[10] S. Helke, T. Neustupny, T. Santen, *Automating Test Case Generation from Z Specification using Isabelle*, In J. Bowen, M. Hinchey, editors, ZUM 1997: *The Z formal Specification Notation*, pages 52-71, Springer, 1997.

[11] P. Stocks, D. Carrington, *Test templates: A Specification-Based Testing Framework.* In Proceedings of the 15th International Conference on Software Engineering, pages 405-414, 1993

[12] Chang, Richardson and Sankar *Structural specification based testing using ADL.*

Bibliography

[1] Jeff Offutt, Aynur Abdurazik, *Generating Tests from UML Specificayions*, George Mason University, April 1999.

[2] T.H Tse, Zhinong Xu , *Test Case Generation for Class-Level Object-Oriented Testing* , University of Hong Kong, May 1996.

[3] Debra J Richardson, Cindy Tittle, Owen O'Malley, *Approaches to Specification-Based Testing*, University of California, December 1989.

[4] Mark T Pronobis, Robert Hillman, Christopher Flynn, *Test Insertion Without Being a Test Expert*,Rome Laboratory.

[5] Edmond M.Clarke, Jeannette M.Wing, *Formal Methods:State of the Art and Future Directions.*, Carnegie Mellon University, ACM Computing Surveys, December 1996.

[6] Perry Alexander, David Barton,Roshan Kamath *System Specification in Rosetta*, University of Kansas,IEEE Engineering of Computer Based Systems Symposium, April 2000.

[7] Dr.Perry Alexander, *A Practical Semantics of Design Facet Interaction* , University of Kansas, September 2000.

[8] Krishna Ranganathan, Dr.Perry Alexander, *Automated Test Vector Generation from Rosetta Requirements*, University of Kansas, May 2000.

[9] G. Myers, *The Art of Software Testing*, chapter 4, 1979.

2. The test vectors are not generated for a structural specification where a system is divided into different components and each component is specified as a Rosetta facet. In such cases, vectors could be generated for each of the components and tested.

3. The test coverage can only be specified using the 'step' approach. This restricts the user to narrow sequences of test inputs. The user should be able to specify inputs within a range using Gaussian distribution, Poisson distribution, etc.

4. The current version of the tool translates the abstract test vectors into WAVES test vectors to test VHDL implementation of the system. It would be useful if the abstract test vectors were translated into various other formats

6.3 Conclusion

Design and implementation are two independent tasks, with the implementation phase depending on an understanding of the design phase for a correct end product. Designing of systems in an abstract way is a complicated and error prone task. There is no smooth flow of the design process from the specification stage to the end product. The number of breaks in the design process increases with the increasing abstraction. It is the presence of such breaks in the life-cycle of a product that necessitates the validation of end-product against the original specification. In this thesis, we have presented a simulated based approach to validate if an implementation is correct. Our technique involves generating the test inputs and the expected outputs from the specification of a system. The implementation of a system is declared correct if the observed behavior matches the expected output obtained from the specification. This process, we believe is last and necessary step before declaring a product "correct"

Currently, test coverage can only be specified using the 'step' approach explained in chapter4. This restricts the user to narrow sequences of test inputs. Ability for the user to specify custom requirements exists, but the framework for understanding such custom requirements is not yet in place. Such a framework will increase the ability of our tool manifold.

application of WAVES and VHDL models conforming to the 1164 standard logic value system.

6.1.1 Evaluation

Evaluation of test information representation and test vectors generation is currently being performed with respect to some systems. Systems like Schmidt Trigger and an alarm clock example by Synopsys are used to evaluate the tool. The alarm clock example is interesting to evaluate since it allows test vector generation for typical register transfer level specifications and systems level specification. The Schmidt trigger example shown in this paper, demonstrates the effectiveness of representing and utilizing test requirements in the vector generation effort. The Schmidt trigger example thus the success of methodology in supporting expressions using IF-THEN-ELSE and AND operators. It covers both relational and logical operators, and serves to demonstrate generation of vectors for nested operators.

Currently more work is done in testing the Satellite Communication up-link system defined by TRW. As the system uses Time Division Multiplexing approach, it forces the test vector generation system to deal with issues such as synchronization. Such issues are critical in Rosetta's application domain. Although results are promising, it is still early to declare the approach successful in actual specification situations.

6.2 Future Work

The current implementation of test vector generator does not support all features of the Rosetta specification language. We enumerate some future tasks that could be done.

1. The current tool supports only primitive data types like boolean, bit, real, integer and their statically determined sub-types. The data types like record, set, tuple and bunch are not yet supported in the current implementation. The test vectors cannot be generated for operators that are not part of the standard Rosetta logical and relational operator set.

2. Obtaining the information from the user-defined requirements: The test scenarios provide us with a class of values for the input variables. This set of values can be huge and hence it is not feasible to generate test vectors for each of these values. By providing test requirements, the user limits the test cases to within the limit of practicality. Requirements provided by the user determine the number of test cases to be used, and the coverage desired. The test requirements can be a range of values for the input parameters, or initial vectors to drive the system to an initial state or a range of values for a characteristic of input parameters. The user defined requirements are obtained in a particular format as explained in chapter 4.

3. Generation of Test vectors from user-defined requirements and Test scenarios:The test vectors that are generated contain the following information

   (a) Values for the input parameters

   (b) Expected values for the output parameters corresponding to the input values

   The generated test cases are referred to as *abstract test vectors* because they are not specific to any testing software. The actual values of input parameters are obtained from the user defined requirements and additional values are generated by applying the boundary testing strategy. The expected output values are obtained by instantiating the inputs in the scenarios with each of corresponding values from test requirements.

4. Translation of Rosetta vectors into WAVES format: As there is no testing software for Rosetta, test vectors that are generated using test scenarios and user defined test requirements have to be translated into a format that is specific to some testing software. We have translated the test vectors generated to a WAVES format. WAVES is IEEE standard 1029.1-1991 for the representation of digital stimulus and response data for both the design and test communities. The format was developed to support users in the exchange of waveform information between different simulator and tester environments.Because waves is a exchange specification, all facets of stimulus and response data must be captured. This test bench tool was developed specifically for the

Chapter 6

Summary and Future Work

6.1 Summary

In this work, we have described the methodology we use for selection and generation of test vectors from a Rosetta specification of a component or a system and the user defined requirements. Test vectors generated provide us with input values and expected output values. The expected output values are compared with the output values from the implementation to check for errors in the implementation. Specification based testing is used to augment and complement implementation based testing. Specification based techniques are basically, implementation based techniques, applied to formal specifications. Generating test scenarios and test cases from the specification and using it against the implementation is a way to establish that the implementation indeed satisfies its specification.

The test vectors generation is described below

1. Generation of Test scenarios: All terms inside the facet are boolean expressions. The test scenarios can be obtained by evaluating the expression to *true* or *false*, depending on the requirements of a particular system. We use the multi-condition strategy as proposed by Myers to generate the specific test conditions from the expressions. According to this strategy, test scenarios should be generated for the expression under test to take all possible values.

5.3.3 Test Vectors

The test vectors are generated from generated test scenarios and test cases obtained from test requirements. We have to make sure that the pre-conditions have to be satisfied for the values of input parameters. In the above case, all the values of *input_voltage* satisfy the pre-conditions. Test scenarios are evaluated by instantiating *input_voltage* with corresponding values. In the first case when the value of is 0.0, the scenario *ACCEPT_1* and the scenario *ACCEPT_4* hold true. Hence the vector generated when the value of *input_voltage* is 0.0 is :

```
ACCEPT_0: (input_voltage =0.0) and (b' = 0) and (output_value' = 0);
```

Now considering the case when *input_voltage* is 1.0, it satisfies the pre-conditions and when the scenarios are evaluated, *ACCEPT_3* and *ACCEPT_4* hold true so the vector that generated for this case is :

```
ACCEPT_1: (input_voltage = 1.0) and (b = 0) and (b' = 0) and (output_value' = 0);
```

The abstract test vectors are generated similarly for all the values of *input_voltage* obtained from test requirements. The Schmidt trigger example thus demonstrates the success of methodology in supporting expressions using IF-THEN-ELSE and AND operators. It covers both relational and logical operators, and serves to demonstrate generation of vectors for nested operators.

```
use testrequirements;
Package schmidt_trigger_REQ is
 Begin logic
 Facet schmidt_trigger_REQ(input_voltage:: in real) is
  Begin logic
       req1: test_req(input_voltage,0.0,5.0,0.5);
  end;
end schmidt_trigger_REQ;
```

Figure 5.8: Test requirements for Schmidt Trigger

in steps of 0.1. The values for the input *input_voltage* that are obtained from user-defined requirements and by applying boundary testing strategy are as follows :

```
input_voltage = 0.0;
input_voltage = 0.5;
input_voltage = 0.9; (From boundary testing strategy)
input_voltage = 1.0;
input_voltage = 1.1; (From boundary testing strategy)
input_voltage = 1.5;
input_voltage = 2.0;
input_voltage = 2.5;
input_voltage = 3.0;
input_voltage = 3.5;
input_voltage = 3.9; (From boundary testing strategy)
input_voltage = 4.0;
input_voltage = 4.1; (From boundary testing strategy)
input_voltage = 4.5;
input_voltage = 5.0;
```

is *output_value*, that is of type *bit*. A state variable *b* is used to store the value of output *output_value*.

5.3.2 Generated Test Scenarios and Test Requirements

The scenarios generated from a Schmidt trigger specification,shown in Figure 2.1, are shown in Figure 5.7. The first test scenario is obtained by requiring the pre-condition to be *true*. Similarly, the test scenarios for the post-condition are obtained by evaluating the post-condition to *true*. The first three acceptance conditions correspond to the three branches of the IF-THEN-ELSE statement.

```
PACKAGE  schmidtTrigger IS
BEGIN logic
FACET schmidt_trigger(input_voltage:: in real;
                      output_value:: out bit) IS
      b :: bit;
BEGIN state_based
 INPUT_0: (input_voltage >= 0.0) and (input_voltage =< 5.0);
 ACCEPT_1: (b'  = 0) and (input_voltage < 1.0);
 ACCEPT_2: (b'  = 1) and (input_voltage > 4.0);
 ACCEPT_3: (b'  = b) and (input_voltage =< 4.0) and
           (input_voltage >= 1.0);
 ACCEPT_4: (output_value'  = b' );
END schmidt_trigger;
END schmidtTrigger;
```

Figure 5.7: Schmidt Trigger Test Scenarios

Since the variable *input_voltage* can take on infinite number of values, there is a need for limiting the input values to plausible ranges. This is done by using the test requirements to specify the input space and coverage, shown in Figure 5.8. This test requirement specifies that the variable *input_voltage* must be varied from 0 to 5 volts (input space) in steps of 0.5 volts.

From the test scenarios we find that there are two boundary values for the input parameter *input_voltage*, 1 and 4. So additional test cases are generated near the values of 1 and 4

```
            signal=[signal y];
            count = count + sampleperiod;
            if (count >= (numberbit *period))
                numberbit = numberbit+1;
            end;
        end;
```

The mechanism for generating the I and Q bit vectors, specified in facet *generate* is explained below. If the modulation type is FSK, SOBPSK or BPSK then all the bits are in phase and so there are no bit vectors in the Q channel. For the other modulation types like QPSK and DEQPSK, the I channel carries all the odd bits and the Q channel carries all the even bits. This is specified in the expressions *l1 to l5*. Once the bits transmitted in each channel are obtained, the next step is to generate the sampled signal. This is done by using the matlab function *bitvectorperiod*. Gaussian noise is then added to the sampled signal. This is done by using the pre-defined matlab function *awgn*. The function *awgn* adds sufficient noise to sampled signal such that the output signal has the right value for Signal-to-Noise Ratio *ratio*. Each of the sampled values is a real number. The function *real2bin* is used to convert the sampled value to 12 bit 2's complement samples.

