APPLICATION OF BAYESIAN NETWORKS TO COVERAGE DIRECTED TEST GENERATION FOR THE VERIFICATION OF DIGITAL HARDWARE DESIGNS

by

Jeffery S. Vance

BS in Computer Engineering, University of Pittsburgh, 2006

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2010

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Jeffery S. Vance

It was defended on

December 9, 2009

and approved by

Steven P. Levitan, Ph.D., Professor

Allen C. Cheng, Ph.D., Assistant Professor

Jun Yang, Ph.D., Associate Professor

Thesis Advisor: Steven P. Levitan, Ph.D., Professor

APPLICATION OF BAYESIAN NETWORKS TO COVERAGE DIRECTED TEST GENERATION FOR THE VERIFICATION OF DIGITAL HARDWARE DESIGNS

Jeffery S. Vance, M.S.

University of Pittsburgh, 2010

Functional verification is generally regarded as the most critical phase in the successful development of digital integrated circuits. The increasing complexity and size of chip designs make it more challenging to find bugs and meet test coverage goals in time for market demands. These challenges have led to more automated methods of simulation with constrained random test generation and coverage analysis. Recent goals in industry have focused on improving the process further by applying *Coverage Directed Test Generation* (CDG) to automate the feedback from coverage analysis to test input generation.

Previous research has presented Bayesian networks as a way to achieve CDG. Bayesian networks provide a means of capturing behaviors of a design under verification and making predictions to help guide test input generation to reach coverage goals more quickly. Previous research has shown methods for defining a Bayesian network for a design domain and generating input parameters for dynamic simulation.

This thesis demonstrates that existing commercial verification tools can be combined with a Bayesian inference engine as a feasible solution for creating a fully automated CDG environment. This solution is demonstrated using methods from previous research for applying Bayesian networks to verification. The CDG framework was implemented by combining the Questa verification platform with the Bayesian inference engine SMILE (Structural Modeling, Inference, and Learning Engine) in a single simulation environment. SystemVerilog testbenches

and custom software were created to automatically find coverage holes, predict test input parameters, and dynamically change these parameters to complete verification with a fewer number of test cases. The CDG framework was demonstrated by performing verification on both a combinational logic design and a sequential logic design. The results show that Bayesian networks can be successfully used to improve the efficiency and quality of the verification process.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

Advances in technology continuously require improved performance and features of integrated circuit designs. Chip designs such as microprocessors, specialized System-on-a-Chip (SOC) designs, and Field Programmable Gate Arrays (FPGAs) continue to grow more complex while being adopted in broader types of applications. As the complexity and size of chip designs grow, verifying the correctness of these designs becomes more challenging for design teams. While verification engineers must meet the new challenges of finding bugs, time-to-market demands require the verification process to be completed quickly. In recent industry practice, the verification phase is commonly the longest phase in hardware design cycles and is the most critical to completing a product in time to meet market demands.

In order to meet the increasing challenges of hardware verification, tools and methodologies have been developed to improve verification flows. These solutions succeed in removing much of the manual labor traditionally involved in verifying hardware designs. Tasks such as writing test cases, running tests, and collecting and evaluating results are now typically automated to a high degree. However, verification continues to be one of the primary challenges in hardware design due to new areas of manual effort that are required.

1.1 MOTIVATION

One of the primary motivations for improving the efficiency and quality of verification cycles is from a business perspective. The highly competitive nature of technology industries puts pressure on companies to release products quickly. Time-to-market is critical to the success of a product since competitive products released earlier to market typically draw the most revenue. Since verification is usually the bottleneck in time-to-market, businesses directly profit from speeding up the verification process.

Ensuring high quality of verification is also important to businesses since the consequences of failing to find bugs can be very costly. Bugs not found until the design is manufactured in silicon are often expensive to fix since it requires repeating much of the development cycle. The consequences of releasing a product with a bug to consumers has been shown to be worse since the company must bear the costs of fixing and replacing defect products as well as endure a potential loss in reputation.

Another motivation in improving verification flows is to provide higher assurance of quality in products for critical systems. Integrated circuits are used in many critical applications such as medical devices, control systems for aviation and automotive industries, and protection systems in nuclear power plants. Regulatory agencies impose strict criteria for engineers to demonstrate the correctness of designs for these products. Improving the verification process allows these companies to more easily meet these criteria while assuring the quality of their designs for critical systems.

1.2 PROBLEM STATEMENT

Many methods for improving the verification flow of hardware designs have been focused on simulation based functional verification. Hardware designs are implemented in a hardware description language and verification engineers create testbenches to simulate the designs at the code level. Recent methods in industry have automated much of this process by using constrained random test input generation, automated checking of data, and coverage monitoring to determine when verification is complete. Using random test inputs allows many design requirements to be verified very quickly with minimal manual effort since no specific test cases are written. Random tests have the additional benefit of possibly generating test scenarios that were not initially thought of.

1.2.1 The Problem

Despite the many improvements in recent functional verification techniques, increasing complexities in designs still make this task a difficult challenge. While much of the manual labor in writing test cases and checking results is removed, there is still often a lot of manual effort required to complete the verification. This is because applying random test inputs alone will not usually produce all the test cases needed to verify the design. The verification team must evaluate the coverage results of the testbench and determine which scenarios remain to be verified. The team can then either write directed tests for the remaining cases, or modify the testbench in attempt to steer the random input generation into the untested scenarios. The manual effort in either solution creates a new bottleneck in the verification cycle that can considerably impact completion schedules for large designs.

Research has been done in *Coverage Directed Test Generation* (CDG) to provide solutions to closing the manual feedback loop from coverage analysis to test input generation. One of these solutions is *dynamic coverage-controlled stimulus generation* [21]. This method attempts to apply machine learning to observe how coverage goals are impacted by changes in input stimulus and dynamically bias input generation parameters to hit the remaining verification scenarios.

S. Fine and A. Ziv (IBM Research Laboratory) have successfully applied Bayesian networks as a method of achieving CDG with dynamic coverage-controlled stimulus generation [5]. In their research, a methodology was developed for creating Bayesian networks to model the relationships between coverage goals, test input parameters, and internal states of a design under verification. It has been shown that this technique can successfully generate test directives from observed coverage holes to guide the verification to completion more quickly [1, 5, 6, 7].

Although previous research has shown success in establishing methods for CDG based verification, there are few solutions for applying these methods in industry using existing verification platforms. Most CDG solutions have been demonstrated with proprietary solutions used by some companies internally for research and development. Despite the continuously enhanced features of commercial verification tools provided by Electronic Design Automation (EDA) vendors, there is currently little guidance for engineers to implement fully automated CDG solutions [16].

1.2.2 The Solution

This thesis demonstrates a feasible solution for creating a fully automated CDG verification environment with existing verification tools and a Bayesian inference engine. This solution

allows verification to meet coverage goals without any manual feedback for coverage closure. This solution was demonstrated using the Questa verification tool from Mentor Graphics [17] combined with the inference engine SMILE (Structural Modeling, Inference, and Learning Engine) [13]. The CDG environment was implemented with both SystemVerilog testbenches and custom C++ programs. Testbenches were designed to detect coverage holes at simulation runtime and dynamically update test input parameters according to inference results. Custom C++ programs made use of SMILE to perform inference calculations on a Bayesian network and generate predictions for new test directives. Bayesian network structures were created using GeNIe (Graphical Network Interface) which is a graphical front-end tool that uses the SMILE engine [13].

