Synthesis and Model Extraction of A Constrained Random Testbench

Thesis Submitted In Partial Fulfillment Of The Requirements
For The Degree Of

Bachelor of Technology
in
Computer Science and Engineering

by

Arnab Sinha

Roll No: 02CS3022

under the guidance of

Dr. Pallab Dasgupta

Department of Computer Science and Engineering
Indian Institute of Technology
Kharagpur
May 2006

Certificate

This is to certify that the thesis titled **Synthesis and Model Extraction of A Constrained Random Testbench** submitted by **Arnab Sinha** to the Department of Computer Science and Engineering in partial fulfillment for the award of the degree of **Bachelor of Technology** is a bonafide record of work carried out by him under my supervision and guidance. The thesis has fulfilled all the requirements as per the regulations of this Institute and, in our opinion, has reached the standard needed for submission.

Dr. Pallab Dasgupta
Dept. of Computer Science and Engineering
Indian Institute of Technology
Kharagpur 721302, INDIA
May 2006

Acknowledgments

This thesis is the result of research performed under the guidance of **Dr. Pallab Dasgupta** at the Department of Computer Science and Engineering of the Indian Institute of Technology, Kharagpur.

I am deeply grateful to my supervisor for having given me the opportunity of working as part of his research group and the huge amount of time and effort he spent guiding me through several difficulties on the way. Without the help, encouragement and patient support I received from my guide, this thesis would never have materialized.

I also acknowledge my senior **Bhaskar Pal** for his encouraging suggestions, sharing the technical skills and the synergy that he brought in our work. My acknowledgments to all the members of Formal-V group, IIT Kharagpur and **Anupam Chattopadhyay** (Institute for Integrated Signal Processing Systems (ISS), University of Technology, Aachen, Germany) for their technical inputs and constant support in my work. Further, I am grateful to my friend Shambaditya Saha (Department of Biotechnology & Biochemical Engineering, IIT Kharagpur) and my parents for their perennial inspiration.

Arnab Sinha
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
May 2006

Contents

1 Introduction **1**
- 1.1 Issues in Formal techniques . . . 2
- 1.2 Issues in Simulation-based techniques . . . 3
- 1.3 Motivation and Objective . . . 3
  - 1.3.1 Automatic Extraction of Assume properties . . . 3
  - 1.3.2 Hardware Accelerated Simulation . . . 5
- 1.4 Work Done . . . 6
- 1.5 Thesis Organization . . . 6

2 The Background **7**
- 2.1 Introduction . . . 7
- 2.2 Relating Constraints with Assume-Properties . . . 8
- 2.3 Assume-Properties . . . 8
  - 2.3.1 A Simple example . . . 9
  - 2.3.2 Constructs in SystemVerilog . . . 9
- 2.4 Constrained Random Test-bench . . . 10
  - 2.4.1 Constructs in SystemVerilog . . . 10
- 2.5 Conclusion . . . 12

3 Hardware Accelerated Constrained Random Test Generation **13**
- 3.1 Introduction . . . 13
- 3.2 The Central Idea . . . 16
- 3.3 Software Preprocessing . . . 18
  - 3.3.1 Computation of Bounds . . . 19
  - 3.3.2 Computation of Boundary Points . . . 20
  - 3.3.3 Strip Computation . . . 21
- 3.4 Synthesis Methodology . . . 23
  - 3.4.1 Synthesis Algorithm . . . 25
- 3.5 Tool Architecture . . . 27
- 3.6 Results . . . 27

3.7 Conclusion 29

4 Automatic Extraction of Assume Properties 32
4.1 Introduction 32
4.2 A Simple Example 34
4.3 Formal Modeling 38
4.4 The Methodology 39
4.4.1 Constrained Environment Automaton Extraction 40
4.4.2 Generation of Assume Property 41
4.4.3 Proof of Correctness 42
4.5 Tool Architecture 42
4.6 Results 43

5 Conclusion and Future Work 45
5.1 Conclusion 45
5.2 Future Directions 46

A SystemVerilog Constrained Random Testbench 47

List of Figures

1.1 The Overall Design Flow 2
1.2 Extraction of assumptions from the test-bench 4
1.3 Hardware Acceleration 5

2.1 Example System 9

3.1 The Overall Idea 17
3.2 Stages of Solution Space Entailment 18
3.3 Constraints plotted as the polygon in X-Y plane 19
3.4 Bounding-box plotted as a rectangle in the X-Y plane 20
3.5 Strip Computation 22
3.6 Synthesis Methodology 24
3.7 Synthesis Block Diagram 27
3.8 Tool Architecture 28
3.9 Synthesis Result 29

4.1 Example System 34
4.2 Example Violation Scenario 35
4.3 State machine extracted from the test bench 36
4.4 Tool Architecture 42

List of Tables

3.1 Results on IBM CoreConnect Bus protocol 28
3.2 Results on AMBA AHB Bus protocol 29
3.3 Results on Hardware resources 30

4.1 Selective test-bench constructs 39
4.2 Results on standard Bus components 44

Abstract

Simulation and formal verification methods are the two important pillars of SoC verification. Unfortunately, both have certain demerits. Simulation techniques suffer due to its non-reliability and unmanageable worst-case time-complexity. In order to reduce this simulation-time, both verification community is now focusing on emulation techniques. But that requires mapping of traditional constrained random test-benches (which usually runs in software) to hardware. On the other hand, the latter requires formal specification of the interface behavior, but this is a non-trivial task. Moreover, validation engineers have significant experience in writing test benches that are also models of the environment, but test-benches are not written in formal languages. This project addresses two problems related to constrained random test benches – (a) they tend to be the main bottleneck in hardware accelerated simulation, and (b) they cannot be directly used by a formal verification tool.For the first problem, this project presents a methodology and a tool *to synthesize constraints in hardware*, which is the core problem in migrating test benches into hardware. For the second problem, the project examines the question - *Can we extract the relevant assumptions from a well structured constrained random test bench?* We present a methodology and a tool to accomplish this task. We report our experience with complex test benches for the ARM AMBA and IBM CoreConnect protocols.

Chapter 1

Introduction

Verification is one of the single biggest challenges in the design of system-on-chip (SoC) devices and re-usable IP blocks. Traditional validation methods have been unable to keep pace with the ever-increasing size and complexity of designs. Despite the introduction of successive new verification technologies, the gap between design capability and verification confidence continues to widen. In practice there are two ways in which design verification is done today. These are *Formal techniques* and *Simulation-based techniques*. In both these forms, formal properties specify the correctness requirements of the design, and the goal is to check whether a given implementation satisfies the formal properties. In formal techniques, it is checked whether all possible behaviors of the design satisfy the given properties. Whereas in simulation-based approach, the properties are checked over a simulation run - the verification is thereby confined to only those behaviors that are encountered during the simulation.

The steps and issues in the current design verification flow are given as follows.

1. The VLSI design flow descends from one abstraction layer to another as shown in Fig 1.1. At each level, the specifications are refined. At each step, verification is essential (a)to check consistency among the behavior of the design and the specification at that level, and, (b) to check consistency among the specifications of current and previous levels.

2. The design architect conceives the design as a set of abstract blocks interconnected through some simple *glue logic*, that together achieves the functionality of the design. Large functional blocks are similarly decomposed into smaller blocks. This process of decomposition continues until the functionality of each block is simple enough to be coded as a single (unit level) module. After the unit level designs are done, these are integrated to build the final RTL implementation.

As the simulation time is one of the major bottleneck, verification community

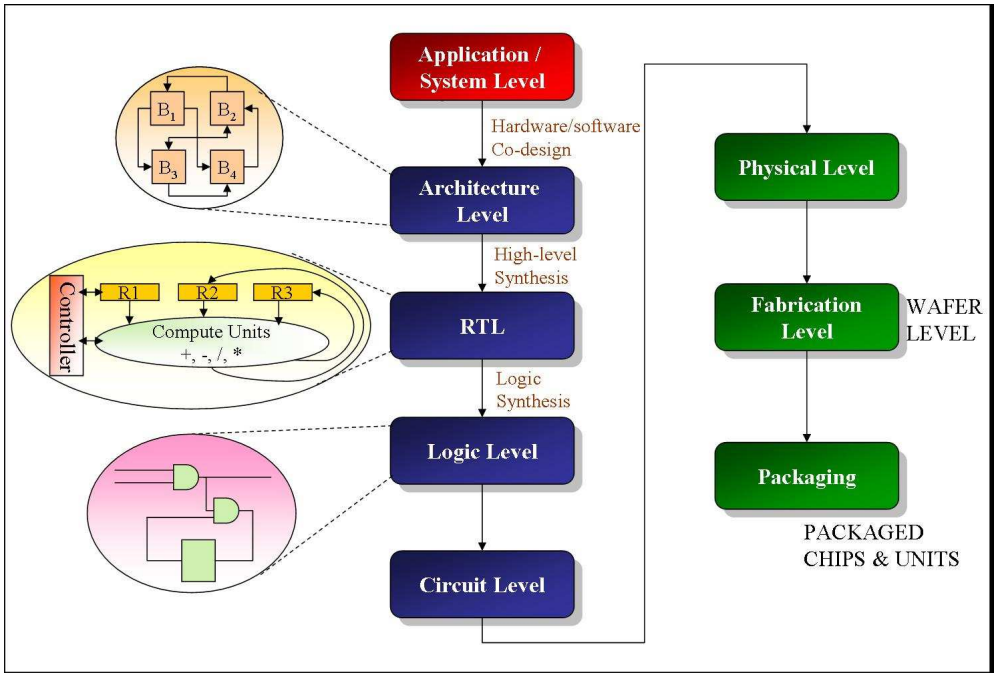

Figure 1.1: The Overall Design Flow

has started migrating from exhaustive simulation to coverage driven simulation techniques. The coverage driven verification approaches ensure that the test-vectors appear according to some probability distribution. This methodology is preferred over conventional methods since more frequent vectors have more occurrence probability. Moreover in order to track the corner cases validation engineers are nowadays preferring constrained random test-benches since generating directed test-cases is once again another non-trivial task. Thus, constrained random test-benches are gaining rapid popularity in the current validation flow. But there are certain issues in the applicability of the constrained random test-benches, which are as follows.

1.1 Issues in Formal techniques

At the unit level, where the module size is small and manageable, formal tools can handle the capacity. This is a big advantage, since the task of achieving a similar degree of confidence through simulation can require thousands of test vectors. However, in order to prove a property in a module, it needs to incorporate a model of the environment. By default, this is the most general environment, thus it is free to behave as it pleases. However, in reality, modules are usually designed under assumptions in *constrained random test-benches* about the environment. Thus an environment shouldn't behave in an arbitrary manner. In absence of these assump-

tions, the formal tool can provide a counter-example, which is actually not a 'true one' for that particular module. Modeling these assumptions is a non-trivial task and requires severe manual intervention. On the other hand validation engineers have significant experience in writing test benches that are also models of the environment. So the question becomes - *Can the relevant assumptions for the module under test be extracted from the constrained random test bench itself?*

1.2 Issues in Simulation-based techniques

Simulation typically takes place in several stages in the design hierarchy. These include, unit level validation for individual modules, block level validation for collection of modules, and system level validation for the integrated design. In each of these cases, thousands of simulation vectors are created to achieve a reasonable confidence. For system level validation, simulation runs into months. Thus the main bottleneck in simulation is the time that it takes. On the other-hand the reality is that for large designs it is the only solution. One of the popular trends to minimize the simulation time is emulation i.e. mapping the test-bench to hardware. In traditional emulation techniques, the DUT is mapped into an FPGA while the test-bench is driven from the software. The constraints present in the *constrained random test-benches* hinder this process of mapping of test-bench, which is a non-trivial task. So the question here we examine - *Can we map the constraints into hardware to reduce the simulation time?*

In summary, constrained random test-benches pose two separate problems in either of the accepted verification methodologies.

1. Can we extract the global as well as local assumptions from a test-bench (to aid the *formal analysis*)?

2. Is it possible to map the constraints in manageable hardware (to reduce the *simulation* time)?

1.3 Motivation and Objective

We have investigated into the problems raised by the constrained random test-benches in the previous section. In the following subsections we categorically explore the issues hidden in them.

