Overture Technical Report Series
No. TR-005

December 2020

Guidelines for using VDM Combinatorial Testing Features

by

Nick Battle
Peter Gorm Larsen

Overture – Open-source Tools for Formal Modelling

Document history

Month	Year	Version	Version of Overture.exe	Comment
December	2010	0.1	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 [Nie&11, Larsen&10].

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 [Paulson97, Bicarregui&94].

Model checking provides an approach to formal testing that is considerably better than ad-hoc testing but not as complete as formal proof [Clarke&99]. 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.

1.2 The Structure of this Document

The document is intended to be read sequentially, but readers who are familiar with the basics of VDM traces can skip Chapter 2 and go straight to the patterns in Chapter 3.

- Chapter 2, Working with Traces introduces the concept of a combinatorial test and explains how to work with traces in VDM.
- Chapter 3, Combinatorial Testing Patterns looks at common ways of using traces that are useful in many different situations.
- Chapter 4, Combinatorial Testing Examples uses two more significant specifications to demonstrate typical usage of traces in real life.
- Appendix A, Combinatorial Testing Syntax defines the formal grammar of traces.
- Appendix B, Overture Screenshots includes screen shots of the Combinatorial Testing perspective in Overture.
- Appendix C, Example Model Listings contains the full listings of the models explored in Chapter 4.

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` class defines a simple operation 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`. Trace names are simple identifiers, optionally separated by slashes (e.g. `item456/interface/all`). This example is the simplest trace possible and indicates that the trace should expand to a single test that just calls `obj.inc()`.

This trace can either be executed in Overture in the Combinatorial Testing perspective (see Appendix B), 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 a semi-colon separator:

```
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, respectively. 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() );
```

Notice that the separator has changed to a comma. 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 any operation call (i.e. not just within `||` operators) 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 in this example, though `?` can be used for any operation call.

Along the same lines as `?`, it is possible to add `*` and `+` operators to any 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. The round brackets are needed around the pair of operation calls because a call binds tightly to the **let**. Without the brackets, you get the following scope error, referring to the "a" in the second call to `obj.dec(a)`:

```
traces
  T1:
      let a in set {6, 7, 10} in
          obj.inc(a); obj.dec(a)
```

```
Error 3182: Name 'Tester`a' is not in scope in 'Tester' (example.vpp) at line 28:29
Type checked 2 classes in 0.12 secs. Found 1 type error
```

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
```

If two variables should take values from the same set of values, it is possible to use a `multiple bind` in a trace, but not a `bind list`. For example:

```
traces
  T1:
      let a, b in set {1, 2, 3} 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(2); obj.dec(1)
Result = [2, 1, PASSED]
Test 3 = obj.inc(3); obj.dec(1)
Result = [3, 2, PASSED]
Test 4 = obj.inc(1); obj.dec(2)
Result = [1, -1, PASSED]
Test 5 = obj.inc(2); obj.dec(2)
```

```
Result = [2, 0, PASSED]
Test 6 = obj.inc(3); obj.dec(2)
Result = [3, 1, PASSED]
Test 7 = obj.inc(1); obj.dec(3)
Result = [1, -2, PASSED]
Test 8 = obj.inc(2); obj.dec(3)
Result = [2, -1, PASSED]
Test 9 = obj.inc(3); obj.dec(3)
Result = [3, 0, PASSED]
Executed in 0.061 secs.
All tests passed
```

As well as defining a variable value from a set, variables 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 new names in the scope that follows. Multiple variable definitions can be declared in one **let** expression. For example:

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

```
> runtrace Tester`T1
Generated 3 tests
Test 1 = obj.inc(2); obj.dec(0)
Result = [2, 2, PASSED]
Test 2 = obj.inc(3); obj.dec(1)
Result = [3, 2, PASSED]
Test 3 = obj.inc(4); obj.dec(2)
Result = [4, 2, PASSED]
Executed in 0.054 secs.
All tests passed
```

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.

Although the example above uses a `let <set bind>` expression, it is also possible to use a **let** `<seq bind>` or **let** `<type bind>`. In a trace context, the ordering that a sequence bind carries does not affect the generation of tests; if the example had **let** `a` **in seq** `[1,2,3]`, the same tests would be generated, and since tests are independent their order is not meaningful. So sequence binds are not particularly useful in traces. However type binds (of finite types) are a shorthand for "all values of this type", which can be useful in some circumstances. This is covered later in Chapter 3.

2.3 Tests with Errors

The examples so far have only includes tests that **PASSED**. This means that they completed the sequence of operation calls without violating any pre-conditions, post-conditions, state invariants, type invariants, recursive measures or dynamic type checks.

If a sequence of operations causes a post-condition failure, then it is certain that there is a problem with the specification - it should not be possible to provoke a post-condition failure with a set of legal calls (ie. ones which pass the pre-conditions and type invariants). On the other hand, if a sequence of operations violates a pre-condition, or a type or class invariant, then it is *possible* that the specification has a problem, but it is also possible that the test itself is at fault (passing illegal values).

The combinatorial testing environment indicates the exit status of the test in the verdict returned in the last item of the results (all `PASSED` above). So if pre/post/invariant conditions are violated during a test, this may be set to `FAILED` or `INDETERMINATE`[1]. If a test fails, then any subsequent test which starts *with the same sequence of calls* as the failed sequence will also fail. These tests are filtered out of the remaining test sequence automatically, and not executed.

For example, if we introduce a pre- and postcondition into our example, we see this behaviour:

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

[1]The tools sometimes call this `INCONCLUSIVE`

```
  pre i < 10
  post total < 20;

