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 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

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, ')' ;

Chapter 3

Combinatorial Testing Patterns

Chapter 4

Combinatorial Testing Examples