Automatic Test Vector Generation in XML from Rosetta Specifications

by

Murthy Kakarlamudi

B.Tech., Electrical and Electronics Engineering,

J.N.T.U College of Engineering, Kakinada, India, 2000

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

______________________________.

Dr.Perrry Alexander (Chair)

______________________________.

Dr.David Andrews (Member)

______________________________.

Dr.Susan Gauch (Member)

______________________________.

Date Accepted

Contents

1 Introduction **1**

1.1 Testing 1

1.2 Motivation 2

1.3 Problem Statement 3

1.4 Proposed Solution 3

1.5 Organization of Thesis 5

2 Background **6**

2.1 Rosetta Specification Language 7

2.2 DVTG 9

2.3 WAVES: A test environment for VHDL 10

2.4 Altera MAX+PLUS II 12

2.4.1 ALTERA Project Verification 12

2.4.2 Input test format for simulating ALTERA MAXPLUS II models 13

2.5 An Introduction to XML 13

2.5.1 XML Building Blocks 14

2.5.2 DTD 16

2.5.3 XML Schema 17

2.5.4 XSLT 18

2.5.5 DOM 18

2.5.6 JAXP 19

2.5.7 Summary 20

3 Design Verification Test Generation 21

3.1 Test Scenarios 22

3.1.1 Logical Operators 24

3.1.2 Relational Operators 29

3.2 Test Requirements 31

3.2.1 General Requirements 32

3.2.2 Initial Vectors and Test Cases 32

4 Implementation Details 34

4.1 Abstract Test Vectors 34

4.2 XML Representation 34

4.2.1 Concrete Test Vectors 37

Generation of test vectors in WAVES format 39

Test format for testing ALTERA MAXPLUS II models 41

4.3 Support for Rosetta Data Types in DVTG 42

4.3.1 Bit Vectors 42

Specifying Test Requirements for BitVectors 43

4.3.2 Sequences 44

Specifying Test Requirements for Sequences 45

Implementation details of Sequences 45

5 Examples **47**

5.1 Schmidt Trigger 47

5.1.1 Functionality and Specification 47

5.1.2 Test Scenarios 48

5.1.3 Test Requirements 49

5.1.4 Output for Abstract Test Vectors in XML Format 50

5.1.5 Concrete Test Vectors in WAVES format 51

5.2 Alarm Clock 52

5.2.1 Abstract Test Vectors in XML 57

5.2.2 Concrete Test Vectors in WAVES format 57

5.3 BitVectors support in Rosetta 58

5.3.1 Specification Illustrating BitVectors 59

5.3.2 Test Scenarios 59

5.3.3 Test Requirements 60

5.3.4 Abstract Test Vectors in XML 60

5.3.5 Concrete Test Vectors in WAVES format 62

5.4 Sequences Support in Rosetta 63

5.4.1 Specication of Sequences 63

5.4.2 Test Requirements 63

5.4.3 Abstract Test Vectors in XML 64
5.4.4 Concrete Test Vectors in WAVES format 65

6 Summary and Future Work **66**

6.1 Summary 66
6.2 Conclusions 67
6.3 Future Work 68

List of Figures

1.1 Flow of generation of test vectors 4

2.1 Schmidt Trigger Specification 7

2.2 Domains and Interactions 9

2.3 WAVES-VHDL Test Bench Configuration 11

2.4 XML Representation 15

2.5 Attributes in an XML Document 16

3.1 Test Requirements Package 31

3.2 Specifying Test Requirements 32

3.3 Test Cases Generated 32

3.4 Requirements file illustrating usage of init function 33

4.1 Flow of generation of Abstract Test Vectors 35

4.2 Abstract Test Vectors DTD 35

4.3 Scenarios file for Schmidt Trigger 36

4.4 Abstract Test Vectors for Schmidt Trigger 38

4.5 Generation of Concrete Test Vectors 39

4.6 XSLT for generating WAVES format 40

4.7 Concrete Test Vectors in WAVES format 41

4.8 Input Test format for ALTERA MAXPLUS II 42

5.1 Schmidt Trigger Specification 48

5.2 Scenarios for Schmidt Trigger 49

5.3 Requirements for Schmidt Trigger Specification 50

5.4 Abstract Test Vectors for Schmidt Trigger 51

5.5 Concrete Test Vectors in WAVES format 52

5.6 System Level Specification for Alarm Clock 53

5.7 Structural Representation Of Alarm Clock 55

5.8 Alarm Clock Specification 56

5.9 Abstract Test Vectors for *mux* component 57

5.10 Concrete Test Vectors for *mux* component 58

5.11 BitVector specification 59

5.12 Scenarios for BitVectors 59

5.13 Requirements for BitVectors 60

5.14 Abstract Test Vectors for BitVectors 61

5.15 Concrete Test Vectors for BitVectors 62

5.16 Specification for Sequences 63

5.17 Requirements for Sequences 64

5.18 Abstract Test Vectors for Sequences 64

5.19 Concrete Test Vectors for Sequences 65

List of Tables

3.1 Truth Table for AND Operator 24

3.2 Truth Table for OR Operator 25

3.3 Final Test Scenarios of OR Operator 26

3.4 Truth Table for NOT Operator 26

3.5 Truth Table for If-Then-Else Operator 27

3.6 Test Conditions for Relational Operators 30

Abstract

Testing is an important phase in the development cycle of any system. Testing involves developing the test cases and running them on the product. It will be useful if the testing process is automated. An automatic test case generator automatically generates test cases. Mostly the Automatic Test case generators generate test cases in a format that is dependent on a particular language or to a particular simulation environment. This limits the flexibility of the tool. Thus if an Automatic Test Case Generator generates test cases in a language independent format then any third party tool can simply transform these language independent test cases into a format that is compatible for their simulation environment. This thesis presents an automatic test case generator DVTG [15], which automatically generates test vectors from Rosetta specifications. DVTG generates the test vectors in XML format. In the process of generating the abstract test vectors, DVTG first generates a set of scenarios from Rosetta specifications. Test scenarios represent a range of values and this range of values can be very high. So to limit the set of test cases, a user specifies test requirements that places constraints on the number of test cases that will be generated. Combining User specified requirements and the scenarios, abstract test vectors in XML will be generated. This XML format can be used to generate concrete test vectors that will be suitable for a particular simulation environment. As part of this thesis work, concrete vectors in WAVES [10] format are generated from XML test vectors. The vectors in WAVES format can be used to test the VHDL implementation of the system.

Acknowledgments

I would like to acknowledge the help of many people during this thesis work, particularly my adviser **Dr.Perry Alexander** for his immense support in providing me the subjects. He is more than a mentor to me. I had real fun working in SLDG lab. I am very thankful to all the SLDG group members Garrin Kimmell, Brandon Morel, Cindy Kong, Jesse Stanley, Justin Ward for all the support they have provided me.

I owe a lot to all my friends Sunil, Ravi, Subhash, Mahesh, Ramu, Kalicut, Pops, Matte, Sudarshan, Jetli, Setti and many more. I can't end this section without mentioning Krishna Ranganathan, my senior in SLDG lab. He has guided me in all the phases of my thesis work. Right from giving the ideas for my thesis till reviewing my thesis for corrections, he helped me a lot.

Finally this work wouldn't be complete without the support of my parents, my sister and my brother-in-law.

Chapter 1

Introduction

1.1 Testing

The importance of testing and its implications with respect to quality of the system developed is boundless [14]. Testing is a critical factor in quality assurance and represents the final review of system specification, design and code generation. Testing is very labor intensive and accounts for approximately 50% of the cost of system development [13]. Testing is the process of executing a program with the purpose of finding an error. A good test case thus, is one that has a high probability of finding an error.

Simulation is commonly used as a means of testing the system. In this technique test cases are designed after the system is developed and then the system is driven with the selected test data to observe the output. Based on the output the correctness of the system is verified. This technique tends to focus more on the implementation of the system and overlooks the actual behavior of the system. This problem can be overcome by using Specification based techniques.

As Systems become complex it will be useful if a designer can represent the system at higher levels of abstraction. Working at higher levels of abstraction gives the designer the power to design a system with incomplete information. System Level specifications provide the designer the capability to work at higher abstraction levels and allows them to focus more on the original requirements. Rosetta, a System Level Design Language is used in this thesis. The test cases are generated from the specifications even before the system is implemented and once the product is developed its implementation can be validated against the test cases previously developed.

1.2 Motivation

The main purpose of testing is to reduce risk. Reducing the risk of using a program also increases our confidence that the developed product will perform as intended. Testing is typically done by choosing the test cases and executing the implementation on the test cases to determine the correctness of the product. Typically the test cases are developed to target a particular functionality of the program and the test cases should be instantiated with values to actually observe the output data. This process is very tedious and repetitious. To effectively test a product, there should be a way to automate test data generation. An Automatic Test Case Generator is a tool that automatically generates test data. Automating test data generation has many advantages. For example if the underlying implementation changes for some reason, then the whole process of generating test cases must be repeated. But if the test data generation is automated then the developer can simply apply the automatic test data

generator and generate new test cases. This potentially reduces the cost of testing manifold.

1.3 Problem Statement

To be effective, an Automatic Test Case Generator should be able to generate the test data in a format that is independent of any language or any simulation environment. There are many automatic test data generators in industry, but most of these generate test data specific to a particular simulation environment. This thesis builds on existing test case generator, Design Verification Test Generator (DVTG). DVTG takes a Rosetta specification as input and generates test vectors in Rosetta. This limits a user unfamiliar with Rosetta from using the tool. This also forces to change the existing tool whenever we want to test a new implementation. It is more useful for the test data generator to generate test cases which are independent of any testing environment. This thesis is an effort into making DVTG generate test vectors in a language independent format.