The CDG framework was demonstrated by performing verification of two example designs: a combinational logic design and a sequential logic design. A 4-bit Wallace Tree multiplier was used to demonstrate the CDG solution for combinational logic. The following experiments were performed to demonstrate the solution for combination logic:

- **Experiment 1:** Generate all possible results of the multiplier using a Bayesian network with parameters based on previous knowledge for generating all results.
- **Experiment 2:** Generate all possible results of the multiplier using a Bayesian network of initialized parameters with no previous knowledge for how to generate results.
- **Experiment 3:** Generate all possible 2-result sequences for the multiplier using a Bayesian network with no previous knowledge of generating results.
- **Experiment 4:** Perform verification using a Bayesian network based on input properties and random distribution settings instead of input values.

- **Experiment 5:** Perform inference using a more complex Bayesian network to reach interesting internal states such as certain full adder carry bits being set.

The CDG solution for sequential logic was demonstrated by targeting a state machine for a vending machine. An experiment was performed using a Bayesian network to track paths through the state machine based on random test input parameters. This demonstrated how Bayesian network inference can be used to reach certain paths through the state machine more often.

Results from the experiments were compared to true random tests to show improved rate of test coverage. Details of Bayesian network inference results were also shown to allow for higher quality tests to be generated. Although the example designs used are very simple compared to the typical complex designs that warrant CDG, the use of both combinational logic and sequential logic targets demonstrate how these techniques can be applied to large variety of more complex digital designs. It is also shown that the computational complexity of Bayesian networks scales efficiently for large models, therefore allowing this technique to be feasable for complex designs.

Research has been done on other methods for achieving CDG besides using Bayesian networks. One method is Model-based CDG in which an abstract model of the design under verification is used to anticipate how much coverage will likely be reached during the construction of new tests [22]. Another approach has been to use constraint extraction as a method of creating new random input constraints from previous simulation data [23]. Research has also been done on applying inductive learning techniques to generate generic rules for closing coverage based on previous simulations [24]. One of the benefits of using Bayesian

networks for CDG is that there exists a large selection of Bayesian inference tools and a large knowledge base of Bayesian techniques due to their application to many types of problems outside of verification.

1.3 OUTLINE OF THESIS

The outline of this thesis is as follows: Chapter 2 gives an overview of verification methods for integrated circuit designs including the motivations for coverage directed generation. Chapter 3 gives an overview of Bayesian networks and how they are generally applied to various applications. This is followed by a summary of previous research that demonstrates how Bayesian network can be applied to verification of digital hardware. Chapter 4 gives a detailed description of the CDG framework implementation including testbench, Bayesian network creation, and custom software for inference calculations. Chapter 5 demonstrates the application of the CDG framework to a 4-bit Wallace Tree multiplier. Chapter 6 presents a solution for applying the CDG framework to a vending machine state machine design. Finally, conclusions for the experiments performed are discussed along with several ways the CDG solution could be improved in future work.

2.0 OVERVIEW OF VERIFICATION METHODS

This chapter gives an overview of standard verification methods used in industry for digital integrated circuit designs. The challenges of functional verification are presented to demonstrate the motivation for *Coverage Directed Test Generation* (CDG). The concepts of constrained random testing and coverage analysis are discussed as well as standard testbench techniques and an overview of a basic CDG solution.

2.1 VERIFICATION OF HDL DESIGNS

Technologies and methods for verifying the correctness of integrated circuits have evolved to address the increasingly large and complex chip designs being made. The majority of verification efforts occur in simulation or analysis of designs implemented in a hardware description language (HDL) such as VHDL or Verilog. This allows bugs to be discovered early in the design process, before the physical circuit layout is designed and before fabrication of the actual hardware occurs. Recent studies have shown that the majority of bugs found in integrated circuits are due to functional design errors and incorrect specifications [20]. These types of errors are introduced during HDL implementation phases.

Performing verification of HDL implementations also allows for a much more thorough verification than what can be achieved by performing tests on the final hardware. HDL design

errors that go undiscovered can result in fabrication of a final product with an unknown functional bug. Such a bug may only occur under extremely rare conditions and therefore is unlikely to be discovered in basic chip-level testing. Most of today's integrated circuit designs have a state space so large that it is not possible to exhaustively test all functions under all plausible scenarios on the final chip that is manufactured. Verification is therefore accomplished on the HDL code, usually by breaking large designs into smaller functional units that are easier to manage.

One of the primary methods currently used for design verification is functional verification. Functional verification deals with simulating the HDL implementation with a series of tests in order to verify the functional requirements of the design are met. Functional verification is the primary focus of this paper and is discussed in more detail in the following section. Other methods for verifying designs include formal verification, which is done by performing an analysis of the design and applying mathematical proofs that a certain property holds true for all possible inputs. Because formal verification has many complexities, it can only be applied in certain circumstances and is therefore not always an alternative to functional verification [21].

2.2 FUNCTIONAL VERIFICATION

Functional verification uses dynamic simulation to verify the functional requirements of the design under test (DUT) are met. Electronic Design Automation vendors supply tools to the industry that allow designs in an HDL to be simulated, showing the effects of input signals, internal design states, and outputs over time. The challenge to verification engineers is to

develop testbenches that simulate all the scenarios necessary to verify the functional requirements are met as well target special scenarios that may be critical to functionality or more likely to have problems.

In previous practice, it was common for testbenches to be implemented in the same hardware description language as the design (i.e., VHDL, Verilog). In more recent practice, the need for automating functional verification has led to the use of Hardware Verification Languages (HVLs) such as ***e*** [9] and SystemVerilog [19]. These languages combine aspects of traditional software programming languages with HDL capabilities. This combination allows for more powerful testbenches that can interface with a DUT while performing advanced software functions. Along with these languages, methodologies such as the Open Verification Methodology (OVM) [14] and e Reuse Methodology (eRM) [4] have been developed to provide reusable solutions and promote industry standard practices to verification engineers. The experiments conducted in this study were implemented using SystemVerilog with OVM.

2.2.1 Basic Verification Methodology

Most methodologies that have been developed for hardware verification use the basic testbench model shown in Figure 2-1. The design under test is interfaced with various testbench components that specialize on certain tasks. A sequencer is responsible for generating sequences of test inputs, usually at a high level of abstraction so it is easy to form test cases. This allows test inputs to be defined in terms of high level constructs such as packets or instructions, removing the complexities of bit-level input test cases. The driver is then responsible for translating the generated sequences into a transaction that is inputted directly to the DUT. The driver adheres to all protocols and handshaking required by the DUT input specifications [21].

A testbench will use monitors on both the inputs and outputs of the DUT to passively monitor transactions and collect data for checking. Monitors will usually only check that the DUT protocols are followed correctly and pass on actual data to other components for data checking. Input data is typically passed to a reference model to predict the expected output of the DUT. The DUT outputs are typically passed to a scoreboard which checks that the outputted data matches the predictions of the reference model. This general approach to verification has become standard in industry, with varying degrees of complexity and customization to suit verification environment needs [10]. The testbenches created for the experiments in this paper use this model.