5.3 Schmidt Trigger

5.3.1 Functionality and Specification

A Schmidt Trigger is a square wave generating circuit component. If the input exceeds a particular value (upper threshold) or goes below a particular value(lower threshold), the output changes. When the input value is between the upper threshold and lower threshold the output retains the value it had in the previous state. The input to the Schmidt Trigger is a voltage(signal of type `real`) according to which the output (of type `bit`) changes. The specification of a Schmidt Trigger component in Rosetta is shown in the Figure 2.1. The input parameter to the component is *input_voltage*,that is of type *real* and the output parameter

```
function sigpower = powersquare(main_signal)
    sigpower =0;
    for i = 1:length(main_signal)
        sigpower = sigpower + (main_signal(i) * main_signal(i));
    end;
    sigpower = sigpower/length(main_signal);
```

4. *bitvectorperiod*

   This function is used to generate a sampled bit vector signal at a given rate given the period of each bit in the original bit vector. The magnitude of generated sampled bit stream is 0.5 for a bit of value 1 and 0.0 for a bit of value 0.

```
function signal = bitvector(genvector,rate,period)
    signal = [];
    totaltime = period * length(genvector);
    magnitude = 0.5;
    sampleperiod = 1/rate;
    numofsamples = totaltime/sampleperiod;
    count = 0;
    numberbit = 1;
    vector = [];
    for i = 1:length(genvector)
        if genvector(i) == 1
           vector = [vector genvector(i)];
        else
           vector = [vector -1];
        end;
    end;
    while count < totaltime
        y = magnitude * vector(numberbit);
```

ratio and modulation type *modType*. A mechanism to generate the actual values of I and Q bit-vectors is provided in the facet *generate*. The facet *generate* takes Signal-to-Noise Ratio *ratio*, Modulation type *modType* as the inputs and outputs are I and Q bit-vectors. Matlab functions like *awgn*, *bitvectorperiod*, *powersquare*, *oddbits*, *evenbits* and *real2bin* are used in the facet. The matlab function *awgn* adds Gaussian noise to a signal in such a way that the output signal has a particular signal to noise ratio given the power of input signal. The other functions are explained below.

1. *oddbits*

   This function is used to return an bit-vector consisting of all the odd bits in a given bitvector.

```
function bits = oddbits(vector);
   count = 1;
   bits = [];
   while count < length(vector)+1
         if (rem(count,2) == 1)
            bits = [bits vector(count)];
 end;
         count = count+1;
   end;
```

2. *evenbits*

   This function is used to return an bit-vector consisting of all the even bits in a given bitvector. The implementation of this function is similar to the *oddbits* function given above.

3. *powersquare*

   This function is used to calculate the power of a bit vector signal.

```
USE matlabpackage;
USE testrequirements;
PACKAGE sigNoise_REQ IS
BEGIN logic
FACET generate(ratio::in real;modType::in integer;
               I,Q::out signal) IS
      bits::bitvector is [1;;0;;1;;0;;0;;0;;1;;0;;0];
      squareI,disI::signal;
      squareQ,disQ::signal;
      initI,initQ::signal;
      powerI::real;
      powerQ::real;
      rate::real is 64000;
      bitperiod::real is 1/9600;
BEGIN logic
  l1: (modType = 0) => (initI' = oddbits(bits)) and
      (initQ' = evenbits(bits));
  l2: (modType = 1) => (initI' = oddbits(bits));
  l3: (modType = 2) => (initI' = bits);
  l4: (modType = 3) => (initI' = bits);
  l5: (modType = 4) => (initI' = oddbits(bits)) and
      (initQ' = evenbits(bits));
  l6:squareI' = bitvectorperiod(initI,rate,bitperiod);
  l7:powerI' = powersquare(squareI);
  l8:disI' = awgn(squareI,ratio,powerI);
  l9:I' = real2bin(disI,12);
  l10:((modType = 0 )  or (modType = 4)) =>(squareQ' =
              bitvectorperiod(initQ,rate,bitperiod));
  l11:((modType = 0 ) or (modType = 4)) =>
             (powerQ' = powersquare(squareQ));
  l12:((modType = 0 ) or (modType = 4)) =>
             (disI' = awgn(squareQ,ratio,powerQ));
  l13:((modType = 0 ) or (modType = 4)) =>
             (I' = real2bin(disQ,number));
END;

FACET sigNoise_REQ(ratio:: in real;modType:: in integer;
                   I,Q::out signal) IS
      test_req(a,b,c,d::integer)::integer;
BEGIN state_based
  l1:test_req(ratio,15.0,30.0,5.0);
  l2:test_req(modType,0,4,1);
  l3:generate(ratio,modType,I,Q);
END;
END;
```


Figure 5.6: Requriements for Satellite Communication Example

in that decision. The synchronizer looks specifically for bit transitions to determine if the proper N symbols are being used to make a decision on the transmitted bit.

5. *Error Correction Decoder:* This block takes the 3 bit vector from the bit synchronizer and gives a decision on the transmitted bit.

6. *Unique Word Detector:* This block correlates the detected bits from the error correction decoder, against the selected unique word. This is implemented similar to a FIR filter with the unique word as the filter taps.

The requirements of the preprocessor system are as follows :

1. The BER (Bit Error Rate) measured at the output of demodulator for FEC code rate 1 shall not exceed *10E-5* for Eb/N values greater than 11.1 dB and BER shall not exceed *10E-3* for Eb/N values greater than 8.3 dB for SBPSK modulation type.

2. The BER (Bit Error Rate) measured at the output of demodulator for FEC code rate 1 shall not exceed *10E-5* for Eb/N values greater than 13.7 dB FSK modulation type.

3. The BER (Bit Error Rate) measured at the output of demodulator for FEC code rate 1 shall not exceed *10E-5* for Eb/N values greater than 12.2 dB and BER shall not exceed *10E-3* for Eb/N values greater than 10.0 dB for BPSK modulation type.

4. The BER (Bit Error Rate) measured at the output of demodulator for FEC code rate 1 shall not exceed *10E-5* for Eb/N values greater than 12.5 dB and BER shall not exceed *10E-3* for Eb/N values greater than 9.7 dB for DEQPSK modulation type.

5. The BER (Bit Error Rate) measured at the output of demodulator for FEC code rate 1 shall not exceed *10E-5* for Eb/N values greater than 11.1 dB and BER shall not exceed *10E-3* for Eb/N values greater than 8.3 dB for SOQPSK modulation type.

5.2.2 Requirements

The user defined requirements for this example is given in the Rosetta package shown in Figure 5.6. In this case the requirements are provided for a property of inputs I and Q,

1. Unique word detected - This bit represents whether the unique word has been detected in the input bit stream.

2. Doppler Estimate - This is a 12 bit vector that gives the integer estimate of the Doppler frequency offset in Hertz.

3. Automatic Gain control threshold - This is a discrete signal provided by the preprocessor indicating whether the input signal power is greater than a particular threshold value.

The various blocks or components in the preprocessor system are as follows :

1. *Resampler :* The function of resampler block is to resample the I and Q bit vectors to a rate that is determined by the modulation rate. The resampling rate is still high enough to track the Doppler offset and is also an integral multiple of modulation rate. The output of resampler is also a 12-bit bit vector, but is generated at a different rate than the input.

2. *Carrier Recovery :* This block is to resolve the frequency uncertainty in the input signal. The frequency uncertainty is mainly due to the Doppler effect. The Carrier Recovery block contains a *voltage controlled oscillator*, *costas loop* and an *automatic gain control* (AGC) unit. Automatic gain controller block determines if the power of input signal is greater than a particular threshold or not. It must be noted that the input signal to the preprocessor is a downconverted signal from the receiver front end that we are not designing.

3. *Decimator :* This block is used to decimate the baseband data samples to a rate that is modulation rate times the number of samples per symbol. This rate is also determined by the modulation rate and is an integral multiple of modulation rate.

4. *Bit Syncronizer :* The 12 bit vectors that are input to this block represent values for individual data samples. Each set of N 12 bit vectors results in a single symbol soft decision. The 3 bit vector represents the decision of transmitted bit and the confidence

The specification that we use specifies a preprocessor block on the receiver side. The main functions of the preprocessor block are :

1. To find the doppler uncertainty.
2. To constantly check if the power of received signal is greater than a particular threshold.
3. To synchronize the bits using the bit-synchronizer block.
4. To detect the unique word by taking the correlation.

It is assumed that there exists a microprocessor that provides some control signals to the preprocessor block. The various control signals are :

1. modType - This gives us the method for carrier recovery.
2. modRate - It is used to determine the interpolation/decimation rates in the resampler and the number of samples per symbol in the bit synchronization block.
3. uniqueWordType - From this value we can determine the sequence against which the incoming bit stream is correlated.
4. burstType - This gives the burst type of a signal.
5. accessType - The access type give the channel that is being used to send the data. The channel could be 5-khz single access or 25-khz TDMA.
6. FECRate - This gives the FEC code rate used to encode.
7. reset - This is a bit that is used to reset the synchronization functions for receiving a new burst or when synchronization is lost.

The inputs to the preprocessor block along with the control signals described above are I and Q bit vectors. These are the 12 bit two's complement samples of in-phase and quadrature waveforms that have been converted to baseband. The outputs from the preprocessor block are :

```
%%% This is the External Test Vectors File in WAVES %%%
%%% This file is automatically generated by the DVTG tool %%%

%       timeIn  setAlarm  setTime alarmToggle  displayTime     alarm
        4          1        -         -             4            -
        1          -        1         -             1            -
        1          0        1         1             1            0
        2          0        1         1             2            0
        3          0        1         1             3            0
        4          0        1         1             4            0
        5          0        1         1             5            1
        6          0        1         1             6            0
        7          0        1         1             7            0
        8          0        1         1             8            0
        9          0        1         1             9            0
        10         0        1         1             10           0
        11         0        1         1             11           0
        12         0        1         1             12           0
```

Figure 5.5: Alarm Clock WAVES file

5.2 Satellite Communication Preprocessor System

5.2.1 Functionality

In Time Division Multiple Access (TDMA), messages sent by different users are interlaced in time. The data from each user is transmitted in slots, with a number of slots comprising a frame. In each time interval only one user is allowed to transmit or receive the data. So each user occupies a cyclically repeating time slot. It is necessary to maintain overall network synchronization in a TDMA system.

Each transmission within a control or slave channel time slot consists of two elements.

1. A synchronization element (Preamble) - a known signal the receiver needs for carrier, bit and data synchronization and a unique word.

2. A data element containing the information.

range of various input variables. There are no additional test cases by applying the boundary testing strategy because all the control inputs are of type *bit*. The resultant sequence of test cases that are obtained from test requirements are as follows :

```
(timeIn = 4) and (setAlarm = 1) -- from the init function.
(timeIn = 1) and (setTime = 1) -- from the init function.
(timeIn = 1) and (setTime = 1) and (setAlarm = 0) and (alarmToggle = 1)
(timeIn = 2) and (setTime = 1) and (setAlarm = 0) and (alarmToggle = 1)
(timeIn = 3) and (setTime = 1) and (setAlarm = 0) and (alarmToggle = 1)
(timeIn = 4) and (setTime = 1) and (setAlarm = 0) and (alarmToggle = 1)
(timeIn = 5) and (setTime = 1) and (setAlarm = 0) and (alarmToggle = 1)
(timeIn = 6) and (setTime = 1) and (setAlarm = 0) and (alarmToggle = 1)
(timeIn = 7) and (setTime = 1) and (setAlarm = 0) and (alarmToggle = 1)
(timeIn = 8) and (setTime = 1) and (setAlarm = 0) and (alarmToggle = 1)
(timeIn = 9) and (setTime = 1) and (setAlarm = 0) and (alarmToggle = 1)
(timeIn = 10) and (setTime = 1) and (setAlarm = 0) and (alarmToggle = 1)
(timeIn = 11) and (setTime = 1) and (setAlarm = 0) and (alarmToggle = 1)
(timeIn = 12) and (setTime = 1) and (setAlarm = 0) and (alarmToggle = 1)
```

The test vector generation first ensures that a test case conforms to pre-conditions specified in the input Rosetta specification. The generated test scenarios are then combined with the user-specified test requirements to generate the abstract test vectors. This is done by instantiating the inputs in the scenarios with each of the corresponding values from test cases.