1.4 Proposed Solution

Given the inherent flexibility and data-neutrality of Extensible Markup Language (XML) [4], applying it to test vector and test requirement representation helps simplify transformation of test vectors into other formats. XML allows a developer working with automatic test case generators to define tags that gives information about the data embedded within them. A Document Type Definition(DTD) [4] is defined as part

of this thesis for representing the abstract test vectors. This DTD provides the rules that define the elements and structure of the new mark up language. It also serves as the guideline for other developers to interface with the application. The process of generating the abstract test vectors in XML format and the subsequent transformation to WAVES and an input format for simulating ALTERA models is shown in Figure 1.1

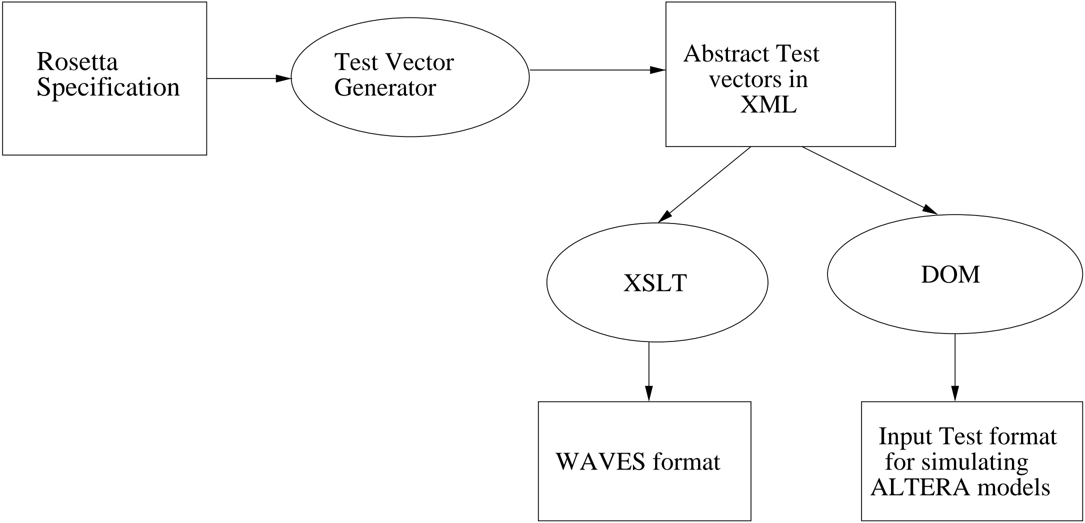

Figure 1.1: Flow of generation of test vectors

The system to be tested is specified in Rosetta. The test vector generator takes this specification and generates the abstract test vectors in XML. These abstract test vectors are further transformed into concrete test vectors. An XSLT, a transformation sheet, is developed to convert abstract test vectors in XML to WAVES, the input test data format for simulating VHDL implementations. DOM that stores the parsed XML as an Object Tree is used to transform the XML abstract test vectors into a input format for simulating models designed using ALTERA MAXPLUS II.

1.5 Organization of Thesis

Chapter 2 provides an introduction to Rosetta, a System Level Design Language (SLDL) [6]. It gives an insight into how Rosetta provides a declarative specification capability. A brief introduction to XML is also provided in this chapter. Additionally the different features of XML that were used in this thesis are discussed in this chapter. The DTD, the template that defines the markup for XML is next introduced. Furthermore this chapter gives an introduction to XSLT. Different formats for testing environments such as WAVES are also described in this chapter.

Chapter 3 introduces the DVTG tool, which automatically generates abstract test vectors in Rosetta. *Chapter 4* describes the implementation issues in this thesis. It discusses the different features of Rosetta that this implementation supports. *Chapter 5* demonstrates how the test cases for examples like Schmidt Trigger, Alarm Clock are generated in XML, and the transition to a corresponding WAVES format. *Chapter 6* summarizes this thesis work and proposes future work that may be done in this field.

Chapter 2

Background

A brief introduction to Rosetta, a System Level Design Language and how it can be used to specify complex systems in an abstract manner is provided in this chapter. An automatic test case generator, DVTG, is also briefly explained in this chapter. A brief overview of XML and the details of XSLT that can be used for transforming XML documents are provided here. Document Object Model (DOM) that is used to store a parsed XML document in a object Tree Representation is also discussed. This chapter also highlights the different formats for testing environments. WAVES is a IEEE specification for testing VHDL systems. ALTERA MAX+PLUS II provides an environment for simulating a model. The input format for ALTERA's simulator is also discussed in this chapter.

```
facet schmidt_trigger(input_voltage :: in real;
                      output_value :: out bit) is
/* local declarations */
b::bit;
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)
        end if;
end facet schmidt_trigger
```

Figure 2.1: Schmidt Trigger Specification

2.1 Rosetta Specification Language

As systems become increasingly complex the need to express them at higher levels of abstraction increases. Rosetta [7] provides an ability to represent a system at a higher level of abstraction. This language allows the designer to specify details about the system that other hardware description languages may not provide. This includes capabilities like expressing constraints on the system and crossing domain boundaries. Developed from the concepts of formal verification and functional programming worlds, it allows the designers to develop and integrate specifications from multiple design domains. The format of a Rosetta specification can be easily understood with an example. Figure 2.1 presents the specification of schmidt trigger.

Facet is the basic unit of specification in Rosetta. The *domain* defines everything from basic semantic unit of any Rosetta specification through systems and components. Each *facet* defines a particular aspect of a component or system from a

particular perspective. The *facet* keyword marks the beginning of any Rosetta specification. It is followed by an optional parameter list that identifies the input and output variables of the specification. The specification in figure 2.1 defines a facet schmidt_trigger with the interface items *input_voltage* and *output_value.* All Rosetta parameters are declared using the notation *x::T*, where x is a variable and T is a type. The scope of the parameters extends throughout the facet. Parameter declarations have an optional mode. Mode *in* indicates that the variable is an input variable and *out* indicates that the variable is an output variable.

The declaration of local variables follows the parameter declaration. In the schmidt trigger specification b is declared to be of type *bit* and is visible over the entire facet. The declarations can also be exported to other facets using the *export* keyword. Referencing the variables outside the facet requires the facet name as the qualifier. When referenced in the facet body all the parameters and variables are referenced without any decoration. For example in the above schmidt trigger specification in figure 2.1 the variable b is referred to as *schmidt_trigger.b* outside the facet.

The specification body is opened by the *begin* keyword that is followed by the specification domain. The domain extends the base definition semantics by adding new definitions specific to a design domain. The above example uses *state_based* domain that provides the basic semantics of state. Some of the existing domains in Rosetta are *state_based, logic, finite state* etc. Figure 2.2 shows the different domains defined for constraint and requirement modeling.

Rosetta is a descriptive specification language. All the expressions in the specification are boolean expressions. The *domain* is followed by a set of terms. *Terms*

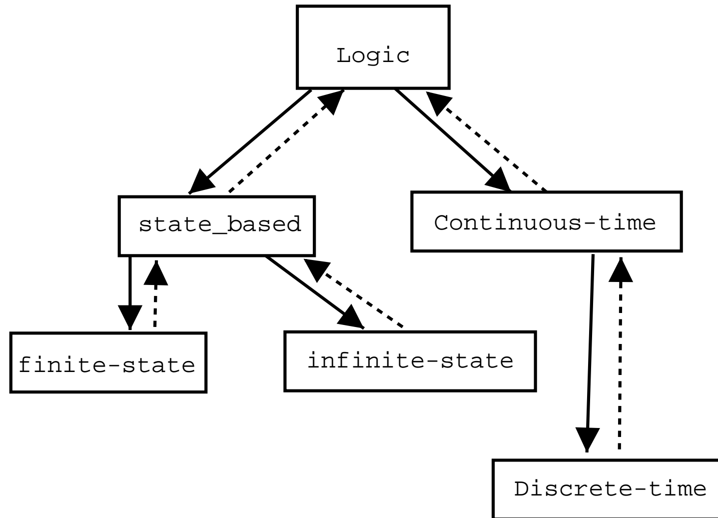

Figure 2.2: Domains and Interactions

define the behavior modeled by the facet. A term is represented using the notation *l:expression* where *l* is the label associated with the expression. Labels have no semantic meaning of their own, but can be used to represent the expression specified in the corresponding term. They are used to refer the term in other definitions as well as when the term is exported. All specifications end with a *end* clause.

2.2 DVTG

DVTG generates test vectors from Rosetta specifications. The task of generating the test vectors is divided into three phases - generation of test scenarios, generating the abstract test vectors from the scenarios and the user specified requirements and translating the abstract test vectors into concrete test vectors.

The Rosetta parser transforms the Rosetta specification and builds a Rosetta Object Model, ROM [12]. ROM is the Object Tree representation for Rosetta. It contains the java classes for all the Rosetta constructs. All the information that is

specified in the Rosetta specification is stored in ROM. Any tool can simply traverse ROM and access all the information that is given in a Rosetta specification.

DVTG traverses ROM and generates the test scenarios. A test scenario is a set of boolean conditions that are constraints on the values of the input and output parameters. There are many strategies that can be used to generate the test scenarios. DVTG uses multi-condition strategy to generate the test cases for logical expressions. Boundary testing is used for relational expressions.

Abstract test vectors are generated from test scenarios and the user defined requirements. The test requirements specify the range of values for the control variables and limits the number of test cases that can be generated. The abstract test cases are generated in the Rosetta format. As a part of this thesis the abstract test cases will be generated in XML format. These abstract test vectors are converted to concrete test vectors in WAVES [10] format.

2.3 WAVES: A test environment for VHDL

The Waveform and Vector Exchange Specification [WAVES] [10] was designed to be the unified testing environment for systems developed using VHDL. The various uses of WAVES include:

1. Defining the test stimuli in the form of digital waveforms or test vectors.
2. Defining the results to be collected, and
3. Controlling the execution of the test stimuli and to collect and compare the results after executing the VHDL simulation.

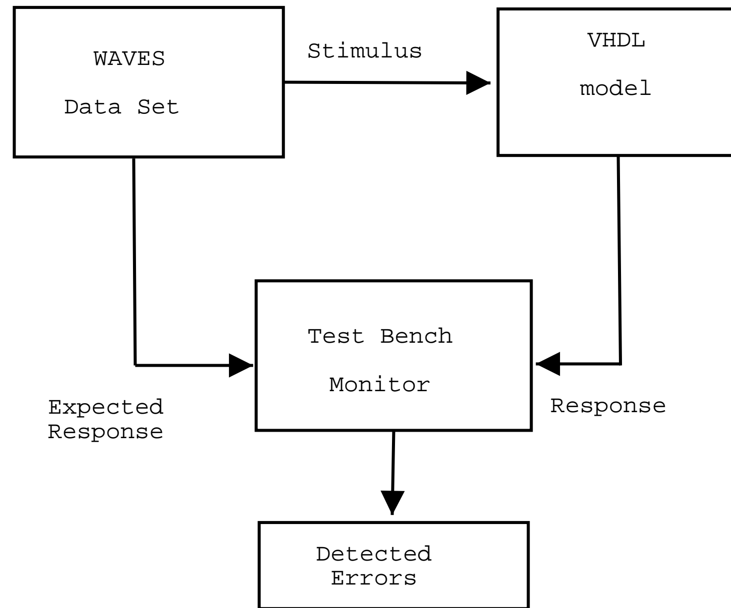

Figure 2.3: WAVES-VHDL Test Bench Configuration

WAVES serves as a test bench for VHDL simulation. WAVES is an IEEE standard and is also a subset of the VHDL standard. This format guarantees conformity among multiple applications and easy integration with VHDL units under test. WAVES test suite consists of a WAVES test bench, a user developed WAVES data set and the VHDL model to be tested. Test vectors can be specified with the help of a principal WAVES dataset element - the External file.

Figure 2.3 shows the usage of test bench in simulation. It consists of a Data set that contains information about the input and the expected output values. The input values are given to the VHDL model. A Test Bench Monitor compares the response from the simulation to the expected output from the WAVES data set. It notifies the error if there are any differences between expected output and the output obtained from the simulation.