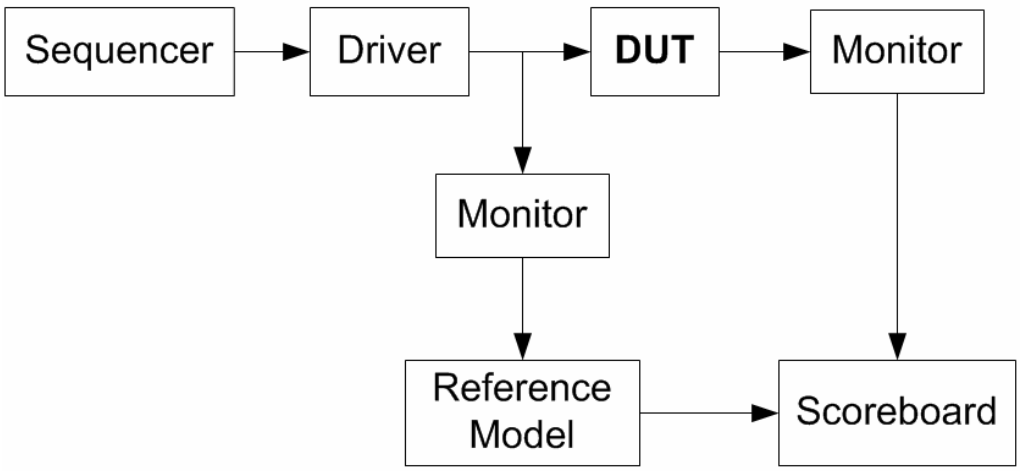


Figure 2-1: Basic Testbench Structure

2.2.2 Constrained Random Input Generation

Traditionally, engineers would write directed tests to complete the verification of a design, with each test targeting a specific requirement. The complexity of designs today make this impractical since the amount of manual labor required to write the sufficient number of test cases

is unrealistic for business. Constrained random input generation has mostly replaced this practice since it removes much of the manual labor required. With relatively minimal manual effort, a random input generator can cover a vast amount of the design state space, verifying many functional requirements. Random inputs provide an additional advantage of potentially creating test scenarios that weren't considered in the verification plan. Therefore the thoroughness of the final verification may actually be better than what was planned and may make up for a weakness in the verification plan [10].

For most complex designs, it is statistically improbable and sometimes impossible to generate all the necessary scenarios randomly. This is particularly difficult for special corner cases that can only occur under rare circumstances. One of the challenges to using constrained random inputs is determining when to wait and hope the required scenarios are generated, and when to stop. Usually after stopping, one must either write directed tests to cover the remaining test scenarios, or modify the random input generators with different constraints in order to help guide the inputs into the necessary test cases. With either approach, a considerable amount of manual effort may be required if the design is complex. This is because it may not be easily apparent to an engineer what inputs are necessary to drive a design into a complex scenario of internal states.

2.2.3 Functional Coverage

The use of constrained random input generation presents the challenge of determining when verification is complete. Coverage monitoring is a method of defining goals in the testbench and tracking how many of these goals have been reached. Functional coverage defines these goals to track specific functions and scenarios that need to be exercised according to the verification plan.

After running simulations, a coverage report can be generated and analyzed by an engineer to identify the coverage holes (untested scenarios). Most HVLs include constructs for defining coverage metrics in the testbench code and EDA simulation tools implement the necessary tasks of tracking and reporting coverage results. The verification engineer therefore only needs to be concerned with defining the coverage goals in the testbench code and interpreting the results from the simulation tools.

2.3 COVERAGE DIRECTED TEST GENERATION

Several challenges to functional verification were discussed in Section 2.1. These challenges involve how to handle the manual effort required to analyze coverage reports, perform analysis of coverage scenarios left untested, and determine how to modify the testbench to reach 100% coverage of the verification plan. For complex designs, a significant amount of manual labor may be involved in each of these tasks. Figure 2-2 shows a simplified model of this process.

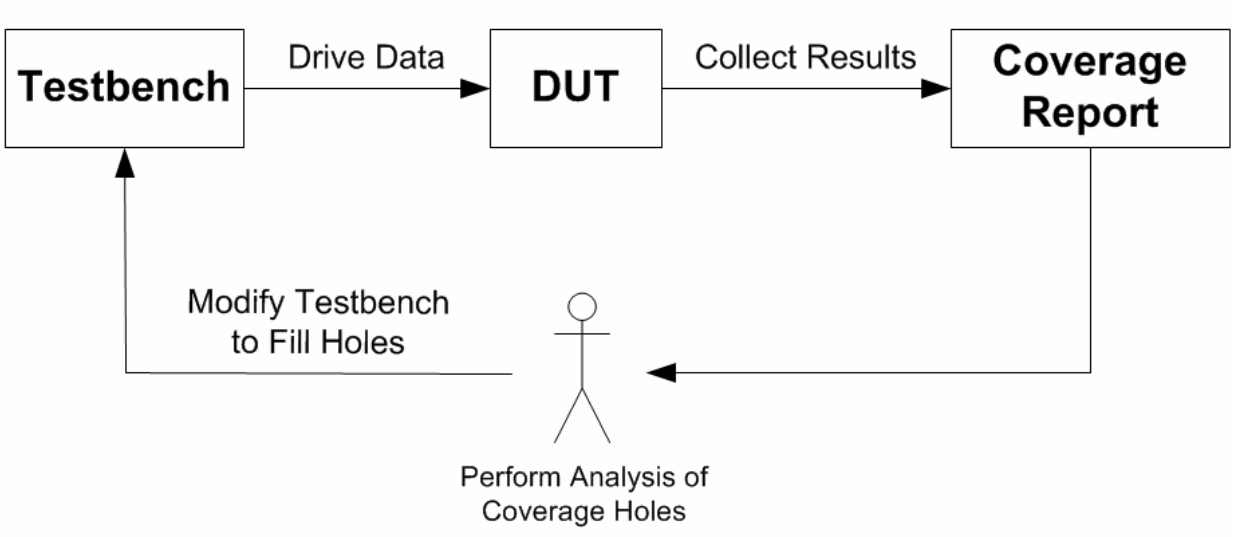


Figure 2-2: Typical Process for Coverage Driven Verification

2.3.1 Closing Coverage with CDG

Sometimes the solution for complete coverage is using directed tests for the missing scenarios, however this may require manual effort in both identifying the necessary inputs and creating a customized environment for verifying the scenario. It has been shown in industry that directed tests should be avoided when possible since the specialized environments for directed tests are usually not reusable. The goal of recent verification methodologies has been to move away from this practice and capitalize on reuse of verification components across design phases and projects [10].

The goal of coverage directed test generation (CDG) is to replace the manual feedback stage shown in Figure 2-2 with automated coverage analysis. This would allow software to programmatically find coverage holes, make predictions on how to reach the untested scenarios, and modify the testbench accordingly. This can be done by having adjustable parameters on the constraints for random input generation. These adjustable parameters can be modified at simulation runtime to make more desired types of inputs more likely to be generated, while reducing the chance of generating undesired inputs.

There are two benefits that can be achieved by applying CDG. The first is that unobserved scenarios can be generated more quickly and allow the verification process to complete sooner. The second benefit is that certain scenarios can be more easily tested multiple times with different input parameters. In many designs, different combinations of inputs exist that may lead to the same test scenario. Although the coverage goal may be met by only exercising one of these input combinations, there is value in exercising the other input combinations as well since there may be a hidden bug that only occurs for certain inputs [5].

2.3.2 Generating Unplanned Scenarios

It is important to consider two seemingly conflicting goals when applying CDG to verification: the goal to reach 100% coverage quickly and the goal to test unplanned scenarios. At best, CDG may allow for complete coverage to be met very quickly. If coverage is closed too quickly, one of the primary values of using random inputs may be lost, which is the potential of generating test scenarios that weren't conceived in the verification plan. This is one of the primary motivations to keep random testing in a CDG solution. One could create a CDG implementation in which sets of directed tests are automatically generated to deterministically drive the scenarios that are needed for coverage. While this may allow verification to be completed, it will not test any unplanned scenarios. The high complexity of many designs increases the chance of a bug slipping through the verification process due to engineers neglecting to consider certain scenarios in the verification plan. Allowing unplanned test scenarios to be generated helps address this issue and provides a more thorough verification of the design.

The need to generate unplanned scenarios impacts how random input constraints are managed in the CDG process. At one extreme, the constraints may be too loose and coverage cannot be closed efficiently or at all. At the other extreme, the constraints may be too tight and result in essentially the same thing as inputting directed tests. The best solution may often be in finding a balance between these extremes so that complete coverage can be met in a reasonable time, while still allowing the benefits of random tests.

