Advanced Test Vector Generation from Rosetta

Masters thesis defense

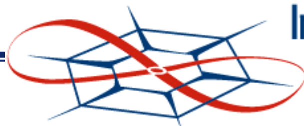
by

Srinivas Akkipeddi

June 27th, 2001

Thesis committee

Dr. David Andrews   Dr. Perry Alexander (Advisor & Chair)   Dr. Jerry James



Overview

- Introduction
- Problem Statement & Proposed Solution
- DVTG Overview
- Generation of Test Scenarios
- Test Requirements
- Test Vector Generation
- Example
- Evaluation
- Related Work
- Summary and Future Work

Introduction

- Growing complexity of systems
  - Abstraction techniques to manage complexity
  - Use of declarative specifications
  - Disadvantage with higher levels of abstraction is "the need for validation"
- Validation techniques
  - Implementation-based testing techniques focus on actual functionality, overlook intended behavior
  - Specification-based testing techniques
    - Help validate implementation with respect to specification
    - Expose any ambiguities or inconsistencies in the specifications
    - Test cases can be designed as soon as the specifications are complete

Problem Statement

- Automate selection of test data from requirements specifications
- Address the problem of generating specific input test values from test requirements
- Translate the generated test cases to a format specific to some testing software

Proposed Solution

- Approach based on a methodology proposed by Richardson for selecting test data from requirements specification

- Test Generation process
  - Generation of test scenarios from input specifications
    - Using the multi-condition strategy proposed by Myers
  - Generation of abstract test vectors from generated scenarios
    - Instantiating inputs in test scenarios with the corresponding values from test requirements
  - Translating abstract test vectors into concrete test vectors format specific to some test software or environment

Rosetta

- A Systems Level Design Language used to design systems at higher levels of abstraction
- Intended to provide specification support suitable for representing
  - Multiple, Heterogeneous perspectives of a single component
  - Systems built from heterogeneous components
- Supports multiple design domains
  - state_based, continuous time, discrete time, logic, finite state, infinite state.
- Basic unit of specification is a facet used to represent a model or component and provides information specific to a domain of interest
- Definition of facets can be done by either defining model properties or by combining previously defined facets

```
facet schmidt_trigger(input_voltage:: in real;
                      output_value:: out bit) is
  b :: bit;     /* State variable */
begin state_based
  /* First pre-condition */
  pre1: (input_voltage > 0.0) and
        (input_voltage < 5.0);
  /* First post-condition */
  post1: if (input_voltage < 1.0)
             then (b' = 0)
             else ( if (input_voltage > 4.0)
                        then (b' = 1)
                        else (b' = b)
                    endif )
         endif;
  post2: (output_value' = b');
end schmidt_trigger;
```

Overview of the entire system

Rosetta Specification

Rosetta Parser

Object Model

Test Scenario Generator

Test Scenarios

Test Vector Generator

WAVES Test Vectors

Test Vector Transformer

Abstract Test Vectors

Test Requirements

Implement in VHDL

VHDL Implementation

Validation of Implementation

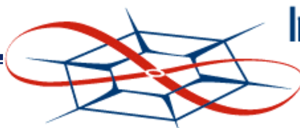
Correct Implementation of Requirements

Testing of Implementation

Correctness of Implementation

Legend:

- System Inputs
- DVTG Tools
- Generated outputs



Test Scenario

- A test scenario is a set of Boolean conditions that provide constraints on values of input and output parameters
  - The input criterion is a constraint on input parameters
  - The acceptance criterion is a constraint on output parameters
- The test scenarios provide classes of tests to be performed on the system
- Use of the multi-condition strategy proposed by Myers
  - enough test scenarios should be generated for the expression under test to take all possible values

Terminology

- Driving values
  - describe input parameters
  - signify that input parameters "drive" system inputs with these values
- Driven values
  - describe output parameters
  - signify that output parameters are "driven" by the system inputs
  - values to observe; specify expected outputs corresponding to driving values
- Controllable predicate
  - consists of driving values
  - values of variables that make up the predicate can be controlled
- Non-controllable predicate
  - consists of driven values
  - values of the variables in the predicate cannot be controlled