1.3.1 Automatic Extraction of Assume properties

In order to model check a component in isolation one needs to incorporate a model of the environment interacting with the component. By default, this is the most

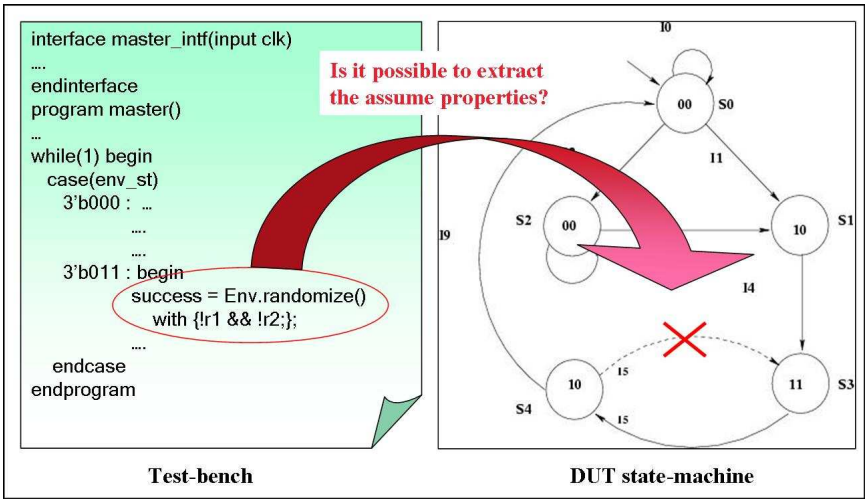

Figure 1.2: Extraction of assumptions from the test-bench

general environment, an environment that can evoke, in any order, any action of the interface between the two. The underlying assumption is that the environment is free to behave as it pleases, and that the component will still satisfy the property. However, in reality, components are usually designed under assumption about the environment. Thus an environment can't behave in an arbitrary manner. In the world of model checking, this concept has given rise to the assume-guarantee style of reasoning, where the model of the environment is restricted by the assumptions provided by the developer. However, if these assumptions are not applied during formal analysis, the formal tool may extract a false counter-example scenario while proving a particular property. That counter-example may be generated for a scenario in which the environment has behaved in an arbitrary manner. This kind of error-reporting provide a false insecurity to the designer.

Thus one of the main challenges in developing a formal verification test plan for an open system (such as a DUT that interacts with its environment) is to formally model the environment of the DUT. The most abstract form of environment modeling is through assume properties, but this may not be adequate since the environment is typically required to satisfy different constraints at different states of the protocol between the DUT and the environment. More refined techniques include the formal specification of an interface automaton, but this is a non-trivial task. There has been some works on assumption generation. In [2], [3], the tool generates the weakest assumption on which a property holds. However, this approach is not helpful in verifying the property under a given environment. In our work, *we aim to aid the formal analysis by taking information from an existing test-bench.* The motivation comes from the fact that validation engineers have significant experience in writing the test-benches and test-benches are also models of the environment.

In recent times, well defined guidelines are being proposed for writing structured constrained random test benches. Moreover, current test-bench modeling languages provide construct to model the state specific constraints (assumptions) in the test-bench. Thus one of the main challenges now is to examine the question - Can we extract the constrained interface automaton from the test bench?

1.3.2 Hardware Accelerated Simulation

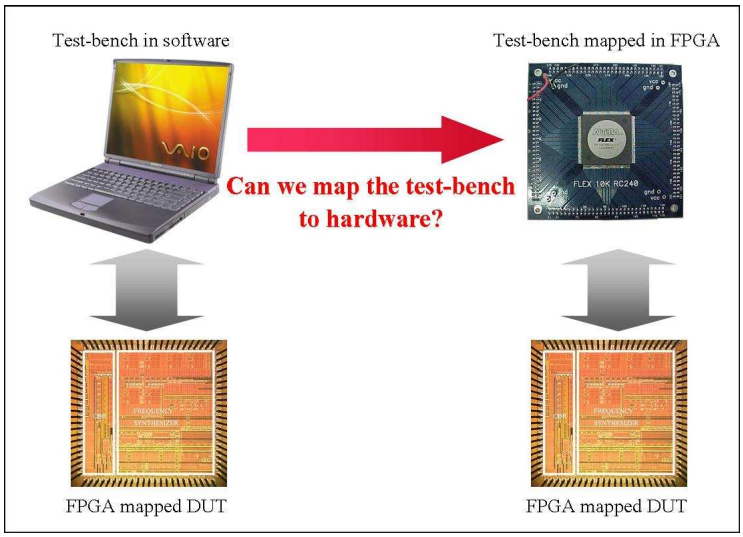

Figure 1.3: Hardware Acceleration

Lately, hardware-acceleration [20, 7] is increasingly being used to reduce the simulation time. The DUT is mapped into an FPGA which helps to reduce the simulation time [32], since the actual hardware runs without any overhead on simulators. However, for large and complex designs, today, the test environment consisting of the models of the other components in the system is often just as large as the DUT. In such cases the test-bench simulation overhead becomes the dominating factor in hardware accelerated simulation. As the test-bench coding style and the associated high-level constructs are not conducive to logic synthesis, the task of mapping a high-level complex test-bench into hardware is non-trivial. The central problem of this synthesis task is to generate constrained random tests in hardware. Although there exists many well-known and efficient constraint solvers in software domain, synthesizing these solvers is non-trivial because most of them use non-deterministic and back-tracking search algorithms [26] while searching for the solution. Using such algorithms in hardware will defeat the central purpose of mapping the test-bench into hardware. To the best of our knowledge, there exists no tool that deterministically generates constrained random tests in hardware. Other tasks in this problem

include providing a *semantic-preserving* translation of the high-level language constructs to equivalent synthesizable logic.

1.4 Work Done

The main contributions of this work are as follows.

1. *Automatic Extraction of Assumptions from the constrained random test-bench:* We present a style for modeling constrained random test-benches. We show that the proposed style can be easily implemented in the current generation test-bench modeling languages like SystemVerilog, OpenVera etc. We present methodology and a prototype tool for automatic extraction of the environment automaton and the relevant constraints from the test-bench, and then mapping these constraints as *assume properties* in the DUT state machine.

2. *Hardware Accelerated Simulation:* We have proposed a deterministic, non-backtracking algorithm to generate random stimuli in accordance with the given constraints. The methodology is sound and complete. The algorithm is fair, thus generates every solution with a non-zero probability.

1.5 Thesis Organization

The thesis is organized as follows. It is described in the following 5 chapters. We give a brief overview of each chapter as follows.

Chapter 2: In this chapter we establish the connection between constraints in constrained random test-bench as well as the assume properties and the various language constructs in modern test-bench languages for expressing the constraints as well as assertions with assume property.

Chapter 3: This chapter describes in detail the methodology of transforming the linear constraints from software test-benches to FPGA.

Chapter 4: The methodology for extraction of assume properties and the environment FSM from the test-bench is described in this chapter.

Chapter 5: Lastly, we conclude our work summarizing the contribution, mentioning the future direction of our research effort in brief.

Chapter 2

The Background

2.1 Introduction

In this chapter we will be discussing the necessary background to describe the work carried out. Simulation is by far the most accepted form of verification in the industry right now. Formal verification techniques are slowly penetrating wherever the systems are fault-critical such as space and cardiac applications.

Simulation techniques are traditionally used since it has a less space-requirement and it requires no added expertise in logic and combinatorics. Unfortunately, the major drawback lies in the fact that, it is expensive in terms of time and also it does not guarantee that the system is bug-free. Certain corner cases always tend to remain hidden. One possible way to overcome that vulnerability is to apply exhaustive set of vectors. But, that is usually an infeasible idea. Suppose, a 32-bit wide address bus is to be verified. It is absolutely impractical idea to apply 2^{32} vectors.

People next tried to apply *directed* test cases. But the two main reasons which made this idea infeasible too are the following.

- Large number of directed test-cases.
- Generating and applying these directed test-cases is a non-trivial task.

These observations has led the industry to move from traditional *directed* [1] to *constrained randomized* [2, 3] approaches. Using this approach, scenarios are generated in an automated randomized fashion under the control of a set of rules, or *constraints* specified by the user. By building randomization into the types of scenarios that are created, generated tests are much more likely to hit corner cases and thereby find more design bugs.

On the other side of the story, formal verification techniques ensure exhaustive checking of the design under test (DUT). But this merit is largely marred by the

state-space explosion. The number of states in the system tend to grow exponentially with the complexity of the system. For example in the previous example of the address-bus, there would be 2^{32} states. This explosion in number of states pose as a challenge before the available space.

2.2 Relating Constraints with Assume-Properties

Observing both the pros and the cons of simulation and formal verification techniques, validation community has gone for certain trade-offs. Instead of applying test-cases exhaustively, the industry and academia prefer randomized test-vector application. In randomized test-plan all the vectors have a non-zero probability of occurrence. But again the same problem arises. *How to track the corner cases?.* One possible approach is to generate directed test cases. But that is always not possible and even if possible, it is definitely a non-trivial task. So, recent shift is towards the constrained random test-bench.

Moreover, the component modules of a System-on-Chip(SoC) are designed expecting certain protocol compliant vectors from the environment. Moreover, we know that, while model-checking [12] the design, the formal property checker should know the appropriate assumptions. Otherwise, it might produce refuting paths which might never exist in practice. Hence, the property verifier should be aware of the *assumptions* made regarding the environment. These assumptions are known as *assume properties.*

Intuitively, the *constraints* in constrained random test-benches (required by simulation tools) and the necessary *assume properties* in the formal model (required by formal verification tools) are equivalent. These constraints provide the assumptions regarding the environment, since the environment is restricted to produce those constraint-compliant vectors only. Note that in our work, we have automatically extracted these constraints (assuming the test-bench to be a complete one) to get the assume-properties.

In the following sections we will be discussing the language constructs of assume-properties and constrained random test-bench.

2.3 Assume-Properties

Usually, in a system-on-chip unit-level modules are separately tested. It is practical to consider that the unit-level modules are not exposed to any arbitrary set of inputs but protocol compliant vectors. Hence, even if we apply randomized test-plan on the DUT, we need to constrain the set of vectors within feasible range of operation. And this leads to constrained random test-benches. The modern verification languages like SystemVerilog provide the support for writing constrained random test-benches.

2.3.1 A Simple example

We consider the verification of a slave device, S, which supports only 2 beat transfer in a Bus protocol. The interface between S and a requesting master device M is shown in Figure 2.1.

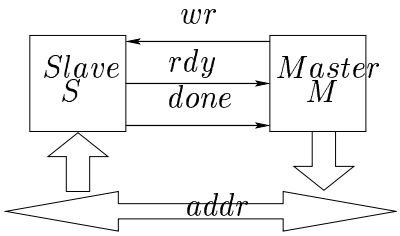

Figure 2.1: Example System

S initially remains in the *idle* state, in which its outputs *rdy* and *done* are low. A master device indicates its intent to write by asserting the *wr* signal and floating an address in the address lines, *addr*. The slaves are memory mapped and identify themselves from their respective address ranges.

For example, whenever S finds that the address floated by the master is outside its address range (say, 0 to 499), it returns to its *idle* state. If the address is within its range, then there are two cases. If the device is not ready to accept the transfer then it lowers the *rdy* signal to delay the transfer. Otherwise it raises the *rdy* signal and completes the 2 beat transfer by asserting the *done* signal in the next two consecutive cycles.

However, the above mentioned functionality can only be satisfied under the following assumption - *Once a transfer is acknowledged, the transfer address must be in the range (0 to 499).* Otherwise, the properties will refute each other. So the test-bench should be written accordingly. In the following subsection we will show how to write the properties as well as the constraints in the SystemVerilog language.

2.3.2 Constructs in SystemVerilog

For automated verification, the specifications should be clearly written in some verification language. We choose SystemVerilog here since it is one of the most popular languages among the validation engineers to express their intent.