3.0 OVERVIEW OF BAYESIAN NETWORKS

This chapter presents an overview of Bayesian networks. The basic principles of Bayesian networks are demonstrated using examples for modeling a domain to by analyzed. The methods for performing inference in order to generate predictions from known evidence are shown. This is followed by examples of applying these methods to functional verification for digital hardware. The basic framework for including Bayesian networks in *Coverage Directed Test Generation* (CDG) is demonstrated with details of how to model the domain and generate predictions appropriate for constrained random testing.

3.1 BASIC BAYESIAN NETWORK CONCEPTS

Bayesian networks are acyclic graphs that model the direct dependence and conditional probabilities between variables in a domain. These models can be used to calculate the probabilities of certain events likely to occur. All the variables being analyzed are represented as nodes in the graph, each with a set of possible mutually exclusive states. Directed arcs are drawn between variables to show a direct dependencies between the variables. Each node in the graph also includes a conditional probability table. A *conditional probability* is the probability of an event given existing knowledge (or evidence) of the state of a related variable. The common notations for the probability of event A given the known evidence of event B is **P(A | B)**. This

probability is also sometimes referred to as *likelihood* since it serves as a measure of how likely the event A has occurred. The conditional probability table for a node in the network lists the probability of each state for each of the possible combinations of state of the parent nodes. Such a table is also called a *joint probability table* since it provides probabilities for every scenario of parent node values [12].

3.1.1 Example Bayesian Network

Figure 3-1 shows an example of a basic Bayesian network modeling the variables likely to cause grass to be wet [2, 11]. In this model, the two variables that may cause wet grass are if it rained and if the sprinkler system had turned on. Each node in the graph represents the state of the variable (true or false), while the arcs show the direct dependencies between these variables. The grass being wet directly depends on whether it rained or the sprinkler system was running. Additionally, the sprinkler system will usually be turned off on days it rains, so the presence of rain directly influences sprinkler state.

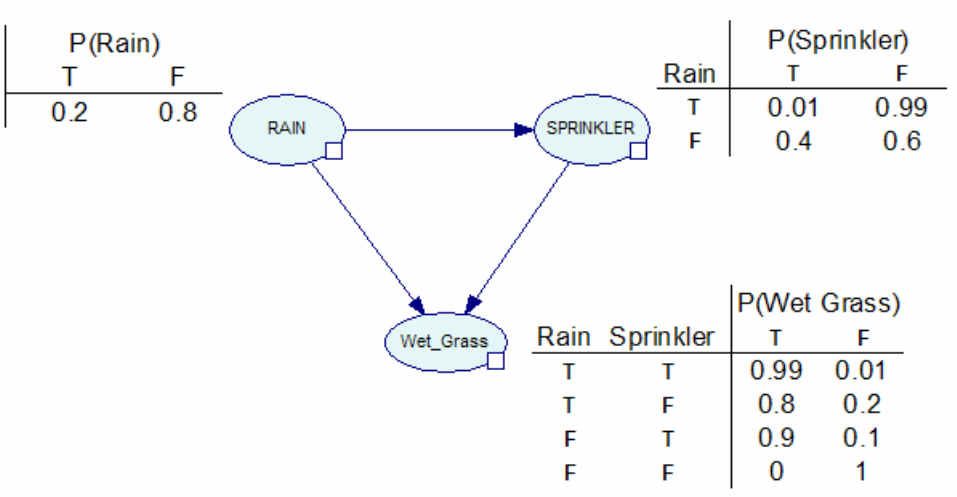


Figure 3-1: Example Bayesian Network (Based on [2, 11])

Each node in the network contains a conditional probability table with the likelihood of each state to occur given the states of parent nodes. The probability of rain occurring, P(Rain), is defined to be 20% on a given day. The sprinkler node is directly dependent on the rain node and therefore the probability distribution table provides different values depending on the presence of rain. If the occurrence of rain is true, there is a 99% probability that the sprinkler system was turned off, vs. a 40% chance of the sprinkler running on a day it doesn't rain. The node for wet grass contains a larger probability distribution table since it depends on both of the other two nodes. A probability of the grass being wet is specified for each of the different combinations of the other node states.

3.1.2 Joint Probability Calculations and Inference

The combination of information stored in each of the conditional probability tables along with the network structure can be used to calculate the joint probability distribution of the entire domain (the individual probabilities of all variable states). This is given by the following equation, referred to as the chain rule for Bayesian networks [11]. With the probability of each variable A_i given its parents $pa(A_i)$, the joint probability distribution for a universe of variables U is given by the following:

$$P(U) = \prod_i P\left(A_i \mid pa\left(A_i\right)\right)$$

This equation shows that the joint probability for all variables in a domain can be calculated as the products of individual conditional probabilities throughout the structure of the network. This allows Bayesian networks to scale efficiently to very large models with relatively low computational complexity.

The joint probability of all variables can also be calculated with evidence of known states specified. This is done using Bayes' rule which shows how to calculate the likelihood of a variable *A* given evidence *e*:

$$P(A \mid e) = \frac{P(e \mid A) \bullet P(A)}{P(e)}$$

Bayes' rule also allows for the belief in this likelihood to be updated as more information is added to the network. In this context, the term *P(A)* is usually considered a prior probability of *A* (previous belief of likelihood) and *P(A|e)* is considered the *posterior* probability of *A* given the known evidence [12]. This provides a way for Bayesian networks to perform statistical inference and predict the most likely states of variables given certain evidence.

3.1.3 Inference with Example Bayesian Network

The example in Section 3.1.1 demonstrated a basic Bayesian network for showing various events that may lead to grass being wet. Inference can be performed to determine the most likely cause of the wet grass. Figure 3-2 shows two versions of inference with the observed evidence that the grass is wet. With the new evidence specified, the network can update the new probability distribution of the states given the new information. The example on the left shows that there is a 65% chance the sprinkler system was running given the observation of wet grass.

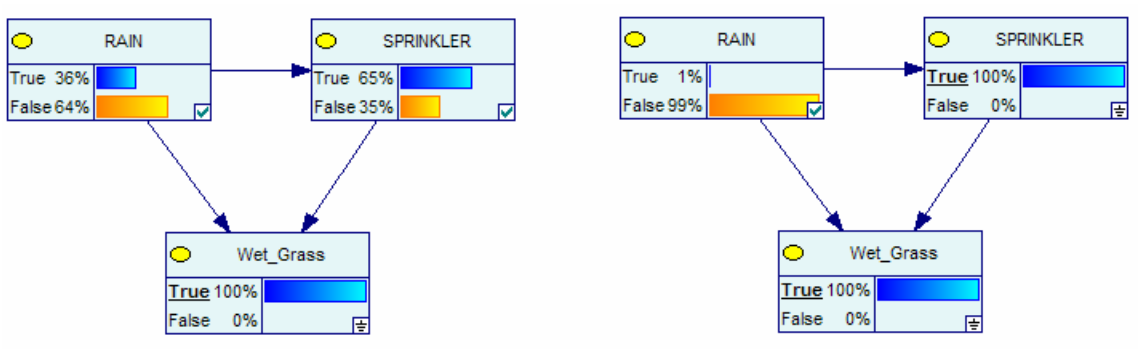


Figure 3-2: Example Bayesian Network with Inference

The second example in Figure 3-2 shows additional evidence being applied to the sprinkler node, indicating the observation that the sprinkler system was in fact on. Updating the network beliefs shows a 99% probability that it did not rain. This type of result is common in Bayesian networks and is referred to as "explaining away" a state. In a model where a state may be caused by two other nodes, the added knowledge that one node was the cause greatly reduces the chance of the other node also being a cause [15]. In this case, the observation that the sprinkler system was on greatly reduces the probability that it had rained.