Test scenario Generation

- Algorithm
  - All the test cases for an operator are listed according to the truth-table definition of operator
  - Cases for which expression evaluates to true are selected
  - Notion of driving or driven values is then used to filter out redundant test cases
  - No scenarios are generated if the expression is a pre-condition
- Every expression in Rosetta facet is a boolean expression
  - Generate different values for predicates in expression by evaluating expression to true
  - If predicates are complex then rules are applied recursively

Logical Operator (OR)

P(x)	Q(y)	P(x) or Q(y)
0	0	0
0	1	1
1	0	1
1	1	1

- Possible test scenarios when P(x) or Q(y) is evaluated to true are

  P(x) = false AND Q(y) = true
  P(x) = true AND Q(y) = false
  P(x) = true AND Q(y) = true

- Disjunction P(x) or Q(y) is true if P(x) is true regardless of value of Q(y)
- P(x) controllable & Q(y) non-controllable, consider only scenarios where P(x) is false
- P(x) non-controllable & Q(y) controllable, consider only scenarios where Q(x) is false
- P(x) non-controllable & Q(y) non-controllable, need to consider all scenarios
- P(x) controllable & Q(y) controllable => need to consider all scenarios if it is not a pre-condition

- Possible test scenario when P(x) or Q(y) is evaluated to false is

  P(x) = false AND Q(y) = false

Example-Test scenario generation

Input Rosetta expression

(a AND b) OR d′

a,b,c :: boolean

- Evaluating this expression to true, generated test scenarios are

  (a AND b) = false AND (d' = true)

- The expression evaluates to true when (a AND b) is false irrespective of value of d'.

- Evaluating expression (a AND b) to false gives the following scenarios

  (a = true) AND (b = false)

  (a = false) AND (b = true)

  (a = false) AND (b = false)

Generated test scenarios

Scenario1: (a = true) AND (b = false) AND (d′ = true)
Scenario2: (a = false) AND (b = true) AND (d′ = true)
Scenario3: (a = false) AND (b = false) AND (d′ = true)

Logical Operators

- Logical operators

Operator	Rosetta expression	Generated test scenarios
and	(x=5) and y′	(x = 5) AND (y′ = true)
or	(x′=0) or (y′=1)	(x′ = 1) AND (y′ = 1) (x′ = 0) AND (y′ = 0) (x′ = 0) AND (y′ = 1)
not	not(x<4.5)	(x >= 4.5)
implies	(x′ implies y′)	(x′ = false) AND (y′ = false) (x′ = false) AND (y′ = true) (x′ = true) AND (y′ = true)
if-then-else	if x=1 then y′=x else z′=x	(x = 1) AND (y′ = x) (x = 0) AND (z′ = x)
nand	(x′>10) nand (y=2.05)	(x′ =< 10) AND (y = 2.05)
nor	(x=<3) nor y′	(x > 3) AND (y′ = false)
xor	x′ xor y′	(x′ = false) AND (y′ = true) (x′ = true) AND (y′ = false)
xnor	x′ xnor y	(x′ = false) AND (y = false) (x′ = true) AND (y = true)

Relational Expressions

- Impossible to obtain all the values of operands for which expression is evaluated to true or false
- For example, consider the expression (x > 20)
  - The expression divides the range of x into two categories. One greater than 20 and the other less than or equal to 20
  - If the expression is evaluated to true then the class of values greater than 20 come under the satisfy class

Operator	Expression	Conditions to evaluate exp.	
		To true	To false
<	**x < y**	**x < y**	**x >= y**
=<	x =< y	x =< y	x > y
>	x > y	x > y	x =< y
>=	x >= y	x >= y	x < y
=	x = y	x = y	x /= y
/=	x /= y	x /= y	x = y

Test Requirements

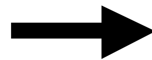
- Indicate coverage desired for generation of specific input values and determine number of test cases to be generated
- Requirements are specified for input parameters
  - Based on the most frequently used range of values, confidence, time constraints, acceptable risk factors, etc

```
package testrequirements is
begin logic
  test_req( v::label; lbound::number; ubound::number; steps::number) :: bunch(number) is
            sel( x::number | (lbound =< x) and (x =< ubound) and
                  exists( n::natural | x = (n*steps + lbound )));

 test_init(num::integer;vector::univ)::univ;

 init(num::integer;vector::univ)::univ;

 export all;
end test_logic;
```