Using the initial vectors provided in the function *init*, the alarm clock system is driven to a state where the internal variable *alarmTime* is set to a value 6 and the *clockTime* variable is set to a value 1. The other vectors are obtained similarly by instantiating inputs in test scenarios with each of the corresponding values from the test cases. Abstract test vectors are translated into a format that is appropriate to the WAVES software. The waves file for the above alarm clock example is as shown in the Figure 5.5.

```
(setTime == 0) and (setAlarm == 0) and (displayTime'  = clockTime);
```

For the fourth labelled item *sound*, test scenarios generated on applying the rules for IF-THEN-ELSE operator are :

```
(alarm'  = 1) and (alarmToggle == 1) and (alarmTime == clockTime);
(alarm'  = 0) and (alarmToggle == 0) and (alarmTime /= clockTime);
(alarm'  = 0) and (alarmToggle == 0) and (alarmTime == clockTime);
(alarm'  = 0) and (alarmToggle == 1) and (alarmTime /= clockTime);
```

5.1.3 Initial Vectors and Test Cases

```
USE testrequriements;
PACKAGE alarmClockBeh_REQ is
BEGIN logic

FACET alarmClockBeh_REQ(timeIn:: in integer;setAlarm,setTime::in
                        bit;alarmToggle::bit) is
BEGIN state_based

 init1: init(1,(setAlarm=1) and (timeIn=4));
 init2: init(2,(setTime =1) and (timeIn=1));
 t1:test_req(timeIn,1,12,1);
 t2:test_req(setTime,1,1,1);
 t3:test_req(setAlarm,0,0,1);
 t4:test_req(alarmToggle,1,1,1);
END;
END;
```

Figure 5.4: Alarm Clock User-Defined Requirements

From user defined requirements, shown in Figure 5.4, we observe that there are some initial vectors. As explained in the previous chapter vectors provided in the *init* function are to be evaluated once before the other test cases. Vectors provided in the function *test_init* are to be evaluated once before every test case. The other test cases are obtained from the given

```
 Facet alarmClockBeh_TEST(timeIn:: in integer;displayTime:: out integer
                          alarm:: out bit;setAlarm::in bit;
                          setTime,alarmToggle:: in bit) IS
       alarmTime,clockTime:: integer;
Begin state_based
       ACCEPT_0: (setTime == 1) AND (clockTime' = timeIn)
                                AND (displayTime' = timeIn);
       ACCEPT_1: (setTime == 0) AND (clockTime' = clockTime + 1);
       ACCEPT_2: (setAlarm == 1) AND (alarmTime' = timeIn)
                                  AND (displayTime'  = timeIn);
       ACCEPT_3: (setAlarm == 0) AND (alarmTime' = alarmTime);
       ACCEPT_4: (setTime == 0) AND (setAlarm == 0)
                                AND (displayTime' = clockTime);
       ACCEPT_5:(alarm'  = 1) AND (alarmToggle == 1)
                              AND (alarmTime == clockTime);
       ACCEPT_6:(alarm'  = 0) AND (alarmToggle == 0)
                              AND (alarmTime /= clockTime);
       ACCEPT_7:(alarm'  = 0) AND (alarmToggle == 0)
                              AND (alarmTime == clockTime);
       ACCEPT_8:(alarm'  = 0) AND (alarmToggle == 1)
                              AND (alarmTime /= clockTime);
End;
```

Figure 5.3: Alarm Clock Specification

true.For the first labelled item *setclock*, applying the rules of IF-THEN-ELSE operator, generated test scenarios are :

```
(clockTime'  = timeIn) and (displayTime'  = timeIn) and (setTime == 1);
(clockTime'  = (clockTime + 1)) and (setTime == 0);
```

Similarly, generated test scenarios for labelled item *settime* are :

```
(alarmTime'  = timeIn) and (displayTime'  = timeIn) and (setAlarm == 1);
(alarmTime'  = alarmTime) and (setAlarm == 0);
```

For the third labelled item *displayClock*, applying the rules for the IMPLIES operator, generated test scenarios are

```
 PACKAGE alarmClockBeh IS
 BEGIN logic

 FACET alarmClockBeh(timeIn::in integer; displayTime::out integer;
                     alarm::out bit; setAlarm::in bit;
                     setTime::in bit;alarmToggle::in bit) IS
       alarmTime :: integer;
       clockTime :: integer;
       increment_time(clk::integer)::integer is clk + 1;

BEGIN state_based
setclock: if %setTime
              then (clockTime' = timeIn) and (displayTime' = timeIn)
              else clockTime' = increment_time(clockTime)
          endif;
setalarm: if %setAlarm
              then (alarmTime' = timeIn) and (displayTime' = timeIn)
              else (alarmTime' = alarmTime)
          endif;
displayClock: (setTime == 0) and (setAlarm == 0) =>
                        (displayTime' = clockTime);
sound: alarm' = if ((alarmToggle == 1) and (alarmTime = clockTime))
                     then 1
                     else 0
                endif;
END alarmClockBeh;
END;
```

Figure 5.2: Alarm Clock Specification

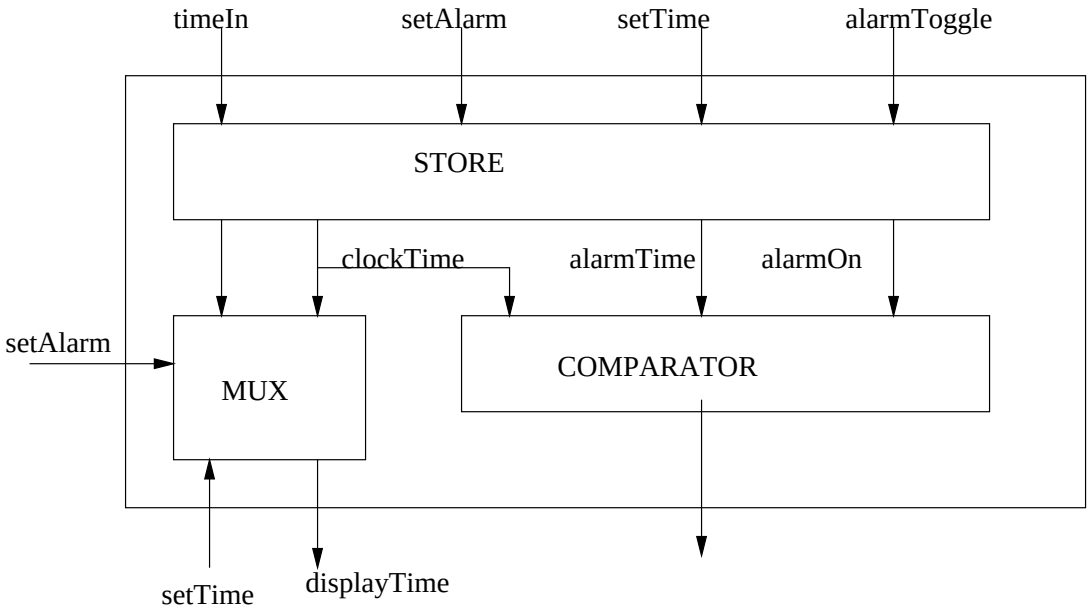

Figure 5.1: Structural representation of the Alarm Clock

clock time is being set. The clock time is incremented in the next state as long as the input *setTime* is *false*. Term *setalarm* handles when the alarm time is being set. If *setAlarm* is not set, then *alarmTime* retains its value from the previous state. Term *displayClock* handles the third requirement, case when clock time is being displayed and term *sound* defines *alarm* output in terms of *alarmToggle* bit and whether *alarmTime* and *clockTime* values are equal. All the terms in the specification have to be true simultaneously. Thus the specification has the same effect as VHDL.

5.1.2 Test Scenarios

We explain generation of input values that satisfy input criteria and output values that satisfy acceptance criteria. The methodology described in chapter 3 is used for generation of these values. The generated scenarios are as shown in Figure 5.3. Examining the test scenarios indicates that the scenarios are generated for each term of the input specification.

In order to generate test scenarios, the expression in the labelled term should evaluate to

3. *alarmToggle* - This control bit indicates whether or not the alarm will ring when current time is equal to alarm time. The alarm will not ring if it is set to low. It is also used to shut off the alarm.

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

Local variables correspond to the state of the clock. The internal variables to the system are

1. *clockTime* - It maintains the current time. If the *setTime* bit is not set then the *clockTime* should be incremented

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

The outputs to the system are

1. *displayTime* - The time that is to be displayed.

2. *alarm* - A bit that indicates that the alarm sound is on.

The specification of an alarm clock system states the following requirements :

1. When *setTime* bit is set, *timeIn* is stored as *clockTime* and output as *displayTime*.

2. When *setAlarm* bit is set, *timeIn* is stored as *alarmTime* and output as *displayTime*.

3. When both *setTime* and *setAlarm* bits are not set, the clock always increments its time value (*clockTime*).

4. When *clockTime* and *alarmTime* are equal, and *alarmToggle* is high, the alarm should be sounded. Otherwise it should not.

Figure 5.2 shows the specification of Alarm clock system in Rosetta. Looking at the requirements and specification above, we can observe that every requirement is defined as a Rosetta expression or a labelled item. The term *setclock* handles the first requirement, case where

Chapter 5

Examples

In this chapter, we present a few examples to explain the methodology of generation of test vectors from Rosetta specification. The examples discussed are the Alarm clock system, a Satellite communication preprocessor system and a Schmidt trigger component. We explain the functionality, generation of test scenarios, test requirements and generation of test vectors for each of the examples.

5.1 Alarm Clock

5.1.1 Functionality

An alarm clock is a time keeping device that has the additional feature of sounding an alarm at a particular time. It also provides the basic capability of setting time and setting alarm time. The inputs to the system are

1. *setTime* - Control bit to indicate that input time should be stored as current time of the alarm clock. The *setTime* and *setAlarm* signals cannot be set simultaneously.
2. *setAlarm* - Control bit to indicate whether that input time should be stored in the internal alarm time.

4.3.2 Conversion

The abstract test vectors that have been generated are not in a format that is specific to some test software. We address the transformation of abstract test vectors generated as a Rosetta facet into WAVES test vectors suitable for VHDL simulation. Since the format of concrete test vectors is specific to the WAVES testing software, stimulus waveforms are applied to the VHDL model, the model produces its output response based on the stimulus presented. The expected and the actual responses can then be compared to find the errors.

```
%%% This is the External Test Vectors File in WAVES %%%
%%% This file is automatically generated by the DVTG tool %%%

%       input_voltage           output_value
            0.1                     0
            0.6                     0
            0.9                     0
            1.1                     0
            1.6                     0
            3.9                     0
            4.1                     1
```

Figure 4.6: WAVES test vectors for Schmidt Trigger

Figure 4.6 shows the WAVES test vectors obtained for the Schmidt Trigger component.The '%' symbol is the commment character in WAVES. The third comment line indicates the list of input and output parameters. The first column is the list of values of input parameter *input_voltage* and the second column specifies the expected values for output parameter *output_value* for corresponding input values.

The test benches are not generated automatically and the user needs to write the test benches for VHDL implementation of a system. Theses test benches could then use the WAVES data set to drive the system during simulation. The results obtained from the simulation are then compared to the expected output values in the WAVES data set. If both are the same, then the implementation conforms to the specification.

oping their WAVES data set and generation of a test bench for applying the stimulus and response to the models. This test bench tool was developed specifically for the application of WAVES and VHDL models conforming to the 1164 standard logic value system. Figure 4.5 is a simplified illustration of the test configuration that is generated.

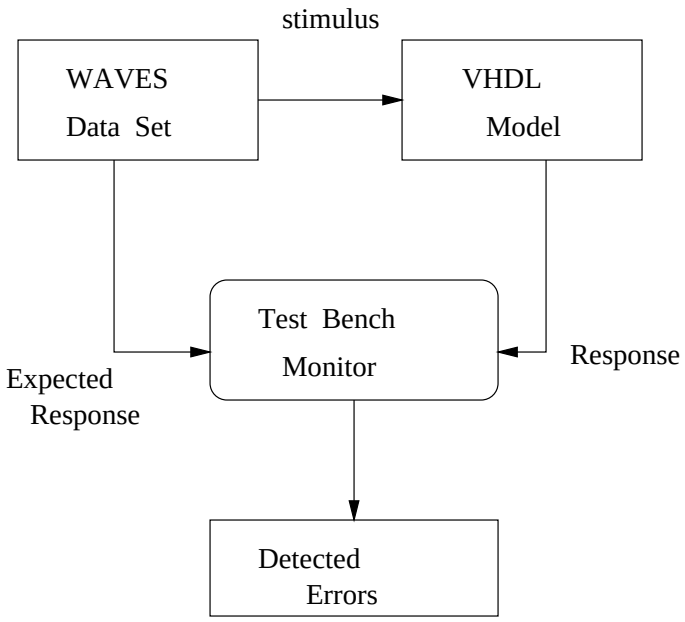

Figure 4.5: WAVES-VHDL Simplified Test Bench Configuration.

Stimulus waveforms that are produced by the WAVES data set are applied to the VHDL model, the model produces its output response based on the stimulus presented. The test bench contains a set of monitoring processes that monitor the expected and actual response values and ensures that they conform to the timing specified in the WAVES data set.

The use of automated test insertion software reduces the need of circuit designers from becoming an expert in testability algorithms, test element implementations, and testability approaches. The structured approach to test insertion that this software provides reduces design time while implementing a verified, efficient test solution. The benefit here is that the designer is required to think up front about the responses and not after the simulation is complete by looking over stacks of simulation reports.