2.4 Altera MAX+PLUS II

The Altera Multiple Array MatriX Programmable Logic User System (MAX+PLUS II) [1] provides a multi-platform, architecture-independent design environment that easily adapts to specific needs. It is a fully integrated package for creating logic designs for Altera Programmable logic devices - including the Classic, MAX 5000, MAX 7000, MAX 9000, FLEX 6000, FLEX 8000 and FLEX 10K families of devices. This software offers a full spectrum of logic design capabilities - a variety of design entry methods for hierarchial designs, powerful logic synthesis, timing-driven compilation, partitioning, functional and timing simulation, linked multi-device simulation, timimg analysis and verification. In MAX+PLUS II a logic design, including all subdesigns, is called a *project*. A project consists of all files in the design hierarchy, including ancilliary input and output files. MAX+PLUS II performs compilation, simulation, timing analysis and programming on one project at a time.

2.4.1 ALTERA Project Verification

MAX+PLUS II provides three applications - the MAX+PLUS II Simulator, the Timing Analyzer and the Waveform Editor - to help test the logic of a compiled project. The Simulator tests the logical operation and internal timing of a project, allowing the user to model a circuit design before it is programmed into a device. For functional, timing or multi-project simulation , the project must be compiled and a *Simulator Netlist File (.snf)* should be generated. The Simulator uses a graphical waveform *Simulator Channel File (.scf)* or an *ASCII Vector File (.vec)* as the source of input

vectors. It is easy to create an Input Vector File which generates a .scf file.

2.4.2 Input test format for simulating ALTERA MAXPLUS II models

An ASCII text file (with the extension .vec) is used as the input for simulation, functional testing, or waveform design entry. Simulation is carried out based on the vector file which specifies the input logic levels. These input logic levels are the vectors that drive the input pins and determine the internal logic levels throughout the project. It also specifies the start and stop times of the vectors and also the intervals at which these vectors can be applied. The Simulator creates an .scf file based on the .vec file. The .vec file can be created using any normal text editor.

2.5 An Introduction to XML

As the abstract test vectors are generated in XML this section provides a brief introduction to XML. XML, the Extensible Markup Language, is a W3C-endorsed standard for document markup [9]. It defines a general syntax used to markup data with simple human readable tags. Data is included in XML documents as strings of text, and the data is surrounded by text markup that describes the data. A particular unit of data and markup is called an *element*. The XML specification defines the rules for the syntax- how the data is represented in tags, how the tags are placed, how the attributes should be defined for a particular tag and so on . XML is a meta-markup language, it doesn't have a fixed set of tags. Users can define their own tags based on their application. This provides a lot of flexibility for specifications. For instance

a chemist can use tags that describe molecules, atoms, and other relevant elements in chemistry, while real estate agent can use elements that describe appartments, rents etc. A tool that deals with generation of test vectors can use it to define tags that define vectors, conditions, and parameters. The generic format of a tagged value can be represented as:

```
<tag> data(attributes) </tag>
```

All forms of data must be embedded in tags. Though XML is flexible in the type of elements it allows to be defined, it mandates a grammar for every XML document. That grammar defines the placement of tags, how tags should be nested and how the attributes should be attached to the elements. XML documents that follow rules defined in the grammar are called *Well-Formed* XML documents.

2.5.1 XML Building Blocks

All the XML documents are made up of the following building blocks.

Elements, *Tags*, *Attributes*, *Entities*, *PCDATA*, *CDATA*.

Elements are the main building blocks of an XML Document.

Tags are used to markup elements. A starting tag like <element> can be used as a beginning of the element and a tag like </element> defines the end of the element.

Entities are used to define common text. They are expanded when they are parsed by the XML parser.

PCDATA is text that is parsed by the XML parser.

CDATA is the text that is not parsed by the XML parser.

Empty Elements are elements that do not have any content. They can be represented as <elementname/> or <elementname></elementname>.

XML Trees: XML documents are represented as trees. Figure 2.4 gives a clear idea about XML documents.

```
<person>
  <name>
     <firstname>Albert</firstname>
     <lastname>Einsten</lastname>
  </name>
  <profession>scientist</profession>
</person>
```

Figure 2.4: XML Representation

Parents and Children Elements: The above XML structure is composed of one *person* element. *name* and *profession* are children for the *person* element. *firstname* and *lastname* are the children for the *name* element.

Attributes: An attribute is a name-value pair attached to the elements start tag. The name and the value are separated by an "=" sign or optionally by a space. For example Figure 2.5 shows a person born in 03/14/1879 and died in 04/18/1955 where *born* and *died* are the attributes.

Uses of XML: XML is used to separate data from HTML. With HTML, data is stored inside HTML. But with XML data can be stored in separate XML files. In this way

```
<person born=''03/14/1879'' died=''04/18/1955''>
  Albert Einstein
</person>
```

Figure 2.5: Attributes in an XML Document

HTML can be used for external data and display and be certain that any changes in the underlying data does not require changes for the HTML.

As computer systems and databases in real world contain data in different formats, it is a major problem for developers working on different database formats to exchange information. By converting and storing all the data in XML this complexity is reduced and the same data can be used by different applications. As XML data is stored in text format it is a software and hardware independent way of sharing the data. XML can also be used to store the data. The data can be stored in files or in databases. Applications can be designed to retrieve the data from the store. Applications can also be written to display the XML data.

The major advantage of XML is that it can be used to create new languages. For instance *WML*(Wireless Markup Language), a language used to markup Internet applications for mobile phones, is an XML derivative.

2.5.2 DTD

A Document Type Definition (DTD) [3] is defined for representing the test vectors in XML. DTD defines formal syntax that defines an XML document. It defines the order in which elements appear in a document. It also defines the attributes that can be defined for a particular element. It is a template that gives an idea about

the document. Any XML parser will fill this template to obtain the resultant XML document. Every XML file can carry a description of its own format with a DTD. This can be used as an interchange format for different groups if they agree on a specific DTD. DTD can be used to verify the data that you receive from the outside world or it can be used to test your own data.

DTD's can be defined based on the specific application. XML documents can be categorized into two classes. If a document conforms to the rules specified in the DTD then it is called a *valid document*. If it does not conform, it is known as *non-valid document*.

2.5.3 XML Schema

An XML Schema [3] defines the legal building blocks of an XML document. It defines all the elements that appear in an XML document. It defines the children for a particular node and also the order in which they can appear and the number of child elements. It also indicates the attributes for a particular element. With Schema you can determine if a particular element is empty. XML Schemas support data types. With support for data types it is easier to describe permissible document content. XML Schemas are represented in XML. So any XML parser can be used to parse a particular XML Schema file and access it using DOM. It can also be transformed using XSLT as it is in XML format. XML Schemas are extensible, and with this feature we can reuse our Schema in other Schemas and multiple schemas can be referenced from the same document.

2.5.4 XSLT

In this thesis an XSLT is used to convert abstract test vectors in XML to WAVES format. A gentle introduction to XSLT follows. XSLT stands for Extensible Style Sheet Transformation [11]. It is a W3C standard defined for transforming XML docments. The style sheet defines a set of rules that defines how information embedded between the tags in XML should be transformed. An XSLT style sheet is an XML document. It has the same features as XML, the data is represented in tags. An XSLT style sheet consists of a set of template rules each of which has the function *"implement a particular rule if a particular node is encountered."* The order of the rules does not have any significance. If more than one rule matches a particular element in the document, then a conflict-resolution algorithm is applied. It resembles text processing languages like Perl. The difference between a serial text processing language and XSLT is that the input is not processed line by line in XSLT. The input XML document is treated like a tree and each template rule is applied to a branch in the tree.

There are many XSLT processors that take a tree structure as input and generates another tree structure as the output. The input tree structure is produced by parsing the XML document and the output tree structure will often be serialized into another XML document.

2.5.5 DOM

DOM stands for Document Object Model [2]. It provides a standard programming interface to the applications. Using DOM a developer can access all the building blocks of XML documents like text nodes, attributes and a lot more. It is designed to

work with any language and any Operating System. The XML document should be stored in the memory before it can be accessed. An XML parser stores the XML in memory. After the document is loaded the information can be accessed and modified using DOM API. It gives capability to create new elements, new nodes and also new attributes. It also gives the capability to delete the elements. A DOM represents a tree view of an XML document. There will be a top level parent element and there will be different child nodes for the parent element. A child node can or cannot have siblings.

2.5.6 JAXP

JAXP stands for *Java Api for Xml Processing* [8]. JAXP leverages the parser standards SAX(Simple API for XML Parsing) and DOM(Document Object Model), so that the XML data can be parsed as a stream of events in the former case or as an object representation of the data in the later case. This package also supports XSLT that can be used to convert the XML data into other formats like HTML for representation purposes. This package also supports namespaces , allowing to work with DTD's which might otherwise cause naming conflicts. This package allows any XML complaint parser to be used from within an application. This package is used in this thesis to parse XML documents. Xalan and Xerces [8] parsers are used as part of this thesis work.

2.5.7 Summary

From the research conducted as part of our work, we found it useful to have an test vector generator that generates the test vectors in a language and platform independent manner. Also generating the test vectors from Rosetta specifications allows the designer to work at a highly abstract level. Based on the discussions in the previous chapters and the introductions of the technologies given in this chapter, we can form a framework for the representation of test-vectors in XML.

Chapter 3

Design Verification Test Generation

Details of the actual problem that this thesis addresses are provided in this chapter. Details of the test vector generator, $DVTG$ [5] are provided. It gives the details about the three phases involved in the generation of concrete test vectors - generation of test scenarios, generation of abstract test vectors in XML and the generation of concrete test vectors from the abstract test vectors. It further discusses the rationale behind selecting XML as the intermediary representation format for the abstract test vectors.

There are automatic test case generators in industry that generate test cases particular to a simulation environment [15]. This makes the tool dependent on a particular language and limits flexibility of the tool to integrate it with other applications. This limitation prevents the user to work with the tool if they are not familiar with the language in which the test vectors are generated. Moreover the implementation

of the tool must be changed every time a new format needs to be generated. DVTG is an automatic test case generator that is used to generate test cases from Rosetta. The following sections give a brief introduction about the specifics of DVTG and how it works.

3.1 Test Scenarios

A test scenario is a set of boolean conditions that are constraints on the values of input and output parameters of a haraware or software module. The methodology used for generating the test scenarios is an extension of the implementation based techniques applied to formal languages. Each test scenario consists of input criteria and acceptance criteria. The constraints on the inputs are specified by the pre-condition and hence the input criteria is obtained from the pre-condition. The constraints on the output for specific input values are specified by the post-condition. The acceptance criteria is therefore obtained from the post-condition. For a given test case and system if the input criteria are satisifed and does not satisfy the output criteria, then there is an error in the implementation.