The properties of the slave S described in previous subsection can be expressed by the following SystemVerilog assertions.

```
property P1;
@(posedge clk) (wr && rdy && !done) |->
(##1 (rdy && done) ##1 (rdy && done));
```

```
endproperty

property P2;
@(posedge clk)
      (rdy && !(addr >=0 && addr < 500))
                |-> (##1 (!rdy && !done);
endproperty
```

Moreover, the assumption should be written as follows,

```
sequence Adrange;
@(posedge clk) (addr < 500 && addr >= 0);
endsequence

property AP1;
@(posedge clk) (wr && rdy && !done) |->
(Adrange ##1 Adrange ##1 Adrange);
endproperty
```

The properties written in this fashion aids in semi-formal verification of the system. At a high level of abstraction the design intent is typically expressed in terms of several high-level correctness requirements. Specification of the exact Boolean functionality of the implementation may neither be practical, nor desirable at the high-level. Therefore properties allow us to express a more relaxed version of the specification, covering the critical correctness requirements of the design, but leaving room for design optimization by not specifying the exact the Boolean functionality. It helps in capturing the essential elements of the design intent in an accurate and non-ambiguous way.

2.4 Constrained Random Test-bench

As already described, the constrained random test-benches help in narrowing down the focus of verification. Suppose in the example described in subsection 2.3.1, we might be interested to observe the behavior of the slave while the *addr* is within $[0, 200]$. We show the test-bench writing style in the following subsection

2.4.1 Constructs in SystemVerilog

For simulation purpose the state-machine of the test-bench with the above-mentioned constraint is specified as follows,

```
//-- Interface declaration
interface master_intf (input clk);
   wire wr, rdy, done;
   wire [9:0] addr;
   //-- Clocking Block
   clocking CBMst @(posedge clk);
     output wr, addr; input  rdy,done;
   endclocking
endinterface

//-- Test-Bench
//-- Some `define s that would be used in
// modeling the env state machine
`define S2 3'b010

//- Input condition
`define I2 !tI.CBMst.rdy && !tI.CBMst.done
`define I3 tI.CBMst.rdy && !tI.CBMst.done

program master (master_intf intf);//-test-bench

  class Master; //-Master device
    rand bit r;//-Internal rand bits
    rand bit [9:0] addr;
  endclass

  //-- virtual interface
  virtual master_intf tI;
  //-- Master instance
  Master Env = new();

  //-- test bench execution starts here
  initial begin
     tI = intf; MasterExec ();
  end
  task MasterExec ();//-Main task definition
    bit [1:0] env_st;//-state encoding
    integer success;

    env_st = `S0; //-Init state

    while (1) begin
```

```
      case(env_st)//-state-mc encode
                  ...
                  ...
       `S2 : begin //- state S2

          //-Inline constraint (C)
         success = Env.randomize() with
             {addr>=0 && addr<=200;};  //-- Constraint: 0 <= addr <= 200

         if (`I2) begin
          tI.CBMst.wr <= 1'b1;
          tI.CBMst.addr <= Env.addr;
          env_st = `S2; //- next state S2
         end

         else if(`I3) begin
          tI.CBMst.wr <= 1'b1;
          tI.CBMst.addr <= Env.addr;
          env_st = `S3; //- next state S3
         end
        end
                ...
                ...
      endcase
     end
    endtask
endprogram //-- End of test-bench
```

2.5 Conclusion

So we have identified the equivalence between constraints and the assume properties and also their constructs in SystemVerilog language. Hence, in order to verify in both the fronts (viz. simulation and formal verification) we are done if we can extract the constraints along with the state-machine of the environment to formally verify the DUT (when the test-bench is given to us). This is precisely what we have described in chapter 4. Moreover, these constraints are solved by some internal solver while simulation. In order to pace up the process through emulation (described in the chapter 1), we need a methodology to map the constraints into synthesizable subset of some hardware description language. This methodology has been described in chapter 3.

Chapter 3

Hardware Accelerated Constrained Random Test Generation

3.1 Introduction

In recent times, there has been a significant change in the verification paradigm from a directed approach to a coverage driven randomized one, primarily intended towards a more complete and exhaustive validation process. The main challenge in developing a randomized test bench for the Design Under Test (DUT) is in constraining its behavior to the correct protocol between the DUT and its environment. An unconstrained random verification process may not generate protocol-compliant test scenarios. Thus the test-bench obeys some *constraints* while generating random tests. *Constraint specification* is an integral part of any randomized test generation framework.

In recent times, hardware verification languages (Open-Vera [5], Specmen-Elite [4], SystemVerilog [31]) support constructs for defining constraints. These can be global constraints as well as constraints applicable to a particular state. The associated tools have in-built constraint solver engines to generate a solution satisfying the constraints.

Lately, hardware-acceleration [20, 7] is increasingly being used to reduce the simulation time. The DUT is mapped into an FPGA which helps to reduce the simulation time [32], since the actual hardware runs without any overhead on simulators. However, for large and complex designs, today, the test environment consisting of the models of the other components in the system is often just as large as the DUT. In such cases the test-bench simulation overhead becomes the dominating factor in hardware accelerated simulation. As the test-bench coding style and the associated high-level constructs are not conducive to logic synthesis, the task of mapping a

high-level complex test-bench into hardware is non-trivial.

The central problem of this synthesis task is to generate constrained random tests in hardware. Most previous research were on using FPGAs as accelerators for solving SAT problems. In [33], a machine based on FPGAs have been implemented using a tree search with forward checking for SAT problems. In [28], a machine based on a dynamic variable ordering heuristic have been modeled. Zhong et al. [34] developed a design for SAT problems utilizing the DavisPutnam algorithm as well as an unimplemented design which used nonchronological backtracking [35]. An incomplete SAT solver based on the GSAT algorithm [30] has been proposed in [22], but the design was not tested on hardware. Lee et al. [25], described an architecture for an implementation of the GENET algorithm using FPGA devices. In [26], a field programmable gate array (FPGA) implementation of a co-processor which uses the WSAT algorithm to solve Boolean satisfiability problems is presented.

An important limitation of all of the implementations described above is that synthesizing these solvers in hardware with limited resources is non-trivial because most of them use non-deterministic and back-tracking search algorithms while searching for the solution. Using such algorithms in hardware will defeat the central purpose of mapping the test-bench into hardware. To the best of our knowledge, there exists no tool that deterministically generates constrained random tests in hardware.

In this project we address the following objective. *Given a set of variables and a set of constraints over these variables, we aim to generate random valuations of these variables in hardware, using limited resources, that satisfies the given constraints.* In its general form the problem is very hard. While there is no restriction on the total number of variables, our methodology solves a system of constraints, where each constraint has at most two variables. Thus in this work we limit ourselves to solving a valuation for a set V of N variables ($N \geq 2$) where each of the constraints are *linear* and involve any of the *two variables* from V. For example, if $V = \{x, y, z\}$ i.e., $N = 3$, the constraints can be as follows.

$$C_1 : x + 2y \leq 5$$

$$C_2 : y + 3z \leq 6$$

$$C_3 : 3x + z \geq 1$$

From our experience in modeling the Verification IPs for several standard bus protocols (e.g. AMBA AHB [19], IBM CoreConnect [23], PCI [24]), we have seen that the associated test-bench constraints can be represented using this model. In recent times, standard test-bench modeling languages (SystemVerilog, OpenVera) support modeling of user-defined constraints in the test-bench. For example, let us consider a Bus which includes a Master device and two Slave devices. The Master, when required, can initiate transfers (read or write) to the Slave devices. The address map of the Slave is [0, 1023], where $Slave_1$ has the range [0, 511] and

$Slave_2$ has the range [512, 1023]. Suppose we wish to verify the Slaves. The address driven in the Bus consists of (*base*, *offset*). A test-bench that models the Master, can drive address to any slave device. However, for slave specific test-cases, it needs to constrain the values of *base* and *offset* in order to provide valid addresses. A SystemVerilog constraint random test-bench fragment is given as follows.

```
‘define AnySlave 3’b001 //defining a state

program master;
 ...
 case (MasterState)
  ...
  ‘AnySlave: begin
    ...
    success = randomize()
    with {base >= 0 && offset >= 0 &&
          (base + offset) <= 511 &&
          (base + 2 * offset) <= 1023 &&
          (base + 2 * offset) >= 512 &&
          (base <= offset);};
  end
 endcase
endprogram
```

Note that any values of *base* and *offset* satisfying these constraints constructs a valid address for the slaves.

The proposed methodology is performed in two steps. In the first step we take the input test-bench constraints and pre-compute the solution region and entail this in some well-defined data structure. For example, the solution region in the above Bus protocol in entailed by the lines:

$$C1 : \text{offset} \geq 0$$

$$C2 : \text{base} + \text{offset} \leq 511$$

$$C3 : \text{base} + 2 * \text{offset} \geq 512$$

$$C4 : \text{base} \leq \text{offset}$$

This processing is done in software. In the second step, we use the stored information to generate a valuation of the random inputs in hardware. The pre-computation and entailment of the solution region makes the search algorithm deterministic, non-backtracking, synthesizable and workable within limited hardware resources.

In short the contributions are:

- A deterministic, non-backtracking algorithm to generate random stimuli in accordance with the given constraints.
- A sound and complete methodology.
- A fair random generator that generates every solution with a non-zero probability.

The project is organized as follows. We present the overall idea around suitable examples in Section 3.2. Section 3.3 presents the software preprocessing phase. Section 3.4 describes the synthesis methodology with suitable examples. Section 3.5 presents the overall architecture of the tool. It also describes how to integrate our framework with the existing verification flow. Section 3.6 presents results on standard Bus protocols and some other circuits.

3.2 The Central Idea

Given a set of linear constraints, each having no more than two variables, our solution space represents a convex region in the N-dimensional hyperspace. Our goal is to randomly choose points in that space.

In Fig 3.1 we illustrate the central idea of our approach with an example. We have constraints in 3 variables x, y and z. The 3D model represents the solution space.We take the projection of the multi-dimensional space into 2D-planes. In Fig 3.1 **GHIJ**, **MNPQ** and **ABCD** are the projections in XY, YZ and ZX planes respectively. Within the bounds of x (i.e. $[x^{lower}, x^{upper}]$), we first select a random value(call it x_0). This restricts the bounds of the other variables. So we get a refined bound for y (i.e. $[y^{lower}, y^{upper}]$ from XY-plane) and z (i.e. $[z^{lower}, z^{upper}]$ from XZ-plane). Now choose a random value for y within the refined bound (say y_0). This again might refine the bound of successive variables (i.e. z in our case). At this stage z only needs to be assigned a random value. Choose a random value (say z_0) within $[z^{lower}, z^{upper}]$. Note that every time a random value is assigned to any k-th variable the given problem is reduced to a problem of $(n-k)$ variables. More importantly, the choice of the value of the k-th variable is done in such a way that guarantees the existence of a solution for the reduced problem involving $(n-k)$ variables.

By virtue of projections, our target in hardware is to get a random value for a constrained variable from the projected polygon in any given plane. Essentially, the hardware should be simple enough to churn out random values in a stipulated time frame, otherwise the whole purpose of hardware acceleration would be defeated. For this, suitable data-structures are required to store the information obtained from each plane. We partition our methodology into two halves - software preprocessing

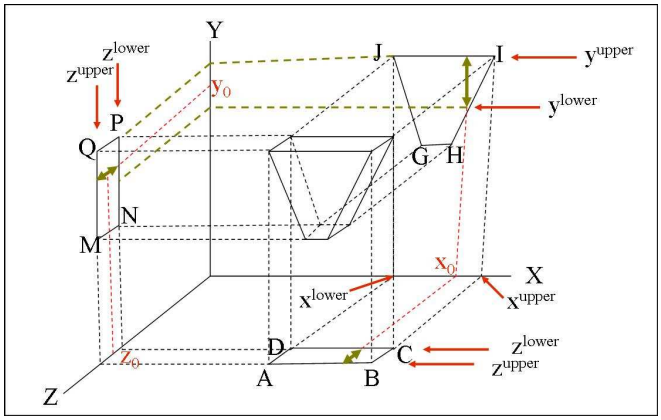

Figure 3.1: The Overall Idea

of the 2D data we have from each plane, followed by hardware synthesis of the oracle targeted to generate valid assignments to the set of constrained variables.

In Section 3.3 we show the extraction of the necessary constraints' information in simple data-structures through strip-computation. Next we use the synthesis algorithm (given in Section 3.4) to integrate the obtained data-structures, and iteratively generate the random values for all the variables to get the overall solution (as explained earlier with reference to Fig 3.1). The software preprocessing (given in the Section 3.3) is applied to all the 2D planes for efficient storage of the constraint information. For the ease of presentation, we discuss it only for the XY-plane.