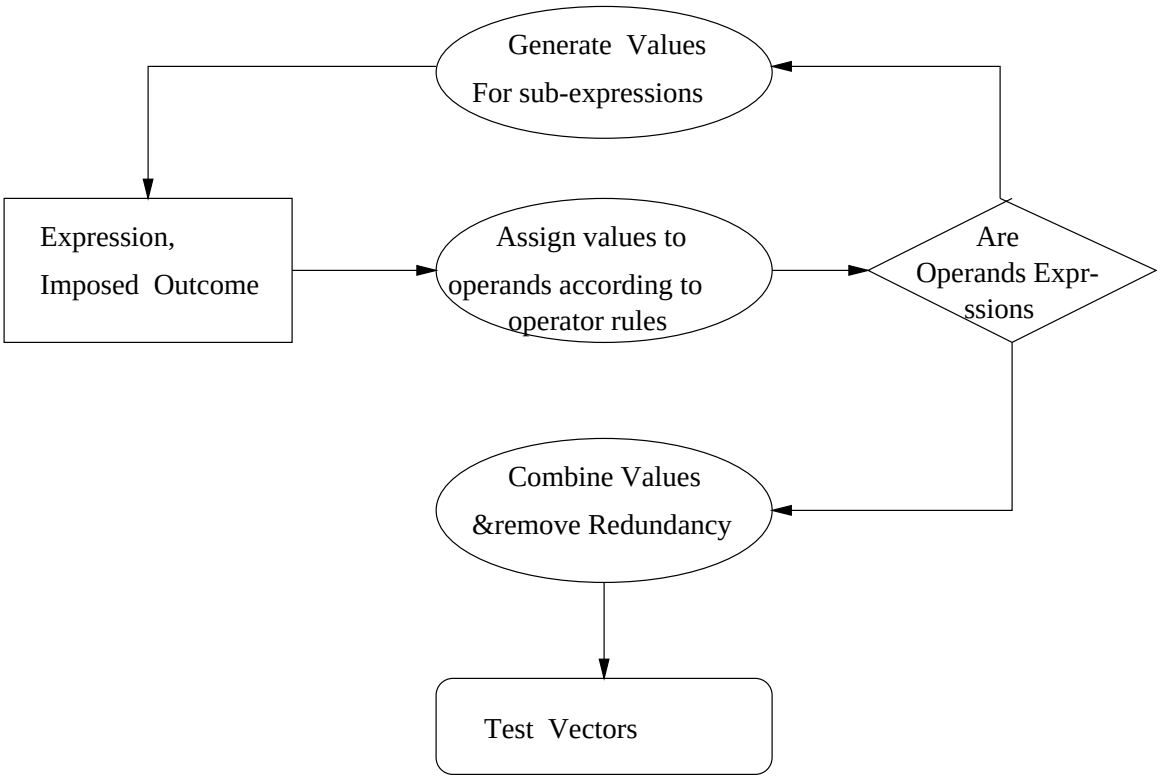

Figure 4.4: Algorithm for generation of Test Vectors

4.3 Conversion of Vectors to WAVES Format

As there is no testing software for Rosetta, abstract test vectors that are generated using the test scenarios and the user defined requirements have to be translated into a format that is specific to some testing software. We have translated the test vectors generated to a WAVES format. The background about WAVES and the details of translation are explained below.

4.3.1 Background on WAVES

WAVES is IEEE standard 1029.1-1991 for the representation of digital stimulus and response data for both the design and test communities. The format was developed to support users in the exchange of waveform information between different simulator and tester environments.Because waves is a exchange specification, all facets of stimulus and response data must be captured. When information is exchanged between environments, assumptions are dangerous and often incorrect. The purpose of the test bench tool is to aid a user in devel-

```
A = 1  AND  B = 0; (vector from "init" function)
A = 1  AND  B = 1; (vector from "test_init" function)
A = 1  AND  B = 4;
A = 1  AND  B = 1; (vector from "test_init" function)
A = 1  AND  B = 5;
A = 1  AND  B = 1; (vector from "test_init" function)
A = 1  AND  B = 6;
A = 1  AND  B = 1; (vector from "test_init" function)
A = 2  AND  B = 4;
A = 1  AND  B = 1; (vector from "test_init" function)
A = 2  AND  B = 5;
A = 1  AND  B = 1; (vector from "test_init" function)
A = 2  AND  B = 6;
A = 1  AND  B = 1; (vector from "test_init" function)
A = 3  AND  B = 4;
A = 1  AND  B = 1; (vector from "test_init" function)
A = 3  AND  B = 5;
A = 1  AND  B = 1; (vector from "test_init" function)
A = 3  AND  B = 6;
```

Figure 4.3: Test Cases when initial vectors are specified in Requirements

Actual values of input parameters are obtained from user defined requirements as explained in the previous section. Additional test cases are generated using the boundary testing strategy. A boundary value is one that is directly on, above or below the limit of a class or range of values specified by a condition. For example, in the condition ($x < 10$), 10 is a boundary value that divides the values of x into different classes. When using real numbers, the step value of a particular variable, specified in test requirements, is used to determine the smallest step size. For example, in the condition ($x > 3.5$) with a step size of 0.05, the smallest step value would be 0.01. Experience shows that test vectors that use boundary conditions have a higher payoff than test vectors that do not. The values generated from the user-defined test requirements are combined with any boundary conditions within the specified bounds to obtain the final abstract test vectors.

which initial vectors are evaluated to reach a particular state. The initial vectors that are obtained from the function `init` are evaluated only once before all the test cases, obtained from generic requirements, are evaluated. The vectors obtained from the `test_init` are evaluated before every test case obtained from the generic requirement.

If the user specifies the requirements for a facet with two input variables A and B as follows :

```
req1: init(1,(A = 1) and (B = 0));
req2: test_init(1,(A = 1) and (B = 1));
req3: test_req(A,1,3,1);
req4: test_req(B,4,6,1);
```

The test cases that the system generates are shown in Figure 4.3. The vectors obtained from the function *init* are evaluated only once. The vectors obtained from the function *test_init* are evaluated once before any test case that is obtained from the function *test_req*.

4.2 Generation of Abstract Test Vectors

Test vectors generated from test scenarios and user defined test requirements contain the following information :

1. Values for the input parameters
2. Expected values for the output parameters corresponding to the input values

The test vector generation first ensures that test cases generated from test requirements, conform to the pre-conditions specified in the input Rosetta specification. The generated test scenarios are then combined with the user-specified test requirements to generate the abstract test vectors. This is done by instantiating the inputs in generated test scenarios with each of the corresponding values from the test requirements. Figure 4.4 gives the flowchart of the algorithm that is used to generate the test scenarios and the test vectors

The package *matlabpackage* contains the declarations of matlab functions in Rosetta format. The user might use the matlab functions to provide the mechanism to generate actual values of input given the property of that input. Figure 4.2 shows the Rosetta package *matlabpackage*. Functions in the package might be predefined matlab functions like *awgn*, a function used to add white gaussian noise, or some user defined matlab functions.

```
PACKAGE matlabpackage IS
BEGIN logic
  pi::real is 3.1416;
  phase::subtype(real) is sel(x::phase | -180 < x < 180);
  signal::subtype(array(real));
  bitsignal::subtype(array(bit));
  sinewave(amp::real;ph::phase;freq::real)::univ;
  bitvectorsignal(input::bitvector;rate::real)::univ;
  bitvectorperiod(input::bitvector;rate,period::real)::univ;
  defaultbitvector(rate::real)::univ;
  powersquare(sig::univ)::real;
  awgn(sig::univ;snr::real;power::real)::univ;
  print(sig::univ)::boolean;
  realTobin(x::univ;y::integer)::univ;
  real2bin(x::univ;y::integer)::univ;
  oddbits(x::bitvector)::bitvector;
  evenbits(x::bitvector)::bitvector;

EXPORT ALL;
END matlabpackage;
```

Figure 4.2: Matlab Package

4.1.3 Initial Vectors and Test Cases

In some cases, it is necessary for the system to reach a state before the generated test cases are tested on the system. The initial test vectors are used to drive the system to a particular state before the test cases are evaluated. The functions `test_init` and `init` declared in the package `testrequirements`, Figure 4.1 are used to specify the initial vectors. Both the functions `test_init` and `init` take a *sequence number* and an *expression* as the arguments. The sequence number is an integer and it provides us the information about the order in

specify this kind of requirement. This function takes a variable, a lower bound, an upper bound and the step value as the arguments. All the numbers within the specified lower and upper bound are selected such that they all vary by the specified step value. A combination of all the numbers selected for different input variables are the different test cases generated.The following example gives an idea of test cases generated when the user provides a requirement for more than one input variable. Requirements can be specified only for the inputs to the system or for the property of input. If the user specifies the requirements for two input variables A and B as follows :

```
req1: test_req(A,1,3,1);
req2: test_req(B,4,6,1);
```

The different test cases that are generated from the above requirements are as follows :

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

4.1.2 Requirements for a property of the input

The user might provide requirements for a property of an input to the system. In such cases, the user must provide a mechanism to generate the actual input values given just the value of property of that input. This mechanism is provided in a Rosetta facet, that is then translated to a matlab function. On execution of this *matlab* function, actual values of input are obtained.

The user defined test requirements are specified for input parameters of a system. These coverage requirements are determined by a number of factors like the most frequently used range of values for signals, confidence, time constraints, the acceptable risk factors, etc.

```
PACKAGE testrequirements IS
BEGIN logic

test_req(label::label;lower_bound,upper_bound,steps :: number ) :: bunch(number
         sel(x :: univ | (lower_bound =< x) and (x =< upper_bound)
         and exists(n :: univ | x = (n * steps + lower_bound)));
test_init(seq::integer;vector::univ)::univ;
init(seq::number;vector::univ)::univ;

EXPORT ALL;
END testrequirements;
```

Figure 4.1: Requirements Package

The Rosetta package, shown in Figure 4.1 gives the declaration of functions that should be used by the user when providing test requirements for a particular system : The package `testrequirements` illustrates our current approach to specifying test requirements in Rosetta. The `test_req` function accepts a variable, a lower bound, an upper bound and the step value. Apart from the variable, that is of type `label`, all the other parameters are of type number, that enables the function to be used over natural numbers, integers, real numbers and boolean variables (that are really maximum and minimum integers). The `test_req` function itself uses the `sel` function to select all the numbers within the specified lower and upper bounds such that they all vary by the specified step value. The functions `test_init` and `init` accepts a sequence number and an The vectors specified in these functions are used to drive the system to a particular state.

4.1.1 Generic Requirements

Generic requirements are requirements where the user provides a range for input variables with a step size. The function `test_req` declared in package `testrequirements` is used to

Chapter 4

Test Requirements and Test Vectors

This chapter explains the general methodology used to generate test vectors from test scenarios and user defined requirements. The chapter can be divided into two parts. In the first part we discuss about test requirements provided by the user. Requirements provided by the user are used to get the actual values of input variables. We also discuss about the initial vectors. The second part of chapter is about generation of test vectors from user defined requirements and generated test scenarios. This involves assignment of values to the operands of expressions that occur in relevant clauses of a specification. Finally, the implementation details of how test vectors are generated using the Rosetta Object Model are explained.

4.1 Test Requirements

In the previous chapter, we explained the generation of test scenarios that provide us with a set of values for the input variables. This set of values can be huge and hence it is not feasible to generate test vectors for each of these values. By providing test requirements, the user limits the test cases to within the limit of practicality. Requirements provided by the user determine the number of test cases to be used, and the coverage desired. In some cases, the user might provide a range of values for a property of input and not directly for the input. In such cases, it is necessary for the user to provide a mechanism so that inputs with the right property or characteristic are generated.

```
B' = IF (A > 50)
        THEN 21
        ELSE 19
     ENDIF;
```

By following the rules for IF-THEN-ELSE operator, the scenarios that we get are

B' = (A > 50) AND 21

B' = (A =< 50) AND 19

But these are not the correct scenarios, the scenarios generated are

(B' = 21) AND (A > 50)

(B' = 19) AND (A =< 50)

In the second scenario since the relational expression (A > 50) is *false*. The values of A less than or equal to 50 evaluate the expression to *false* and hence, these values are selected.