Input criteria can be as specific as a single value or it can be generalized to a range of values. Thus an input variable can be initializd to a single value or it can be constrained within a range of values. The single value is used when the system needs to be driven to a particular state before the test vectors can be generated. The range of values are used to generate the abstract test vectors in the range. Similarly the acceptance criteria specify a single value or a range of values for the output parame-

ters. Thus the test scenarios provide a class of tests that will serve as inputs for the simulation of the system.

As all the terms inside a facet are boolean expressions, the test scenarios are generated by evaluating the expressions to *true* or *false*. Operators joining the variables and literals of various types are used to form the expressions. According to the *multi condition strategy* that is used to generate the test scenarios, enough test scenarios should be generated for an expression to take all possible values. For example, in the case of an arithmetic expression that has an integer outcome, test scenarios should be generated such that the result of the expression ranges from the smallest to the largest integer. In the case of boolean the entire range of outcomes are covered in the set of *true* and *false*. As all Rosetta terms are boolean expressions and they evaluate to *true*, test scenarios are generated such that the operands in the expression take all possible values that make the expression *true*.

In this work, input parameters are referred to as *driving* values. This indicates that the input parameters drive the system to a particular state. Output parameters are referred as *driven* values. This signifies that the output parameters will be driven to a particular state. Rosetta expressions are specified as predicates. A predicate is called *controllable* if the values of the variables that build the predicate are driving values. As the driving values are used to control the system, the predicate is called as a *controllable* predicate. Similarly if the predicate contains only driven values, then it cannot be controlled and hence it is called as *non-controllable*.

Rosetta expressions will be composed of either logical operators or relational operators or both. The following paragraphs gives information about how the different

logical and relational operators are processed in DVTG.

3.1.1 Logical Operators

Logical Operators are handled according to the canonical definition of each operator. Primarily all possible test cases for the operator based on the truth table are generated. After that redundant test cases are eliminated. For instance, consider a binary expression. It contains two operands and an operator. If both the operands of an operator have driving values then there are no output values to be observed and the test scenarios for this expression can be ignored. In the following paragraphs the terms **P(x)** and **Q(y)** are used. Both are predicates over x and y respectively.

- **And Operator**

P(x)	Q(x)	P(x) and Q(x)
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.1: Truth Table for AND Operator

Table 3.1 shows the truth table for the AND Operator. From the above table we can infer that the expression P(X) and Q(y) is true only when both the predicates P(x) and Q(y) are true. Hence the only test scenario that is generated for an **AND** expression is:

$$\mathbf{(P(x) = true)\ AND\ (Q(y) = true)}.$$

If the **AND** expression is a pre-condition then the test scenario can be eliminated as there are no output values to be observed.

- **Or Operator**

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

Table 3.2: Truth Table for OR Operator

Table 3.2 shows the truth table for the OR operator. From the above table it can be observed that the OR expression is true when either or both the operands are true. The OR expression is false when both the operands are false. So, the test scenarios that can be generated when the OR expression is evaluated to true are shown below:

$$(P(x) = false) \;\; AND \;\; (Q(y) = true) \tag{3.1}$$

$$(P(x) = true) \;\; AND \;\; (Q(y) = false) \tag{3.2}$$

$$(P(x) = true) \;\; AND \;\; (Q(y) = true) \tag{3.3}$$

By considering driving and driven values for the variables, P(x) is controllable when x is a driving variable and non-controllable if x is a driven variable. If P(x) is a controllable predicate in the expression (P(X) OR Q(y)) then the only scenario of interest is when P(x) is false. This is because the above OR expression will be true always regardless of the value of Q(y) when P(x) is true. The same applies to Q(y).

After eliminating the redundant test scenarios the final test scenarios are shown

in the table 3.3. In table 3.3 the left side columns indicate the driving and the

x	y	Test conditions considered		
0	0	3.1	3.2	3.3
0	1	-	3.2	-
1	0	3.1	-	-
1	1	-	-	-

Table 3.3: Final Test Scenarios of OR Operator

driven values. 1 indicates that a variable is a driving variable and 0 indicates that a variable is a driven variable. Thus the test scenarios that will be generated for an OR expression will be different.

- **Not Operator**

P(x)	not(P(x))
0	1
1	0

Table 3.4: Truth Table for NOT Operator

Not is a unary operator. From the truth table 3.4 it can be inferred that the NOT expression evaluates to true when the predicate P(x) is false. So the only test scenario that is generated is:

$$\mathbf{(P(x) = false)}$$

- **If-Then-Else Operator**

  If-Then-Else operator contains three predicates. The predicate in the *Then* condition is evaluated to true if the *if* condition is true. *Else* condition is evaluated to true if the *if* condition is false. An *if* expression of the form **If P(x) Then Q(y) Else R(z)** is equivalent to the disjunction of the form **(P(x) and Q(y)) or (not(P(x)) and R(z))**. Based on this transformation, the truth table is shown in Table 3.5.

P(x) and Q(y)	not(P(x)) and R(z)	(P(x) and Q(y)) or (not(P(x)) and R(z))
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.5: Truth Table for If-Then-Else Operator

From the above truth table, evaluating the if expression to true, we get the following test scenarios

$$not(P(x) \ \ and \ \ Q(y)) \ \ and \ \ (not(P(x)) \ \ and \ \ R(z)) \tag{3.4}$$

$$(P(x) \ \ and \ \ Q(y)) \ \ and \ \ not(not(P(x)) \ \ and \ \ R(z)) \tag{3.5}$$

The above test scenarios can further be simplified using *De Morgans* laws. Considering the test scenario 3.4

$$not(P(x) \ \ and \ \ Q(y)) \ \ and \ \ (not(P(x)) \ \ and \ \ R(z))$$

By expanding the NOT Operator using De Morgans laws we obtain:

$$not(P(x)) \;\; or \;\; not(Q(y)) \quad and \quad (not(P(x)) \;\; and \;\; R(z))$$

Applying the test scenario generation algorithm for the OR Operator, we get three scenarios:

$$P(x) \;\; and \;\; not(Q(y)) \quad and \quad (not(P(x)) \;\; and \;\; R(z)) \tag{3.6}$$

$$not(P(x)) \quad and \quad Q(y) \quad and \quad (not(P(x)) \;\; and \;\; R(z)) \tag{3.7}$$

$$not(P(x)) \quad and \quad not(Q(y)) \quad and \quad (not(P(x)) \;\; and \;\; R(z)) \tag{3.8}$$

First test scenario can be eliminated from the above 3 test scenarios as there is a contradiction in **P(x) and not(P(x))**. Thus the resultant 2 test scenarios are:

$$not(P(x)) \quad and \quad Q(y) \quad and \quad R(z)$$

$$not(P(x)) \;\; and \;\; not(Q(y)) \quad and \quad R(z)$$

It can be observed that the above two scenarios are contradictory as one test scenario contains **Q(y)** and the other test scenario contains **not(Q(y))**. So we can infer that the predicate **Q(y)** does not have any significance when the

predicate **P(x)** is false. So by ignoring **Q(y)** we get a single scenario:

$$not(P(x)) \;\; and \;\; R(z)$$

The final simplified scenario is:

$$(P(x) \;\; = \;\; false) \;\; and \;\; (R(z) \;\; = \;\; true)$$

Applying the similar technique to simplify test scenario 3.5 the final scenario is obtained as:

$$(P(x) \;\; = \;\; true) \;\; and \;\; (Q(y) \;\; = \;\; true)$$

Thus the final list of test scenarios generated for an If-Then-Else expression can be summarized as follows:

$$(P(x) \;\; = \;\; true) \;\; and \;\; (Q(y) \;\; = \;\; true)$$

$$(P(x) \;\; = \;\; false) \;\; and \;\; (R(z) \;\; = \;\; true)$$

3.1.2 Relational Operators

A Relational expression is an expression is an expression that contains operands with a relational operator. To generate the test conditions, test values are obtained that cause the relational expressions to evaluate to either true or false. The test scenarios generated for the relational operators are shown in table 3.6. In a relational expression,

Operator	Expression	True Expressions	False Expressions
<	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

Table 3.6: Test Conditions for Relational Operators

the right hand side operand divides the whole expression into two classes. One of the classes evaluate the expression to true and the other class evaluates it to false. For example the expression `a < 10` is divided into two classes. One class contains all the values of a that are less than 10 and the other class contains all the values of a that are greater than 10.

In some cases there can be logical expressions that have relational expressions as an operand. If these sub expressions generates many values for the operand, then they are combined according to the rules of the operator joining the sub-expressions. The following example shows a scenario where the relational and the logical expressions are combined in an expression. Considering an *if* expression of the following form:

```
if(input1<10)
  then ((output1 = 5) and (output2 = 20)
  else ((output1 = 10) and (output2 = 30)
end if
```

The scenarios that were generated for the above expressions are as follows:

```
scenario1:(input1 < 10) and (output1 = 5) and (output2 = 20)
scenario2:(input1 >= 10) and (output1 = 10) and (output2 = 30)
```

3.2 Test Requirements

Scenarios represent the boolean conditions specified on the input and output variables. They typically represent a range of values for the input and output variables. This range can be huge and the number of values that are generated can be very high. A user can place constraints on the values with a requirements file that limits the number of values that can be generated. The requirements should be set such that there is a practical limit on the range of values that can be generated.

Requirements can be directly specified on the input variables or there can be cases where a user can specify a range of values over a property of an input and not the input directly. In such cases it is necessary for the user to provide a mechanism so that inputs with the right property or characteristics are generated. Test requirements are generally specified on the input parameters of the components. The input parameters act as control variables for any component. These control variables can directly modify the output variables or they modify the data variables inside a component. The data variables are declarations that were made locally in the component. Figure 3.1 shows

```
package testrequirements is
begin logic
  test_req(parameter::label;lower_bound,upper_bound,setps::number);
  test_init(seq::number;vector::univ);
  init(seq::number;vector::univ);
export all;
end package testrequirements;
```

Figure 3.1: Test Requirements Package

the package that is used to specify the test requirements for a particular system. The function *test_req* takes a variable and it specifies a lower bound, upper bound and an

increment for the variable. The functions *test_init* and *init* accepts a sequence number and a vector. These vectors are used to drive a system to a particular state.