Generic Requirements

- Uses the function "test_req", that takes a variable, lower bound, upper bound and step size
  - All numbers within the specified lower and upper bound are selected such that they vary by step size
  - Requirements can be specified for input parameters or for the property of input parameters

```
test_req(A,1,3,1);
test_req(B,4,6,1);
```



```
(A = 1)  AND  (B = 4);
(A = 1)  AND  (B = 5);
(A = 1)  AND  (B = 6);
(A = 2)  AND  (B = 4);
(A = 2)  AND  (B = 5);
(A = 2)  AND  (B = 6);
(A = 3)  AND  (B = 4);
(A = 3)  AND  (B = 5);
(A = 3)  AND  (B = 6);
```

Initial Vectors

- Specified using the functions "test_init" and "init", that take a sequence number and the vector as arguments
- Sequence number provides information about the order in which vectors are used for the system to reach a particular state
- Vectors obtained from the "init" function are used to evaluate scenarios only once before all the other test cases
- Vectors obtained from "test_init" are used to evaluate scenarios before every test case obtained from generic requirement "test_req"

```
req1: init(1,(A=1) and (B=0));
req2: test_init(1,(A =1) and (B=1));
req3: test_req(A,1,2,1);
req4: test_req(B,4,5,1);
```

```
A=1 and B=0 (vector from init function)
A=1 and B=1 (vector from test_init function)
A=1 and B=4
A=1 and B=1 (vector from test_init function)
A=1 and B=5
A=1 and B=1 (vector from test_init function)
A=2 and B=4
A=1 and B=1 (vector from test_init function)
A=2 and B=5
```

Test Requirements (cont.)

- Generic requirements are specified for property of the input
  - User must provide a mechanism to generate actual values of input given the value of its property in a Rosetta facet
  - Rosetta facet is translated to a matlab function
  - Makes use of predefined or user-defined matlab functions that are declared in a Rosetta package “matlabpackage”

Abstract Test Vectors

- Actual test cases generated from the specification. They contain
  - Values of input parameters
  - Expected values for output parameters corresponding to input parameters
- Generated by combining test scenarios and test requirements
- Use of Boundary testing strategy proposed by Myers
  - Boundary value is one that is directly on, above or below the limit of the range of values specified in a condition
  - For example in the condition (x > 10)
    - three equivalence classes (x < 10) (x= 10) (x > 10)
    - boundary values (x = 9) (x = 10) (x=11)

Abstract Test Vectors (cont..)

- The vectors are obtained by instantiating input parameters with the corresponding values obtained from test requirements
- Generic format, not specific to any testing software
- Abstract test vectors are translated into concrete test vectors for traditional simulation systems
- In this case they are translated into WAVES test vectors for VHDL simulation

WAVES format was developed to support users in the exchange of waveform information between different simulator and tester environments. Stimulus waveforms that are produced by the WAVES data set are applied to VHDL model, the model produces its output response based on stimulus presented.

Example : Schmidt Trigger

```
facet schmidt_trigger(input_voltage:: in real; output_value:: out bit)  is
   b :: bit;
begin state_based
   l1: (input_voltage > 0.0) and (input_voltage < 5.0);
   l2: if (input_voltage < 1.0) then (b' =0)
          else ( if (input_voltage > 4.0)  then (b' = 1) else (b' = b)  endif )
       endif ;
  l3: (output_value' = b');
end schmidt_trigger;
```

```
FACET schmidt_trigger_TEST( input_voltage::in real; output_value:: out bit)  IS
   b :: bit;
BEGIN state_based
   INPUT_0    :  ( input_voltage > 0.0 ) AND ( input_voltage < 5.0 );
   ACCEPT_0:  ( input_voltage < 1.0 ) AND ( b' = 0 ) ;
   ACCEPT_1:  ( input_voltage > 4.0 ) AND ( b' = 1 ) ;
   ACCEPT_2:  ( input_voltage >= 1.0 ) AND ( input_voltage =< 4.0 ) AND ( b' = b );
   ACCEPT_3:  ( output_value' = b' );

END schmidt_trigger_TEST;
```

Example : Schmidt Trigger (cont..)