- The problem $\boldsymbol{C}$ in the 2D XY-plane can be defined as

$$\boldsymbol{C} =< V, C >$$

where, V is the set of constraint variables and C is the set of constraints.

$$V = \{x, y\}$$

$$C = \{C_1, C_2, \ldots, C_m\}$$

- The problem is modeled as a *LP model* (linear programming problem) where

$$C_i : (a_{i1}x + a_{i2}y)\ \theta\ a_{i3}$$

where

$$\theta \in \{\geq, \leq\},\ \ i = 1 \ldots m$$

- Our goal is to generate a synthesizable hardware in some HDL (which is verilog in our case) which has non-zero probability of generating all the *integer solutions* satisfying the constraints in the solution-space.

- We will demonstrate our approach through the following running example.

$$\tilde{C} = \langle \{x, y\}, \{C_1, C_2, \ldots, C_5\} \rangle$$

where

$$C_1 : x + y \geq 1.0$$

$$C_2 : 0.2x + 0.2y \leq 1.0$$

$$C_3 : y \leq 2.5$$

$$C_4 : x - y \leq 1.0$$

$$C_5 : -x + y \leq 1.0$$

Note that we restrict the test-bench designer to model the test-bench constraints in accordance to the above model. We believe that in-spite of this restriction, the constraint specification language is quite rich in practice - a fact that we demonstrate by modeling the AMBA AHB and IBM CoreConnect Bus protocols.

3.3 Software Preprocessing

In this section we will show the various stages of entailment of the solution space for a given problem $\boldsymbol{C}$, along with the supporting theorems and their brief proofs. Our goal is to decompose the solution space of each variable into a finite number of entailed regions. The stages of the computation (as given in Fig 3.2) are described in the following subsections.

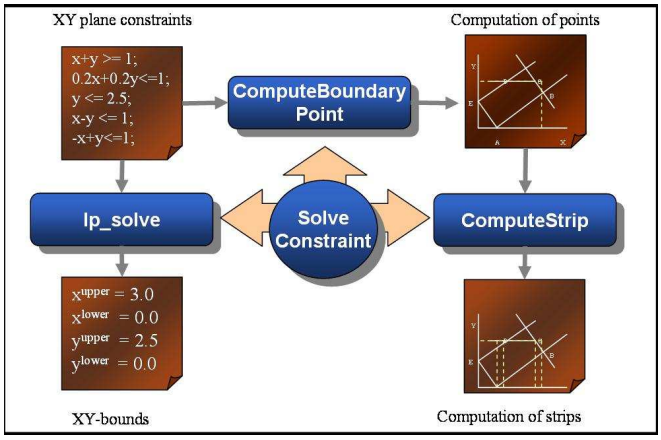

Figure 3.2: Stages of Solution Space Entailment

3.3.1 Computation of Bounds

The linear constraints when represented graphically will give a polygon representing the solution space as shows in Fig 3.3. In order to check the satisfiability of $\boldsymbol{C}$, we pre-compute the minimum area orthogonal rectangle (i.e. edges parallel to either X or Y axes) which contains the polygon. Necessarily the edges parallel to X and Y axes give the feasible bounds for X and Y respectively. We are using a standard solver lp_solve (version 5.1) [36], which uses the SIMPLEX algorithm (a well known linear programming technique) [37]. *Note that we use this solver not to locate the integer solution points, (which is in fact not possible in LP model as it works in the real domain) but to compute the feasible, optimum bounds for x and y.* However, lp_solve does not aim to give bounds on variables. It simply optimizes an objective function. So we run SIMPLEX twice, for each variable (once maximizing, and once minimizing that variable) and this requires a total of $2|V|$ runs. Also, we allow backtracking in this software level. However, all these are done in the pre-processing stage and for only once.

As lp_solve generates real bounds, we transform these to integer bounds. This transformation methodology is given in section 3.3.3. We denote this rectangle as the *bounding-box* ($\boldsymbol{B}$) for the problem $\boldsymbol{C}$. Fig 3.4 shows the bounding-box $\tilde{\boldsymbol{B}}$ for the problem $\tilde{\boldsymbol{C}}$.

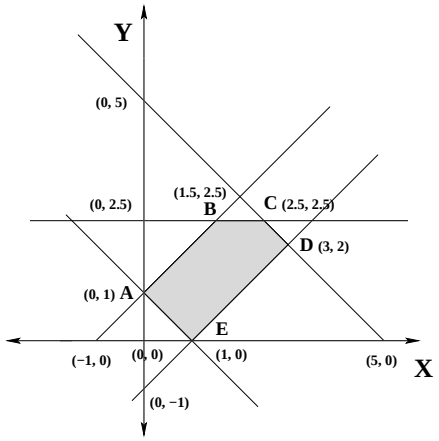

Figure 3.3: Constraints plotted as the polygon in X-Y plane

Theorem 1 *The bounding-box for a polygon is a complete but not sound representation of the solution space.*

Proof: *Completeness: Let $P'(\tilde{x},\tilde{y})$ be a feasible point which lies outside $\boldsymbol{B}$. Then $\tilde{x} \notin [\, x^{lower}, x^{upper}]$ or $\tilde{y} \notin [y^{lower}, y^{upper}]$. As $\boldsymbol{B}$ contains the polygon, it contains the entire feasible solution space. This implies P' is not a feasible point. But that is a contradiction.*

Soundness: As for soundness, it is easy to see that there may exist points inside the rectangle which are not inside the polygon.

Hence the bounds provided by the lp_solve entail a region which is complete but not sound in nature. □

For example, in Fig 3.4, the rectangle **ONPM** generated by the lp_solve contains the solution polygon **ABCDE**. However, the points inside the triangle **DME** are not solution points.

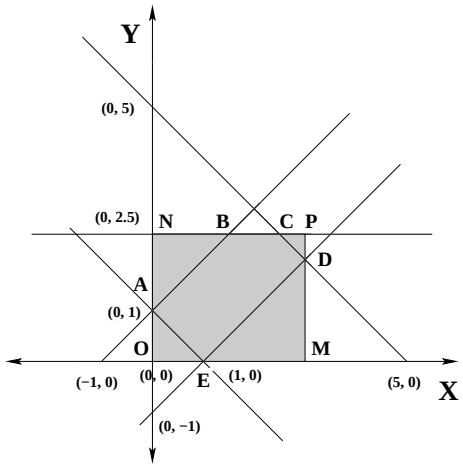

Figure 3.4: Bounding-box plotted as a rectangle in the X-Y plane

3.3.2 Computation of Boundary Points

As the boundary-box is not a sound representation, our next aim is to compute the boundary points on the convex hull. Each constraint divides the plane into two halves. So whenever two linear constraints intersect, there is a intersection of two half-planes. It is a standard result that half-planes are convex and a set of intersecting convex planes give rise to another convex plane [29]. Even if the polygon is not bounded, we truncate it to make it to a large finite value to make it bounded. So essentially we get a bounded convex polygon.

Following the previous result, we can also say that in a convex polygon, each edge contributes at-most two vertices of the polygon. Using these two results, we compute the vertices of the polygon. The algorithm is as follows (We assume that all the constraints are non-redundant).

Algo. 3.3.1 ComputeBoundaryPoint

ComputeBoundaryPoint($\boldsymbol{C}$, $\boldsymbol{B}$)
begin
1: $V(polygon) \leftarrow$ NULL,
where $V(polygon)$ is the set of vertices.
2: Use lp_solve to get the first vertex, P_{active} which
is on edge C_{active}.
3: $P \leftarrow \{P_i \mid C_i$ intersects C_{active}
such that $C_i \neq C_{active}$ $P_i \neq P_{active}\}$
$P_{min} \leftarrow P_i$ where distance between P_i and P_{active}
is minimum and within $\boldsymbol{B}$, $P_i \in P$
4: If $P_{min} \notin V(polygon)$ then
4.1 $V(polygon) \leftarrow V(polygon) \cup \{P_{min}\}$
4.2 $P_{active} \leftarrow P_{min}$
4.3 $C_{active} \leftarrow C_{min}$, where C_{min}
intersects C_{active} at P_{min}
4.4 Goto Step 3
Else Return $V(polygon)$
EndIf
end

For our problem $\tilde{\boldsymbol{C}}$, $V(polygon) =$ { (1.5,2.5), (2.5,2.5), (3,2), (1,0), (0,1) }.

3.3.3 Strip Computation

As the previously computed convex hull can be of arbitrary shape, our next step of computation is to classify the whole polygon region into a collection of trapeziums or triangles. We sort the vertices of the polygon according to their non-decreasing x co-ordinates and drop vertices having same x-coordinate, except one (this step is done by procedure **Sort** in Algo 3.3.2). We define the region $[x_j,\ x_{j+1}]$ as a strip. We denote the strip by $S_{x_j x_{j+1}}$ where $x_j < x_{j+1}$ and no other vertex(v) exists with x-coordinate (x_v) such that $x_j < x_v < x_{j+1}$. We also define the set of constraints which crosses through a strip, as the *overlapping constraint set* for the strip. The algorithm for strip computation is given as follows.

Algo. 3.3.2 ComputeStrip

ComputeStrip($\boldsymbol{C}$, $V(polygon)$)
begin
1: $S \leftarrow$ NULL, where S is set of strips.
2: $StripCount \leftarrow$ **Sort** ($V(polygon)$).

3: for (i=1 to $StripCount$) do
  3.1: $S_{x_i x_{i+1}} \leftarrow \langle \{x_i, x_{i+1}\}, \{c_j \mid c_j \ crosses \ S_{x_i x_{i+1}}\} \rangle$
  3.2: $S \leftarrow S \cup S_{x_i x_{i+1}}$
 Endfor
4: Return S.
end

We show that such overlapping constraint set for a strip always has the cardinality two. This is important as we use this claim during synthesis to generate the solution points.

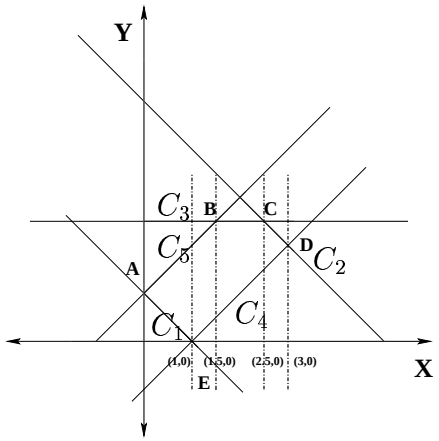

Figure 3.5: Strip Computation

Theorem 2 *The cardinality of overlapping constraint set for any strip in a convex polygon is 2.*

Proof: *The cardinality cannot be 1, otherwise the polygon cannot be bounded. Next, we prove that the cardinality cannot be more than 2 through contradiction. Let the cardinality of overlapping constraint set be more than 2. This is possible if there lies vertices inside the strip or if multiple constraints crosses the strip without intersecting each other. For the first case, let there be a vertex P (x_p, y_p) inside a strip $S_{x_j x_{j+1}}$. Then $x_j < x_p < x_{j+1}$. But that contradicts the definition of a strip. For the second case, the polygon ceases to be a convex polygon. Hence the result.* □

The lp_solve tool can give real bounds to the strips. In those cases, we refine them by taking the inner integer bound. For example, if the bound is $(1.89, 4.65)$, we take $(2, 4)$ as the refined bound. By this method we do not lose any feasible integer point or incorporate any infeasible integer points. In our example the strips are the following (see Fig 3.5).

$$\langle\{0,1\},\{C_1,C_5\}\rangle, \quad \langle\{1,1.5\},\{C_4,C_5\}\rangle,$$
$$\langle\{1.5,2.5\},\{C_3,C_4\}\rangle, \quad \langle\{2.5,3\},\{C_2,C_4\}\rangle$$

We refine the bounds as follows,

$$\langle\{0,1\},\{C_1,C_5\}\rangle, \quad \langle\{1,1\},\{C_4,C_5\}\rangle,$$
$$\langle\{2,2\},\{C_3,C_4\}\rangle, \quad \langle\{3,3\},\{C_2,C_4\}\rangle$$

3.4 Synthesis Methodology

The synthesis algorithm takes the following inputs.

- The lower and upper bounds for all the variables (which we call the maximum feasible bounds).
- The set of strips S in all the 2D planes, computed in the preprocessing step.

With reference to Fig 3.6, we illustrate the synthesis algorithm for generating a valuation of the variables fulfilling the constraints (which is formally stated later in this section). In the example, we work with only three variables (x, y, z). However, the methodology is generic for any finite number of variables.