3.2.1 General Requirements

These requirements specify a range of values for the input variables. They are specified using the Rosetta function *test_req*. With this function a user specifies a lower bound, an upper bound and an increment for all the variables. The combination of all the numbers for different variables are the different test cases selected. This is illustrated with a sample example in figure 3.2. Figure 3.2 defines two *test_req* functions that

```
req1: test_req(var1, 1, 2, 1);
req2: test_req(var2, 2, 3, 1);
```

Figure 3.2: Specifying Test Requirements

initializes two variables var1 and var2. The test cases that will be generated are shown in Figure 3.3.

```
var1 = 1 var2 = 2
var1 = 1 var2 = 3
var1 = 2 var2 = 2
var1 = 2 var2 = 3
```

Figure 3.3: Test Cases Generated

3.2.2 Initial Vectors and Test Cases

In some situations, for the test cases to be generated a system needs to be driven to a particular state. The initial test vectors are used to drive the system to a initial state. The functions *test_init* and *init* are used to achieve this functionality. These functions

take a sequence number and an expression as an argument. The sequence number is an integer and it determines the order in which the initial vectors are evaluated. Figure 3.4 illustrates this concept.

```
package store_req is
begin logic
 facet store(timeIn::in time;setAlarm::in bit;setTime::in bit;
             toggleAlarm::in bit;clockTime::out time;
             alarmTime::out time;alarmOn::out bit) is
 test_req(param::sequence(char);lower_bound,upper_bound,increment::real)::bit;
 begin state_based
I1:init(1,(alarmTime = 10.0));
I2:init(2,(clockTime = 20.0));
I3:init(3,(alarmOn = 1.0));
l4:test_req(timeIn,5.0,6.0,1.0);
 end facet store;
end package store_req;
```

Figure 3.4: Requirements file illustrating usage of init function

Chapter 4

Implementation Details

4.1 Abstract Test Vectors

The main goal behind generating test vectors is to specify expected output values for specified input values. Vectors are obtained from test scenarios that are in turn obtained from the specification and user specified requirements. The DVTG tool restricts the test cases generated from test requirements to preconditions specified in the Rosetta specification. Scenarios generated from Rosetta specification are combined with user requirements to generate the abstract vectors. The process in represented in Figure 4.1.

4.2 XML Representation

The format for the XML representation for abstract test vectors is described in Figure 4.2. The DTD(Document Type Definition) [3] is presented and then the DTD is described with an example.

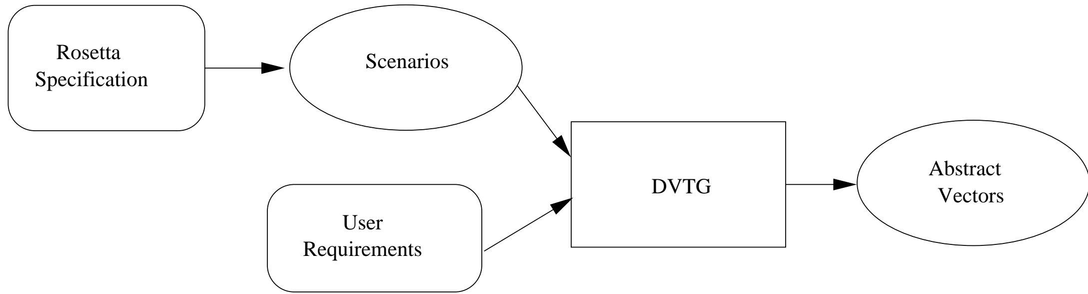

Figure 4.1: Flow of generation of Abstract Test Vectors

```
<!ELEMENT vectorslist (vector)*>
<!ELEMENT vector (condition)*>
<!ELEMENT condition (parameter,value)*>
<!ELEMENT parameter (#PCDATA)*>
<!ELEMENT value (#PCDATA)*>

<!ATTLIST parameter mode CDATA #REQUIRED*>
```

Figure 4.2: Abstract Test Vectors DTD

The XML document has *vectorslist* as the root element. *vector* is the child element for the root. The *vectorslist* root element can have many *vector* child elements. A *vector* has *condition* as the child element. There can be many *condition* elements in a *vector* element. Each *condition* has *parameter* and *value* as child elements. A *parameter* has a name and an attribute associated with it. An attribute is used to decorate a node. It gives more information about the node. Here the *parameter* node has *mode* as the attribute. It defines if the parameter is of input mode or output mode. A *condition* also has *value* as a child that stores the value of the parameter.

This is explained with a simple example for a schmidt Trigger. Figure 4.3 shows the scenario file for the schmidt trigger specification. From the specification in figure 4.3 for *schmidtTrigger* we can infer the following facts. The component has an input variable `input_voltage` and an output variable `output_value`. This

```
package schmidtTrigger is
begin logic
facet schmidt_trigger(input_voltage::in real;output_value::out bit) is
b :: bit;
begin state_based
  ACCEPT_1:(input_voltage >= 0.0) and (input_voltage =< 5.0);
  ACCEPT_1:(input_voltage < 1.0) and (b' = 0);
  ACCEPT_2:(input_voltage > 4.0) and (b' = 1);
  ACCEPT_3:(input_voltage =< 4.0) and (input_voltage >= 1.0) and (b' = b);
  ACCEPT_1:(output_value = b');
end facet schmidt_trigger;
end package schmidtTrigger;
```

Figure 4.3: Scenarios file for Schmidt Trigger

specification has a local variable declaration `b` that is of type bit. In this specification `input_voltage` becomes the control variable and the declared variable becomes the data variable. According to the specification the value of the input variable should lie between 0.0 and 5.0. According to the output condition `output_value = b'`, the value of the output variable equals the value of `b` in its next state. The value of the output variable is 0 if the input value is less than 1.0. If the input value lies between 1.0 and 4.0 then the value of the output variable equals the value of the declared variable in the previous state. If the value of the output variable is greater than 4.0 then the output value equals 1.0. The XML representation of the abstract test vectors is shown in Figure 4.4.

Figure 4.4 shows the Abstract Test Vectors in XML for Schmidt Trigger. The XML file starts with *<vectorslist>* element and is the collection of all the vectors. This is followed by the *<vector>* element. This element contains an abstract test vector. This element contains two *<condition>* elements. The first *<condition>* element stores the information about the input parameter *input_voltage*. It contains two ele-

```
<vectorslist>
  <vector>
     <condition>
          <parameter mode = "in">input_voltage </parameter>
          <value> 0.0 </value>
     </condition>
     <condition>
          <parameter mode = "out">output_value </parameter>
          <value> 0.0 </value>
     </condition>
  </vector>
  <vector>
     <condition>
          <parameter mode = "in">input_voltage </parameter>
          <value> 0.5 </value>
     </condition>
     <condition>
          <parameter mode = "out">output_value </parameter>
```

ments *<parameter>* and *<value>*. The attribute *mode* = *"in"* for the *<parameter>* node specifies that input_voltage is an input variable. The *<value>* element stores the value for the *input_voltage* variable. The other *<condition>* element stores the data about the output variable *output_value.* The *<parameter>* element stores the name and mode of the variable and the *value* element stores the value of the variable. We have a set of *<vector>* elements that store information about all the abstract test vectors. This information is ended by an ending *</vectorslist>* tag.

4.2.1 Concrete Test Vectors

As there is no testing software for Rosetta, abstract test vectors that are generated from DVTG are transformed into a format that is specific to some testing environment. DVTG converts the abstract test vectors into two formats. One is WAVES format, an input format for testing VHDL implementations. The other is an input format

```
          <value> 0.0 </value>
     </condition>
  </vector>
  ............
  ............
  ............
  <vector>
     <condition>
          <parameter mode = "in">input_voltage </parameter>
          <value> 4.0 </value>
     </condition>
     <condition>
          <parameter mode = "out">output_value </parameter>
          <value> 1.0 </value>
     </condition>
  </vector>
  <vector>
     <condition>
          <parameter mode = "in">input_voltage </parameter>
          <value> 4.5 </value>
     </condition>
     <condition>
          <parameter mode = "out">output_value </parameter>
          <value> 1.0 </value>
     </condition>
  </vector>
</vectorslist>
```

Figure 4.4: Abstract Test Vectors for Schmidt Trigger

for simulating models designed in ALTERA MAXPLUS II. This process is shown in figure 4.5.

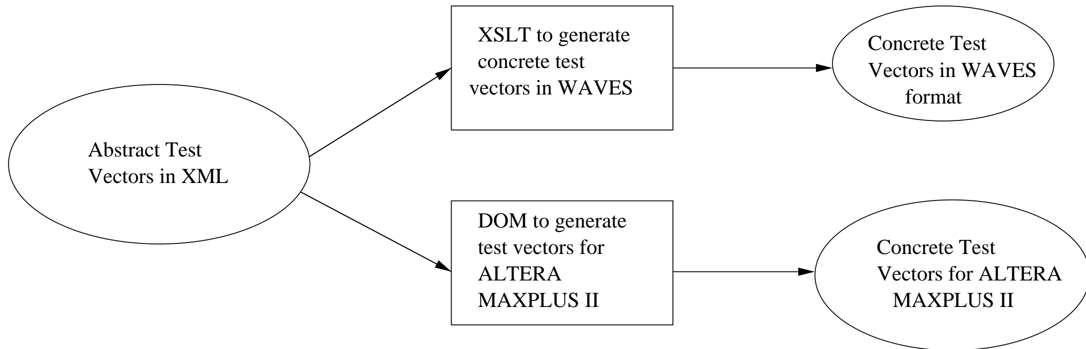

Figure 4.5: Generation of Concrete Test Vectors

Generation of test vectors in WAVES format

The abstract test vectors that have been generated in XML are transformed into WAVES, the test format for simulating the VHDL implementations. This functionality is achieved by using XSLT. An XSLT has been developed as part of this thesis to generate test vectors in WAVES format. XSLT takes an input XML document and generates the output based on the rules specified in the transformation sheet. The style sheet used as part of my thesis is shown in figure 4.6.

This XSLT has different templates specifying the action to be taken when a particular node is encountered. The XSLT processor walks through the XML document and when the appropriate node is encountered corresponding action is executed. In the transformation sheet in Figure 4.6 there are four templates. The first template matches the beginning <*vector*> element and prints all the input and the output vari-

```
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
        <xsl:text>%</xsl:text>
        <xsl:value-of select="count(vectorslist/vector)"/>
        <xsl:apply-templates select="/vectorslist/vector[1]/condition/parameter"/>
        <xsl:text>
        </xsl:text>
        <xsl:apply-templates select="/vectorslist/vector"/>
        <xsl:text>
        </xsl:text>
</xsl:template>
<xsl:template match="/vectorslist/vector">
        <xsl:apply-templates select="condition/value">
        </xsl:apply-templates>
        <xsl:text>
        </xsl:text>
</xsl:template>
<xsl:template match="/vectorslist/vector[1]/condition/parameter">
        <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="condition/value">
        <xsl:value-of select="."/>
        <xsl:text>&#9;</xsl:text>
</xsl:template>

</xsl:stylesheet>
```

Figure 4.6: XSLT for generating WAVES format

ables. The following templates extracts values for all the variables and prints them in a format shown in Figure 4.7.

```
input_voltage output_value
        0.0       0.0
        0.5       0.0
        1.0       0.0
        1.5       0.0
        2.0       0.0
        2.5       0.0
        3.0       0.0
        3.5       0.0
        4.0       1.0
        4.5       1.0
```

Figure 4.7: Concrete Test Vectors in WAVES format

Test format for testing ALTERA MAXPLUS II models

ALTERA MAXPLUS II has a simulation environment that allows users to simulate the models designed using that tool. To simulate the model the test requirements should be specified in a specific format. This thesis generates concrete test vectors in that format.