3. When the scenarios are generated for a expression, there is every chance that the sub-expressions might be redundant. The tool removes redundancy in the generated scenarios. For example the scenarios generated for a nested IF-THEN-ELSE expression will have redundancy when the rules are applied.

```
IF (input_voltage < 1.0)
    THEN (b' = 0)
    ELSE (IF (input_voltage > 4.0)
              THEN (b' = 1)
              ELSE (b' = b)
         ENDIF)
ENDIF;
```

The scenarios generated by applying the rules are

```
(b'  = 0) AND  (input_voltage < 1.0)
(b'  = 1) AND  (input_voltage > 4.0) AND (input_voltage >= 1.0)
(b'  = b) AND (input_voltage =< 4.0) AND (input_voltage >= 1.0)
```

From the above scenarios generated we observe that there is redundancy in the second scenario. If it is given that input voltage is greater than 4, then it is redundant to say that input voltage is greater than one. After the redundancy is removed the scenarios are

```
(b'  = 0) AND  (input_voltage < 1.0)
(b'  = 1) AND  (input_voltage > 4.0)
(b'  = b) AND (input_voltage =< 4.0) AND (input_voltage >= 1.0)
```

4. Consider the following expression :

((A AND B) = false) AND (D = true)

The l.h.s. is another logical expression whose value is to be evaluated to false. The list of scenarios for *(A AND B)* when the expression is evaluated to *false* are :

(A = false) AND (B = false)

(A = false) AND (B = true)

(A = true) AND (B = false)

The r.h.s. is just a identifier. Combining both the scenarios the list of test scenarios for the expression *(A AND B) OR (D')* are :

(A = false) AND (B = false) AND (D' = true)

(A = false) AND (B = true) AND (D' = true)

(A = true) AND (B = false) AND (D' = true)

2. Consider an expression

```
IF (A > 50)
        THEN (B' = 20)
        ELSE (B' = B + 1)
ENDIF;
```

The above expression is evaluated to true and the scenarios generated by applying the rules for IF-THEN-ELSE operator are :

(A > 50) = true) AND (B' = 20) = true

(A > 50) = false AND (B' = B + 1) = true

Applying the rules for relational operators, the following scenarios generated are :

(A > 50) AND (B' = 20)

(A =< 50) AND (B' = B + 1)

3.3 Implementation Details

The present Rosetta suite has the parser that parses the specification and builds a parse tree. This parse tree is then used to build an object model of Rosetta items. An item is the basic Rosetta semantic unit. All items have Type, Value, Label and String fields. The test scenario generator makes use of this object model for obtaining the fields. Every expression in the facet is in the form of a tree structure with the node as the operator and branches as left operand and right operand. The rules to generate test scenarios for logical operators are applied recursively until the operands are just identifiers or literals or sub-expressions without any logical operators. When the sub-expressions are relational expressions, the rules of relational operators are applied to evaluate the expression to the boolean value that is obtained and the scenarios are thus generated. The following examples give more information as to how the scenarios are generated :

1. Consider an expression with just the logical operators

```
(A  AND B) OR (D')
```

In the object model this expression is stored as a tree structure as follows

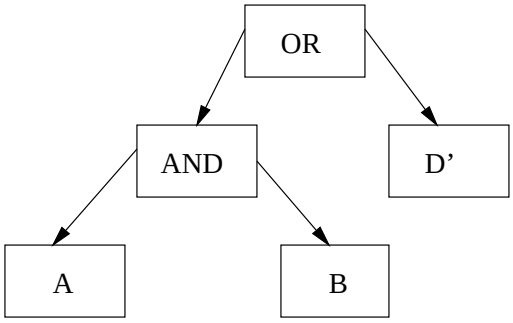

Figure 3.2: Tree Structure for the expression (A AND B) OR (D')

The test scenario generator tool first generates the values for the operands corresponding to the *OR* operator. In this case the r.h.s. is a non-controllable operator and the l.h.s. operator is a controllable operator so the test scenarios generated are as follows. The expression is evaluated to *true*.

Operator	Expression	Test conditions to evaluate expression	
		to true	to false
<	x < y	x < y	x >= y
=<	x =< y	x =< y	x > y
>	x > y	x > y	x =< y
>=	x >= y	x >= y	x < y
=	x = y	x = y	x /= y
/=	x /= y	x /= y	x = y

Table 3.7: Test conditions for relational operators

If the relational expression is a part of or a sub expression in a logical expression then the value to which the expression has to be evaluated is obtained by applying the rules for logical operators. The assigned boolean values determine whether the test scenarios are to be generated by evaluating the expression to *true* or *false*. If the relational expression is not a operand in a logical expression then the test scenarios are generated by evaluating the expression to *true* .

In a relational expression, the r.h.s operand divides the range of l.h.s into classes. One of these classes evaluate the expression to *true* and the other classes evaluate the expression to *false*. Depending on the operator, these classes fall in the *satisfy* or *non-satisfy* category of the expression. This is the process used to generate the test conditions for the relational operators.

For example, consider the expression x > 20 . This expression divides the range of x into two categories. One class of values that are greater than 20 and the other the values that are less than or equal to 20. If the expression has to be evaluated to false then the second class comes under the *satisfy* category and vice-versa.

As mentioned above, some logical expressions have the relational expressions as an operand. If each of the sub-expressions yields a number of values for the operands, then these are combined according to the rules of operator joining the sub-expressions.

(not(P(x)) And R(z)) are both *false*, the **If-Then-Else** expression evaluates to *false*. We also note that the terms **(P(x) And Q(y))** and **(not(P(x)) And R(z))** can never be *true* at the same time, since that would be a contradiction. Hence, the possible test scenarios for the **If-Then-Else** expression are as enumerated as below :

$$\mathbf{not((P(x)\ AND\ Q(y)))}\quad \mathbf{AND}\quad \mathbf{(not(P(x))\ AND\ R(z))}$$
$$\mathbf{(P(x)\ And\ Q(y))}\quad \mathbf{AND}\quad \mathbf{not(not(P(x))\ AND\ R(z))}$$

We notice that each of the above test scenarios are in terms of basic logical operators and can be further simplified using De Morgan's laws. We obtain the final test scenarios by recursively applying the test generation rules for the basic operators. Examining the driving and driven values does not help reduce the number of test scenarios any further. This is because, by definition, the operator covers the two cases when the **If** term condition is *true* and *false*. This gives us the list of final test scenarios generated for an **If-Then-Else** expression. These scenarios are summarized below.

$$\mathbf{(P(x) = false)}\quad \mathbf{AND}\quad \mathbf{(R(z) = true)}$$
$$\mathbf{(P(x) = true)}\quad \mathbf{AND}\quad \mathbf{(Q(y) = true)}$$

3.2.2 Relational Operators

In order to obtain the test conditions for relational operators, we obtain test values that cause the relational expression to evaluate to *true* or *false* as shown in Table 3.7. For the logical operators since the values that the operand could take were just *true* or *false* all the values of operands that evaluate the expression to *true* or *false* were generated. In the case of a relational expression it is impossible to get all the values of operands for which the expression will evaluate to *true* but it is possible to get a class of values. The following table gives the list of test scenarios generated when a relational expression is evaluated to *true* or *false*.

P(x)	Q(y)	SCENARIOS
Non-Controllable	Non-Controllable	1. P(x) = false AND Q(y) = false 2. P(x) = false AND Q(y) = true 3. P(x) = true AND Q(y) = true
Non-Controllable	Controllable	1. P(x) = false AND Q(y) = false
Controllable	Non-Controllable	1. P(x) = true AND Q(y) = true
Controllable	Controllable	1. P(x) = false AND Q(y) = false 2. P(x) = false AND Q(y) = true 3. P(x) = true AND Q(y) = true

Table 3.5: Scenarios for IMPLIES operator when expression is evaluated to true

We find that considering the driving and driven variables does not help reduce the number of test scenarios. This is because, regardless of whether the operands are driving or driven variables, the expression evaluates to *false* only when the left operand is *true* and the right operand is *false*. So the tool generates the single test condition as indicated above.

The IF-THEN-ELSE Operator

The IF-THEN-ELSE expression can be transformed to an expression in terms of the basic logical operators. The expression *IF P(x) THEN Q(y) ELSE R(z)* equals (P(x) AND Q(y)) OR (NOT(P(x)) AND R(z)). Using this transformation, we obtain the truth table for the **If-Then-Else** operator shown in Table 3.6.

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 3.6: Truth table for the If-Then-Else operator

We observe from the truth table, that when the term **(P(x) And Q(y))** and the term

P(x)	**Q(y)**	**SCENARIOS**
Non-Controllable	Non-Controllable	1. P(x) = false AND Q(y) = true 2. P(x) = true AND Q(y) = false 3. P(x) = true AND Q(y) = true
Non-Controllable	Controllable	1. P(x) = true AND Q(y) = false
Controllable	Non-Controllable	1. P(x) = false AND Q(y) = true
Controllable	Controllable	1. P(x) = false AND Q(y) = true 2. P(x) = true AND Q(y) = false 3. P(x) = true AND Q(y) = true

Table 3.4: Scenarios for NOR operator when expression is evaluated to false

enumerated below :

$$(\mathbf{P(x) = false}) \quad \mathbf{AND} \quad (\mathbf{Q(y) = false})$$

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

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

By taking into consideration whether the expressions *P(x)* and *Q(y)* are controllable or non-controllable, we can remove some scenarios generated. We know that if the left operand is *true*, then the right operand has to be *true* in order to evaluate the IMPLIES expression to *true*. Hence, when P(x) is controllable and Q(y) is not, we consider only the test scenario where P(x) and Q(y) are both *true*. Similarly, if the right operand is *false*, then the left operand has to be *false* to evaluate the IMPLIES expression to *true*. Hence, if only Q(y) is controllable, we consider only the test scenario where both P(x) and Q(y) are *false*. The final test scenarios after the elimination process are summarized in Table 3.5.

When the IMPLIES expression is *false*, the only test scenario generated is :

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

NOR expression to *true* are :

$$(\mathbf{P(x) = false}) \quad \mathbf{AND} \quad (\mathbf{Q(y) = false})$$

and the list of possible test scenarios to evaluate the NOR expression to *false* are :

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

In the case when the NOR expression is evaluated to *true*, we find that considering the driving and driven variables does not help reduce the number of test scenarios. This is because, regardless of whether the operands are driving or driven variables, the expression evaluates to *true* only when both the operands evaluate to *false*. So the tool generates the single test condition as indicated above.But considering the the driving and driven variables helps reduce the number of test scenarios, when the NOR expression evaluates to *false*. If P(x) can be controlled, then we are only interested in the test scenarios where P(x) is *false*, because a disjunction is always *true* whenever P(x) is *true* regardless of value of Q(y). A similar argument holds when Q(y) is *controllable* and P(x) is *non-controllable*. The final test scenarios generated when the NOR expression is *false* are summarized in Table 3.4.

The IMPLIES Operator

The IMPLIES expression can be written in terms of basic logical operators. The expression *P(x) => Q(y)* equals NOT(P(x)) OR Q(y).

We observe that the IMPLIES expression evaluates to *false* when the left operand P(x) is *true* and the right operand Q(y) is *false*. The expression evaluates to *true* for all other values of P(x) and Q(y). The list of possible scenarios when the expression evaluates to *true* is as

where P(x) is *true*, because the NAND expression is *true* whenever P(x) is *false*, regardless of value of Q(y). A similar argument holds for Q(y) depending on whether y is driving or driven variable. Table 3.3 gives the scenarios generated when the NAND expression is *false*.

P(x)	**Q(y)**	**SCENARIOS**
Non-Controllable	Non-Controllable	1. P(x) = false AND Q(y) = false 2. P(x) = false AND Q(y) = true 3. P(x) = true AND Q(y) = false
Non-Controllable	Controllable	1. P(x) = false AND Q(y) = true
Controllable	Non-Controllable	1. P(x) = true AND Q(y) = false
Controllable	Controllable	1. P(x) = false AND Q(y) = false 2. P(x) = false AND Q(y) = true 3. P(x) = true AND Q(y) = false

Table 3.3: Scenarios for NAND operator when expression is evaluated to true

When the NAND expression is *false*, the only test scenario that is generated is :

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

We find that considering the driving and driven variables does not help reduce the number of test scenarios. This is because, regardless of whether the operands are driving or driven variables, the expression evaluates to *false* only when both the operands evaluate to *true*. So the tool generates the single test condition as indicated above.

The NOR Operator

The NOR operator functions as the negation of OR operator. The NOR expression evaluates to *true* only when both the operands are *false*, and the expression evaluates to *false* when either or both of operands are *true*. Hence the list of possible test scenarios to evaluate the

The XNOR Operator

The XNOR expression evaluates to *true* when both its operands are *true*, or when both operands are *false*. From this property, we enumerate the list of possible test scenarios when the XNOR expression is *true*.

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

When the XNOR expression is *false*, the scenarios generated are :

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

We then examine the driving and driven values for XNOR expression. We derive that if only one of the predicates is *controllable* and the other *non-controllable*, or when both are *non-controllable*, we need to consider both the test scenarios.

The NAND Operator

The NAND operator is the negation of AND operator. Hence, the NAND expression evaluates to *true* when either or both the operands are *false*. When the expression is *true* the list of possible test scenarios is as enumerated below :

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

Considering the driving and driven variables reduces the number of test scenarios when (i) P(x) is *non-controllable* and Q(y) is *controllable*; and (ii) P(x) is *controllable* and Q(y) is *non-controllable*. If P(x) can be controlled, then we are only interested in the test scenarios

The NOT operator is a unary operator and by definition the result of NOT operation on a expression *NOT(P(x))* is evaluated to *true* only if *P(x)* is *false*. The scenarios generated when the expression is *true* or *false* are given below.

When the expression is evaluated to *true* the only scenario generated is :

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

When the expression is evaluated to *false* the only scenario generated is :

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

Considering the driving and driven variables does not help reduce the scenarios.