Input 1: Maximum feasible bounds for x, y and z (For example, for x it is $[x^{lower}, x^{upper}]$).
Input 2: The set of strips in each plane.
Methodology:
Step 1: Choose a random value $x_0 \in [x^{lower}, x^{upper}]$.
Step 2: Refine the bounds for y (given as $[y^{lower}, y^{upper}]$ in plane XY) and z (given as $[z_1^{lower}, z_1^{upper}]$ in plane XZ).
Step 3: The strips S_{y2} (completely covered), S_{y1} and S_{y3} (partially covered) belong to updated $[y^{lower}, y^{upper}]$.
Step 4: Choose a random value $y_0 \in [y^{lower}, y^{upper}]$.
Step 5: Obtain the bounds for z (given as $[z_2^{lower}, z_2^{upper}]$ in plane YZ) and refine z to obtain the final bound (given as $[z^{lower}, z^{upper}]$).
Step 6: Choose a random value $z_0 \in [z^{lower}, z^{upper}]$.

To generate a random number, x, within a given bound we do the following. Let $[x_{min}, x_{max}]$ be the minimum and maximum bounds for x.

1. First, we compute the difference, D = x_{max} - x_{min}. Let us assume that D be of k bits, i.e., $2^k \geq \mathrm{D} \geq 2^{k-1}$.

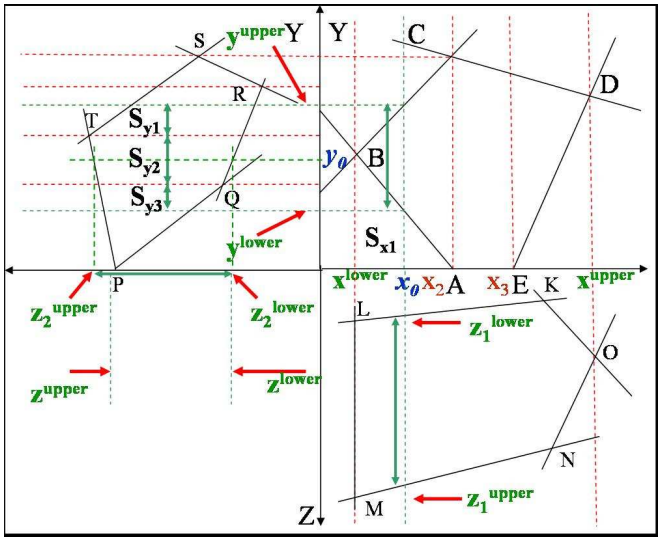

Figure 3.6: Synthesis Methodology

2. We use a k-bit Linear Feedback Shift Register (LFSR) to generate a k-bit random number R. Now, there can be two possibilities.
   (a) R $\leq$ D. In this case, we accept the random number.
   (b) R $>$ D. In this case, we flip the first set bit of R starting from MSB. This ensures that modified R will be less than D.
3. We add R to x_{min} to get the desired valuation for x.

Example 1 *Let x_{min} = 2 and x_{max}=14. Therefore D = 12. The width of the LFSR becomes 4-bits. Suppose the 4-bit LFSR generates R (=4'b1110). So, R > B. Thus we check R from the MSB. It has a '1' at the MSB. So we flip it. The modified R becomes 4'b0110 (decimal 6), which becomes less than D.*

Lemma 1 *The random generator generates every value between x_{min} to x_{max} with a non-zero probability.*

After getting a random number for x (say x_0) between x_i to x_{i+1}, we generate a random value for y. This requires the following steps.

1. Substitute x_0 in the constraints (given in the overlapping constraint set for $S_{x_i x_{i+1}}$), we get the bounds for y. Let the bounds be y^{lower} and y^{upper}. Thus we need to generate a random value between y^{lower} and y^{upper}.
2. We use the same algorithm (given earlier) for random number generation within a given bound, to solve for y.

Example 2 *In our example problem (as given in Section 3.2), variable y has the initial bounds [0, 2.5]. However, if x gets the random value 3 from the LFSR, the associated strip constraints C_4 and C_2 refines the bounds of y as follows.* $yTemp^{lower}$ = 3 − 1 *(as from* C_4*)* = 2 *and*
y^{lower} = *Maximum (0,* $yTemp^{lower}$*)* = 2
$yTemp^{upper}$ = 5 − 3 *(as from* C_2*)* = 2 *and*
y^{upper} = *Minimum (2.5,* $yTemp^{lower}$*)* = 2.
Thus the refined bounds of y becomes [2,2].

3.4.1 Synthesis Algorithm

Let CT be the set of all the constraints and VT be the set of all variables. At any given instant, the set of variables with already generated random values are defined as GV (generated variables), and the set containing the rest is defined as NGV (non-generated variables). Initially, NGV is equal to VT. Moreover the bound for each variable is initialized to its maximum feasible value. Next, we choose a variable (say x_i) from NGV. After getting a random solution for x_i, we push x_i in GV and update the bounds for x_j (where $j>i$) in all the X_iX_j-planes by taking the intersection of the existing bound and obtained bound from X_iX_j-plane. We iterate till we get a random value for all the variables, i.e, until NGV becomes empty. The algorithm for generating the solution is as follows.

Algo. 3.4.1 GenerateSolution

GenerateSolution(CT, VT)
begin
1: $NGV \leftarrow VT$, $B_{x_i} \leftarrow [x_i^{lower}, x_i^{upper}]$, $\forall i \in$[1,n]
obtained from lp_solve // Initialization step
2: $i \leftarrow 1$.
3: while ($|NGV|$) do
3.1: Take x_i from NGV.
3.2: $X_i \leftarrow$ Random Value for $x_i \in [x_i^{lower}, x_i^{upper}]$.
3.3: $GV \leftarrow GV \cup \{x_i\}$.
3.4: $NGV \leftarrow VT$ - GV.
3.5: For each variables $x_j \in NGV$ $(j > i)$ do
3.5.1: $B_{x_j}^{temp} \leftarrow$ bound for x_j computed using GV.
3.5.2: $B_{x_j} \leftarrow B_{x_j}^{temp} \cap B_{x_j}$.
EndFor
3.6: $i \leftarrow i + 1$;
EndWhile
4: Solution $\leftarrow (X_i, X_{i+1}, X_{i+2}, \ldots, X_n)$.

end

Theorem 3 *Algorithm GenerateSolution guarantees that the valuation of the first k variables is done in a way that the remaining $(n-k)$ variables have a satisfiable assignment.*

Proof: *We start with maximum feasible bound B_{x_i} for i-th variable (i=1 to $n-1$). Let, after obtaining a satisfying valuation $\tilde{x_k}$ for k-th variable, there exists no satisfying value for $(k+1)$-th variable. This implies the bound for x_{k+1} (call it $B_{x_{k+1}}$) after refinement (say $B'_{x_{k+1}}$) is infeasible. $B_{x_{k+1}}$ was a feasible bound. Also, $B'_{x_{k+1}} \subseteq B_{x_{k+1}}$. Moreover, $B'_{x_{k+1}} \neq \phi$ (since otherwise it implies that, no part of the polygon intersects with $x_k = \tilde{x_k}$, i.e. $\tilde{x_k} \notin B_{x_k}$ which is not possible). This implies $B_{x_{k+1}}$ contained an infeasible region, which is a contradiction. Hence, there exists a satisfying assignment for $(k+1)$-th variable whenever we choose a valid valuation of k-th variable.* $\square$

Theorem 4 *All valid valuations occur with a non-zero probability using GenerateSolution.*

Proof: *Consider a valid tuple $\tilde{x} = (\tilde{x_0}, \tilde{x_1}, \ldots \tilde{x}_{n-1})$. If we show that on choosing '$\tilde{x_k}$' for k-th variable '$\tilde{x}_{k+1}$' lies in the refined bound for $(k+1)$-th variable, then applying this reason recursively we prove that all the n-elements of tuple are considered with non-zero probability and hence the tuple has non-zero probability of occurrence.*

Let on choosing $\tilde{x_k}$ for k-th variable we exclude $\tilde{x}_{k+1}$ from $B_{x_{k+1}}$. This is possible only if X_kX_{k+1}-plane possesses some constraints (since in order to refine $B_{x_{k+1}}$ k-th and $(k+1)$-th variable should constitute some constraint, otherwise no valuation of k-th variable can affect the bound of $(k+1)$-th variable). As $\tilde{x}$ is valid, we have the following for all C_i-s in X_kX_{k+1}-plane:

$$(a_{i1}x_k + a_{i2}x_{k+1}) \ \theta \ a_{i3} \tag{3.1}$$

where $\theta \in \{\geq, \leq\}$.

Let C_p and C_q be the member of the overlapping constraint set for the strip constituting $\tilde{x_k}$. As $\tilde{x}_{k+1}$ is out of the refined bound,

$$(a_{p1}x_k + a_{p2}x_{k+1}) \ \neg\theta \ a_{p3}$$

or,

$$(a_{q1}x_k + a_{q2}x_{k+1}) \ \neg\theta \ a_{q3}$$

But that is a contradiction to Equation 3.1. $\square$

Theorem 3 and theorem 4 respectively show that our methodology is sound and complete.

Fig 3.7 shows the architecture of a typical test generator synthesized by our method. We consider k variables namely x_1, x_2, ..., x_k. The block x_i^{fit} takes a random value from the LFSR and fits it into the current range of x_i. The block AdjustForx_i represents a combinational logic for updating the ranges of all variables, x_j, where $j > i$.

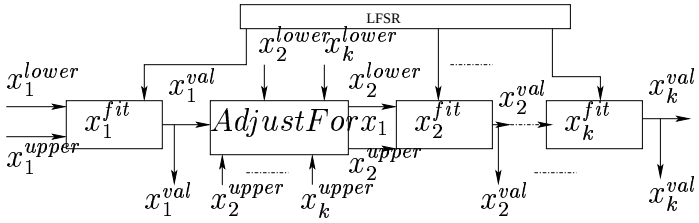

Figure 3.7: Synthesis Block Diagram

3.5 Tool Architecture

The architecture of the tool is shown in Figure 3.8. The test-bench is written in SystemVerilog. We assume that the rest of the test-bench (except the constraint constructs) is synthesizable. The tool accepts the constraints specified by the verification engineer in the test-bench as input. It then performs the software preprocessing and synthesis operations. The output is a synthesizable module that generates the targeted constrained stimuli. This module is then integrated with the rest of the synthesized test-bench. Note that only the shaded blocks are automated by the proposed tool.

3.6 Results

The proposed test generation methodology is fully automated. It only takes the constraints from a test-bench, and directly generates the hardware module. Thus it can be easily instantiated appropriately during accelerated simulation.

Table 3.1 and 3.2 show the results on IBM CoreConnect and AMBA AHB on-chip Bus protocols. The first column shows the number of variables, the second column shows the number of planes and the fourth column shows the number of strips. The third column indicates the number of input constraints (cons). The fifth column shows the interface details for the generated module. The sixth and seventh columns show the number of sequential and combinational elements in the generated circuit respectively. We have used Design Compiler [21] from Synopsys for synthesizing the hardware.

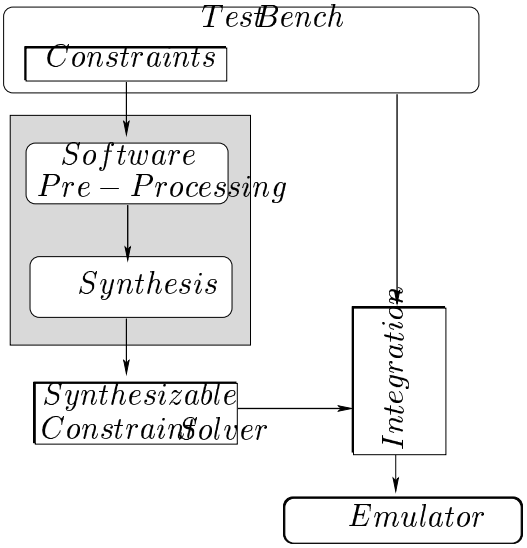

Figure 3.8: Tool Architecture

No of Vars	No of Planes	No of Cons	No of Strips	Mod Int	Seq Elm	Comb Elm
IBM CC DCR Slave						
2	1	4	1	input: 2 output: 10	59	1710
IBM CC DCR Master						
2	1	4	1	input: 13 output: 11	54	2057
IBM CC OPB Master						
3	2	8	2	input: 14 output: 10	74	1892
IBM CC OPB Arbiter						
3	2	6	2	input: 5 output: 3	12	192

Table 3.1: Results on IBM CoreConnect Bus protocol

Note that the circuit elements for IBM CC OPB and AMBA AHB Arbiters are quite less compared to the others. This is because the width of the control signals for OPB Arbiter is very small, whereas other components include constraints involving address and data buses.