In the example of Figure 3-2, the inference results gave a full probability table for each of the parent nodes. Another useful calculation is the *Most Probable Explanation* (MPE), which determines the most likely state of each of the nodes and the probability of that network configuration occurring [12]. This is similar to calculating a full probability table of each node and identifying the states with the highest probability. The primary difference is the algorithms used for MPE can be calculated much faster and ensure the estimates for each of the variables agree on the common explanation. For example, a network configuration could have two variables with highest probability estimates for states that are less likely to occur together. The MPE calculation would then likely not estimate a configuration with both of these states.

Therefore MPE calculations may often be more appropriate for analysis than calculating the maximum of each individual probability table.

3.1.4 Learning Network Parameters

Bayesian networks are used for a variety of applications where statistical conclusions need to be made from previous observations of data. The values in the probability distribution tables can be learned from historical data using the *Expectation Maximization* (EM) algorithm. This algorithm can determine the most likely network parameters when working with complete and incomplete sets of data [12]. Figure 3-3 shows a common process for applying Bayesian networks to any domain under analysis. A person must collect observations from the domain, use this data to learn network parameters, and then query the network for probabilities of states to make predictions. As more observations are made, sets of data can be repeatedly used to relearn network parameters, gradually increasing the accuracy of the model.

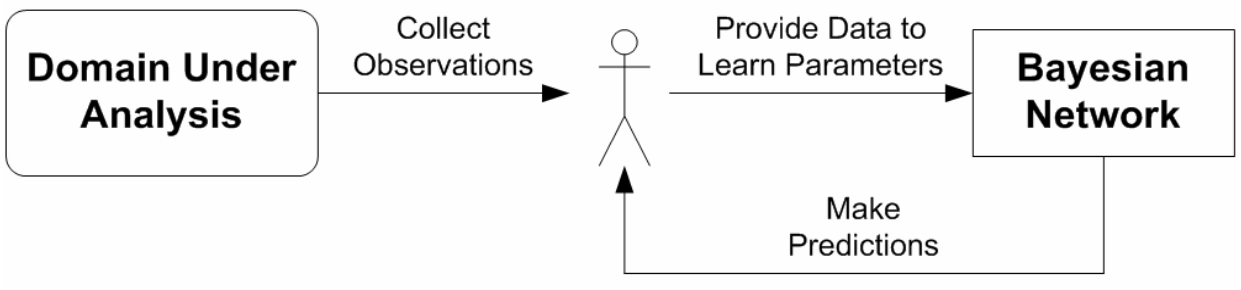


Figure 3-3: Basic Model for Applying Bayesian Networks

A classic example of applying Bayesian networks is to diagnose an illness given certain symptoms. Doctors can track the history of symptoms that have been observed for patients over

large periods of time, then use this information to generate a Bayesian network with potential diseases as parent nodes and the symptoms as children. Since each disease may result in different combinations of symptoms with different probabilities of each occurring, the network can combine all the data to quickly diagnose a new patient along with a probability of this diagnosis being correct. Similar to the wet grass example, this is done by setting the evidence of observed symptoms on the children and applying Bayes' rule to generate posterior probabilities of the parent nodes.

When generating Bayesian network parameters from historical data, it is important to realize that Bayesian networks can only provide probability calculations for states that have been observed. For diagnosing patients, if a doctor never observed a certain combination of symptoms occurring together, the network would be unable to predict a diagnosis. Bayesian networks can therefore be used for keeping a concise summary of past observations and making predictions based on how frequent certain observations were made.

It is also useful to consider the value of Bayesian networks over other methods of tracking trends. One could conceivably perform some of the same analysis Bayesian networks provide using simple tables of historical data. For diagnosing an illness in which one has a headache, one could simply parse through all the previous cases in which headaches were found and count which illness was the most commonly occurring cause. However this approach requires maintaining historical data in huge tables that do not scale well to large domains. If the analysis were to include 100 possible symptoms, the required table of data would have to be at least 2^{100} entries. This is clearly too large to be manageable for most studies and is a realistic scenario for some applications.

3.2 APPLYING BAYESIAN NETWORKS TO VERIFICATION

The goal in applying Bayesian networks to functional verification of chip designs is to reach a fully automated framework for CDG. The manual effort normally required in coverage driven verification (Figure 2-2) can be replaced by a Bayesian network to track previous behaviors and generate predictions to reach certain coverage goals.

3.2.1 A Framework for CDG

Figure 3-4 shows the CDG framework model with all manual effort removed from the process. The testbench collects the outputs from the design under test and provides data to the Bayesian network to learn the network parameters. The network uses this information to make MPE calculations and provide this data to the testbench. The testbench tracks the status of the coverage goals from the coverage reporting tools at simulation runtime. The testbench then determines how to modify input generation parameters to reach the coverage holes based on the MPE predictions of the Bayesian network.

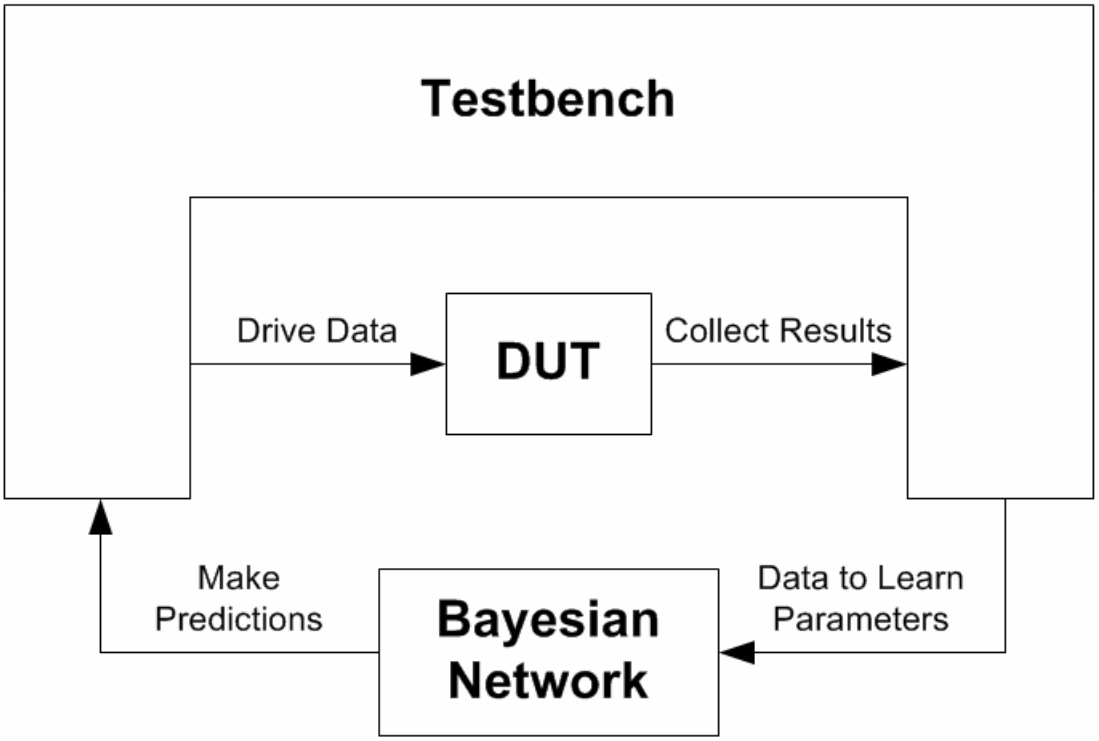


Figure 3-4: Automated CDG using a Bayesian Network

In this CDG model, the testbench and Bayesian network each mutually depend on the other for information. The normal process of applying Bayesian networks (Figure 3-3) involves manually making observations from a domain and training the network with this data to be a more accurate model. Without this information, the network cannot make any predictions. The testbench fulfills this dependency by continuously providing observations of outputs and internal states from the DUT. The longer the test environment runs in simulation, the more accurate the Bayesian network will become.