The XOR Operator

The XOR operator evaluates to *true* only when one of the operands is *true* and the other is *false*. The scenarios generated when the XOR expression is evaluated to *true* are :

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

From the definition, we know that for an XOR expression to evaluate to *true*, the two predicates should have contrasting values. Based on this property, we conclude that if only one of the predicates is *controllable*, while the other is *non-controllable*, or when both are *non-controllable*, we need to consider both the test scenarios.

When the XOR expression is *false*, the scenarios generated are :

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

the expression is *true*, the maximum possible scenarios that can be generated are :

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

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

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

If P(x) can be controlled, then we are only interested in the test scenarios where P(x) is *false*, because a disjunction is always *true* whenever P(x) is *true* regardless of value of Q(y). A similar argument holds when Q(y) is *controllable* and P(x) is *non-controllable*. The final test scenarios generated when the OR expression is *true* are summarized in Table 3.2.

P(x)	**Q(y)**	**SCENARIOS**
Non-Controllable	Non-Controllable	1. P(x) = false AND Q(y) = true 2. P(x) = true AND Q(y) = false 3. P(x) = true AND Q(y) = true
Non-Controllable	Controllable	1. P(x) = true AND Q(y) = false
Controllable	Non-Controllable	1. P(x) = false AND Q(y) = true
Controllable	Controllable	1. P(x) = false AND Q(y) = true 2. P(x) = true AND Q(y) = false 3. P(x) = true AND Q(y) = true

Table 3.2: Scenarios for OR operator when expression is evaluated to true

The only scenario that is generated when the expression is *false* is :

$$(\mathbf{P(x) = false}) \quad \mathbf{AND} \quad (\mathbf{Q(y) = false})$$

This is because, regardless of whether the operands are driving or driven variables, the expression evaluates to *false* only when both the operands evaluate to *false*.

The NOT Operator

When the value of expression is *false*, the list of possible test scenarios are :

$$(\mathbf{P(x) = false}) \quad \mathbf{AND} \quad (\mathbf{Q(y) = false})$$

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

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

Considering the driving and driven variables reduces the number of test scenarios when (i) P(x) is *non-controllable* and Q(y) is *controllable*; and (ii) P(x) is *controllable* and Q(y) is *non-controllable*. If P(x) can be controlled, then we are only interested in the test scenarios where P(x) is *true*, because the AND expression is *false* whenever P(x) is *false*, regardless of value of Q(y). A similar argument holds for Q(y) depending on whether y is driving or driven variable. Table 3.1 gives the scenarios generated when the AND expression is *false*.

P(x)	**Q(y)**	**SCENARIOS**
Non-Controllable	Non-Controllable	1. P(x) = false AND Q(y) = false 2. P(x) = false AND Q(y) = true 3. P(x) = true AND Q(y) = false
Non-Controllable	Controllable	1. P(x) = false AND Q(y) = true
Controllable	Non-Controllable	1. P(x) = true AND Q(y) = false
Controllable	Controllable	1. P(x) = false AND Q(y) = false 2. P(x) = false AND Q(y) = true 3. P(x) = true AND Q(y) = false

Table 3.1: Scenarios for AND operator when expression is false

The OR Operator

By definition, the outcome of an `OR` operation evaluates to *true* if either of the terms i.e either of operands in the `OR` expression is *true*. In a disjunction of form `P(x) OR Q(y)`, when

both the predicates are driven variables, all the test scenarios are considered to evaluate the expression.

Every expression in a Rosetta facet is a boolean expression. We generate the different values for the predicates in the expression by evaluating the expression to *true*. If the predicates are complex, the rules are applied recursively and values for simple predicates, not involving logical operators are generated, by evaluating them to *true* or *false* depending on the value assigned to them.

The specific algorithm for test case generation is explained below in detail for each of the logical operators.In the subsections P(x) and Q(y) are assumed to be predicates over the parameters x and y. We consider single variable predicates for sake of simplicity. In reality, predicates can have any number of variables, and will be handled properly by our tool. The various values generated for the simple predicates P(x) and Q(y) when the expression is evaluated to *true* and *false* are shown. If P(x) and Q(y) are complex predicates, then the process is continued recursively.

The AND Operator

By definition, the outcome of an `AND` operation evaluates to *true* if both the terms i.e operands in the `AND` expression are both *true*. If the AND expression is a pre-condition, we can eliminate the test scenario because we have no values to observe for the expression.

The value of an expression depends on the requirements. If the value of an expression is *true* then the only test scenario generated for the `AND` expression is:

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

We find that considering the driving and driven variables does not help reduce the number of test scenarios. This is because, regardless of whether the operands are driving or driven variables, the expression evaluates to *true* only when both the operands evaluate to *true*. So the tool generates the single test condition indicated above.

tree. Then the corresponding test generation algorithm is used for each boolean operator depending on its boolean value. The boolean value assigned to the operator determines if the test scenarios should be generated to evaluate the predicate to *true* or *false*. The following subsections of this chapter explain how the scenarios are generated for all the logical and relational operators in Rosetta. The expression from the object model is always evaluated to *true* and the other sub-expressions are evaluated to *true* or *false* depending on the requirement.

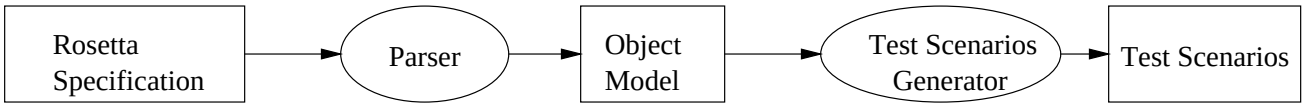

Figure 3.1: Test Scenario Generation Process

Throughout the thesis, we use the terms *driving* and *driven* values to describe the input and output parameters respectively. Inputs to the component are the parameters that drive the system and the expected outputs corresponding to the driving values (inputs) are the driven parameters. A predicate is a *controllable predicate* when the values of variables that make up the predicate are driving values or can be controlled. A predicate is a *non-controllable predicate* when the values of variables that make up the predicate are driven values or cannot be controlled but are obtained by driving the system with the input parameters.

3.2.1 Logical Operators

Test scenarios are generated for expressions with logical operators using the canonical definition of each operator. First, all the test cases possible for an operator are listed according to the truth-table definition of the operator. Test cases for which the expression evaluates to *true* or *false* depending on requirements, are selected. The notion of driving and driven values is then used to filter out redundant test cases, if any from the selected ones. If all the predicates of an operator are driving values, then we can eliminate the test scenarios because we have no output values to observe for these driving values. On the other hand, if

in the implementation are detected when, given a test case that satisfies the input criteria, the generated output does not satisfy the acceptance criteria. The input criterion may be as specific as actual input values or as general as a range of values that the input parameters can take. The acceptance criterion specifies a value or range of values for the output parameters that are acceptable to the system. The test scenarios, therefore, provide classes of tests to be performed on the system.

3.2 Generation of test scenarios from specification

Since all terms inside the facet are boolean expressions, test scenarios are generated by evaluating the expressions to *true* or *false*, depending on the requirements of a system. The expressions are formed by joining variables and literals of various types with operators. We use the multi-condition strategy as proposed by Myers [9] to generate the specific test conditions from the expressions. According to this strategy, enough test scenarios should be generated for the expression under test to take all possible values. For example, consider an arithmetic expression on having an integer outcome. According to the strategy used, test scenarios generated should be such that the outcome of the expression ranges from the smallest integer to the largest integer. In case of boolean expressions the whole range of outcomes is covered in the set of *true* and *false*.All Rosetta expressions are boolean expressions. In the case where the outcome of an expression is known, the test case are generated to satisfy this value. For example, if the known outcome of an expression is *true*, we generate test scenarios, such that operands in the expression take all possible values, that make the expression *true*.

The process of test scenario generation is summarized in Figure 3.1. The present Rosetta suite has a parser that parses the specification and generates an Object Model for the specification. The expressions obtained from the object model are in the form of a tree structure. If the expression is a formula then each node of the tree corresponds to operator and the branches represent operands or predicates. Test scenarios generation is done by simple traversal algorithm to determine the operator and its boolean value, at each level of the

Chapter 3

Generation of Test Scenarios

In this chapter we explain the methodology used to generate test scenarios from a Rosetta specification. Test scenarios are generated using the specification based testing techniques. The techniques used are discussed in detail in this chapter. We discuss the various operators supported by Rosetta, and the technique used to handle each of these operators. We then illustrate the test scenario generation process with simple examples.

3.1 Test Scenarios

The methodology that we use is an extension of implementation based testing techniques applied to formal languages. Richardson, O'Malley and Tittle [3] have proposed a methodology to generate test scenarios from specifications in any axiomatic specification language. This methodology is an extension of implementation-based testing strategy applied to formal specifications. It proposes that, a test scenario should be generated for each pre-condition and post-condition pair. As explained in section 2.4, a test scenario consists of an input criteria and an acceptance criteria. The pre-condition defines constraints upon the inputs to the component, hence the input criteria is obtained from the pre-condition. Similarly, the post-condition defines the constraints on the generated output for input values that satisfy the input criteria. The acceptance criteria is hence obtained from the post-condition. Errors

In general *test requirements* are properties that must be satisfied during testing. For example, reaching statements are the requirements for statement coverage. In our work we refer to *test requirements* as the requirements on test implementation and test coverage. They are used to constrain the otherwise infinite number of test cases that fall within a particular test scenario. In this work, the test requirements are specified by the user.

A *test case* or *test vector* is a general software artifact that includes test case input values, expected outputs for the test case, and any inputs that are necessary to put the system into the state that is appropriate for the test input values [1]. The specific values for the input are obtained from the test requirements provided by the user. By evaluating the test scenarios for a particular value of input, a description of output parameter for that specific value of input is obtained. It is to be noted that the specific values of input should satisfy the pre-condition.

Abstract test vectors are vectors that are generated from the user defined requirements and the test scenarios. They are in Rosetta format and cannot be directly used on any kind of implementation of a component where the specification has been written and hence we call them *abstract test vectors*. In this work, we translate abstract test vectors to WAVES format.

Concrete test vectors have the same information as the abstract test vectors, like input values and expected output values. They are obtained by simple translation of *abstract test vectors*. The concrete test vectors are specific to some testing software. In our work, we translate the abstract test vectors to a WAVES format.

The return state test conditions constrain values of both the input and output parameters. In their paper *Structural specification based testing using ADL*, [12] Chang, Richardson and Sankar describe a method to develop test cases from Assertion Description Language (ADL) specifications. They claim that their method complements traditional code based techniques like statement coverage and branch coverage.

In their paper *Test templates: A specification-based testing*, [11] Stock and Carrington describe a new method for specification-based testing using *test-templates* and *test template frameworks*. The *test template frameworks* deals with functional unit testing, that is, defining and deriving tests from operations defined in the specification. Their main strategy is partition testing but the framework is flexible enough to incorporate other strategies too. The framework also provides a platform to apply, classify and compare the results due to the different testing strategies.

Most of the algorithms for generation of test vectors, are aimed to detecting errors in the implementation of a system with the main focus being functional testing. These algorithms do not consider whether the implementation conforms to its specification. The approach presented in this thesis is different. We generate test scenarios/vectors to test whether the implementation conforms to the formal specification of a system, without considering internal details of the implementation.

2.4 Terminology

A *test scenario* or a *test condition* is a set of boolean conditions that provide constraints on the values of input and output variables. The input criterion is a constraint on the input parameters; it may be as specific as actual input values or as general as a range of values that the input parameters can take. The acceptance criterion is a constraint on the output parameters, and it specifies a value or range of values for the output parameters that are acceptable to the system. The test scenarios, therefore, provide *classes of tests* to be performed on the system. Throughout the thesis we use the terms *test scenario* or a *test condition* interchangeably unless explicitly stated otherwise.

class. The test-model referred above is a finite-state machine that composes of a set of states and a set of transitions among the states. Each state is obtained through the state-space partition of a class. Each transition consists of a method that might change the value of an object from source state to target state. Partition analysis to get the states mainly involves the reduction of predicates into a disjunctive normal form, that gives a disjoint partition of value spaces. While performing the partition of state-space of a class, the input space of each method is also to be partitioned. While generating the test cases from the test-model, either the finite-state testing techniques could be used or the data-flow testing techniques could be used.

The main difference between the approach of the authors and the approach we take is as follows; While they define the test scenarios as a sequence of operations and the expected result, we define them as built of input and acceptance criterion, that is the set of input data and the expected results. Hence, while they use the information from the test-model that they build, to obtain the test scenarios, we use the expressions in the facets to obtain ours.We obtain all this information from the user defined requirements while they derive the test cases from the test-model that they build by partition analysis.