We now analyze the synthesis complexity of the proposed approach. To do that we have constructed some complex constraints examples. Table 3.3 show that the complexity of the polygon and the number of strips (in a 2-D plane) depends on the number of constraints in the same plane. Also, the amount of hardware logic

No of Vars	No of Planes	No of Cons	No of Strips	Mod Int	Seq Elm	Comb Elm
AMBA AHB Master						
6	4	12	6	input: 29 output: 27	161	4135
AMBA AHB Slave						
4	3	8	3	input: 16 output: 14	96	1892
AMBA AHB Arbiter						
4	3	9	3	input: 9 output: 7	42	334

Table 3.2: Results on AMBA AHB Bus protocol

directly depends on the number of strips in a plane. So given the number of planes N and the maximum number of strips in a plane M, the amount of hardware logic L is given by $O(N \times M)$. Figure 3.9 shows the distribution of circuit complexity (sum of seq and comb elements) vs (no_plane $\times$ no_strips), which mostly follows an uniform distribution. However, note that the flop-count is $O(|V|)$ (required only for the LFSRs and the output buffers).

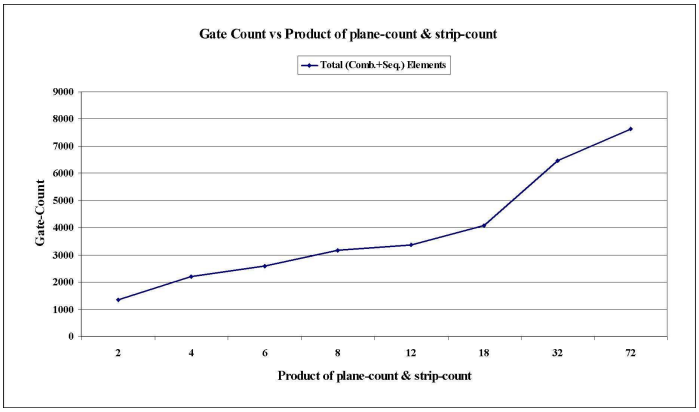

Figure 3.9: Synthesis Result

3.7 Conclusion

In this project we have proposed a novel methodology for constrained random test generation in hardware.

Limitations: The limitations of the proposed methodology are given as follows.

No.of Vars	No. of Planes	No. of Cons	No. of Strips	Mod Int	Seq Elm	Comb Elm
2	1	4	2	input 10 output 8	24	1313
2	1	5	4	input 10 output 8	24	2175
3	2	6	3	input 14 output 12	36	2562
3	2	8	4	input 14 output 12	36	3140
4	3	8	4	input 18 output 16	48	3331
4	3	12	6	input 18 output 16	48	4033
4	4	16	8	input 18 output 16	48	6405
4	6	24	12	input 18 output 16	48	7585

Table 3.3: Results on Hardware resources

- Each constraint involves at most two variables, though there is no restriction on the total number of variables.
- The solution region needs to be convex in nature. Our results show that this is the most usual scenario in practice. However, there may exist situations where the region may be concave.

Advantages: The advantages of the proposed methodology are given as follows.

- The software pre-processing step utilizes the full power of the lp_solve tool. This step also computes the strips and stores them in efficient data structures, which are used in the synthesis algorithm. These data structures allow the generated hardware to be optimized in hardware resources.
- The synthesized hardware generates every solution deterministically with a non-zero probability (fair).
- The methodology is sound and complete.

The software pre-processing step may backtrack, but the synthesis algorithm is a deterministic one. The pre-processing over-head is one time and as we aim at accelerating the test generation within minimal hardware resource, the software pre-processing complexity is not a major concern. The results show that the generated

hardware logic is manageable. The generated module can be easily integrated in the current hardware accelerated verification flow.

Chapter 4

Automatic Extraction of Assume Properties

4.1 Introduction

Capacity limitations of existing formal property verification (FPV) tools have led to the adoption of the *assume-guarantee* style of FPV, where each component module of a design is verified in isolation under given constraints on its environment. Recent languages for formal property specification, such as PSL and SVA support *assume* properties that model the constraints on the environment, and *assert* properties that model the behavior expected from the design module under test (DUT). The DUT is expected to satisfy the *assert* properties under all input scenarios that satisfy the *assume* properties. The FPV tool formally checks whether this is indeed the case.

Recent experience shows that the task of defining a set of assume properties for a module is a nontrivial task. This is mainly because, (a) it is hard to model the behavior of the environment, which is a product of the behaviors of the other components of the system, and (b) different constraints apply at different states of the protocol between the module and its environment. It is hard to model the states of the protocol purely through properties.

Traditionally, in simulation based verification, the test bench models the environment of the module. It solves (at least to some extent) both of the above difficulties – the first, by using behavioral models of the other components of the system, and the second, by maintaining the state of the protocol. The focus of this project is to ask: *Can we extract the assume properties from the test bench?*

The foremost question that may appear in the mind of the FPV practitioner is: *Is the test bench a "complete model" of the environment?* In general the answer is negative, because a test bench may not exercise all the choices of the environment. On the other hand, the industry appears to be moving towards randomized test generation, where the choice points of the environment are randomized, thereby

guaranteeing that every choice of the environment is taken with nonzero probability. Therefore, a properly structured randomized test bench is indeed a *complete* model of the environment.

Verification engineers are usually more comfortable in driving the interface between a module and its environment through test benches as compared to the task of modeling the environment through formal assume properties. Unfortunately even when we use randomized test benches, simulation does not hit all the interesting behaviors in reasonable time, because some of these scenarios have a very low probability of occurrence. If we can extract a formal model from the test bench, then a FPV tool can verify such behaviors exhaustively.

If we treat the random choice points in a constrained random test bench as non-deterministic choices of the environment, then (theoretically) we should be able to extract a non-deterministic finite state machine from the test bench. However, there are two major problems. Firstly the extraction problem becomes very difficult when we support all features of a test bench language. Secondly, the choices of the environment are not purely random, but *constrained random*, that is, they are random within some given constraints. These constraints are in fact the *assumptions* about the environment that we require for formal analysis.

Therefore, a constrained random test bench has an underlying non-deterministic finite state machine, where the states are the states of the protocol between the module and the test bench. At each state of this machine the test bench drives inputs to the module randomly under one or more given constraints. Our goal is to extract this state machine from the test bench and automatically generate the assume properties from the state-specific constraints. This project presents the preliminary formal methodology to achieve this task. We believe that this is the first project to propose the extraction of assume properties from constrained random test benches.

The main contributions of this project are as follows.

1. We identify a fragment of the some standard test bench language (e.g. SystemVerilog) from which it is convenient to extract an abstract state machine and the state specific assume properties. We show that the fragment is expressive enough to model constrained random test benches of standard protocols such as IBM CoreConnect and ARM AMBA.

2. We present a prototype tool for automatic extraction of the underlying state machine and assume properties from the test-bench.

3. We present a methodology for modeling the environment in terms of the abstract state machine and assume properties to enable formal verification of the assertions on the DUT.

Our existing tool accepts the test bench in the proposed test-bench language subset and extracts the assume properties.

The chapter is organized as follows. In the next Section, we present an example system (DUT & the test-bench) that is used in the rest of the chapter. In Section 3, we provide the formal modeling for the constrained random test-bench and the DUT. Section 4 provides the methodology for extraction of *assume properties* from the given test-bench. We provide the tool architecture in Section 5. The experimental results are shown in Section 6.

4.2 A Simple Example

We consider the verification of a slave device, S, which supports only 2 beat transfer in a Bus protocol. The interface between S and a requesting master device M is shown in Figure 4.1.

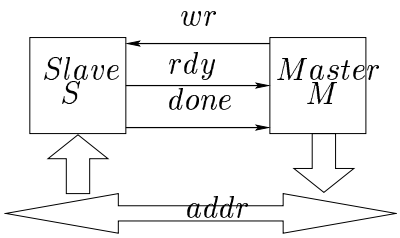

Figure 4.1: Example System

S initially remains in the *idle* state, in which its outputs *rdy* and *done* are low. A master device indicates its intent to write by asserting the *wr* signal and floating an address in the address lines, *addr*. The slaves are memory mapped and identify themselves from their respective address ranges.

For example, whenever S finds that the address floated by the master is outside its address range (say, 0 to 499), it returns to its *idle* state. If the address is within its range, then there are two cases. If the device is not ready to accept the the transfer then it lowers the *rdy* signal to delay the transfer. Otherwise it raises the *rdy* signal and completes the 2 beat transfer by asserting the *done* signal in the next two consecutive cycles.

The functionality of S can be expressed by the following SystemVerilog Assertions.

```
property P1;
@(posedge clk) (wr && rdy && !done) |->
(##1 (rdy && done) ##1 (rdy && done));
endproperty

property P2;
```

```
@(posedge clk)
      (rdy && !(addr >=0 && addr < 500))
               |-> (##1 (!rdy && !done);
endproperty
```

The first property expresses the requirement that whenever a transfer is acknowledged by S by asserting the *rdy* signal, the transfer must be completed by asserting *rdy* and *done* in the next two cycles. The second property expresses the requirement that the slave must return to the *idle* state (by lowering *rdy* and *done*) when *addr* holds an address that is outside its range.

Curiously, given no restrictions on the inputs to S, the properties $P1$ and $P2$ cannot be satisfied if the master presents an address outside the range of S in the 2^{nd} beat of the 2 beat transfer. The scenario is shown in Figure 4.2.

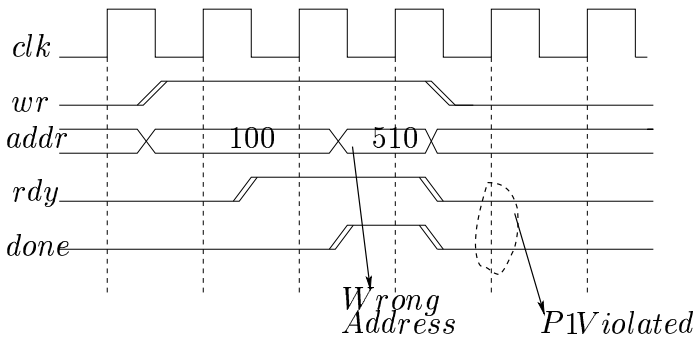

Figure 4.2: Example Violation Scenario

However, properties $P1$ and $P2$ can be satisfied if we add the following properties.

Assumption: Once a transfer is acknowledged, the transfer address must be in the range (0 to 499). This assumption may be expressed by the following SVA assume property.

```
sequence Adrange;
@(posedge clk) (addr < 500 && addr >= 0);
endsequence

property AP1;
@(posedge clk) (wr && rdy && !done) |->
(Adrange ##1 Adrange ##1 Adrange);
endproperty
```

The question that we ask in this project is - could we have extracted the above assume property from a constrained random test bench for the DUT (slave, S)? To

support our argument, we observe that a typical randomized test bench for the slave will have two random choices. The first choice decides whether the master initiates a transfer or whether it remains in the *idle* state. The second choice randomly selects the value of the address bits. The choice of the *addr* appears in both cycles of the 2 beat transfer and in both the cases, the choice is constrained within the address range of the slave S (DUT).

A sample test bench for the DUT is given in the Appendix. This test bench uses the subset of the test-bench language proposed in this project (given in section 4.4). The underlying state machine of the test bench in shown in Figure 4.3.