The testbench depends on the Bayesian network for predictions on how to reach certain test cases. However, the network can only provide predictions on how to reach events that have already been observed. In the event that that network cannot provide a prediction, the testbench must either continue running with true random (unbiased) test inputs or “guess” the constrained

random generation parameters. In either case, the hope is eventually the testbench will exercise the desired event, at which point the Bayesian network has the data needed to update the probability distribution tables. The Bayesian network is then able to make predictions for how to repeat the event in the future.

3.2.2 Reaching Unobserved Events

Since the network cannot make predictions for unobserved states, it is primarily useful for generating predictions to reach an unobserved combination or sequence of states for which each individual event was previously observed. To clarify this concept, this paper will distinguish between the terms *event* and *state*. In this context, a *state* refers to a single more basic observed coverage goal of a design under test, such as a certain output or internal signal value. This could be the output of a multiply instruction, an internal carry bit being set, or the presence of a load instruction in a certain stage of a pipelined processor. The term *event* will be used to refer to a scenario made up of a combination or sequence of states.

A verification plan may include many complex coverage events that are made up of sequences or combinations of more basic states that are individually easier to observe. For example, if generating instructions for a pipelined processor, many add and branch instructions may have been observed, but never in consecutive stages of the pipeline. The Bayesian network can guide the testbench to more likely reach this event since it can make individual predictions for generating both types of instructions. Many of these types of events are also called *cross coverage* goals since they refer to certain combinations of more basic coverage goals occurring together [10].

The Bayesian network is also useful for learning how to recreate a certain scenario of interest. An obscure bug might be found in a design for which it is difficult to pinpoint exactly what conditions led to the error. The network can help repeat such a scenario and provide information to determine the combination of states that led to the bug being discovered.

3.2.3 Applying Bayesian Networks

Bayesian networks have been successfully applied to hardware verification in recent research (see [1, 5, 6, 7]). Basic methods have been presented for defining the network and generating queries to make predictions for a specific DUT. The most basic method is to manually construct the Bayesian network based on the domain knowledge of the design under test.

Figure 3-5 shows an example Bayesian network for a multi-core processor (see [5]). The parent nodes of the network are chosen to be the input parameters of the constrained random input generators. In this example, the input parameters are the weights to randomly generate a given CPU instruction (i.e., read, write), and the weights to determine which CPU core is active (or both). The other end of the network contains the coverage variables being tracked in the verification process. These may be used to track coverage goals such as verifying whether every type of command is tested with either core enabled and ensuring every possible response is generated.

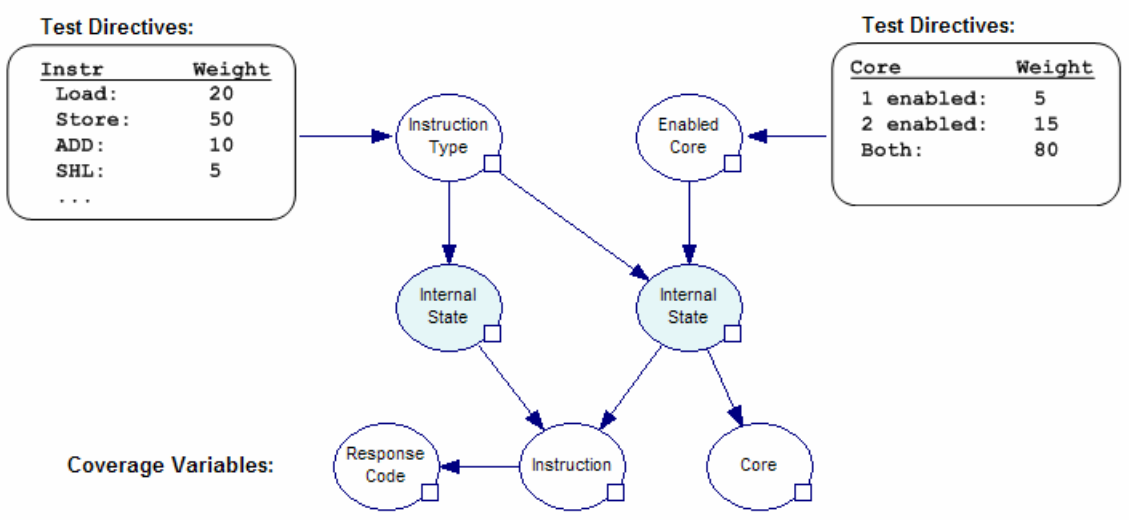


Figure 3-5: Example Bayesian Network for Verification (Based on [5])

The internal nodes of the network represent internal states of the design under test. The arcs of the network capture the direct dependencies, starting with how the test input directives influence the internal states and how the internal states influence the outputs and coverage variables. This model also uses hidden nodes (indicated with shaded color) to model internal states that are not observed during simulation. The hidden nodes allow complex internal states that may influence coverage variables, but are not directly observable. They also serve to reduce the complexity of the Bayesian network. Once the Bayesian network is defined, data sets can be generated from simulation runs to capture each of the variable states for different test directives. The *Expectation Maximization* (EM) learning algorithm, as described in Section 3.1.4, can then be used to update the probability distribution tables of the network based on the observed test data. At this point, the network can be queried to make predictions for test inputs [5].

Using the Bayesian network in Figure 3-5 to make predictions can be done by setting evidence to the coverage variables for the state that is desired. For example, to determine how to most likely generate a load command with a certain response code and core 1 enabled, the

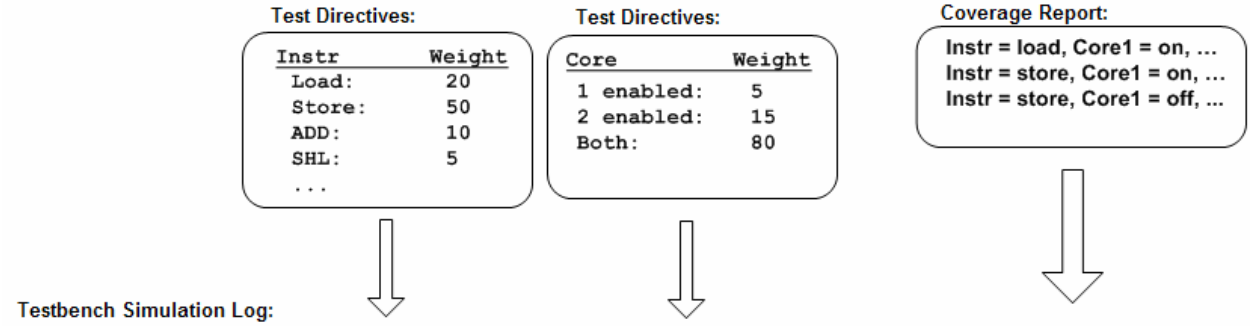
Instruction node would be set with evidence “load”, the Core node set to “1 enabled”, and the Response Code node set to evidence of the desired response code. Inference can then be performed by applying Bayes’ rule as described in Section 3.1.2. The likelihood of the parent node states will then be updated given the specified evidence of the coverage variables. The resulting probability estimations on the inputs will directly indicate the best test generator directives to use. The testbench can then be automatically updated to use the test directives with the highest probability for reaching the desired coverage goal. Bayesian inference engines such as SMILE [13] implement the necessary constructs and algorithms for defining a Bayesian network and performing all of these actions.

3.2.4 Choosing Domain Variables for the Network

One of the challenges to applying Bayesian networks in the way described in the previous section is the requirement of applying expert knowledge to build the network structure. Although the effort of building the probability distribution tables is reduced through the EM learning algorithm, one must take care to manually build the network structure, choosing which directives, coverage variables, and internal states should be used. Previous research (see [7]) has shown that using too many input directives and coverage variables will lead to a network that is too complex to be useful. The EM learning algorithm will not work if the model is too large. Even if the model is small enough to allow learning, larger models require much larger data sets to have accuracy. Otherwise, the network may make predictions based on a few randomly coincident states that have no real correlation. Therefore the best structure usually requires an expert to determine a few input directives that have most influence toward reaching a smaller subset of coverage goals.