2.3.3 Other related work

In this section, we give a brief overview some of the other work being done in the area of specification-based testing. In their paper, *Automating Test Case Generation from Z specifications using Isabelle*, [10] Helke, and Santen describe a way of automating test case generation from Z specifications. They make use of the Isabelle theorem prover to automate the generation of test cases. The Isabelle theorem prover is used to generate the disjunctive normal form from the Z schemas, and using it to obtain and simplify the test cases.

For Assertion Definition Language (ADL), two kinds of test conditions can be derived. They are, *call-state test conditions* and *return-state test conditions*. Call state test conditions are derived from the input conditions and they describe constraints on the input parameters while the return state test conditions are derived from both input and output conditions.

user could provide some initial vectors in the requirements file that can be evaluated to reach a particular state. The methodology used to generate the test cases that conform to the full predicate criteria described in their work is similar to the methodology we use. Different combinations of inputs are generated such that each predicate P in the Rosetta facet results in *true* or *false*, depending on the requirement. They generate the test cases that make the transition *valid* and *invalid*. Similar to our work an algorithm is run to identify and remove the redundant test cases. We do not have the *Transition-pair Coverage* because we are concerned with only a single transition and not all the transitions.

2.3.2 Testing object-oriented programs from formal specification

Object-oriented software testing is generally performed at four levels:

1. Method-level testing - testing of an individual method in class.
2. Class-level testing - testing of interactions among components of an individual class.
3. Cluster-level testing - testing of interactions among objects.
4. System-level testing - testing the external inputs and outputs visible to the users of a system

Objects in object oriented programming are based on data abstraction, that can be defined at two different levels. At the more abstract level, known as the specification level, objects are represented by object types. They specify only the expected interfaces and behaviors of object. At more concrete level objects are represented by classes.

In their paper *Test case Generation for Class-Level Object-Oriented Testing*, [2] Tse and Xu present a new technique to generate the test cases for class-level object-oriented testing. It integrates the testing techniques based on algebraic specifications, model-based specifications, and finite-state machines. In their approach, test cases contain various sequences of method invocations with various test data. The test cases are generated from a test-model to test the state-dependent scenarios on the behaviors of interactions among methods of a single

2.3 Related Work

2.3.1 Generating Tests from UML Specifications

In the paper, *Generating Tests from UML specifications*, [1] Offutt and Abdurazik describe a way of generating test data from UML state charts. UML state charts are based on finite state machines and are used to represent the behavior of an object. The state of an object is the combination of all attribute values and objects the object contains. This is similar to what a state of an object means in Rosetta. The state of a facet is the combination of all the parametized and internal variables. UML categorizes transitions into five types *high-level*, *compound*, *internal*, *completion* and *enabled* transitions.The authors describe different levels of testing and they are as follows

1. *Transition Coverage:* The test set T must satisfy every transition in the specification graph.
2. *Full Predicate Coverage:*For each predicate P on each transition,T must include tests that cause each clause c in P to result in a pair of outcomes where the value of P is directly correlated with the value of c.
3. *Transition-pair Coverage:*For each pair of adjacent transitions, T contains a test that traverses the pair in sequence.

The main entry point has three objects

1. a UML specification parser
2. a full predicate test case generator
3. a transition-pair test case generator.

A prefix generation algorithm is used in test case generation algorithms to create the values necessary to reach a particular state. This is similar to the approach used in our work. The

by revealing specification failures. Fault-based testing techniques postulate that faults exist in the implementation and select test cases to detect those faults if they exist. Even if the specification is correct, these hypothesized faults may still be indicative of faults that might be introduced in the implementation.

The following testing techniques treat the specification as an oracle to be violated.

3. *Oracle/Error-Based Testing*

   Oracle/Error-based testing applies error-based techniques to the implementation while explicitly attempting to force a violation of a oracle in a specification. A form of oracle/error based testing is incompleteness testing where test cases are selected for portions of input domain left unspecified.

4. *Oracle/Fault-Based Testing*

   Oracle/Fault-based testing focuses on detecting specific faults in the implementation by transferring resulting errors to violate the specification. For a potential fault, we select the revealing condition up to a post condition that references an error and attempts to force one of the source or the variant to satisfy the post condition and the other to violate it.

We use the error-based testing technique mentioned above for the generation of test scenarios from the specification. We assume the Rosetta specification that we have is correct and then use generated test scenarios to generate the test vectors for particular values of input. Test vectors provide us with expected outputs that are compared to the outputs of implementation in the hope of uncovering errors. Our terminology is slightly different from that used in their work. They use the term *test-case* as we define the term *test-scenarios* whereas we use the term *test-case* to mean specific values given to the input variables.

2.2.2 Approaches to Specification based testing

In their paper "Approaches to Specification-Based Testing",[3] Richardson, O'Malley and Tittle propose different approaches to specification based testing by extending a wide variety of implementation based testing techniques to be applicable to formal specification languages. In the same paper the authors describe these approaches for the Anna and Larch specification languages.

According to the authors a test case consists of an input criterion and an acceptance criterion. The input criterion may be as specific as actual test data or as general as a condition on the input domain or output range. The output criterion is a condition describing whether or not execution of this test case is acceptable or whether an error has been revealed.

The test data selection can be classified as error based or fault based. Error based techniques are used to reveal specific types of errors. Fault based testing selects test data that focus on detecting particular types of faults where a fault is a mistake in the source code. The following approaches are applicable to algebraic specifications, pre/post conditions, assertions and procedural design languages. The specification-based testing approaches discussed below select test cases as they are applied to pre/post condition pairs.

1. *Specification/Error-Based Testing*

   Specification/Error-based testing attempts to detect errors in the implementation that derive from misunderstanding the specification or errors in the specification. The general approach is to select the test cases that would detect potential errors in the representation. Symbolic evaluation of specification partitions the input space. The form of a specification partition depends on the type of specification language. Evaluation of a pre/post-condition language, partitions the domain by the pre-conditions while evaluation of a algebraic specification language selects specification paths partitioned by the procedural constructs.

2. *Specification/Fault-Based Testing*

   The goal of Specification/Fault-based testing is to detect faults in the specification

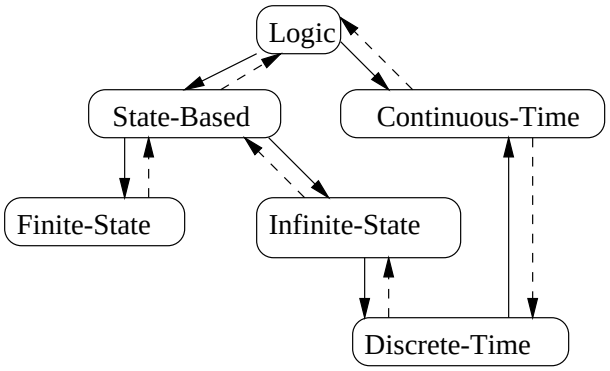

Figure 2.2: Domains and Interaction

program is written.

The relationship between specifications and testing includes ways that information derived from a formal or informal specification can be used to assist testing but also other issues such as testability. This includes both dynamic testing, where we execute the implementation under test (IUT) and compare the output with that expected and static testing where we compare the specification and the code without actually executing the IUT. Specification based verification uses conventional testing methods where the program under test is repeatedly stimulated and outputs and/or other values are observed and compared against expected values. The benefits of using a formal specification language when verifying are :

1. Test results are compared against a formal specification, not a human interpretation of a specification document.

2. Powerful regression testing. Other methods tend to fall apart when the implementation is changed. This happens because these methods compare the test against a "golden" test, representing a particular correct test run.

Most of the research conducted in the area of specification-based testing has been limited to theory and there is a need for practical solution that addresses this concern.In our work, we attempt to leverage some of this research and provide a tool that automatically generates test vectors from high level formal specifications written in the Rosetta specification language.

```
Facet schmidt_trigger(
                     input_voltage:: in real;
                     output_value :: out bit) is
                   /* state_variable */
           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)
                       ENDIF)
             ENDIF;)
                   /* second post-condition */
     post2: (output_value' = b');
end schmidt_trigger;
```

Figure 2.1: Schmidt Trigger Rosetta Specification

2.2.1 Specification based testing

The term specification-based testing as used in the document, usually refers to testing based solely on the specification, i.e testing not using any information from the implementation.Specification based testing offers many advantages in software testing. The specification of a software product can be used as a guide for designing functional tests for the product. The specification precisely defines fundamental aspects of a software, while more detailed and structural information is omitted. Formal specifications provide a simpler, structured and more formal approach to the development of functional tests than using non-formal specifications. One advantage of producing tests from specifications is that the tests can be created earlier in the development process, and be ready for execution before the program is finished or the system is built. Additionally, when tests are generated the test engineer can find inconsistencies in the specifications, allowing the specifications to be improved before a

The basic unit of specification is termed a *facet*. A facet is a model of a component or system that provides information specific to a domain of interest. To support heterogeneity in designs, each facet may use a different domain model to provide domain-specific vocabulary and semantics. A facet is a parameterized construct to encapsulate all Rosetta definitions from basic unit specifications through components and systems. The definition of facets is achieved by directly defining model properties or by combining previously defined facets. The former technique allows users to choose a specification domain and specify properties that must hold in that domain. The latter technique allows users to select several models and compose a system model that consists of elements from each facet domain. Vectors are generated for the facets that are defined using the former technique. The form of a Rosetta specification can be understood by examining the Schmidt trigger functional specification facet from Figure 2.1. The specification body is opened by the `begin` keyword immediately followed by the specification domain. Three terms are provided where one of them is the pre-condition and the other two terms are post-conditions.

Different domains are discrete time, continuous time, finite state, performance constraints, and, most recently testing domains. Figure 2.2 shows a collection of domains defined for requirements and constraint modeling. Solid arrows in the figure represent homomorphism while the dashed arrows represent incomplete interactions. Rosetta allows designers to develop and integrate specifications from multiple design domains using *interaction*. Description of domains and interactions is beyond the scope of this thesis.

2.2 Background on Specification Based Testing

Much research has been done in the area of specification-based testing, and the significance of test generation from formal specification has gained wide acceptance in the industry and the software engineering community as a whole. We use these research efforts as the basis of our test generation methodology. In this section, we discuss some approaches to specification-based testing.

Chapter 2

Background and Related Work

In this chapter we give a brief description of Rosetta as a specification language and how it is used to specify complex systems in an abstract way. We also discuss specification based testing and compare it with the implementation based testing. The different approaches to specification based testing techniques are also discussed. Finally, we discuss some related work being done in specification based testing.

2.1 Rosetta Specification Language

Rosetta is a System Level Design Language used to design systems at higher levels of abstraction [6]. As system complexity increases, the need to specify a system at higher abstraction levels becomes important. Rosetta provides some important design characteristics: (i) Integration of information from multiple heterogeneous sources; (ii) The ability to work at high levels of abstraction with incomplete information; and (iii) Support for model composition. By allowing the user to define and compose models, it allows flexibility and heterogeneity in design modeling and supports meaningful integration of models representing all aspects of system design. Developed from concepts from the formal verification and functional programming communities, Rosetta allows designers to develop and integrate specifications from multiple design domains.

itations of current implementation with the future work to be done in order to remove the limitations in the current work is given.The main contribution through this thesis is a methodology to generate test scenarios and abstract test vectors from Rosetta specifications.

function and on execution of this matlab function, the actual input values are obtained.

The requirements could also contain some initial vectors so that the system reaches a particular state before each test case. In this case before driving the system for a particular test case, the system is driven to reach the required state by going through the set of vectors provided by the user.It is to be noted that the values of only the input variables can be controlled by the user. Hence test requirements are specified only for the input variables.

The test vectors that are generated from the scenarios and the user-defined requirements are in our own generic format with sufficient information to translate them into a format specific to some testing software. In our case, we translate the vectors into WAVES format.

1.4 Organization of the thesis

Chapter 2 provides background on the Rosetta language. It gives an overview of how the Rosetta facets provide the declarative specification capability. We also discuss some of relevant research work done in the area of specification-based testing and test case selection.

In Chapter 3, details on the strategies used to generate test scenarios from the expressions that occur in the Rosetta facets are provided. We discuss the various operators supported by Rosetta, and the algorithm used to generate test conditions for each of these operators. The implementation of these strategies is explained with some examples.

Chapter 4 describes the generation of abstract test vectors and concrete test vectors. We first explain how the requirements specified in Rosetta are used to know what values the variables are likely to take and how many tests need to be performed for adequate coverage. We then explain how test scenarios are combined with the test requirements to generate abstract test vectors. Finally, the translation of abstract test vectors, represented in Rosetta, into WAVES test vectors for VHDL simulation is described.