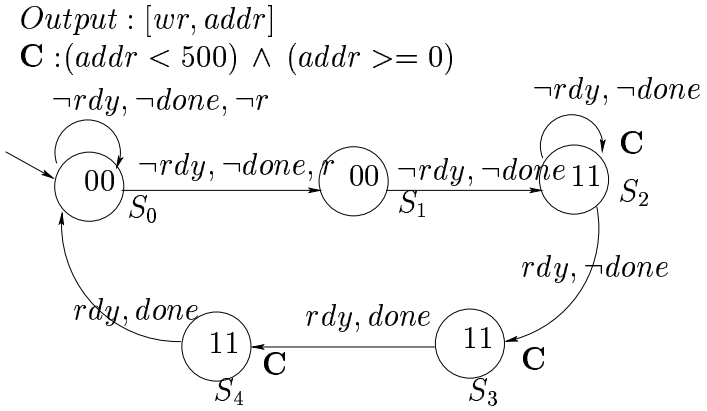

Figure 4.3: State machine extracted from the test bench

Our tool extracts this state machine from the test bench. The state machine is annotated with constraints on the choice of the values of the random variables. After extracting this state machine, our tool generates a state machine and a collection of assume properties as shown below. These constitute the environment of the DUT, S, when we run a FPV tool on S to verify $P1$ and $P2$.

```
//-We use the following ‘define s for
//-rest of the SV modelings
‘define S0 3’b000
‘define S1 3’b001
‘define S2 3’b010
‘define S3 3’b011
‘define S4 3’b100

//- Input condition
‘define I0  !rdy && !done && !r
‘define I1  !rdy && !done && r
‘define I2  !rdy && !done
‘define I3  rdy && !done
‘define I4  rdy && done
```

```
interface SlaveAssumeP
    (input rdy, input done, input wr,
     input addr, input r, input clk);

logic [1:0] env_st;

//-- Transition relation encoding
always @(posedge clk) begin
//- Initialization
env_st = `S0;
//-Transition Relation
env_st <=
    (env_st == `S0) && `I0  ? `S0 :
    (env_st == `S0) && `I1  ? `S1 :
    (env_st == `S1) && `I2  ? `S2 :
    (env_st == `S2) && `I2  ? `S2 :
    (env_st == `S2) && `I3  ? `S3 :
    (env_st == `S3) && `I4  ? `S4 :
    (env_st == `S4) && `I4  ? `S0 :
   env_st;
end

//-Assume properties for
//-constraint C at different states
//-Sequence that represents valid
//-range for S

sequence Adrange;
    (addr >=0 && addr < 500);
endsequence

property A0;
@(posedge clk)
    (env_st == `S2) |-> Adrange;
endproperty

assume_A0: assume property (P0);

property A1;
@(posedge clk)
    (env_st == `S3) |-> Adrange;
endproperty
```

```
assume_A1: assume property (P1);

property A2;
@(posedge clk)
    (env_st == `S4) |-> Adrange;
endproperty

assume_A2: assume property (P2);
endinterface
```

These assume properties together satisfy the required environment assumption of S.

4.3 Formal Modeling

In this section, we provide a formal description of the proposed modeling style for the test-bench. The essence of any constrained random test-bench lies in the presence of a set of random variables. The test-bench functionality is non-deterministic in nature. The choice of taking a transition from a given state thus depends on the present-valuation of the random variables. In some states, the valuation of the random variables are constrained by the given constraints.

The environment is modeled as a nine-tuple machine:

$$Env : \langle S_{env}, s_{env}, I_{DUT}, I_{RI}, R_{env}, O_{env}, L_{env}, CON_{env}, F_{env} \rangle$$

where,

S_{env}: The set of states.
s_{env}: The set of initial states, $s_{env} \subseteq S_{env}$.
I_{DUT}: The set of inputs (from the DUT).
I_{RI}: The set of random variables.
$R_{env} : S_{env} \times \{I_{DUT} \cup I_{RI}\} \rightarrow S_{env}$ is the transition function.
O_{env}: The set of outputs of the test bench.
$L_{env} : S_{env} \rightarrow 2^{O_{env}}$ is a function that labels each state with the set of outputs true in that state.
CON_{env}: The set of user-specified Boolean constraints over $\{I_{RI} \cup O_{env}\}$.
$F_{env} : S_{env} \rightarrow 2^{CON_{env}}$ is a relation that maps each state with the set of constraints.

To illustrate further, consider the test bench in our example. S_{env}=$\{S_0, S_1, S_2, S_3, S_4\}$ (the states of environment automaton), s_{env}=$\{S_0\}$ I_{DUT}=$\{rdy, done\}$ (i.e. the outputs of S), and I_{RI}=$\{r\}$. The transition relation (R_{env}) is depicted in Figure 4.3.

The output-set (O_{env}) is {wr, $addr$}. The set of constraints, (CON_{env}) ={C}. The constraint mapping F_{env} is shown in Figure 4.3.

In the next section, we show how to implement our test-bench specification style in our proposed test-bench writing language.

Constrained Random Test-bench Specification Language:

To implement the proposed style, only a sub-set of some standard test-bench language functionality is supported. The chosen test-bench language constructs are given in Table 4.1.

Construct	Functionality
Interface	interface is used to model the signal directions.
Clocking Block	clocking block is used the synchronizing clock for the interface.
Assignment Statement	Blocking, Non-Blocking. Blocking used for internal assignments. Non-Blocking drives interface signals.
Selection Statement	if-else, if-elseif-else, case-endcase. case statement is used for state encoding. if statement is used for transition encoding.
Loop Statement	while while statement models test-bench iterations.
Class	class is used for modeling design variables.
Task	task is used for modeling of design functionality (state machine).
Constraint Block	constraint expression, in-line constraint. constraint expression in a class acts as global constraint for the entire device functionality. in-line constraint is used to model state-specific constraints. It may override global constraints.

Table 4.1: Selective test-bench constructs

4.4 The Methodology

In this section we first show how to extract the constrained environment automaton from the given type of test-benches. The constraints at a state defines the constraints on O_{env}. For example, the constraint C_1 in state S_1 (see Figure 4.3) restricts the test-bench output $addr$ within 0 to 499. The steps of the methodology are given as follows.

4.4.1 Constrained Environment Automaton Extraction

Given a test-bench TB (in the proposed format), we extract out the states and transition from TB. Next we map the constraints (if any) given in a specific TB state.
The algorithm to extract a constrained environment automaton Env from TB is outlined as follows.

Algo. 4.4.1 ALGORITHM AUTOMATONEXTRACTION

ALGORITHM AUTOMATONEXTRACTION (TB, Env)
begin
1. Find out state variable $StateVar$ from *case* statement in TB.
2. Find out the initialization of $StateVar$.
   Mark it as the start state in Env i.e. s_{env}.
3. For each state encoding $StateVar_i$ of $StateVar$
   in the *case* statement.
   3.1 Create a separate state in Env i.e. compute S_{env}.
4. In every state encoding $StateVar_i$ of $StateVar$
   do the following.
   4.1 Find out the *in-line* constraint Con.
   4.2 Map Con as restrictions on O_{env} in the
      appropriate state $s_i \in S_{Env}$.
   4.3 For every transition encoding $Trans$, given in the form
      of *if-elseif* in TB do the following.
      4.3.1 Find out the *condition* $TransC$ in $Trans$ (*if-elseif*).
      4.3.2 Find out the new assignment of $StateVar$
         in $Trans$ as $StateVar_j$.
      4.3.3 Find out the assignment of output variables,
         Out in $Trans$.
      4.3.4 Create a transition $Tr \in R_{env}$ on $TransC$ in s_i.
      4.3.5 Assign s_j as the next state of s_i on $TransC$.
      4.3.6 Assign the value of the state variables of s_j as Out.
end

Theorem 5 *The extracted automaton in Algorithm 4.4.1 is semantically equivalent to the formal model described in section 4.3.*

Proof: To establish semantic equivalence, we prove the correctness of translation of each construct allowed in Table 4.1. We first identify the initialization of state variable i.e. s_{env} *(line 2 of Algo 4.4.1) from the first blocking statement on the state variable in the* **`task`**. *Each of the case items gives the state encoding i.e.* S_{env}

*(*for-loop *in line 3). Line 4.1 extracts the elements of the set CON_{env}, while line 4.2 defines the labeling function L_{env}. The* for-loop *in line 4.3 defines the transition relation R_{env} by identifying $TransC$ ($I_{DUT} \cup I_{RI}$) (line 4.3.1). The output variables, O_{env} and their respective labeling L_{env} are also extracted (in line 4.3.3).*

These enables extracting a semantically equivalent formal model of the Env (in the proposed format) from the given test-bench. □

4.4.2 Generation of Assume Property

The generation of assume properties from a constrained random test-bench requires two main steps.

- Modeling the transition relation of the Environment.
- Mapping the test-bench constraints as *assume properties* in specific states of the extracted Environment state machine.

Given the extracted Environment automaton Env in the earlier step, we generate the transition relation TR_{Assume} and the *assume properties* $AssumeP$ by the following algorithm.

Algo. 4.4.2 ALGORITHM GENASSUMEPROP

ALGORITHM GENTRANSRELATION (Env, TR_{Assume}, $AssumeP$)
begin
//– Transition Relation Encoding
1. Initialize the state variable S_{Assume} to s_{env} (start state).
2. For every state $s_i \in S_{env}$ starting from $s_{env}(startstate)$
   1.1 For every transition (s_i, $TransC$, s_j)
      //-Assign S_{Assume} to s_j on condition $TransC$ from s_i.
      1.1.1 Assign $S_{Assume} = (s_i \;\&\&\; TransC)\ ?\ s_j$;
   1.2 EndFor
2. EndFor
//- Creating Assume Properties
3. For every state $s_i \in S_{env}$ starting from $s_{env}(startstate)$
   //–Check whether the state has a constraint
   3.1 if (F_{env} (s_i))
      //– Model the constraint as an assume property
      Add an assume property $assumeP_i$:
      $assume$ property (CON_{env_i});
   3.2 endif
4. EndFor
end

For example, the assume properties for the example Slave device are given as follows.

4.4.3 Proof of Correctness

The results of the simulation of the DUT using the given test-bench is covered by the formal analysis using the extracted assume properties.

To establish the above claim, we need to prove that the constraints in the test-bench are correctly propagated as *assume properties* at appropriate states in the DUT state machine.

Theorem 5 proves the correctness of Algo 4.4.1. Moreover, the proof of correctness for the Algo 4.4.2 is intuitive. The assume properties of the environment are propagated from Env to $AssumeP$ through constraint mapping mentioned in *for*-loop of line 3 in Algo 4.4.2.

Thus the constraints mentioned in TB finally get mapped into the state-machine required by the formal property checker as assume-properties in the appropriate states.

4.5 Tool Architecture

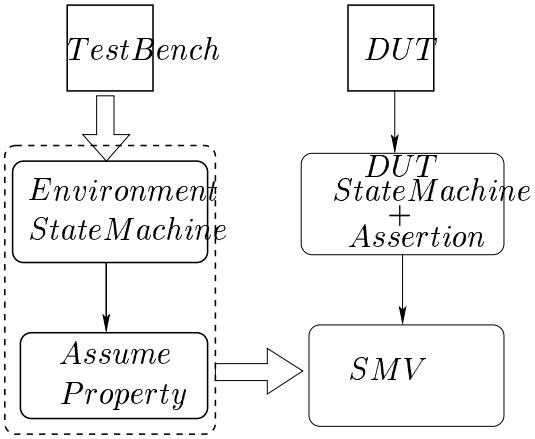

Figure 4.4: Tool Architecture

In this section, we describe the tool architecture as shown in Fig 4.4. The front-end of the tool consists of a parser, which parses a constrained random test-bench modeled in the prescribed format (see section 4.3). It next extracts out the Environment state machine along with the state specific constraints. The tool then generates the assume properties for the DUT. These assume properties are fed to

SMV, a formal property checker from Carnegie Mellon University. SMV, on the other front, takes the DUT description and the associated assertions or properties. The formal analysis of the DUT is then performed in presence of the extracted assume properties. The results from SMV helps us to compare the simulation results with the formal analysis. This in turn helps to prove the correctness of the approach.

4.6 Results

We have tested this methodology on ARM AMBA AHB and IBM CoreConnect Bus components. The test-benches are modeled using the mentioned test-bench constructs of the proposed language. The designs are modeled in Verilog. The assertions/properties for the designs are modeled in SystemVerilog Assertions. The formal analysis has been carried out using SMV [16]. It has been found out that analysis without using assumptions has led to refutation of one or more properties in all of these cases. Next we have extracted and used the assumptions from the associated test-benches. We have seen that in most of the cases the refuted scenarios have been translated to success. In some of the cases, the bug remains there and the design needed to be corrected. The complete analysis is shown in Table 4.2. The design and environment circuit complexities are shown in column 1 and column 2 respectively. The number of assume properties has been shown in column 3. The number of SystemVerilog assertions is shown in column 4. The CPU runtime (in seconds) is shown in column 5. The analysis has been done on a PC with Intel Xeon CPU 2.80 GHz, with 4 GB RAM.