Another decision one must make when building the network is how to model the input parameters in the network. The more traditional way of using Bayesian networks is to use the values of the inputs as the input variables. In the case of processor instructions, the node would have a state for each instruction, tracking how often a load or store instruction occurred. For applying Bayesian networks to verification, previous research has instead used the weights of the test generation parameters rather than the actual values of the inputs. In this case, the input nodes for the processor might be the weighted percent probability of generating a load instruction [1].

Figure 3-6 shows an example of creating a simulation data set to be used for learning parameters of a Bayesian network (based on [1]). The two inputs to the data set are the settings of the test directives and the observed coverage event for each test case. The input directives are captured as the probability weight used on each test input. For example, the first three test cases used directives that the instruction had a 20% chance of being a load, a 50% chance that the instruction is store, and a 30% chance of Core 1 being enabled.



Testbench Simulation Log:

P(Instr = load)	P(Instr = store)	P(Instr = Add)	P(Instr = SHL)	P(Core1 = enabled)	. . .	INSTR	RESP	CORE
0.2	0.5	0.1	0.05	0.3		add	overlow	both
0.2	0.5	0.1	0.05	0.3		store	OK	2
0.2	0.5	0.1	0.05	0.3		load	OK	both
0.5	0.1	0	0.4	0.5		load	OK	both
0.5	0.1	0	0.4	0.5		store	INV	both
0.5	0.1	0	0.4	0.5		load	OK	1
. . .	. . .	. . .	. . .	. . .		. . .	. . .	. . .

Figure 3-6: Example Data Set Generation from Testbench (Based on [1])

The primary motivation for using test directives as opposed to actual input values is driven by the goals of constrained random test generation for verification. As discussed in Section 2.3.2, constrained random tests aim at providing rich data sets to reach complex scenarios (combinations of internal design states), including scenarios that were unplanned. If random test generation is constrained too tightly, the verification process may be hindered. If actual input values are used as Bayesian network nodes, there is a possibility the network will make predictions that prevent rich data sets from being generated. For the processor example, this might result in an abundance of load instructions. This result may reach the desired coverage goal quickly, however the test generator would then be overly constrained and unable to generate other instructions mixed with the load instructions, therefore reducing the chance of reaching more interesting scenarios that need to be verified [1].

An interesting aspect of using the probability distribution settings as network input states instead of the actual input values is that the Bayesian network does not maintain any knowledge of the actual generated inputs. It only tracks the input generation settings and the resulting coverage events that occurred. Therefore the inputs to the Bayesian network are considered *soft evidence* since they do not always correlate to the same observed inputs [1]. The same input directives will produce slightly different results over time, which is desired for constrained random testing. While the alternative of using actual input values in the network can be useful for generating tests, the result will be predictions that are more deterministic. This method would be similar to automatically generating directed tests. If the network is queried to generate a load instruction, it will simply give the settings that will most likely generate the instruction deterministically *every* time. Using soft evidence will instead result in predictions that are likely to generate a load instruction *most* of the time.

This chapter has presented an overview of Bayesian networks and how they can be applied to functional verification to reach a fully automated CDG test environment. The next chapter will describe the implementation of the CDG framework, including the use of existing verification tools, Bayesian network creation, testbench implementation, and custom programs to perform inference. This is followed by a series of tests in Chapter 5 to demonstrate the use of the CDG solution.

4.0 IMPLEMENTATION OF CDG SOLUTION

This Chapter describes the implementation details of a completely automated solution for *Coverage Directed Test Generation* (CDG) using commercially available tools and industry standards for verification. The functional verification tools will be described as well as testbench implementation and verification methodologies applied. This is followed by a description of the Bayesian inference tools used for constructing the Bayesian networks and performing inference at simulation runtime for generating test input parameters.

4.1 FUNCTIONAL VERIFICATION TOOLS AND TESTBENCH

The Questa verification platform was used for dynamic simulation of a design under test (DUT) and execution of testbenches for verifying the design [17]. Testbenches were implemented using the SystemVerilog verification language [19] along with the Open Verification Methodology (OVM) [14]. This allowed for a complex verification environment to be created relatively quickly with reusable components. The use of current industry standards for testbench design also help demonstrate how the solution can be adopted for other designs.

A different testbench was created for each of the example DUTs used for experiments, discussed further in Chapters 5 and 6. In all cases, the testbenches used the structure of Figure 4-1. This is similar to the generic testbench structure previously shown in Figure 2-1 except

without a reference model and scoreboard for checking results. This was done since the purpose of these experiments was to demonstrate the relationships between test input generation and coverage closure. In all experiments, the designs were simple enough to assure no bugs existed with the initial basic tests that were performed. Therefore there was no need to actually check the outputs of the designs for correctness during the experiments. However in a real verification scenario, these components are essential to finding bugs in a DUT.

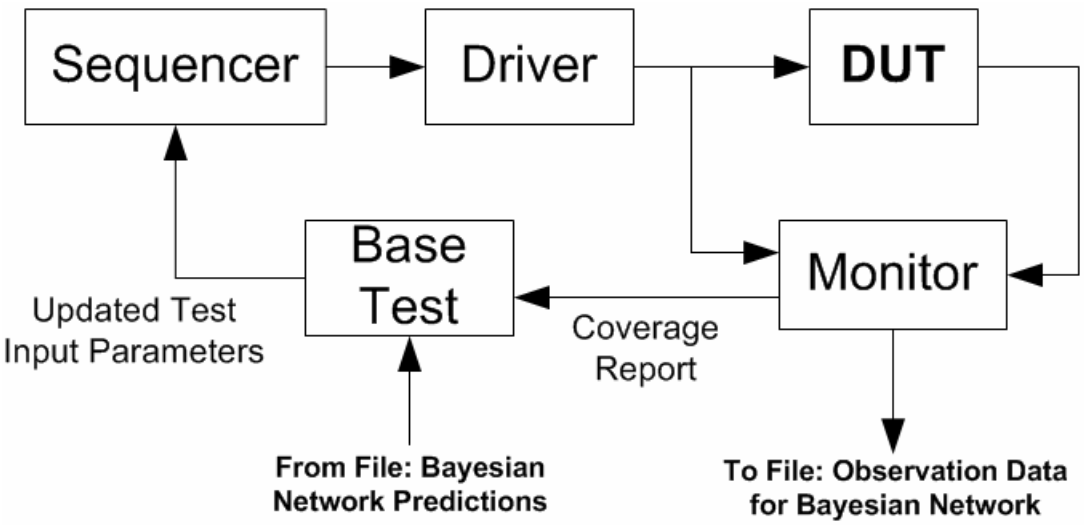


Figure 4-1: Testbench Structure for CDG Solution

The testbench used a sequencer with control variables for test generation parameters. These control variables were changed at runtime by the base test being executed to change the probability distribution for generating inputs. Initially, all control variables were set to equal values to allow a flat probability distribution for input generation. Doing this resulted in purely random (unconstrained) input generation with an equal chance for all inputs occurring. During simulation time, the base test component would periodically check the coverage report from the monitor to determine which cover points have not been reached. The base test would then

change the input generation parameters of the sequencer according to the values predicted by the Bayesian network to more likely reach the needed state. Bayesian network predictions were written to a log file on a periodic basis as more data was learned from the simulation. An example log file is shown in Appendix C, which shows a list of Most Probable Explanation (MPE) calculations for each of the possible outputs of a 4-bit multiplier. When determining which inputs are needed to drive the DUT, the testbench would look up the MPE in the log file, indexed by the result that is needed.