Converting the abstract test vectors in XML to this format is implemented using DOM. The input XML abstract test vectors are parsed and the information is stored in DOM. DOM is an Object Tree representation for the input XML document. An application has been developed as part of this thesis that walks through DOM and generates the output in the required format. The input format for testing the models designed using ALTERA MAXPLUS II is shown in Figure 4.8. The test format in Figure 4.8 contains the input parameters and their corresponding initial, final and the increment values. It also contains all the output parameters.

```
START 0;
STOP 5;
INTERVAL 0.5;
INPUTS input_voltage;
OUTPUTS output_value;
```

Figure 4.8: Input Test format for ALTERA MAXPLUS II

4.3 Support for Rosetta Data Types in DVTG

In the *Schmidt Trigger* specification there are two variables of type *real* and *bit.* In addition to these data types Rosetta has other data types. Data Types in Rosetta can be classified into two categories, *Primitive Data Types and Composite Data Types.* Primitive data types are atomic and cannot be decomposed. Composite data types are not elemental values and they can be defined by describing their contents. Support for different data types in DVTG is discussed in the following section. DVTG supports all the primitive data types like integers, reals, booleans and bits. It also provides support for composite data types such as bit-vectors and sequences. The following sections discuss the different composite data types and their support in DVTG.

4.3.1 Bit Vectors

A Bit Vector is a sequence of bits. Each element of a bitvector can be either 0 or 1. A bitvector is declared by encapsulating the bits with in square brackets. The different elements in a bitvector are separated by commas. [7] A Bitvector can be defined as

bitvector::type is sequence(bit);.

All the operations that can be performed over bits are generalized to bitvectors. The operations are performed by operating on the same indexed bits from the two bitvec-

tors. If any of the bitvectors is longer then the shorter bitvector is padded to the left with 0's. To allow bitvectors of specific lengths, a special subtype of bitvector is defined. A wordtype former takes a natural number as an argument and selects all the bitvectors that have the length as the specified natural number.

```
wordtype(n::integer)::set(bitvector) is sel(b::bitvector | #b = n);
```

The definitions

```
word::subtype(bitvector) is wordtype(16);
byte::subtype(bitvector) is wordtype(8);
nybble::subtype(bitvector) is wordtype(4);
```

define new types called word, byte, and nybble that have 16, 8 and 4 for lengths respectively.

Specifying Test Requirements for BitVectors

The specification of requirements for bit vectors is the same as specifying the test requirements for primitive data types like reals, integers etc. Specifying the test requirements for primitive data types is acheived by using *test_req* Rosetta function. This function will be specifying a lower bound, upper bound and an increment for the variables. For example the function declaration `test_req(var,1.0,5.0,1.0)` initialises 1.0 as the lower bound, 5.0 as the upper bound and increments in steps of 1.0 for the variable `var`. The test requirements for bit vectors are specified similarly. There will be a lower bound that is specified as a bitvector and an upper bound specified as bitvector. The increment is specified as a real number. This is illustrated with

a sample example. `test_req(var,[0,0,0,0],[1,1,0,0],1)` initialises `var` that is of type bitvector to have [0,0,0,0] as the lower bound, [1,1,0,0] as the upper bound and 1 as the increment. Rosetta supports *bv2nat()* and *nat2bv()* functions that facilitate the conversion of a bitvector to a natural and a natural to a bitvector respectively. The increment can thus be specified as a natural number.

BitVectors are containers of bits. They have only 0's and 1's as the variable elements. Rosetta also supports composite data types that are containers of elements that may not be 0's and 1's. These data types are *Sets* and *Sequences.* Sets provide a container for a specified type that is not indexed and does not contain duplicate items. *Sequences* index elements allowing individual elements within the *Sequence* to be accessed.

4.3.2 Sequences

The features of *arrays* and *lists* are combined into a single data structure called *Sequence* [7]. Sequences are indexed so that individual elements can be randomly accessed with their index. For illustration purposes if S = *[1,2,1]* then s(0) = 1, s(1) = 2 and s(2) = 1. A Sequence can have multiple instances of the same value. In the above example element '1' appears in two instances. The sequence "[ ]" former forms sequences by extension. The order of the elements in the sequence is the same as the lexical ordering in the former.

Specifying Test Requirements for Sequences

The specification of test requirements for sequences is different from specifying the test requirements for primitive data types. The test requirements for primitive data types contains specifying the lower bound, upper bound and increments for the variables. But the requirements for Sequences contain only initialized values and do not contain any final values or any increment. After the initial values are specified the individual elements in the sequence can be accessed with an index. `test_req(var,[apples,oranges,grapes])` is an example of specifying the requirement for sequences. `var` is of type *Sequence* and the initialized value is `[apples,oranges,grapes]`. The elements are indexed from 0 and the individual elements can be accessed with the index.

Implementation details of Sequences

The initialized values for the sequences are extracted from the test requirements file by parsing the test requirements. The variable name and the initialised values for the variable are stored in a data structure. The values are stored in the order in which they appear in the test requirement file. In the vector generator these values are acquired and based on the functionality specified in the scenarios file the corresponding operation is performed and the concrete test vectors are generated. For example if `a` is initialised to the sequence `[apples,oranges,grapes]`, then `a[0]` corresponds to the element `apples`, `a[1]` corresponds to the element `oranges` and so on.

Rosetta supports another composite datatype *sets*. Sets are also collection of elements, but there is no ordering among the elements of the set, so the individual elements cannot be accessed with an index. Generating test vectors for Sets is simi-

lar to generating the test vectors for Sequences. But while extracting the initialised values for the variables from the user specified test requirements and storing them in a data-structure, the order of the elements in which the elements are stored need not be considered. Moreover with a set, test vectors cannot be generated for individual elements. Vectors should be generated for the entire collection of elements.

Chapter 5

Examples

In the chapters thus far the methodology of generation of test vectors from Rosetta specifications has been discussed. In this chapter this methodology is illustrated with some sample examples. Schmidt Trigger and Alarm Clock examples are used to illustrate this methodology. Abstract and Concrete test vectors for both the examples are discussed here.

5.1 Schmidt Trigger

5.1.1 Functionality and Specification

A Schmidt Trigger is a square wave generating circuit component with hysteresis. If the output value exceeds a particular value (upper threshold) or if it goes below a certain value (lower threshold) the output value changes. If the input value is in between the upper threshold and the lower threshold then the output value remains the same.

```
facet schmidt_trigger(input_voltage :: in real;
                      output_value :: out bit) is
/* local declarations */
b::bit;
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)
       end if;
out:(output_value = b')
end facet schmidt_trigger;
```

Figure 5.1: Schmidt Trigger Specification

From the specification in Figure 5.1 we can see that the component has *input_voltage*, an input variable that is of type real. *output_value* is the output variable that is of type bit. The domain included is *state_based*. The first term *pre1* indicates that the input variable *input_voltage* should be between 0.0 and 5.0. The term labeled *out* describes the relationship between the output variable and the locally declared variable. The second term labeled *post1* describes the actual functionality of the component. If the value of the input variable is less than 1.0 then the value of the declared variable is 0. If the value is greater than 4.0 then the declared variable is 1. If the value of the input variable lies between 1.0 and 4.0 then the value of the declared variable equals the value of the variable in the previous state.

5.1.2 Test Scenarios

Based on the conditions specified in Figure 5.1 a scenarios file is generated. As shown

```
facet schmidt_trigger(input_voltage::in real;output_value::out bit) is
b::bit;
begin state_based
ACCEPT_1:(input_voltage < 1.0) and (b' = 0);
ACCEPT_2:(input_voltage > 4.0) and (b' = 1);
ACCEPT_3:(input_voltage =< 4.0) and (input_voltage >= 1.0) and (b' = b);
ACCEPT_4:(output_value = b');
end facet schmidt_trigger;
```

Figure 5.2: Scenarios for Schmidt Trigger

in Figure 5.2 the conditions ,that were specified in the specification in Figure 5.2, are represented as boolean conditions in the scenarios file. The first scenario is obtained by requiring the pre-condition to be true. The test scenarios for the post condition are obtained by evaluating the post condition to true. Three acceptance conditions are obtained by evaluating the IF-THEN-ELSE statements.

5.1.3 Test Requirements

The vectors are generated from the scenarios. From the scenarios in Figure 5.2 it can be inferred that the number of test vectors that can be generated can be very large and there must be a limit on the number of test vectors that can be generated. This is achieved by specifying test requirements. These test requirements specify the initial values, final values and increments for all the input variables. Figure 5.3 shows the test requirements.

test_req is the declaration of the function that is used to initialize the input variables. The term labeled *l1* actually specifies the initial value, final value and increment for input voltage. The input variable is initialized to 0.0 and the maximum value is 5.0. It should be incremented in steps of 0.5.

```
package schmidt_trigger_REQ is
begin  logic
  facet schmidt_trigger_REQ(input_voltage::in
              real;output_value::out real) is
  test_req(a,b,c,d::real)::real;
  begin state_based
  l1:test_req(input_voltage,0.0,5.0,0.5);
  end facet schmidt_trigger_REQ;
end package schmidt_trigger_REQ;
```

Figure 5.3: Requirements for Schmidt Trigger Specification

5.1.4 Output for Abstract Test Vectors in XML Format

The test vectors are generated from the generated test scenarios and the user specified test requirements. The abstract test vectors are shown in Figure 5.4 are obtained by evaluating the variables in the test scenarios file in Figure 5.2 with the values of the variables from the test requirements in Figure 5.3.

```
<vectorslist>
  <vector>
    <condition>
       <parameter mode = "in">input_voltage </parameter>
       <value> 0.0 </value>
    </condition>
    <condition>
       <parameter mode = "out">output_value </parameter>
       <value> 0.0 </value>
    </condition>
  </vector>
  <vector>
    <condition>
       <parameter mode = "in">input_voltage </parameter>
       <value> 0.5 </value>
    </condition>
    <condition>
       <parameter mode = "out">output_value </parameter>
       <value> 0.0 </value>
    </condition>
```

```
  </vector>
..........
  <vector>
    <condition>
      <parameter mode = "in">input_voltage </parameter>
      <value> 4.0 </value>
    </condition>
    <condition>
      <parameter mode = "out">output_value </parameter>
      <value> 1.0 </value>
    </condition>
  </vector>
  <vector>
    <condition>
      <parameter mode = "in">input_voltage </parameter>
      <value> 4.5 </value>
    </condition>

    <condition>
      <parameter mode = "out">output_value </parameter>
      <value> 1.0 </value>
    </condition>
  </vector>
</vectorslist>
```

Figure 5.4: Abstract Test Vectors for Schmidt Trigger

5.1.5 Concrete Test Vectors in WAVES format

Abstract Test vectors are converted to WAVES format using a style sheet, XSLT. The abstract test vectors in Figure 5.4 are given as an input to XSLT and the output is the WAVES format shown in Figure 5.5.

```
input_voltage output_value
        0.0       0.0
        0.5       0.0
        1.0       0.0
        1.5       0.0
        2.0       0.0
        2.5       0.0
        3.0       0.0
        3.5       0.0
        4.0       1.0
        4.5       1.0
```

Figure 5.5: Concrete Test Vectors in WAVES format

5.2 Alarm Clock

An Alarm Clock is a time keeping device. It provides the basic capability of displaying time, setting time, setting alarm and sounding the alarm. The following are the requirements for the alarm clock:

- When the *setTime* bit is set, the *timeIn* is stored as *clocktime* and output as the displaytime.
- When the *setAlarm* bit is set, the *timeIn* is stored as *alarmTime* and output as the displaytime.
- when the *alarmToggle* bit is set, the *alarmOn* bit is toggled.
- When the *clockTime* and *alarmTime* are equivalent and *alarmOn* is high then the alarm should be sounded, otherwise it should not.
- The clock increments its time value when the time is not being set.