Test Requirement - ***test_req(input_voltage,0.1,4.9,0.5);***

Generated Vectors

```
FACET schmidt_trigger_TEST_VECTORS(input_voltage:: in real;
                                   output_value:: out bit) IS
  b :: bit;
BEGIN state_based

  ACCEPT1:  ( input_voltage = 0.1 ) AND ( b' = 0 ) AND ( output_value' = 0 );
  ACCEPT2:  ( input_voltage = 0.6 ) AND ( b' = 0 ) AND ( output_value' = 0 );
  ACCEPT3:  ( input_voltage = 0.9 ) AND ( b' = 0 ) AND ( output_value' = 0 );
  ACCEPT4:  ( input_voltage = 4.1 ) AND ( b' = 1 ) AND ( output_value' = 1 );
  ACCEPT5:  ( input_voltage = 4.6 ) AND ( b' = 1 ) AND ( output_value' = 1 );
  ACCEPT6:  ( input_voltage = 1.0 ) AND ( b' = 1 ) AND ( output_value' = 1 );
  ACCEPT7:  ( input_voltage = 1.1 ) AND ( b' = 1 ) AND ( output_value' = 1 );
  ACCEPT8:  ( input_voltage = 1.6 ) AND ( b' = 1 ) AND ( output_value' = 1 );
  ACCEPT9:  ( input_voltage = 2.1 ) AND ( b' = 1 ) AND ( output_value' = 1 );
  ACCEPT10: ( input_voltage = 2.6 ) AND ( b' = 1 ) AND ( output_value' = 1 );
  ACCEPT11: ( input_voltage = 3.1 ) AND ( b' = 1 ) AND ( output_value' = 1 );
  ACCEPT12: ( input_voltage = 3.6 ) AND ( b' = 1 ) AND ( output_value' = 1 );
  ACCEPT13: ( input_voltage = 3.9 ) AND ( b' = 1 ) AND ( output_value' = 1 );
  ACCEPT14: ( input_voltage = 4.0 ) AND ( b' = 1 ) AND ( output_value' = 1 );

END schmidt_trigger_TEST_VECTORS;
```

Comment character

List of input and output parameters

```
% input_voltage    output_value
      0.1               0
      0.6               0
      0.9               0
      4.1               1
      4.6               1
      1.0               1
      1.1               1
      1.6               1
      2.1               1
      2.6               1
      3.1               1
      3.6               1
      3.9               1
      4.0               1
```

Evaluation

- Alarm clock system
  - It allows test vector generation for typical register transfer level specification
  - Tests If-Then-Else, And, Implies operators
  - Tests the tool when initial vectors are provided in the test requirements
- Schmidt Trigger component
  - Covers relational and logical operators, tests for nested operators
  - Demonstrates effectiveness in representing and utilizing test requirements.
- Satellite Communication preprocessor defined by TRW
  - Tests the tool when the test requirements are provided for the property of the input signal to preprocessor (Signal-to-Noise ratio)
  - Deals with synchronization and timing issues.

Related Work

- Krishna Rangarajan– Automated test generation from Rosetta specifications
  - Similar methodology but improved test generation techniques
  - Support for *packages* and *functions* in Rosetta
  - Allow user to specify initial values for internal state variables
  - Ability to generate test vectors for input parameters from properties specified for those inputs
  - Demonstrated automatic generation of abstract and concrete test vectors
- Offutt et al – Generating tests from UML specifications
  - Similar methodology but different techniques used
- Tse et al – Test case generation for class-level object oriented testing
  - Technique for test generation is different
- Stocks and Carrington – Introduce the use of test templates (TTs) and test template frameworks (TTFs) for specification based testing
  - Test Templates similar to Test Scenarios, while TTFs provide structuring of the test space

Summary

- Specification-based testing techniques are used to augment and complement implementation-based testing
- Generation of test vectors from specification and using it against the implementation is a way to establish that implementation conforms to the specification
- Achieved the primary objective of automatically generating test vectors from Rosetta specification.
- Enables testing intended behavior as well as actual functionality
- Examples demonstrate success of the methodology in small test situations

Future Work

- Handling of broader sets of types
  - composite types - records, tuples
  - user-defined types
- Support for “structural specification”
- Generate concrete test vectors in other data formats
- Support for different custom coverage requirements
  - user can then specify, for example Gaussian distribution of test cases over a specified range.