traces
  T1:
      let a in set {6, 7, 10} in
          obj.inc(a){1, 5}
```

```
> runtrace Tester`T1
Generated 15 tests
Test 1 = obj.inc(6)
Result = [6, PASSED]
Test 2 = obj.inc(6); obj.inc(6)
Result = [6, 12, PASSED]
Test 3 = obj.inc(6); obj.inc(6); obj.inc(6)
Result = [6, 12, 18, PASSED]
Test 4 = obj.inc(6); obj.inc(6); obj.inc(6); obj.inc(6)
Result = [6, 12, 18, Error 4072: Postcondition failure: post_inc in 'Counter' at line 15:16, FAILED]
Test 5 = obj.inc(6); obj.inc(6); obj.inc(6); obj.inc(6); obj.inc(6)
Test 5 FILTERED by test 4
Test 6 = obj.inc(7)
Result = [7, PASSED]
Test 7 = obj.inc(7); obj.inc(7)
Result = [7, 14, PASSED]
Test 8 = obj.inc(7); obj.inc(7); obj.inc(7)
Result = [7, 14, Error 4072: Postcondition failure: post_inc in 'Counter' at line 15:16, FAILED]
Test 9 = obj.inc(7); obj.inc(7); obj.inc(7); obj.inc(7)
Test 9 FILTERED by test 8
Test 10 = obj.inc(7); obj.inc(7); obj.inc(7); obj.inc(7); obj.inc(7)
Test 10 FILTERED by test 8
Test 11 = obj.inc(10)
Result = [Error 4071: Precondition failure: pre_inc in 'Counter' at line 14:11, INCONCLUSIVE]
Test 12 = obj.inc(10); obj.inc(10)
Test 12 FILTERED by test 11
Test 13 = obj.inc(10); obj.inc(10); obj.inc(10)
Test 13 FILTERED by test 11
Test 14 = obj.inc(10); obj.inc(10); obj.inc(10); obj.inc(10)
Test 14 FILTERED by test 11
Test 15 = obj.inc(10); obj.inc(10); obj.inc(10); obj.inc(10); obj.inc(10)
Test 15 FILTERED by test 11
Executed in 0.075 secs.
Some tests failed or indeterminate
```

The `inc` operation now has a pre-condition that the argument must be less than 10 and a postcondition that the resulting total must be less than 20. The trace makes 1 to 5 calls to the `inc` operation with arguments 6, 7 and 10, respectively.

The first three tests are fine, but `Test 4` fails because the fourth call to `inc(6)` pushes the total over the limit. This is therefore a post-condition `FAILED` test, and the error message is listed along with the results of the earlier operation calls. Test 5 then tries to do the same, but adds a further call. This must fail in the same place as `Test 4`, because`Test 4` is the “stem” of `Test 5`. Therefore this test is “`FILTERED` by `Test 4`”. Similarly, `Test 8` fails and Tests 9 and 10 are filtered by this failure.

Test 11 fails on the first call to `inc(10)`, since the argument must be less than 10. This produces an `INDETERMINATE` error because we are not sure whether this is a problem with the trace or the specification being tested. Lastly, Tests 12 to 15 are filtered by Test 11, since they would behave the same way.

At the end of the run, the `runtrace` command indicates that some tests failed or were indeterminate, just to remind you.

2.4 Trace Reduction

A trace may expand to many millions of tests and these may take many hours to execute. This can make large traces difficult to work with, both while the trace is being developed and subsequently when traces are used to check that a change to a specification is sound.

Therefore the trace system provides the means to reduce the number of tests that are generated. This limited number can then be checked more quickly, which naturally does not provide the confidence of a full execution, but is convenient to work with when a sample of tests is sufficient.

Trace reduction can be achieved in four different ways:

- *RANDOM* reduction. This is the simplest kind of reduction, and is used to randomly select tests from the full set. For example, a 1% RANDOM reduction of a trace that expands to a million tests would select 10,000 tests from the set. The pseudo-random selection is seeded, so that a consistent subset of the tests can be selected.

- *SHAPES_NOVARS* reduction. The "shape" of a test means the sequence of operation names (regardless of arguments passed), and the idea of shaped reduction is to preserve *at least one test* of every shape in the full set. So for example, a test that calls `[opA(1), opB(2), opC(3)]` would be considered the same shape as a test that calls `[opA(111), opB(222), opC(333)]`, but different to a test that calls `[opX(1), opY(2)]`. So a shaped reduction of 1% of one million tests would try to select 10,000 tests, but it also guarantees to include at least one test of every shape within the million, even if that means the reduction is (say) 1.5%.

- *SHAPES_VARNAMES* reduction. The simple interpretation of a shape above only looks at the names of the operations called. This second kind of shaped reduction looks at the variable names used in the `let` bindings and constants as well as the names of the operations. So with this kind of reduction, `[opA(a), opA(b)]` is different to `[opA(x), opA(y)]`, even if the values of "`a`" and "`x`" can sometimes be the same. Typically, different variable names are used in different parts of a complex trace and so this reduction method is trying to select all of the different parts of a trace, even if the sequence of operation names produced is the same as another part of the trace.

- *SHAPES_VARVALUES* reduction. The third type of shaped reduction is even more specific about what constitutes a shape, taking into account the *value* of the variables used as well as their names. So `[opA(a)]` will be considered a different shape to another `[opA(a)]` elsewhere, as long as "`a`" is bound to a different value.

The following simple trace, using the `Counter` example from previous chapters, illustrates RANDOM reduction:

```
traces
  T1:
      let a in set {1, ..., 1000} in
      let b in set {1, ..., 1000} in
          ( obj.inc(a); obj.dec(b) );
```

```
> filter
Usage: filter %age | RANDOM | SHAPES_NOVARS | SHAPES_VARNAMES | SHAPES_VARVALUES | NONE
Trace filter currently 100.0% NONE (seed 0)

> filter random
Trace filter currently 100.0% RANDOM (seed 0)

> filter 1%
Trace filter currently 1.0% RANDOM (seed 0)

> seedtrace 1234
Trace filter currently 1.0% RANDOM (seed 1234)

> runtrace Tester`T1
Generated 1000000 tests, reduced to 10000, in 1.571 secs.
Test 66 = obj.inc(1); obj.dec(66)
Result = [1, -65, PASSED]
Test 155 = obj.inc(1); obj.dec(155)
Result = [1, -154, PASSED]
Test 323 = obj.inc(1); obj.dec(323)
Result = [1, -322, PASSED]
Test 419 = obj.inc(1); obj.dec(419)
Result = [1, -418, PASSED]
...
Test 999815 = obj.inc(1000); obj.dec(815)
Result = [1000, 185, PASSED]
Test 999894 = obj.inc(1000); obj.dec(894)
Result = [1000, 106, PASSED]
Test 999992 = obj.inc(1000); obj.dec(992)
Result = [1000, 8, PASSED]
Executed in 4.932 secs.
```

The "filter" and "seedtrace" commands are used to set the filtering required, then the trace is executed as normal. But we see that the million tests have been reduced to 10,000 (taking a few seconds). Then instead of trying every combination of "a" and "b", the filtering selects them at random, starting with `a = 1, b = 65` and ending with `a = 1000, b = 992`.

The next example illustrates shaped reduction. The trace joins together two **let** bindings, repeating each call once and twice. This would normally produce 36 tests.

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

```
> filter 1
Trace filter currently 1.0% RANDOM (seed 0)

> filter shapes_novars
Trace filter currently 1.0% SHAPES_NOVARS (seed 0)

> rt Tester`T1
Generated 36 tests, reduced by SHAPES_NOVARS, in 0.001 secs.
Test 1 = obj.inc(1); obj.dec(1)
Result = [1, 0, PASSED]
Test 2 = obj.inc(1); obj.inc(1); obj.dec(1)
Result = [1, 2, 1, PASSED]
Test 7 = obj.inc(1); obj.dec(1); obj.dec(1)
Result = [1, 0, -1, PASSED]
Test 8 = obj.inc(1); obj.inc(1); obj.dec(1); obj.dec(1)
Result = [1, 2, 1, 0, PASSED]
Executed in 0.006 secs.
All tests passed
>
```

Here, we try to reduce this to 1%, but using the SHAPES_NOVARS option. A reduction of 1% of 36 tests ought to produce a single test (reduction will never produce zero tests), but in fact the shaped reduction produces four. You can see that there is one case of each "shape": one `inc` and one `dec`; two `inc`'s and one `dec`, and so on. So the idea is that the shaped reduction has given a representative sample of all of the possible shapes, disregarding variable names or values.

Reducing this trace using SHAPES_VARNAMES produces the same number of shapes, since the two calls always use the same variable names. But reduction using SHAPES_VARVALUES regards all of the tests as different shapes, because the variable/value/operation combinations are different in all of the tests.

In practice, the most useful reductions are RANDOM and SHAPES_NOVARS. Random reductions give a simple way to cut down a large number of tests. Shaped reduction is choosing one example of every "path" that the trace is taking the specification through, which is often closely related to the different use cases that the system has.

2.5 How does Trace Expansion Work?

The sections above have given an overview of all the trace operators, and there are some examples of combinations of operators. But to see how traces are expanded in general, we need to look at traces from a different point of view. The syntax of traces is deliberately made similar to the syntax of VDM-SL, but to understand how operators combine to produce multiple tests, it helps to look at operators as though they followed a separate "expansion" grammar. In the description that follows, a `set` is a set of tests:

- `set = object.opname(args)`. The simplest form of a trace is a set that comprises a single call to an operation or function with arguments. The arguments can be symbolic, and bound to various values by the **`let`** operator described below.
- `set = set1; set2; ...; setn`. A set of tests may be formed from an ordered sequence of sets. This expands to all possible selections of one test from each of the sets. In its simplest form, this could be a sequence of operation calls which therefore just expands to one test. But a combination of sets of tests results in a set of the product of the sizes of those sets.
- `set = set1 ?`. A set of tests may be formed from another set with a `?` operator. This produces the same set, but includes a "skip" step.
- `set = set1` {`n[, n]`}. A set of tests may be formed from another set with a {`n`} or {`n1, n2`} operator. This produces a set with every member of the original set repeated `n` times, or between `n1` and `n2` times (inclusive).
- `set = set1 *|+`. A set of tests may be formed from another set with a `*` or `+` operator. This produces another set with every member of the original set repeated from 0 to N times (with `*`) or 1 to N times (with `+`). The value of N is tool dependent, but defaults to 5.
- `set = set1 | set2 | ... | setn`. A set of tests may be formed by combining a number of other test sets with a `|` operator. This produces a set with the union of the other sets.
- `set = || (set1, set2, ..., setn)`. A set may be formed from the permutations of a number of other sets. This produces a set with each permutation of each selection of one test from each set.
- `set = let <multiple bind> [be st <cond>] in set1`. A set of tests may be formed from a `multiple bind`, which expands to the substitution of all the possible the bound values in the original set.
- `set = let <name> = <exp> [, <name2> = <exp2>, ...] in set1`. A set of tests may be evaluated in a scope that defines name/value pairs. This does not increase the number of tests in the set, but just binds free variables.

For example, if (for brevity) we say that a test with a single call to `obj.opA()` is written as "[A]", and similarly "[B]" and "[C]" for other operation calls, and "[-]" for a skip, then we can say the following trace operators produce these sets of tests:

```
A? = { [A], [-] }
A;B = { [AB] }
A;B? = { [AB], [A] }
A* = { [-], [A], [AA], [AAA], [AAAA], [AAAAA], ... }
A+ = { [A], [AA], [AAA], [AAAA], [AAAAA], ... }
A{3} = { [AAA] }
A{1,3} = { [A], [AA], [AAA] }
A | B = { [A], [B] }
A | B? = { [A], [B], [-] )
|| (A, B, C) = { [ABC], [ACB], [BAC], [BCA], [CAB], [CBA] }
|| (A, (B;C)) = { [ABC], [BCA] }
|| (A, B+) = { [AB], [BA], [ABB], [BBA], [ABBB], [BBBA], ... }
let a in set {1,2,3} in A(a) = { [A(1)], [A(2)], [A(3)] }
let b : bool * bool in B(b) = {
    [B(mk_(true, true))], [B(mk_(true, false))],
    [B(mk_(false, true))], [B(mk_(false, false))]
}
let z = 1 in B(z) = { [B(1)] }
```

Note that the repeat limits in a trace (like {1,3}) must be numeric literals. But values in a multiple bind set or sequence can be variables, either bound earlier in the trace or other fields within scope of the trace inside the object or module where it is defined. Similarly, the values in the right hand side of let definitions can be variables within the trace or the object/module scope.

2.6 Language Considerations

Combinatorial tests are available for both VDM-SL and VDM++/VDM-RT. The process of trace expansion and execution is very similar in all cases, but there are some differences that are described below.

2.6.1 Traces in VDM-SL

Traces are added in a "traces" section within a VDM-SL specification. This can either be within one or more modules or within a flat specification. The name of the traces in a module are implicitly exported, so they are referred to as ``<modulename>`<tracename>``. You can omit the module name if it is the default module.

The VDM-SL specification that is equivalent to the example used above is like this:

```
module Counter
exports all
definitions
```

```
state S of
    total:int
init s == s = mk_S(0)
end

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

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

module Tester
imports from Counter all
definitions

traces
    T1: Counter`inc(1)*;

end Tester
```

And in the VDM-SL command line, that would be executed as follows. Note that Tester is not the default module, so the trace name is qualified:

```
> modules
Counter (default)
Tester
> runtrace Tester`T1
Generated 6 tests in 0.002 secs.
Test 1 = skip
Result = [(), PASSED]
Test 2 = inc(1)
Result = [1, PASSED]
Test 3 = inc(1); inc(1)
Result = [1, 2, PASSED]
Test 4 = inc(1); inc(1); inc(1)
Result = [1, 2, 3, PASSED]
Test 5 = inc(1); inc(1); inc(1); inc(1)
Result = [1, 2, 3, 4, PASSED]
Test 6 = inc(1); inc(1); inc(1); inc(1); inc(1)
Result = [1, 2, 3, 4, 5, PASSED]
Executed in 0.019 secs.
All tests passed
>
```

This trace is very similar to the VDM++ example. The `Counter` module has a single state that is equivalent to the VDM++ “total” instance variable. Note that this is reset to zero automatically before each test is executed. This is because each test re-initialises the specification, and the module state has an “**init**” clause that sets the total to zero.

Notice also that the operation calls are not applied to a Counter object, unlike VDM++.

2.6.2 Traces in VDM++ and VDM-RT

Traces are added in a "traces" section within a VDM++ or VDM-RT specification, inside one or more classes. In effect, the name of the trace is a public static symbol, so it is referred to as `` <classname>`<tracename> ``, as we have seen in the examples above. You can omit the class name if that is the default class.

Although a VDM++ trace is effectively a static scope, and can call static operations directly (similar to a VDM-SL trace), every test execution occurs in a *new instance* of the containing class – in our examples, in a new Tester instance. This means that objects created within the Tester's construction will be freshly initialised and ready for use in each test run. In the example, the Counter object `obj` is created for each test, because the instance variable is initialised at construction.

2.6.3 Expansion and Execution Considerations

The process of running a combinatorial test has two phases: expanding the trace to a number of test definitions; and subsequently executing those definitions. The trace expansion typically does not take very long, since it is only constructing a tree of iterators that are capable of generating the tests one after another. The subsequent execution of those tests can obviously take a long time, depending on how many there are.

We have seen (above) how the specification is initialised before test execution, and the state of the module or class is available to the trace, but care must be taken if operations or functions within the environment are used as part of a trace. This is because some expressions are evaluated during trace expansion and some during test execution. For example:

```
...
functions
  private static range: int * int -> set of int
  range(a, b) == {a, ..., b};

values
  Z = 100;

traces
  T1: let x in set range(3, 5) in
      let y in set range(x, x+2) in
          obj.inc(Z + x + y);
```

In this case, the `range` function is used to create a set for the multi-binds, and this is executed during *expansion*, once for "x" and three times for "y". Similarly, the right hand side of simple let definitions are executed during trace expansion. But the addition of `Z + x + y` in the argument to `inc` is called during *execution* (once for each test).

Trace generation starts inside a fresh object instance of the class (or initialised module) that contains the trace. So if operations are called during trace *expansion*, these can modify state and so affect subsequent operation calls elsewhere in the expansion. This can become very confusing, and it is not a recommended trace design strategy! On the other hand, calling functions as part of the trace expansion can make traces easier to understand and can provide the means to build complex sets that would be difficult to construct directly within the trace statements.

When a test is listed in the trace output, the arguments that are passed to operation calls are shown as literals, if possible. As seen in the examples here, a call to `obj.inc(1)` is shown, rather than `obj.inc(a)`. This is possible whenever arguments can be easily evaluated. For example the `Z + x + y` case above would produce `Test 1 = obj.inc(106)`, which is 100 + 3 + 3. But if the argument is a more complex expression involving operation applies or new object creation, these cannot be evaluated and so the argument expression is listed "as is". For example, if the trace above is changed to call `obj.inc(max(x, y))`, the test would be listed with "`x`" and "`y`" rather than their current values:

```
> runtrace Tester`T1
Generated 9 tests in 0.006 secs.
Test 1 = inc(max(x, y))
Result = [3, PASSED]
Test 2 = inc(max(x, y))
Result = [3, PASSED]
...
```

Chapter 3

Combinatorial Testing Patterns

3.1 Data and Process Traces

Traces fall into two broad categories, although you can have a mixture of both in a specification. One category is focussed on the behaviour of a single function or operation when presented with a large variety of different data structures; the other category is focussed on the process behaviour of a system when exercised by a large number of different operation call paths.

In one sense this distinction is artificial. You can create a trace that explores many process paths and also creates a wide variety of data values to pass to the operations called on the way. But these aspects of a specification's behaviour are often separable, and little advantage is gained by trying to test everything at once.

The patterns described in this chapter are generally useful in one or other of the trace styles. The styles are also illustrated in more detail by the examples in Chapter 4.

3.2 Traces and Test Operations

After expanding a trace to all of the call sequences that match, a test is ultimately just a sequence of operation or function calls. But these calls do not have to be the primary operations that drive the specification. In some cases, it makes sense for traces to expand to a set of tests that call *test operations* that maintain their own state and exercise the main specification, checking the responses and the main state. Checks like this, that do not directly form part of the constraints of the specification, are usually called *validation conjectures*. The task of the testing operations is therefore to check that, whatever the call sequence made by the trace, the validation conjectures for the specification are maintained.

For example, a specification may describe how a sequence of parts are produced as calls are made to `newPart(), adjustPart(), completePart()`. The process of creation of individual parts may well involve preconditions, postconditions, type invariants and so on. But there may also be a requirement that (say) over time, the total number of parts of type A and type B never differ by more than a tolerance. This is a validation conjecture: it is not directly stated in the constraints of the specification, but it is a behaviour that must be manifest by the system over time. Therefore the trace(s) for such a system can call the main operations via test operations, like `testNewPart()` and so on, and the `testCompletePart()` operation can check the history of parts created to validate that the tolerance is always respected, regardless of the sequence of operations that the trace tries. Note that these test operations can maintain their own state that is private and separate from that of the main specification.

In such cases, it makes sense to add all of the test operations to a separate class or module, to make it clear that they are not part of the main specification. This is illustrated in the Basket Service example in Chapter 4.

3.3 Common Patterns

Experience has shown that the testing of many specifications with traces requires several common "patterns" to create data selections or sequences of operations. These are presented in this section.

3.3.1 Sets for "let" bindings

The set of values that is used by a **`let`** `multiple-bind` is usually shown as an enumeration of literals in examples. But the set value can use any VDM expression that yields a set. The following cases are generally useful:

- Set comprehensions can be used to select values from a larger set that meet the membership predicate. This is similar to the use of the **`be st`** clause, but the filter acts on the members of the set rather than the values that are bound:

```
let pair in set {mk_(a, b) | a, b in set VALUES & a > b} in ...
```

- The `power` operator can be used to produce all possible subsets from another set, though you often have to eliminate the empty set, which is produced by the operator:

```
let options in set power ALL_OPTIONS \ {{}} in ...
```

- If you are using finite types (i.e. types that have a finite number of values), then you can use a multiple type bind to conveniently choose all of the possible values of that type:

```
let p1, p2 : Product in ...
```

- Often elements have to be chosen from a set such that they are different to previous selections from the same set. This can sometimes be done in a single **`let`** bind by using a **`be st`** clause that states that the values are different. But a common usage is to make a selection from a set that has had the first choice(s) eliminated:

```
  let a, b in set S be st a <> b in ...
  let c in set S \ {a, b} in ...
  let d in set S be st d not in set {a, b, c} in ...
```

- A common requirement is to select permutations of a set. This can be done by using the looseness of a set bind. Note the cardinality check in the k-permutation example (k=3), and the use of the set pattern and the set-of-set {S} in the full permutation example - in this case you have to know the size of the set to match the pattern:

```
-- Every k-permutation of 3 values from S
  let p1, p2, p3 in set S be st card {p1, p2, p3} = 3 in ...
-- Permutations of all values from S
  let {p1, p2, p3, p4, p5} in set {S} in ...
```

- It is sometimes useful to be able to select all k-combinations from a set, rather than k-permutations. This is similar, but with a **be st** discriminator that selects one unique combination from the possible orderings (here, `c1 < c2 and c2 < c3`):

```
-- Every k-combination of 3 values from S
  let c1, c2, c3 in set S be st card {c1, c2, c3} = 3
      and c1 < c2 and c2 < c3 in ...
```

3.3.2 Bracketing operations with headers and trailers

A very common requirement is to make a number of fixed setup calls, followed by a large number of different test calls, followed by a number of fixed closedown calls. This pattern emerges naturally from a semi-colon separated trace sequence with a complex "middle":

```
traces
  T:
    SetSystemDate(221220);
    LoadCertificates(RefData);
        test1() |
        test2() |
        test3() |
        test4() |
        test5() |
        test6() |
        test7();
    EndTransaction();
```

This produces a set of seven tests, calling `test1` to `test7`, each of which is sandwiched by calls that set up the system and close down the transaction. Clearly the body containing the alternative tests can be arbitrarily complicated, for example using variable binds to generate many possibilities.

3.3.3 Graph searching by traces

Traces that explore all of the possible uses cases of a system frequently need to search a graph that describes the possible paths through the use case. This is translated into a trace by generating a call for each step through the graph, where the trace expands to the possibilities at each step. For example:

```
values
  TESTSTATES : map TestState to map Event to TestState =
    {
        <READY> |->
        {
            <SEND>    |-> <SENT>
        },

        <SENT> |->
        {
            <RX_NACK> |-> <RESEND1>,
            <RX_ACK>  |-> <END>,
            <TIMEOUT> |-> <RESEND1>,
            <PARTIAL> |-> <SENT>
        },
        ...
    }

traces
  AllTransitions:
      let s1 = <READY> in
      let ev1 in set dom TESTSTATES(s1) in
      let s2 = TESTSTATES(s1)(ev1) in
      let ev2 in set dom TESTSTATES(s2) in
      let s3 = TESTSTATES(s2)(ev2) in
      let ev3 in set dom TESTSTATES(s3) in
      let s4 = TESTSTATES(s3)(ev3) in
      let ev4 in set dom TESTSTATES(s4) in
      let s5 = TESTSTATES(s4)(ev4) in
      let ev5 in set dom TESTSTATES(s5) in
          execute([ev1, ev2, ev3, ev4, ev5]);
```

Here the system starts in state `s1`, which must be `READY`. From there, a number of events are possible given by the domain of a lookup of the state in the `TESTSTATES` map. One of these events is selected as `ev1`, which then moves us to state `s2`, and so on. After five events have been generated, they are passed to an `execute` operation which uses the events to test that particular path through the graph.

Note that a graph searching cascade like this can generate tens of thousands of possibilities very quickly, even with comparatively simple graphs.

3.3.4 Building data in stages

A cascade of let bindings can be used to build a complex data structure in stages, rather than each binding having to create a complete value. For example:

```
EPATest:
    let a, b, c, d, e in set    -- Pick five branches
    {
        mk_B([1],        0),
        mk_B([-1, 2],    1),
        mk_B([1, 2, 0],  2)
    }
    in

    let branches in set         -- Between one and five of them
    {
        [a],
        [a,b],
        [a,b,c],
        [a,b,c,d],
        [a,b,c,d,e]
    }
    in

    let epa = mk_InputFile(     -- EPA InputFile from those B values
        mk_Header(),
        mk_PaymentSummary(getPS(branches)),
        {
            mk_Branch(
                mk_SummaryOfCharge(mk_token(mk_("MID", B)), ...),
                {
                    mk_RecordOfCharge(... mk_("TxnNum", B, i) ...)
                    | i in set inds branches(B).ROCs
                },
                {
                    mk_Adjustment(mk_token(mk_(B, i)))
                    | i in set {1, ..., branches(B).ADJs}
                }
            )
            | B in set inds branches
        },
        mk_Trailer(getTR(branches) + 1)
    )
    in

    -- Finally, transform the EPA file into the various output formats.
    (
        transformAudit(epa);
        transformC4D(epa)
    );
```

This example creates `InputFiles`, which contain various records that relate to electronic payments taken from a branch of a business. The objective of the trace is to check a large number of different input files with different numbers of branches and various record types.

Creating such files in a single trace step would be difficult, if not impossible. But here, we see that the generation starts with a selection of “B” values from a set of possibilities. Then sequences of between one and five of these values is created. Then these “branch” sequences are used in nested set comprehensions to create an `InputFile`. Lastly, every InputFile created is processed by a couple of operations.

3.3.5 Oracle functions

In many cases, the expected result of an operation or function call is too complicated to predict simply in a trace - assuming you have a support function that checks the result with (say) a post-condition:

```
assert: seq1 of nat * seq1 of nat +> bool
assert(data, expected) == data = expected
post RESULT = true;
```

Here, we assume that the "`data`" argument passed comes from some processing in the specification, but where does the "`expected`" value come from? It could be a literal in the trace, but this is not practical for non-trivial examples, so it is common to create support functions that produce answers that are correct *by definition*. Such functions are called *oracles*. For example:

```
Scenarios:
  let s = mk_Service(...) in
  let q in set ... in
  let mk_(first, second) in set ... in
  let test = mk_Test(
      [s],
      keyStrokesFor(q, first, second),     -- Expected keystrokes
      basketFor(first, second))            -- Expected basket
  in
      run(test);
```

In this example, a `Test` record is created that includes a Service, the expected keystrokes and the expected basket result for a retail application in a given scenario (use case). The `run` operation uses the key strokes to drive the specification under test and then check that the expected basket is produced correctly. Both of these functions are oracles.

3.3.6 Using positive and negative sense checks

The natural way to think about testing a specification is to consider all of the success paths and then design traces that exercise those paths. But in many cases, a specification is also required to "fail" in the correct way; that is, there certain inputs or call sequences that require a specific error condition to be generated, even though that is a failure in some sense.

There is an example of this in the Luhn specification described in Chapter 4. The Luhn algorithm is a kind of checksum. Therefore it is *required* to fail if a piece of data is corrupted in specific ways, which shows that the checksum is doing its job, detecting the corruption. So the Luhn specification tests deliberately corrupt a piece of data and then verify that the algorithm generates a *different* check digit – i.e. they verify that a check with the original check digit fails correctly:

```
checkFail: seq1 of nat * nat * nat ==> bool
```

```
  checkFail(data, expected, base) ==
      return luhn(data, base) <> expected    -- Expect failure
  post RESULT = true;
```

3.4 Avoiding Test Explosions

It is extremely easy to write a comparatively simple looking trace definition that expands into a collection of tests that is so large that it is not practical to execute, either because it would take too long or because the generation process takes up too much memory.

The following tips will help you to avoid this pitfall:

Start small: It is tempting to write traces as clearly as possible to start with, and that may lead to the binding of values from sets of data or types with many values. The combinatorial expansion process will then either multiply these data sizes together, or in some cases generate combinations that depend on the product of the *factorial* of the data sizes. So a modest set of 50 values might therefore generate of the order of 10^{64} tests (i.e. the factorial of 50). So start small: design and test traces with small example sets and types, and only expand the data selections when you can see that the trace is expanding as you require.

Split up traces: The alternation operator, `|`, will join together two sets of tests in a trace. It may therefore be tempting to write a single trace that is composed of many parts, testing different parts of the system, joined by alternation. This has the advantage that the whole specification can be tested by running one trace. But it also means that the trace expands to the sum of all of the tests within all of the parts. That is much better than a product or factorial combination, but if the parts are genuinely separate, you will be able to do more tests in the parts by separating them into multiple traces.

Be careful with multiple-binds and power sets: As mentioned above, factorial scaling is extremely expensive. This occurs most commonly with multiple-binds that are used for k-permutations, and with the **`power`** operator on sets. As suggested above, in these case start with small sets and increase them with care!

Chapter 4

Combinatorial Testing Examples

This Chapter looks at two significant specifications and their traces. Extracts of the specifications are given, but the full listing of both can be found in Appendix C.

4.1 The Luhn Check Digit Model

4.1.1 Background

The Luhn[1] check digit algorithm is commonly used in commercial systems to provide a simple means to check the integrity of a number, such as a credit card number or a product barcode.

The simplest version of the algorithm is able to check strings of decimal digits. A more complex version is defined for checking strings of digits in an arbitrary numerical base and encoding. In both cases, the check digit produced is a valid character in the numerical base of the input. The example presented here is the more complex algorithm.

The Luhn algorithm can detect a wide range of common transcription errors, for example when adjacent digits are swapped in the input. But its error detection is not perfect. In particular, there are specific patterns of input corruption that are *not detected* by the algorithm. The testing of the specification has to verify both the correct detection of most corruptions, and the the correct failure to detect the known weaknesses.

The discussion below starts by describing the structure and traces used to test a standard version of the Luhn algorithm. Then we look at a *non-standard* implementation from a real life example, and show how combinatorial tests not only identify the problem, but also show that the level of error checking provided by the non-standard implementation is weaker than the standard one.

4.1.2 The Structure of the Specification

The Luhn algorithm is defined in a singe class, called "LUHN".

The first part of the specification defines the encoding for the digit strings that are to be checked. The algorithm is defined for a given *base*, and therefore a set of characters for the base must be

[1] See https://en.wikipedia.org/wiki/Luhn_algorithm

defined. In the simple base-10 case, the ten characters are usually '0', '1', ..., '9', but they could be any selection of ten distinct characters. Similarly, other bases require more or fewer characters, and the choice is arbitrary. A given choice is defined by a `Mapping` type in the model:

```
public Mapping = inmap char to nat
inv m == rng m = {0, ..., card rng m - 1};
```

The top level Luhn algorithm is then a function which takes a `Mapping` as well as a base and a string of characters, returning a single check digit:

```
public luhns: seq1 of char * nat1 * Mapping -> char
luhns(string, base, mapping) ==
    let encoded = [ mapping(string(i)) | i in set inds string ] in
        (inverse mapping)(luhn(encoded, base))

pre (elems string subset dom mapping) and (card rng mapping = base)
post RESULT in set dom mapping;
```

The precondition checks that the string passed is only comprised of characters in the domain of the mapping, and that the base is consistent with the size of the mapping. The postcondition checks that the character returned is also a member of the mapping.

The body of the function uses the mapping to encode the input string into a sequence of numbers. These are then processed by a lower level `luhn` function to produce a check value, which is finally mapped back to a character using the inverse of the mapping.

The `luhn` core function is passed a sequence of numbers and a base, returning another number of that base:

```
public luhn: seq1 of nat * nat1 -> nat
luhn(data, base) ==
    let remainder = total(data, base) mod base in
        (base - remainder) mod base
pre forall i in set inds data & data(i) < base
post RESULT < base and (total(data, base) + RESULT) mod base = 0;
```

The Luhn check value is defined in terms of a `total` function. The Luhn result is difference between the total function and the base (modulo the base). The precondition checks that all of the values in the input string are within the base, and the postcondition checks that the result is within the base and that adding the total value to the result gives zero (modulo the base).

Lastly, the `total` function is passed the string of input values and produces a total, which is the sum of the digits of the value multiplied by a factor which alternates between 1 and 2, such that the *rightmost* factor is always 2.

4.1.3 Testing Approach

The natural approach to test the algorithm is to produce a check digit for some strings with known check digits, such as a credit or debit card number[2], where the Luhn calculation is in base-10 and computed over the first 15 digits of the number:

```
> print Test`luhn10("492912341234999")
= '5'
Executed in 0.006 secs.
```

This is a perfectly valid way to test, and quickly gives confidence that the LUHN specification is roughly correct. But of course there may be many edge cases and strange combinations of digits that cause issues. This is where combinatorial tests give us more testing power.

The style of combinatorial testing needed for a specification like LUHN is the "data" style, where traces are used to generate a large number of input data strings, which are then verified by a single function. See 3.1.

The first trace creates a large set of test strings to check whether the specification produces a result that does not violate any of the constraints:

```
values
  charToCodeMap10 : LUHN`Mapping =
    let decimal = "0123456789" in
      { decimal(a) |-> (a-1) | a in set inds decimal };

traces
  FirstN:
    let a,b,c,d in set dom charToCodeMap10 in
    (
      luhn10([a])       |
      luhn10([a,b])     |
      luhn10([a,b,c])   |
      luhn10([a,b,c,d])
    );
```

This uses the domain of the `Mapping` for base 10 to produce *all possible* sequences of 1, 2, 3 and 4 digits, calculating the Luhn check digit for each. Note the following:

- Unlike the ad-hoc tests of various credit card examples, this trace does not include the correct answer for each test. Rather, it depends on the postcondition of the `luhn` function, which says that the `total` function plus the check digit must be zero (mod 10). This is generally the case for combinatorial tests, unless you produce an oracle function instead of a

[2]No, this isn't my real credit card number!

postcondition. In this case, such an oracle would have to reproduce the Luhn algorithm, and so this would add little value (i.e. the oracle could be wrong as well).

- The trace only includes strings of length 1 to 4, not 5 and beyond. Why not? The reason is a judgement, made by the tester. A string with a single value may well be a special case; similarly a string of two is the minimal example of a multi-character string. The Luhn algorithm treats alternate places in the string differently, so adding three and four values covers cases for the second pair. Tests of five or more values are probably adding less value, and at some point you simply have to stop. This trace produces 40,000 tests. If you have time, it would be easy to extend it to produce a million, but if the first 40,000 work there is unlikely to be a problem in the remainder.

After the `FirstN` trace, there is a second one called `FirstN_16`, which does the same processing but in base 16 rather than base 10. This naturally produces more test cases (because the alphabet is larger), so to keep the trace size manageable, it covers strings of length 1 to 3. As before, this can be extended (with diminishing returns) if you have time.

So at this point, we have much greater confidence that the specification is producing results that meet the internal constraints. But recall that the correct Luhn specification is also *unable* to detect certain changes in its input. This is an unusual case where the algorithm is required to produce a particular "wrong" behaviour, in a sense. So next we can design traces that test this particular aspect.

The first trace checks that the algorithm correctly detects all single-digit errors. This means all cases where a single digit is changed to another value:

```
AllOneDigitErrors:
  let base = 16 in
  let input = conc [[a, a] | a in set {0, ..., base-1}] in
  let expected = LUHN`luhn(input, base) in
  let pos in set inds input in
  let replacement in set {0, ..., base-1} \ {input(pos)} in
  let corrupt = input(1, ..., pos-1) ^
                [replacement] ^ input(pos+1, ..., len input)
  in
    checkFail(corrupt, expected, base);
```

This trace is mostly simple definitions. The test uses `base` 16; an `input` string is created formed from pairs of all digits from the base; the `expected` Luhn check digit for the string is calculated. There are duplicated pairs because the Luhn algorithm treats odd and even positions differently, so both should be tried for every value. Then the `pos` value considers every position in the string and `replacement` takes the value of every other possible digit at that position. The `corrupt` value is then the original input with the digit at `pos` replaced by `replacement`. This corrupt string ought to produce a different Luhn check digit to the `expected` value from the original input, which is verified by `checkFail`.

So having verified the correct detection of all single digit errors, we have to check for the cases that are *not detected* by the algorithm. This happens in two special cases, firstly for swapped pairs of digits in the input, where one digit is zero and the other is base-1 (e.g. 'F' in hexadcimal or '9' in base-10). The trace to prove this is as follows:

```
AllAdjacentTranspositions:
  let base = 16 in
  let a, b in set {0, ..., base-1} be st a <> b in
  let pass = (a = 0 and b = base-1) or (a = base-1 and b = 0) in
  let expected = LUHN`luhn([a, b], base) in
    checkResult([b, a], expected, base, pass);
```

This trace produces all `[a, b]` pairs for the given base, where the two values are different. The expected Luhn check value is calculated for the pair and then the result is checked for the swap of the two values, `[b, a]`. The `pass` value is true for the special cases where the algorithm explicitly should *not* detect the swap - the `checkResult` operation uses the pass argument to call `checkOK` or `checkFail`.

The second special case that is not detected by the Luhn algorithm is with the exchange of some pairs of digits. Specifically it will not detect 22 < - > 55, 33 < - > 66 or 44 < - > 77. The trace to verify this is as follows:

```
AllTwinErrors:
  let base = 10 in
  let a, b in set {0, ..., base-1} be st a <> b in
  let pass = {a, b} in set {{2, 5}, {3, 6}, {4, 7}} in
  let expected = LUHN`luhn([a, a], base) in
    checkResult([b, b], expected, base, pass);
```

The trace generates all possible pairs of values for the base, and then creates a pair from one value and checks that a substitution with a pair of the other is detected as a change. Note that the `pass` value is true for the cases that should not be detected.

The final trace to test the algorithm checks that a prefix of an arbitrary number of zero characters does not affect the check digit value:

```
ZeroPadding:
  let base = 16 in
  let input = [x | x in set {0, ..., base-1}] in
  let expected = LUHN`luhn(input, base) in
  let number in set {1, ..., 8} in
  let padding = [p-p | p in set {1, ..., number}] in
    checkOK(padding ^ input, expected, base);
```

The call to `checkOK` verifies that the check digit produced from the original input is the same as that produced for the same with a varying prefix of between 1 and 8 zeros.

4.1.4 Real World Luhn, the ACME Example

The sections above describe a formal specification of the standard Luhn algorithm, and traces that can be used to verify that the specification is correct, compared to the published algorithm.

It is always useful to have a formal specification of an algorithm, but in the real world, implementations are often created by programmers (with the best of intentions) on a rather less formal basis. In some cases, their implementations are not correct, though they may still pass the simple ad-hoc levels of testing that are usually performed.

The authors are familiar with a particular case of an invalid Luhn implementation written by the ACME[3] Company. ACME is in the business of tracking the movement of a large number of parcels around the country. Each parcel is marked with a (hexadecimal) barcode, and these have a base-16 Luhn check digit. ACME employees use barcode scanners to check parcels in and out of depots and delivery vans, and the software in the scanners checks the Luhn check digit on each one. The same Luhn software is used to create the barcodes for new parcels.

Unfortunately, the Luhn implementation used by ACME had a fault. The software was originally written in Visual Basic for base-10 and subsequently extended to cover base-16. But internally, a VB library to convert decimal characters to numbers was silently ignoring hexadecimal characaters. This meant that although the barcodes produced by the software could be checked by the scanners, the check digits produced were not the standard check digits.

This went unnnoticed for some considerable time. But eventually the software used to produce the barcodes had to be updated as part of a planned system enhancement, leaving the barcode scanners unchanged. The informal specifications of the system stated that it used the Luhn base-16 algorithm, and that was duly re-implemented (correctly) in the upgrade.

Of course, when the new barcodes were checked with the old scanners, the check digits *sometimes* failed - not always, because the invalid algorithm was only incorrectly handling hexadecimal characters, but it was still a serious problem.

This caused a great deal of confusion about which implementation was correct! It led to the creation of the formal model above and this was used to demonstrate that the ACME implementation was at fault.

But what was the invalid algorithm? And what were its error detection characteristics? This had to be discovered by careful reverse engineering of the ACME Visual Basic code, and was captured in an ACME variant of the formal model, which we present in this section.

The ACME Total Function

The error in the ACME implementation was tracked back to the `total` function that is used to multiply and add the digits of the input. First we show the standard algorithm for the core of the `total` function:

[3]Not their real name!