The specification of alarm is shown in Figure 5.6. The parameterized list gives the

```
use time_types;
package alarm is
begin logic
facet alarmClockBeh(timeIn::in time; displayTime::out time;alarm::out bit;
                 setAlarm::in bit;setTime::in bit; alarmToggle::in bit) is
  alarmTime :: time;
  clockTime :: time;
  alarmOn :: bit;

begin state_based
  setclock: if setTime = 1
                      then (clockTime' = timeIn) and (displayTime' = timeIn)
                      else clockTime' = clockTime
                      end if;
  setalarm: if setAlarm = 1
                      then (alarmTime' = timeIn) and (displayTime' = timeIn)
                      else alarmTime' = alarmTime
                      end if;
  displayClock: setTime = 0 and setAlarm = 0 => displayTime' = clockTime;

  tick : setTime => clockTime' = increment_time(clockTime);

  armalarm: if alarmToggle = 1
                      then alarmOn' = -alarmOn
                      else alarmOn' = alarmOn
                      end if;
  sound : alarmOn and %(alarmTime = clockTime);

  end facet alarmClockBeh;

end package alarm;
```

Figure 5.6: System Level Specification for Alarm Clock

inputs to the system and the outputs from the system. Inputs correspond to the control variables to the clock.

timeIn - contains the current time input and can be used to set either the alarm time or the clock time.

displayTime - is the time being currently displayed.

alarm - drives the audible alarm.

setAlarm and *setTine* control whether the alarm time or clock time are currently being set.

alarmToggle - causes the alarm set state to toggle.

The state of the clock is represented by the local variables. The internal variables of the system are:

clockTime - This bit maintains the current time.

alarmTime - It stores the value of the time associated with sounding the alarm.

alarmOn is "1" when the alarm is set and "0" otherwise.

Inspecting the specification in Figure 5.6 indicates that each requirement is defined as a labeled term. Term *setClock* handles the case where the clock time is being set. Term *setalarm* handles when the alarm time is being set. Term *alarmarm* handles the toggling of the alarm set bit. *tick* causes the clock time to be incremented. The *sound* term defines the alarm output in terms of the alarmOn bit and whether the alarmTime and clockTime values are equal. Figure 5.7 shows the flow of inputs and

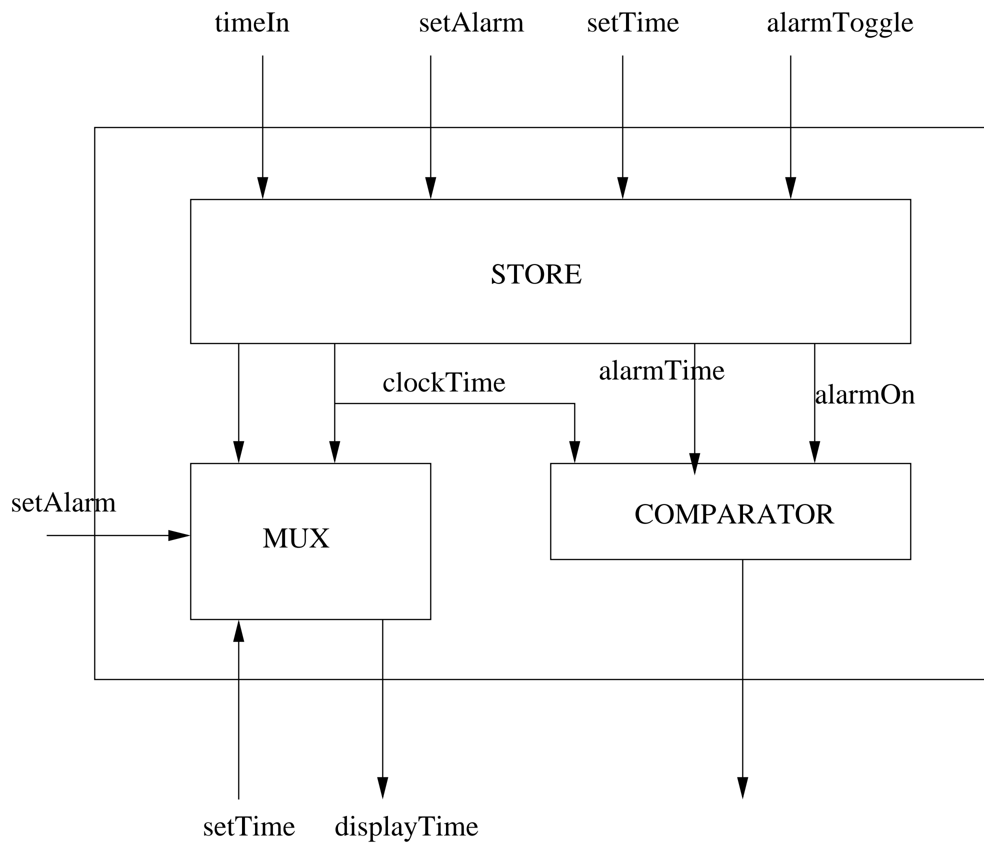

Figure 5.7: Structural Representation Of Alarm Clock

outputs diagrammatically. From Figure 5.7 we see that there are three components *store*, *mux* and *comparator* that comprise the alarm clock. As DVTG generates test vectors for individual components of a complex system, a single component *mux* is taken and its generation of concrete test vectors are illustrated here. The specification of a component of alarm clock, *mux* is shown in Figure 5.8.

```
use time_types;
package alarm is
begin logic
facet mux(timeIn::in time; displayTime::out time;
                  clockTime::in time; setAlarm::in bit; setTime::in bit) is
begin state_based
  l1: %setAlarm => (displayTime' = timeIn);
  l2: %setTime => (displayTime' = timeIn);
  l3: %(-(setTime xor setAlarm)) => (displayTime' = clockTime);
end mux;
end alarm;
```

Figure 5.8: Alarm Clock Specification

From the specification in Figure 5.8 it can be inferred that *timeIn* that is of type *time* is an input variable. *clockTime* is also an input variable of type *time*. *setAlarm* and *setTime* are input variables of type *bit*. As the concept of state is used the facet includes a state_based domain.

The actual specification of alarm clock is specified in terms. The first term *l1* specifies that if the *setAlarm* bit is set then the output variable *displayTime* in its next state equals the input variable *timeIn*. The second term *l2* also specifies that the output variable *displayTime* equals the input variable *timeIn* when the *setTime* bit is set. The last term *l3* specifies that when both the *setTime* and *setAlarm* bits are set simultaneously then the value of the *displayTime* equals the *clockTime*.

5.2.1 Abstract Test Vectors in XML

The test vectors are generated from the generated test scenarios and the user specified test requirements. Variables in Test scenarios are evaluated by instantiating with the values from the test requirements. The abstract test vectors are shown in Figure 5.9

```
<vectorslist>
  <vector>
    <condition>
        <parameter mode = "in">setAlarm </parameter>
        <value> 0.0 </value>
    </condition>
    <condition>
        <parameter mode = "in">setTime </parameter>
        <value> 0.0 </value>
    </condition>
    <condition>
        <parameter mode = "in">clockTime </parameter>
        <value> 6.2 </value>
    </condition>
    <condition>
        <parameter mode = "in">timeIn </parameter>
        <value> 5.1 </value>
    </condition>
    <condition>
        <parameter mode = "out">displayTime </parameter>
        <value> 6.2 </value>
    </condition>
  </vector>
............
</vectorslist>
```

Figure 5.9: Abstract Test Vectors for *mux* component

5.2.2 Concrete Test Vectors in WAVES format

Abstract Test vectors are converted to WAVES format using a style sheet, XSLT. The abstract test vectors in Figure 5.9 are given as an input to XSLT and the output is

the WAVES format shown in Figure 5.10.

```
setAlarm setTime clockTime timeIn   displayTime
0.0      0.0     6.2       5.1      6.2
1.0      0.0     6.2       5.1      5.1
0.0      1.0     6.2       5.1      5.1
1.0      1.0     6.2       5.1      6.2
```

Figure 5.10: Concrete Test Vectors for *mux* component

The above shown Schmidt Trigger and Alarm clock examples illustrate the support for primitive data types like reals, integers, booleans, bits in DVTG. DVTG also supports composite data types, data types whose elements are composed of primitive data types. Composite data types provides the user more flexibility while specifying the components. A user can represent more functionality with composite data types than with primitive data types. For example if a user wants to perform boolean operations on a collection of bits instead of a single bit, then defining a data type that contains a set of bits and performing operations on that data type will be more useful than performing on the individual bits. Such a collection of bits is called a BitVector in Rosetta. The following example illustrates the support for bit vectors in DVTG.

5.3 BitVectors support in Rosetta

A BitVector is a sequence of bits. All the operations that can be performed over bits are generalized to bitvectors. The following sections shows the specifications, abstract and concrete test vectors for bitvectors.