We demonstrate how the test scenarios and the test vectors given the test requirements are generated for some Rosetta specifications like alarm clock, satellite communication and Schmidt trigger in chapter 5. In Chapter 6, a summary of our contributions and the lim-

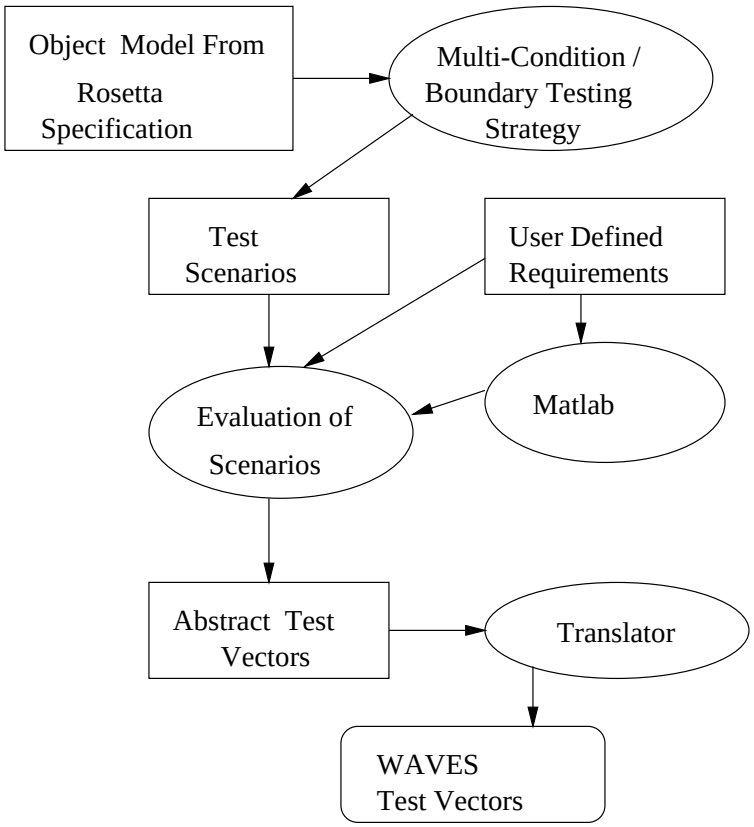

Figure 1.2: Test case Generation Information flow

A test scenario is a set of boolean conditions that are constraints on the values of input and output parameters. A number of strategies can be used to generate the scenarios for the expressions in the facet. We use the multi-condition strategy for the logical expressions. For the relational expressions the boundary testing strategy is used.

The scenarios generated and the user defined test requirements are used to generate the abstract test vectors. The test requirements specify the coverage requirements of a tester, and are used to restrict the number of test vectors that are generated.In addition to the test requirements the boundary testing strategy is also employed to determine the test cases.The number of test cases generated depends on the test coverage required by the user.

When the requirements are specified for some property of input rather than the input values, the user has to provide information about the generation of actual input values. This information has to be specified in Rosetta facet. The Rosetta facet is translated to a matlab

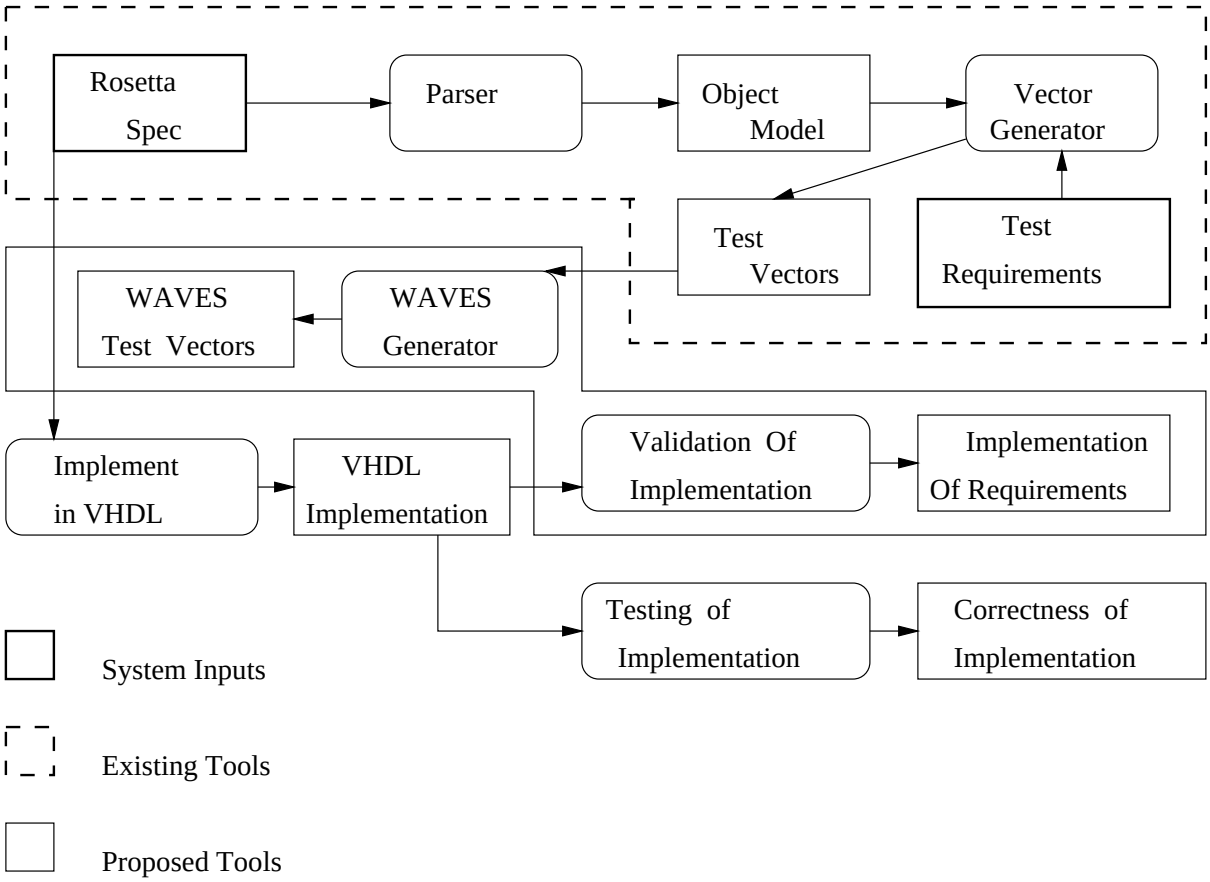

Figure 1.1: Testing of implementation against the specification

test requirements given by the user are specified for some property of input.The generation of specific test cases from the test scenarios given the test requirements by the user is also addressed.

1.3 Proposed Solution

The task of generating the test vectors is divided into three distinct phases - generation of test scenarios, generation of abstract test vectors from the scenarios, and translating the abstract vectors obtained into WAVES [4] format. Figure 1.2 gives an overview of test vector generation process.

The Rosetta parser builds the object model from the Rosetta specification of a component.

A system can be analyzed formally and its behavior can be predicted even before the system is built. This enables the identification of potential problems very early in the design cycle of a system.

Rosetta [6] is a system level design language used to design systems at higher levels of abstraction. Rosetta models individual facets, that can be combined using boolean operators, to model the complete system. We have established the importance of testing the implementation of a system against the specification. This brings the need for a Rosetta tool that has the capability of using a specification to generate test cases. A *test case* includes the values of input variables and the expected output when the implementation is run using the values of input variables.The system is driven with selected input test data, and the output of simulation is monitored and verified against the expected output. If the two outputs match, then the test is a success, else it indicates an error in the system. Figure 1.1 explains in detail the approach we take to test if the implementation conforms to the specification of a system. In the figure, the implementation of the system is assumed to be in VHDL, but could theoretically be any simulation language. The correctness of an implementation is tested by comparing the expected output values generated from the specification with the outputs obtained from the implementation.

1.2 Problem statement

This thesis addresses the problem of automatically generating the test cases for systems specified in Rosetta. This is done by generating the test scenarios from the specification. The test scenarios give us a set of values for the input variables. Unfortunately, this set of values could be huge and it is not feasible to exhaustively test the system. Hence, it is necessary to know what values the variables are likely to take and how many tests need to be performed for adequate coverage. These are called the test requirements and are provided by the user. These test requirements given by the user might also include some initial vectors that drive the system to a particular state before the test case is executed. We also address the problem of generating the actual values for the input variables, when the

been added does not interfere, or cause problems, with previously tested code. Thus black box testing establishes whether the implementation of component satisfies its requirements. It makes use of specification based testing strategies.

In implementation based testing techniques, the intended behavior of implementation is overlooked. The disadvantage of this approach is that while it may help verify that the implementation is correct, it does not ensure that the correct system has been implemented. This can be overcome by using specification based testing techniques. The relationship between specifications and testing includes ways where information derived from a formal or informal specification can be used to assist testing but also other issues such as testability. This includes both dynamic testing, where we execute the implementation under test (IUT) and compare the output with that expected and static testing where we compare the specification and the code without actually executing the IUT.

Specification based verification uses conventional testing methods where the program under test is repeatedly stimulated and outputs and/or other values are observed and compared against expected values. The advantages of specification based verification are:

1. It helps testing intended behavior as well as actual functionality.
2. It helps to expose any ambiguities or inconsistencies in the specifications.
3. Test cases can be designed as soon as the specifications are complete.
4. Tests are done from a user's point of view

1.1 Motivation

As systems become increasingly complex, the design process effectively uses abstraction to manage the complexity. Such abstraction techniques initially result in a series of operational decomposition techniques, used for small or medium scale systems. For complex systems, declarative specifications are gaining acceptance in the industry. The formalisms provide better understanding of system requirements and requisite rigor for mission-critical systems.

Chapter 1

Introduction

A typical development process for reliable systems consists of cyclic iterations of requirements analysis, system design, implementation, and validation. Such a development process should support validated transitions from requirement specifications to system models and from abstract design specifications to detailed implementation. There are two common methods for validation, namely verification and testing. In the face of complexity as well as acceptance problems in industry as encountered by verification techniques such as model checking and theorem proving, there is increasing need for testing. Traditionally, there are two main approaches to testing: white-box (or structural) testing and black-box (or functional) testing.

White box testing is carried out using implementation-based testing techniques. The control structures of procedural design are used to derive the test cases. These test cases are such that they test each independent path of implementation at least once. They also test the loop and logical-decision constructs along with the internal data structures. Hence, white box testing ensures that the algorithm to build the component has been implemented correctly, in terms of data and control structures. Black box testing is based upon systems level requirements.

Black box testing is used to find whether valid inputs are accepted and the obtained outputs conform to the requirements. Exhaustive black box testing is usually impossible, thus a small set of tests is used. Regression tests should also be run to be sure that any new code that has

List of Tables

List of Figures

Contents

Abstract

Implementation based testing techniques select test data based on the information obtained from the implementation. They tend to ignore the intended behavior of a system and focus on the actual behavior [3]. Specification-based verification uses conventional testing methods where the program under test is repeatedly simulated and outputs and/or other values are observed and compared against the expected values. Selecting test data from the specifications enables testing intended behavior as well as actual functionality [3]. Testing the implementation against the specification ensures that the implementation satisfies its specified functionality. We thus try to ensure that the correct system has been built.

In this thesis, we present a methodology for generating test scenarios and test vectors from Rosetta specifications. The methodology used is derived from specification based testing techniques that are an extension of implementation based testing applied to formal specification languages. This is done by generating test scenarios from the specification giving us a set of values for input variables. Unfortunately, this set of values could be huge and it is not feasible to exhaustively test the system for each of these values. Actual values that an input variable can take are obtained from user defined requirements. These test requirements given by the user might also include some initial vectors that drive the system to a particular state before a test case is executed. When the requirements provided by the user do not specify the actual values for input variables, we use Matlab tool to generate the actual values of input variables.

Abstract test vectors are generated from user defined requirements and test scenarios. These abstract test vectors, represented in Rosetta, can be directly translated into concrete test vectors that are used as specific inputs to simulation runs. The mechanism to translate the generated test cases into a format that is specific to some testing software is also addressed. We transform the abstract Rosetta vectors into WAVES test vectors suite for VHDL simulation.

Acknowledgments

I would like to thank Dr. Perry Alexander, my advisor and committee chairman, for his guidance and support through this project. I am thankful for the suggestions, and freedom that I received during the course of this work. I would also like to thank him for providing me with an opportunity to work under him and for guiding me through my graduate studies.

I would like to thank Dr. Jerry James and Dr. David Andrews for consenting to be on my committee.

I would also like to thank all the members in SLDG group for their support. Thanks to all and sundry who have made my stay and experience in Lawrence a memorable one.

To my parents

Advanced Test Vector Generation from Rosetta

by

Srinivas Akkipeddi

B.E. (Electronics & Communications Engineering),
University of Allahabad, Allahabad, India, 1999

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 requiremet for the degree of Master of Science

Professor in Charge

Committee Members

Date Thesis Accepted