IBM CC DCR Master				
DUT Ckt Elements	Env Ckt Elements	No. of Assume	No. of Assertion	CPU time
Inputs: 3 Outputs: 5 Seq: 5 Comb: 124	Seq: 5 Comb: 42	2	3	0.01 sec
IBM CC DCR Slave				
Inputs: 5 Outputs: 3 Seq: 3 Comb: 75	Seq: 28 Comb: 404	5	3	0.02 sec
IBM CC OPB Arbiter				
Inputs: 9 Outputs: 3 Seq: 8 Comb: 251	Seq: 31 Comb: 388	4	4	0.02 sec
AMBA AHB Arbiter				
Inputs: 12 Outputs: 4 Seq: 8 Comb: 528	Seq: 20 Comb: 304	7	6	0.02 sec

Table 4.2: Results on standard Bus components

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In the previous chapters we discussed about the recent design verification technologies, namely, simulation and formal verification along with their merits and demerits. We found that simulation is time-consuming, while the formal techniques demands expertise of the validation engineer in building the formal model. We have also discussed that *constrained random test-benches* are among the recent trends in the electronic design verification. Our discussion included how the popular verification languages support writing constrained random test-benches. But this feature in those languages raised the vital question of synthesizability and extraction, from languages like SystemVerilog into explicit formal model. We approached to solve both of these problems in two different ways. As constrained random test-benches help us in simulating real-world environments, we believe, this is a significant contribution in aiding the verification community at both the ends. We present the summary of the approaches taken.

- As the hardware devices are faster than the software routines, the simulation time is reduced. This technique is usually referred to as the *emulation* technique. This implies the necessity of mapping the software test-benches to synthesizable hardware. Thus, we tried to synthesize the linear constraints (given in constrained random test-benches) into synthesizable hardware. The goodness of the proposed methodology lies in the fact that we are not compromising with the fairness (i.e. all the valid points have a non-zero probability of occurrence) and also it has a manageable hardware cost.

- On the other hand, in order to assist the verification engineer in formally checking the design, we have suggested a definite style of writing constrained random test-benches, from which we have developed a methodology to extract the formal model of the test-bench along with the embedded constraints. This

would help in formal verification of the design without actually writing the test-bench in another formal language.

5.2 Future Directions

Having solved the problem of extraction of the assume-properties from a given test-bench, now we aim to attack a more difficult problem. The test-benches are usually not written at the module level. They are written in the system-level. The advanced constructs allow the verification engineer to write the test-bench in more complex manner e.g. layered test-benches. But quiet contrary to the simulation methods, the formal methodologies are used to verify the module-level design since the methods cannot handle the exponential growth in the complexity. So it is needless to say that we need some interpreter which will generate the formal model of the environment for the given module along with the realistic assume properties. Now this interpreter should be intelligent enough to propagate and detect the signal values at primary input pins which might affect the given module. So in short, we will be investigating into the following question - "*Can we use the system-level test-bench to extract the module-level assume-properties?*"

Appendix A

SystemVerilog Constrained Random Testbench

The example SystemVerilog constrained random test bench is given as follows.

```
//-- Interface declaration
interface master_intf (input clk);
   wire wr, rdy, done;
   wire [9:0] addr;
   //-- Clocking Block
   clocking CBMst @(posedge clk);
     output wr, addr; input  rdy,done;
   endclocking
endinterface

//-- Test-Bench
//-- Some `define s that would be used in
// modeling the env state machine
`define S0 3'b000
`define S1 3'b001
`define S2 3'b010
`define S3 3'b011
`define S4 3'b100

//- Input condition
`define I0 !tI.CBMst.rdy&&!tI.CBMst.done&&!Env.r
`define I1 !tI.CBMst.rdy&&!tI.CBMst.done&&Env.r
`define I2 !tI.CBMst.rdy && !tI.CBMst.done
`define I3 tI.CBMst.rdy && !tI.CBMst.done
`define I4 tI.CBMst.rdy && tI.CBMst.done

program master (master_intf intf);//-test-bench

  class Master; //-Master device
    rand bit r;//-Internal rand bits
```

```
  rand bit [9:0] addr;
endclass

//-- virtual interface
virtual master_intf tI;
//-- Master instance
Master Env = new();

//-- test bench execution starts here
initial begin
   tI = intf; MasterExec ();
end
task MasterExec ();//-Main task definition
  bit [1:0] env_st;//-state encoding
  integer success;

  env_st = `S0; //-Init state

  while (1) begin
    case(env_st)//-state-mc encode
     `S0 : begin  //- state S0

      if (`I0) begin //-IDLE
        tI.CBMst.wr <= 1'b0;
        env_st = `S0; //-S0
      end

      else if(`I1) begin //-Rdy for Trans
        tI.CBMst.wr <= 1'b0;
        env_st = `S1; //- next state S1
      end
     end

     `S1 : begin //- state S1
      //-Inline constraint (C)
       success = Env.randomize() with
           {addr>=0 && addr<500;};

       if (`I2) begin
        tI.CBMst.wr <= 1'b1;
        tI.CBMst.addr <= Env.addr;
        env_st = `S2; //- next state S2
       end
      end

      `S2 : begin //- state S2

        //-Inline constraint (C)
       success = Env.randomize() with
           {addr>=0 && addr<500;};
```

```
        if (`I2) begin
         tI.CBMst.wr <= 1'b1;
         tI.CBMst.addr <= Env.addr;
         env_st = `S2; //- next state S2
        end

        else if(`I3) begin
         tI.CBMst.wr <= 1'b1;
         tI.CBMst.addr <= Env.addr;
         env_st = `S3; //- next state S3
        end
       end

      `S3 : begin //- state S3

        //-Inline constraint (C2)
        success = Env.randomize() with
            {addr>=0 && addr<500;};

        if (`I4) begin
         tI.CBMst.wr <= 1'b1;
         tI.CBMst.addr <= Env.addr;
         env_st = `S4; //- next state S4
        end
       end

      `S4 : begin //- state S4
        if (`I4) begin
         tI.CBMst.wr <= 1'b0;
         env_st = `S0; //- next state S0
        end
       end
     endcase
    end
  endtask
endprogram //-- End of test-bench
```

Bibliography

[1] Savot, T., and Seviora, R. E., "Directed Simulation for Automatic Detection of Failures", In the Proc. of 1997 World Congress on System Simulation, pp. 432-441, September 1997.

[2] Aharon, A., Goodman, D., Levinger, M., Lichtenstein, Y., Malka, Y., Metzger, C., Molcho, M., and Shurek, G., *Test program generation for functional verification of PowerPC processors in IBM*, In Proceedings of the 32nd Design Automation Conference, pages 279-285, June 1995.

[3] Ahi, A., Burroughs, G., Gore, A., LaMar, L., Linand, C., and Wieman, A., *Design verification of the HP9000 series 700 PA-RISC workstations*, Hewlett-Packard Journal, 14(8), August 1992.

[4] e Reuse Methodology (eRM), www.verisity.com/products/erm.html

[5] *OpenVera LRM 2.0*, http://www.open-vera.com.

[6] "Synthesizable verification solutions", Duolog Technologies, 2002.

[7] "RTL and Behavioral Synthesis:A Case Study" www.cs.hongik.ac.kr/~dspark/hls.html

[8] A Reference Verification Methodology for Vera, The Synopsys Verification Avenue Technical Bulletin, Vol. 4, Issue 1, February 2004.

[9] Assumption Generation for Software Component Verification, Giannakopoulou, D., Corina S. and Barringer, H., In *Proceedings of the* 17^{th} *IEEE International Conference on Automated Software Engineering (ASE'02).*

[10] Automatic Assume Guarantee Analysis for Assertion-Based Formal Verification, Dong Wang and Jeremy Levitt, *In Proceedings of the ASPDAC 2005,pg561-566.*

[11] ARM AMBA Specification Rev. 2.0, http://www.arm.com

[12] Clarke, E.M., Grumberg, O., and Peled, D.A., *Model Checking*, MIT Press, 2000.

[13] IBM CoreConnect Bus Specification. www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/

[14] Learning Assumptions for Compositional Verification, Jamieson M. Cobleigh, Giannakopoulou, D. and Corina S., In *Proceedings of the TACAS 2003*, LNCS 2619, pp. 331-346, 2003.

[15] OpenVera Language Specification 2.0 http://www.open-vera.com/

[16] The SMV System http://www.cs.cmu.edu/ modelcheck/smv.html

[17] Verification Methodology Manual for SystemVerilog, Bergeron, J., Cerny, E., Hunter, A., Nightingale, A. 2005, XVIII, 510 p., Springer.

[18] Verification Reuse Methodology, Essential Elements for Verification Productivity Gains
http://www.verisity.com/resources/whitepaper/erm.html

[19] AMBA Specification Rev2.0
http://www.arm.com/products/solutions/AMBA_Spec.html

[20] "Accelerating ASIC Verification with FPGA Verification Components"
www.fpgajournal.com/articles/20040921_einfo.htm

[21] Design Compiler.
www.synopsys.com/products/logic/logic.html

[22] Hamadi, Y., and Merceron, D., Reconfigurable architectures: A new vision for optimization problems, in Principles Practice Constraint Programming CP97, 1997, pp. 209215.

[23] IBM CoreConnect Bus Specification.
www-306.ibm.com/chis/techlib/techlib.nsf/techdocs/

[24] PCI Bus Specification 3.0. http://www.pcisig.com/members
/downloads/specifications/conventional/PCI_LB3.0-2-6-04.pdf

[25] Lee, K., T., Leong, W., H., P., Lee, H., K., Chan, T., K., Hui, S., K., Yeung, K., H., Lo, M., F., and Lee, M., H., J., An FPGA implementation of GENET for solving graph coloring problems, in IEEE Symp. Field-Programmable Custom Computing Machines, 1998, pp. 284285.

[26] Leong, W. H. P., Sham, W. C., Wong, C. W., Wong, Y. H., Yuen, S. W., and Leong, P. M., "A Bitstream Reconfigurable FPGA Implementation of the WSAT Algorithm", IEEE Transactions on VLSI Systems, Vol. 9, No. 1, February 2001.

[27] Palnitkar, S., Design Verification with e, Prentice Hall Professional Technical Reference, August, 2003.

[28] Sayama, T., Yokoo, M., and Sawada, H., Solving satisfiability problems using logic synthesis and reconfigurable hardware, in Proc. 31st Hawaii Int. Conf. System Sciences, 1998, pp. 179186.

[29] Schrijver, A., 1986. Theory of Linear and Integer Programming. Wiley Interscience series in discrete mathmatics. John Wiley & Sons, December.

[30] Selman, B., Levesque, H., and Mitchell, D., A new method for solving hard satisfiability problems, in Proc. 10th Nat. Conf. Artificial Intell. (AAAI-92), San Jose, CA, 1992, pp. 440446.

[31] SystemVerilog 3.1a Language Reference Manual.
http://www.eda.org/sv/SystemVerilog_3.1a.pdf

[32] Wageeh, N, Mohamed., Wahba, M, Ayman., Salem, M. Ashraf., and Sheirah, A. Mohamed., "FPGA Based Accelerator for Functional Simulation", In the Proceedings of the ISCAS, 2004.

[33] Yokoo, M., Sayama, T., and Sawada, H., Solving satisfiability problems using field programmable gate arrays: First results, in Proc. 2nd Int. Conf. Principles Practice Constraint Programming, 1996, pp. 497509.

[34] Zhong, P., Martonosi, M., Ashar, P., and Malik, S., Accelerating Boolean satisfiability with configurable hardware, in IEEE Symp. Field-Programmable Custom Computing Machines, 1998, pp. 186195.

[35] Zhong, P., Ashar, P., Malik, S., and Martonosi, M., Using reconfigurable computing techniques to accelerate problems in the CAD domain: A case study with Boolean satisfiability, in Proc. Design Automation Conf., 1998, pp. 194199.

[36] http://lpsolve.sourceforge.net/5.1/

[37] http://www-unix.mcs.anl.gov/otc/Guide/faq/linear-programming-faq.