5.3.1 Specification Illustrating BitVectors

The specification in Figure 5.11 shows the sample specification using bitvectors. The facet contains two input variables A and B. The length of these bitvectors is 4. The outout variable is C is also of type bitvector. The term *l1* specifies the actual functionality. It states that the output variable is the logical *and* between the two input variables.

```
package bvector is
begin logic
  facet bvet(A::in bitvector(4);B::in bitvector(4);C::out bitvector(4)) is
  begin state_based
  l1:C = A and B;
  end facet bvet;
end package bvector;
```

Figure 5.11: BitVector specification

5.3.2 Test Scenarios

Figure 5.12 shows the scenarios file for the specification in Figure 5.11. The specification in Figure 5.11 is given as an input to the scenario generator and the final scenarios obtained are shown in Figure 5.12.

```
package bvector is
begin logic
  facet bvet(A::in bitvector(4);B::in bitvector(4);C::out
                                              bitvector(4)) is
  begin state_based
  ACCEPT_1:(C = A and B);
  end facet bvet;
end package bvector;
```

Figure 5.12: Scenarios for BitVectors

5.3.3 Test Requirements

Figure 5.13 shows the requirements file for bitvectors. The requirement specification for bit-vectors is the same as specifying the test requirements for reals. The function *test_req* takes the variable name, its lower bound specified as a bit-vector, the upper bound specified as bit-vector and the increment. The increment is specified as a real number. The requirement specification in Figure 5.13 contains two *test_req* functions that initialize two variables.

```
package bvector_req is
begin logic
   facet bvector_req is
   test_req(param::set(char);ip::bitvector(4);maximum::bitvector(4);inc::real)
           ::bit;
   begin state_based
   l1:test_req(A,[0,0,0,0],[0,1,1,1],1);
   l2:test_req(B,[0,0,0,0],[0,1,1,1],1);
   end facet bvector_req;
end package bvector_req;
```

Figure 5.13: Requirements for BitVectors

5.3.4 Abstract Test Vectors in XML

The abstract test vectors are generated by instantiating the variables in test scenarios in Figure 5.12 by the values specified in the test requirements in Figure 5.13. The abstract test vectors are shown in Figure 5.14.

```
<vectorslist>
  <vector>
     <condition>
        <parameter mode = "in"> B </parameter>
        <value> [0,0,0,0] </value>
     </condition>
     <condition>
        <parameter mode = "in"> A </parameter>
        <value> [0,0,0,0] </value>
     </condition>
     <condition>
        <parameter mode="out"> C </parameter>
        <value> [0, 0, 0, 0 ] </value>
     </condition>
  </vector>
...........
...........
...........
  <vector>
     <condition>
        <parameter mode = "in"> B </parameter>
        <value> [0,1,1,1] </value>
     </condition>
     <condition>
        <parameter mode = "in"> A </parameter>
        <value> [0,1,1,1] </value>
     </condition>
     <condition>
        <parameter mode="out"> C </parameter>
        <value> [0, 1, 1, 1 ] </value>
     </condition>
  </vector>
</vectorslist>
```

Figure 5.14: Abstract Test Vectors for BitVectors

5.3.5 Concrete Test Vectors in WAVES format

There is an XSLT, a transformation sheet, developed as part of this thesis that generates the concrete test vectors. Abstract test vectors in Figure 5.14 are given as an input to this transformation sheet and the resultant concrete test vectors are shown in Figure 5.15.

```
    B            A             C
[0,0,0,0]    [0,0,0,0]    [0, 0, 0, 0 ]
[0,0,0,0]    [0,0,0,1]    [0, 0, 0, 0 ]
[0,0,0,0]    [0,0,1,0]    [0, 0, 0, 0 ]
...........
...........
[0,1,1,1]    [0,1,0,1]    [0, 1, 0, 1 ]
[0,1,1,1]    [0,1,1,0]    [0, 1, 1, 0 ]
[0,1,1,1]    [0,1,1,1]    [0, 1, 1, 1 ]
```

Figure 5.15: Concrete Test Vectors for BitVectors

BitVectors are indexed collection of bits. Rosetta also provides data types that are collection of elements where the elements may not be simple reals or bits. The elements can be strings, numbers, bitvectors etc. Sets and Sequences are the data types that define the collection of elements. As Sequences define an indexed collection of elements generating test vectors for elements other than reals, integers will be different. The following example shows the generation of concrete test vectors for Sequences.

5.4 Sequences Support in Rosetta

5.4.1 Specication of Sequences

Sequences are indexed collection of elements. Figure 5.16 shows a specification containing sequences. This specification contains an input parameter A that is of type Sequence and an output parameter B that is also of type Sequence. Term *l1* specifies the actual functionality of the specification.

```
package SequenceType is
begin logic
  facet Sequence_Type(A::in sequence(char);B::out sequence(char)) is
  begin state_based
  l1:B = A;
  end facet Sequence_Type;
end package SequenceType;
```

Figure 5.16: Specification for Sequences

5.4.2 Test Requirements

Specifying Test Requirements for Sequences are different that specifying test requirements for numbers. *test_req* function initializes only a lower bound that is a collection of elements. Each element can be accessed with an index. Figure 5.17 shows a specification that provides requirements for Sequences.

```
package SequenceType_REQ is
begin logic
  facet Sequence_Type_REQ is
  S::sequence(character);
  test_req(param::sequence(char);initial_value::sequence(S))::int;
```

```
  begin state_based
  t1:test_req(A,[apples,oranges,grapes,oranges]);
  end facet Sequence_Type_REQ;
end package SequenceType_REQ;
```

Figure 5.17: Requirements for Sequences

5.4.3 Abstract Test Vectors in XML

Figure 5.18 shows the abstract test vectors for Sequences. These are obtained by combining the scenarios obtained from the specification and the user defined requirements.

```
<vectorslist>
  <vector>
     <condition>
        <parameter> A </parameter>
        <value> apples </value>
     </condition>
   <condition>
        <parameter> B </parameter>
        <value> apples </value>
     </condition>
  </vector>
  ......
  ......
  <vector>
     <condition>
        <parameter> A </parameter>
        <value> oranges </value>
     </condition>
     <condition>
        <parameter> B </parameter>
        <value> oranges </value>
     </condition>
  </vector>
</vectorslist>
```

Figure 5.18: Abstract Test Vectors for Sequences

5.4.4 Concrete Test Vectors in WAVES format

Figure 5.19 shows the Concrete Test Vectors in WAVES format for Sequences. Abstract Test Vectors are given as an input to a Transformation Sheet that is developed as part of this thesis and the resultant output is shown in Figure 5.19.

```
  A               B
apples          apples
oranges         oranges
grapes          grapes
oranges         oranges
```

Figure 5.19: Concrete Test Vectors for Sequences

Chapter 6

Summary and Future Work

6.1 Summary

The importance of testing in the development of any system cannot be understated. It accounts for approximately 50% in the development cycle of the product. The traditional approach for testing is implementation based testing. But this technique tends to overlook the intended behavior of the system and focus on the correctness of the implementation. This problem is overcome by using specification based testing techniques that derive tests from requirements.

Testing involves developing test cases and running the test cases on the implementation. Developing test cases is an repetitious and tedious process. Thus automating the process of generating the test cases will reduce testing costs. An automatic test case generator automatically generates test cases. There are many test case generators by now that generate test cases in a particular language. This limits the ability of the tool to integrate with other applications. It will be very useful if the automatic test

case generator can generate test cases in a format that is independent of any language or any simulation environment.

In this thesis we have developed a tool that automates the process of generating the test cases from a Rosetta specifications. The tool generates test scenarios and test vectors for input Rosetta requirement specifications. The test scenarios only give a range of values for testing, but not the actual input test cases. This range can be very large. So in order to limit the number of test cases that will be generated, a user can specify the test requirements for the coverage area desired.

In this thesis we have chosen XML as the language independent format for the test cases. This format is chosen because XML is getting widely deployed in the industry as a standard way of representing data. The abstract test vectors are generated in XML. The XML test vectors are then converted to concrete test vectors. As part of this thesis, the abstract test vectors are converted into WAVES, a format for testing the VHDL implementations and into input test format for simulating the models designed using ALTERA MAX+II software.

6.2 Conclusions

Specification based Testing techniques should be used to augment Implementation based techniques, but not to completely replace them. Using the specifications from System Level Design Languages allows the systems to be represented at higher abstraction levels and the problem associated with being not able to concentrate on requirements is solved with this testing technique.

Automated test case generation will be of immense use in the field of testing. The purpose of representing the test cases in a language independent format is served well by XML. As part of this thesis work, the XML abstract test vectors are transformed into two different formats. The concrete test vectors in these formats can be used to simulate VHDL models and models designed using ALTERA MAXPLUS II software. This XML format can be used to generate concrete test vectors for most of the simulation environments.

6.3 Future Work

The existing implementation of test vector generator does not implement all the features of Rosetta. Future work includes:

1. This tool supports only a limited set of data types like integer, real, bit, boolean, bit-vector, sequences. The support for user defined data types is not supported in this tool.

2. The support for structural specifications is not provided in this tool. When there are multiple facets in a single package, this tool cannot generate test vectors for the entire package. The test vectors will be generated for the individual facets.

3. In structural specifications, there should be support for generating the test cases for facets that do not have any interface parameters. If a particular facet does not have any input parameters or any output parameters to be observed then with this tool test cases cannot be generated. But when this facet is part of a *package* then it should be able to import the variables from the package and able

to generate the test cases.

4. This tool requires the input format for the test requirements to be in Rosetta. It will be more useful if the test requirements can be specified in XML.

5. The process of simulating the design can also be automated. A test harness will be used to instantiate the inputs and feed those inputs to the program to test its functionality. As this tool generates the test inputs in XML format if a test harness is developed that takes XML input form, then these abstract test vectors can be used for simulation purposes.

Bibliography

[1] Description of input format for simulating ALTERA MAXPLUS II models. World Wide Web: http://www.altera.com.

[2] Java Language API. World Wide Web: http://java.sun.com/j2se/1.4/docs/api.

[3] XML, DTD, XML Schema Tutorial. World Wide Web: http://www.w3schools.com.

[4] XML Specications. World Wide Web: http://www.w3c.org.

[5] Srinivas Akkipeddi. Advanced Test Vector Generation From Rosetta. Master's thesis, University Of Kansas, 2001.

[6] Perry Alexander. Details of rosetta. World Wide Web: http://www.sldl.org.

[7] Perry Alexander, Cindy Kong, David Barton, Peter Ashenden, and Catherine Menon. Rosetta strawman. 2002.

[8] Apache. Details of Java Parsers for XML. World Wide Web:http::://xml.apache.org.

[9] IBM. IBM Developers Tutorial. World Wide Web:
`https://www6.software.ibm.com/developerworks/education/xmlintro`.

[10] J.P.Hanna, H.L.Hirsch, T.H.Noh, and R.R.Vemuri. *Using WAVES and VHDL for Effective Design and Testing*. Kluwer Academic Publishers, 1997.

[11] Michael Kay. *XSLT Programmer's Reference, 2nd edition*. Wrox Press Inc, 2000.

[12] Garrin Kimmell, Joseph Kuehn, Kylie Williams, and Jesse Stanley. *Rosetta Developers Guide*. Universiry Of Kansas, December 2002.

[13] Bodan Korel. Automated Software Test Data Generation. *IEEE*, 1990.

[14] Roger S Pressman. *Software Engineering A Practioner's Approach*. Mc-Graw Hill, 2001.

[15] Krishna Ranganathan. DVTG, Design Verification Test Generation from Rosetta Specifications. Master's thesis, University Of Cincinnati, 2001.