(4)
Result = [1, -3, PASSED]
Test 2 = obj.inc(1); obj.dec(5)
Result = [1, -4, PASSED]
Test 3 = obj.inc(1); obj.dec(6)
Result = [1, -5, PASSED]
Test 4 = obj.inc(2); obj.dec(4)
Result = [2, -2, PASSED]
Test 5 = obj.inc(2); obj.dec(5)
Result = [2, -3, PASSED]
Test 6 = obj.inc(2); obj.dec(6)
Result = [2, -4, PASSED]
Test 7 = obj.inc(3); obj.dec(4)
Result = [3, -1, PASSED]
Test 8 = obj.inc(3); obj.dec(5)
Result = [3, -2, PASSED]
Test 9 = obj.inc(3); obj.dec(6)
Result = [3, -3, PASSED]
Executed in 0.066 secs.
All tests passed
```

This example produces a test for every combination of "a" and "b" values, which is therefore nine tests. Note also that the variables defined are in scope throughout the clauses below, so the "a" variable could be used to define the set of "b" values:

```
traces
    T1:
        let a in set {1, 2, 3} in
            let b in set {a, ..., a + 2} in
                ( obj.inc(a); obj.dec(b) );

> runtrace Tester`T1
Generated 9 tests
Test 1 = obj.inc(1); obj.dec(1)
Result = [1, 0, PASSED]
Test 2 = obj.inc(1); obj.dec(2)
Result = [1, -1, PASSED]
Test 3 = obj.inc(1); obj.dec(3)
Result = [1, -2, PASSED]
Test 4 = obj.inc(2); obj.dec(2)
Result = [2, 0, PASSED]
Test 5 = obj.inc(2); obj.dec(3)
Result = [2, -1, PASSED]
Test 6 = obj.inc(2); obj.dec(4)
Result = [2, -2, PASSED]
Test 7 = obj.inc(3); obj.dec(3)
Result = [3, 0, PASSED]
Test 8 = obj.inc(3); obj.dec(4)
Result = [3, -1, PASSED]
Test 9 = obj.inc(3); obj.dec(5)
Result = [3, -2, PASSED]
Executed in 0.059 secs.
All tests passed
```

As well as defining a variable value from a set, a variable can be used to simplify calculations that would otherwise have to be made in the arguments to operation calls. These simpler `let` definitions do not increase the number of tests generated from the trace, they just introduce a new name in the scope that follows:

```
traces
    T1:
        let a in set {1, 2, 3} in
            let b = a + 1 in
                ( obj.inc(a); obj.dec(b) );

> runtrace Tester`T1
Generated 3 tests
Test 1 = obj.inc(1); obj.dec(2)
Result = [1, -1, PASSED]
Test 2 = obj.inc(2); obj.dec(3)
```

```
Result = [2, -1, PASSED]
Test 3 = obj.inc(3); obj.dec(4)
Result = [3, -1, PASSED]
Executed in 0.036 secs.
All tests passed
```

Lastly, note that if repetitions are added to a clause within a `let` body, they bind tightly to the operation call rather than the entire `let` clause. If you want to repeat the entire `let`, you have to bracket the whole clause and add a repetition to that. For example:

```
traces
    T1:
        let a in set {1, 2, 3} in
            obj.inc(a){1, 2}
    T2:
        ( let a in set {1, 2, 3} in
            obj.inc(a) ){1, 2}

> runtrace Tester`T1
Generated 6 tests
Test 1 = obj.inc(1)
Result = [1, PASSED]
Test 2 = obj.inc(1); obj.inc(1)
Result = [1, 2, PASSED]
Test 3 = obj.inc(2)
Result = [2, PASSED]
Test 4 = obj.inc(2); obj.inc(2)
Result = [2, 4, PASSED]
Test 5 = obj.inc(3)
Result = [3, PASSED]
Test 6 = obj.inc(3); obj.inc(3)
Result = [3, 6, PASSED]
Executed in 0.044 secs.
All tests passed

> runtrace Tester`T2
Generated 12 tests
Test 1 = obj.inc(1)
Result = [1, PASSED]
Test 2 = obj.inc(2)
Result = [2, PASSED]
Test 3 = obj.inc(3)
Result = [3, PASSED]
Test 4 = obj.inc(1); obj.inc(1)
Result = [1, 2, PASSED]
```

```
Test 5 = obj.inc(2); obj.inc(1)
Result = [2, 3, PASSED]
Test 6 = obj.inc(3); obj.inc(1)
Result = [3, 4, PASSED]
Test 7 = obj.inc(1); obj.inc(2)
Result = [1, 3, PASSED]
Test 8 = obj.inc(2); obj.inc(2)
Result = [2, 4, PASSED]
Test 9 = obj.inc(3); obj.inc(2)
Result = [3, 5, PASSED]
Test 10 = obj.inc(1); obj.inc(3)
Result = [1, 4, PASSED]
Test 11 = obj.inc(2); obj.inc(3)
Result = [2, 5, PASSED]
Test 12 = obj.inc(3); obj.inc(3)
Result = [3, 6, PASSED]
Executed in 0.03 secs.
All tests passed
```

The difference may seem subtle, but the effect is significant. T1 behaves like a simple "$\{1, 2\}$" repetition for each of the `let` values, whereas T2 produces either one or two cases from the *entire set* created by the `let` clause.

Chapter 3

Combinatorial Testing Patterns

Chapter 4

Combinatorial Testing Examples

Appendix A

Combinatorial Testing Syntax

traces definitions = '**traces**', [named trace, { '**;**', named trace }] ;

named trace = identifier, { '**/**', identifier }, '**:**', trace definition list ;

trace definition list = trace definition term, { '**;**', trace definition term } ;

trace definition term = trace definition, { '**|**', trace definition } ;

trace definition = trace binding definition
| trace repeat definition ;

trace binding definition = trace let def binding
| trace let best binding ;

trace let def binding = '**let**', local definition, { '**,**', local definition },
'**in**', trace definition ;

trace let best binding = '**let**', multiple bind, ['**be**', '**st**', expression],
'**in**', trace definition ;

trace repeat definition = trace core definition, [trace repeat pattern] ;

trace repeat pattern = '*'
| '+'
| '?'
| '{', numeric literal, ['**,**', numeric literal, '}'] ;

trace core definition = trace apply expression
| trace concurrent expression
| trace bracketed expression ;

trace apply expression = call statement ;

trace concurrent expression = '||', '(', trace definition,
',', trace definition,
{ ',', trace definition }, ')' ;

trace bracketed expression = '(', trace definition list, ')' ;