```
-- The standard algorithm:
let multipler = (len data) mod 2 + 1,
    code = hd data,
    product = code * multipler
in
    sumDigits(product, base)
```

The `sumDigits` function adds the digits of its argument in the given base (e.g. the sum of 123 in base 10 is 1+2+3=6). Here is the equivalent in the ACME implementation:

```
-- The ACME algorithm:
let multipler = (len data) mod 2 + 1,
    code = hd data,
    product = code * multipler
in
    if hasNonDecimals(product, base)
    then code
    else sumDigits(product, base)
```

Notice that the ACME algorithm has an extra test for `hasNonDecimals`. This function is true if there are any A-F characters in the product. If there are none, then the result of the ACME `total` function will be exactly the same as the standard; but if there are non-decimal characters, the function returns the `code` value - i.e. the unmultiplied and unsummed input value. Obviously this is wrong.

Using this new specification, we could show that it seemed to produce ACME check digits that agree with the old ACME software, both with and without hexadecimal characters. But how could we test this specification further with traces, if we were not sure of the error detection characteristics of the ACME algorithm?

Clearly, without a complete analysis of the ACME algorithm we could not *predict* its error detection characteristics. But by using the traces for the standard algorithm, we could easily *discover* whether the ACME version was detecting the same, or more, or fewer error cases. The results were as follows:

FirstN and FirstN_16: These traces still operate correctly under the ACME variant because they are testing for violation of constraints in the specification rather than checking individual check digit values. This tells us that the ACME algorithm does produce something that meets all of the pre- and postconditions defined.

AllOneDigitErrors: This trace produces no errors with the standard algorithm, because it is capable of detecting all single digit errors. However, the ACME variant produces six errors. On closer examination, we find that it cannot dectect substitutions of 3 < - > 6, 5 < - > A, 7 < - > B.

AllAdjacentTranspositions: This trace has two exceptions that are not detected by the standard algorithm, 0 < - > F and F < - > 0. The ACME variant has these two exceptions as well, but in addition it has 40 others.

AllTwinErrors and ZeroPadding: These traces behave in the same way as the standard algorithm.

So, with the help of a formal model and combinatorial testing, we could show the ACME company not only that their implementation was at fault, but that the invalid implementation was significantly weaker at error detection than the standard algorithm (and we could characterize its weaknesses).

4.2 The Basket Service Model

4.2.1 Background

The Basket Service model describes the behaviour of a simple client-server system that manages a basket of retail items. Initially a basket is empty, and can have multiple items added and cancelled (removed), before finally the entire basket is settled (the customer pays for the items) or cancelled.

4.2.2 The Structure of the Specification

The Basket Service is modelled by a single class, called BasketService, which includes the current basket in its state:

```
types
public Product = seq1 of char;

private BasketItem ::
  iid       : nat                    -- Unique per basket
  product   : Product
  amount    : real;

private Basket ::
  items : [seq of BasketItem]
inv
  basket ==
    basket.items <> nil =>
      let ids = { item.iid | item in seq basket.items } in
        card ids = len basket.items;

instance variables
  basket : Basket := mk_Basket(nil);
```

The client-server messages are modelled by record types for each transaction type, either adding or cancelling an item from the bsaket, or cancelling or settling the entire basket. For example:

```
public AddItemRequest ::
  uid       : nat
  product   : Product
  amount    : real;

public CancelItemRequest ::
  uid       : nat
  iid       : nat;

public CancelBasketRequest ::
  uid       : nat;

public SettleBasketRequest ::
  uid       : nat;

public Response ::
  type      : <OK> | <FAILED> | <SEQERR>
  sid       : nat
  message   : [seq of char]
  iid       : [nat]
  total     : real;
```

In addition to the simple values that populate the basket, the requests carry a `uid` (a *user* message ID), and responses include a `sid` (a *server* message ID). The protocol requires that both user IDs and server IDs are incremented for every message sent or received. Responses also include the total amount of the basket so far (which may be zero, if it is empty).

The processing of each message type is specified as a *pair* of operations, like `addItem` and `addItemImpl`. The first is the public interface which checks the arguments passed and the message sequencing before calling the second operation; the second is private, assumes the arguments are correct, and performs the core processing of the request. The postcondition on the first operation defines the overall behaviour *in detail*; the precondition on the second operation verifies that the arguments are valid. For example, the `cancelItem` operations are as follows:

```
public cancelItem: CancelItemRequest ==> Response
cancelItem(request) ==
(
  if lastuid <> nil and request.uid <> lastuid + 1
  then return seqerrResponse("Invalid sequence")
  else
  (
    lastuid := request.uid;

    if basket.items = nil
    then return errorResponse("Basket is empty", nil)
    else if request.iid not in set basketItems(basket)
         then return errorResponse("No such item", request.iid)
         else return cancelItemImpl(request)
  )
)
post
  cases RESULT.type:
    <SEQERR> ->
      basket = basket~  -- sid changes, nothing else
      and lastuid = lastuid~
      and iid = iid~
      and sid = sid~ + 1,

    <FAILED> ->
      basket = basket~  -- lastuid and sid change
      and lastuid = request.uid
      and iid = iid~
      and sid = sid~ + 1,

    <OK> ->
      lastuid = request.uid
      and iid = iid~
      and sid = sid~ + 1
      and len basket.items = len basket~.items - 1
      and RESULT.total = basketTotal(basket.items)
      and RESULT.iid not in set basketItems(basket)
  end;

private cancelItemImpl: CancelItemRequest ==> Response
cancelItemImpl(request) ==
(
  basket.items := [ item | item in seq basket.items & item.iid <> request.iid ];
  return okResponse(request.iid)
)
pre lastuid = request.uid
  and basket.items <> nil
  and request.iid in set basketItems(basket);
```

Notice how the main public operation performs basic checks and then delegates to the private "Impl" operation, which actually performs the update. The postcondition on main operation has a detailed postcondition that considers each possible result type and states the changes in the system state and message counters that result. You can see that if the result is `<OK>`, then the basket is edited to remove a single item and return the new basket total.

From a combinatorial testing perspective, this style of specification allows us to make lots of experimental calls to the public interface operations, in the confidence that the constraints will very precisely check that the operations are modifying the state of the server correctly. Although note that the server only checks the sequence of the user message IDs; it is still up to the "caller" to check that the server message IDs returned are in sequence.

4.2.3 Testing Approach

The obvious way to create a simple test of this model is to write a sequence of public interface calls, and check that the return response from each is as expected. For example:

```
  public testAdHoc: () ==> seq of bool
  testAdHoc() == let s = new BasketService() in return
  [
    s.cancelItem(mk_BasketService`CancelItemRequest(0, 99))
      = mk_BasketService`Response(<FAILED>, 1, "Basket is empty", nil, 0),
    s.cancelBasket(mk_BasketService`CancelBasketRequest(1))
      = mk_BasketService`Response(<FAILED>, 2, "Basket is empty", nil, 0),
    s.addItem(mk_BasketService`AddItemRequest(2, "apples", -99))
      = mk_BasketService`Response(<FAILED>, 3, "Amount invalid", nil, 0),
    s.addItem(mk_BasketService`AddItemRequest(3, "apples", 1.23))
      = mk_BasketService`Response(<OK>,     4, nil, 1, 1.23),
    s.addItem(mk_BasketService`AddItemRequest(99, "pears", 2.34))
      = mk_BasketService`Response(<SEQERR>, 5, "Invalid sequence", nil, 1.23),
    s.addItem(mk_BasketService`AddItemRequest(4, "pears", 2.34))
      = mk_BasketService`Response(<OK>,     6, nil, 2, 3.57),
    s.cancelItem(mk_BasketService`CancelItemRequest(5, 1))
      = mk_BasketService`Response(<OK>,     7, nil, 1, 2.34),
    s.cancelItem(mk_BasketService`CancelItemRequest(6, 99))
      = mk_BasketService`Response(<FAILED>, 8, "No such item", 99, 2.34),
    s.settleBasket(mk_BasketService`SettleBasketRequest(7))
      = mk_BasketService`Response(<OK>,     9, nil, nil, 2.34)
  ]
  post elems RESULT = {true};
```

This test operation returns a sequence of booleans, each one made from an assertion that a call to a particular public operation will return a particular Response message. The postcondition effectively asserts that the whole test passes.

As discussed above, this type of ad-hoc testing is perfectly valid, but there are very many paths through the server behaviour that are not exercised by such tests. Here, combinatorial traces can give us a great deal more testing power. The Basket Service model lends itself to the *process* style of combinatorial testing, discussed in 3.1. But we also have to consider creating supporting test operations that maintain the "client" state of the system in order to make sensible sequences of interface calls on the server with a trace. See 3.2.

To create test operations, we need to be able to track the server message ID (to check that returned values are continuous) as well as creating a contiguous client message ID for new requests. We also need to be able to track the item IDs returned when new items are added to the basket, so that the cancelItem operation can select items that are in the basket. Lastly, it would be useful to be able to create errors in the IDs, so that we can check this behaviour as well. This leads to a `Tests` class with state and operations as follows:

```
class Tests

-- State for the combinatorial tests.
instance variables
  SID : nat              := 0;
  UID : nat              := 0;
  iids : set of nat      := {};
  server : BasketService := new BasketService();

operations
  private addItemTest: bool * BasketService`Product * real ==> BasketService`Response
  addItemTest(ok, product, amount) ==
  (
    if ok then  UID := UID + 1;
    let response = server.addItem(mk_BasketService`AddItemRequest(UID, product, amount)) in
    (
      if response.type = <OK> then iids := iids union {response.iid};
      checkSID(response.sid);
      return response
    )
  );

  private checkSID: nat ==> ()
  checkSID(s) == SID := s
  pre s = SID + 1;
```

So the `addItemTest` operation (conditionally) increments the client UID value and uses this to build a Request. If the result is `<OK>`, the IID returned is added to the set. The server SID is checked and updated, and the response is returned so that the tests from the trace show the result details.

Similar test operations are created for the other server interface operations, which similarly use the state data to prepare their requests. For example, the `cancelItemTest` operation includes a line to select an arbitrary IID to cancel, or return nil if there are no items in the basket (i.e. skip the operation):

```
  if iids <> {} then
    let iid in set iids in
      ... -- cancel item iid
  else return nil
```

Given that all of the server interface operations check their actions in detail, it is tempting to create a combinatorial test that just tries all of the operations in every possible order, including with a *false* argument to cause error cases:

```
traces
  TryEverything: ||
  (
    addItemTest(true, "apples", 1.23),
    addItemTest(false, "pears", 4.56),
    cancelItemTest(true),
    cancelItemTest(false),
    cancelBasketTest(true),
    cancelBasketTest(false),
    settleBasketTest(true),
    settleBasketTest(false)
  );
```

This approach does have some value. It produces 8! tests (40,320), like this, selecting test number 1234 to illustrate:

```
> runtrace TryEverything 1234
```

```
Generated 40320 tests in 0.001 secs.
Test 1234 = addItemTest(true, "apples", 1.23);
            cancelItemTest(true);
            settleBasketTest(true);
            cancelItemTest(false);
            cancelBasketTest(true);
            cancelBasketTest(false);
            settleBasketTest(false);
            addItemTest(false, "pears", 4.56)
Result = [mk_Response(<OK>, 1, nil, 1, 1.23),
          mk_Response(<OK>, 2, nil, 1, 0),
          mk_Response(<OK>, 3, nil, nil, 0),
          nil,
          mk_Response(<FAILED>, 4, "Basket is empty", nil, 0),
          mk_Response(<SEQERR>, 5, "Invalid sequence", nil, 0),
          mk_Response(<SEQERR>, 6, "Invalid sequence", nil, 0),
          mk_Response(<SEQERR>, 7, "Invalid sequence", nil, 0), PASSED]
Excluded 40319 tests
Executed in 0.024 secs.
All tests passed
```

Although this permutation approach might show up some corner cases, most of the permutations do not represent a realistic call sequence of a client (like this example!). At most a permutation will add one item to the basket and only settle one basket. To be more realistic, we need more basket activity before it is either settled or cancelled. That could be achieved by adding more `addItemTest` or `settleBasketTest` calls to the trace, but even if we only add two, that takes the total number of permutation up to 10! (3,628,800), which will take a very long time to execute.

A more considered approach yields a larger number of realistic test sequences. Here, we build a trace using a typical sequence of actions from a client, some of which are alternated or optional (to cover varying possibilities). At the end, the whole sequence is duplicated ($\{2\}$) to produce tests that complete one basket and then another:

```
TwoBaskets:
  (
    addItemTest(true, "apples", 1.23) |
    addItemTest(false, "apples", 1.23);
    ( cancelItemTest(true) |
      cancelItemTest(false) )?;

    addItemTest(true, "pears", 2.34) |
    addItemTest(false, "pears", 2.34);
    ( cancelItemTest(true) |
      cancelItemTest(false) )?;

    ( cancelBasketTest(false) |
      settleBasketTest(false) )?;

    cancelBasketTest(true) |
    settleBasketTest(true)
  ) {2}
```

This trace means that we always start by adding, or failing to add an item; then we either cancel or fail to cancel it or we skip this step; then we add or fail to add a second item; then we cancel or fail to cancel or we skip; then we fail to cancel the basket or fail to settle the basket; and lastly we successfully cancel or settle the basket; and overall we repeat all *combinations* of these sequences twice.

This produces far more realistic test sequences and expands to 46,656 tests. Here is one selected for illustration:

```
> runtrace TwoBaskets 4321
Generated 46656 tests in 0.001 secs.
Test 4321 = addItemTest(true, "apples", 1.23);
            skip;
            addItemTest(true, "pears", 2.34);
            skip;
            skip;
            cancelBasketTest(true);
            addItemTest(true, "apples", 1.23);
            cancelItemTest(true);
            addItemTest(false, "pears", 2.34);
            cancelItemTest(true);
            skip;
            cancelBasketTest(true)
Result = [mk_Response(<OK>, 1, nil, 1, 1.23),
         (),
         mk_Response(<OK>, 2, nil, 2, 3.57),
         (),
         (),
         mk_Response(<OK>, 3, nil, nil, 0),
         mk_Response(<OK>, 4, nil, 1, 1.23),
         mk_Response(<OK>, 5, nil, 1, 0),
         mk_Response(<SEQERR>, 6, "Invalid sequence", nil, 0),
         nil,
         (),
         mk_Response(<SEQERR>, 7, "Invalid sequence", nil, 0), PASSED]
Excluded 46655 tests Executed in 1.486 secs.
All tests passed
```

You can see that this is a more rational client call sequence (though only just!). So by composing a trace that is focussed on more realistic behaviour, we have managed to produce a comparable number of tests to the full permutation approach (of the order of 40K), but each one should be adding more value than a random permutation of interface calls. Of course, both traces can be used in the full test suite.

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, ‘)’ ;

Appendix B

Overture Screenshots

Appendix C

Example Model Listings

The VDM models listed in this appendix are described in detail in Chapter 4

C.1 The LUHN Check Digit Model

C.2 The Basket Service Model

References

[Bicarregui&94] Juan Bicarregui and John Fitzgerald and Peter Lindsay and Richard Moore and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. *FACIT*, Springer-Verlag, 1994. 245 pages. ISBN 3-540-19813-X.

This book is a tutorial on the process of formal reasoning in VDM. It discusses how to go about about building proofs and provides the most complete set of proof rules for VDM-SL to date.

[Clarke&99] E. Clarke and O. Grumberg and D. Peled. *Model Checking*. The MIT Press, 1999.

[Larsen&10] Peter Gorm Larsen and Kenneth Lausdahl and Nick Battle. Combinatorial Testing for VDM. In *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods*, pages 278–285, IEEE Computer Society, Washington, DC, USA, September 2010. ISBN 978-0-7695-4153-2.

[Nie&11] Nie, Changhai and Leung, Hareton. A Survey of Combinatorial Testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, February 2011.

[Paulson97] Lawrence C. Paulson. Generic Automatic Proof Tools. In Robert Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, pages 23–47, MIT Press, Cambridge, MA, USA, 1997.