It should be noted that this implementation requires the Bayesian network to generate MPE calculations for *all* anticipated coverage states in advance on a routine basis. A more efficient solution can likely be achieved by having the Bayesian network generate only a single MPE calculation on demand, as needed by the testbench at runtime. This will be discussed more in Chapter 7 as part of the conclusions and future work.

The monitor of the testbench also logged in a file the outputs (and some internal values) of the DUT along with the corresponding input parameters that led to those outputs. This log consisted of a tab delimited listing with each column corresponding to one of nodes of the Bayesian network. The resulting data sets were used by the Bayesian inference engine to learn network parameters. During simulation runtime the testbench would continue to add results to the log, providing a continuously growing data set. This allowed the Bayesian network to relearn network parameters periodically, each time becoming a more accurate model of the domain.

4.2 BAYESIAN NETWORK TOOLS

The tools used for implementing Bayesian networks and performing inference consisted of the inference engine SMILE (Structural Modeling, Inference, and Learning Engine) and the graphical front-end tool GeNIe (Graphical Network Interface) [13]. Both tools are developed by the Decision Systems Laboratory at the University of Pittsburgh and have over ten years of development and field testing. Figure 4-2 shows the general structure of the CDG solution and how GeNIe and SMILE were integrated into the verification environment.

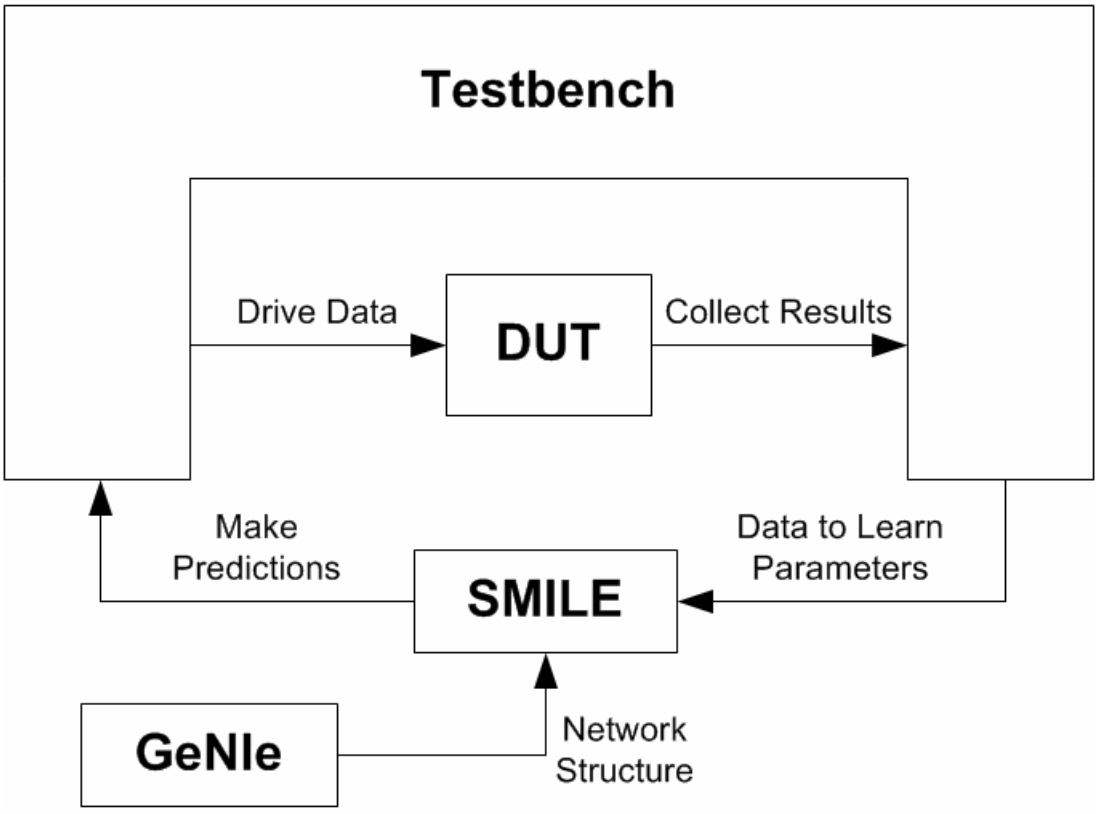


Figure 4-2: CDG Solution with GeNIe and SMILE

GeNIe was used as a graphical editor to manually create the structure of the Bayesian networks. This included the node names, names of states for each node, and arcs for showing direct dependencies between nodes. GeNIe also allows one to manually define probability distribution tables, set evidence to nodes, perform inference, and analyze results through graphs. The tool can also parse logs of data and apply the Expectation Maximization (EM) algorithm for learning network parameters. Therefore GeNIe was also used for doing some basic stand-alone inference calculations as initial experiments and examples (shown in Chapters 5 and 6). Networks built in GeNIe were saved to files which could be opened during simulation runtime of the testbench.

The inference engine SMILE is used by GeNIe for network definition and inference calculations. This engine is also available as a C++ Application Programming Interface (API), allowing custom programs to be created for building Bayesian networks and performing specific inference calculations. The CDG solution used custom C++ programs with the SMILE API to perform the following operations:

- Open the Bayesian network file previously created in GeNIe
- Open testbench log files and learn network parameters from the data by applying the Expectation Maximization (EM) learning algorithm.
- Set evidence to coverage variable nodes for anticipated coverage states.
- Perform Most Probable Explanation (MPE) calculations for test input parameters given specified evidence.
- Generate a log file of all MPE calculations for each anticipated coverage goal. This log file was used by the testbench to determine how to change test input parameters.
- Save the Bayesian network with updated parameters back to a file.

When creating the Bayesian networks in GeNIe, the network parameters (probability distribution tables) were not specified. This ensured that the initial state of the networks at the beginning of testbench execution had no previous knowledge of the domain other than network structure. Therefore, no predictions could be generated by the inference engine at the start of simulation ensuring all predictions were based on newly observed events. In the event the testbench required predicted test inputs from the Bayesian network that were not available, the testbench would simply continue using flat distribution settings (unconstrained random inputs).

An important consideration when applying GeNIe and SMILE to the CDG solution was how to represent data. These tools require discrete data (as opposed to continuous data) for many operations. Discrete data in these tools are interpreted as strings, with each string corresponding to a different discrete state. For example, a 2-state node in the network could simply have the states “True” and “False”, or a 3-state node could have values “small”, “medium”, and “large”. Integer or floating point values are interpreted as continuous data by the inference tools and require discretization. For this reason, all values logged from the testbench were recorded as strings, with each string corresponding exactly to state names used when drawing the Bayesian network in GeNIe.

The SMILE API includes functionality to automatically parse tab delimited data files and correlate the columns and values to network parameters for learning. Testbench logs used headings for each column corresponding exactly to the string used when naming nodes in GeNIe. This ensured that when the network was opened for learning parameters, SMILE would correctly correlate the columns of testbench data to each node of the network, with all logged string values corresponding directly to a node state. To ensure numeric data was interpreted as strings, letter prefixes were added to the numbers for logging. For example, a positive 8 was logged as “p_8”

and a negative 8 as “n_8”. In some cases binary values were used in the form of “b_00110”. Defining all these strings in the Bayesian network ahead of time and using the same convention for testbench logs ensured SMILE could correctly correlate all data from the testbench to Bayesian network parameters.

This chapter presented a detailed description of the CDG solution using Questa for dynamic simulation and SMILE for Bayesian inference calculations. Chapters 5 describes several experiments that were performed to see how well the CDG solution works for a basic combinational logic design. This is followed by experiments for applying the CDG solution to a sequential logic design in Chapter 6.

5.0 BAYESIAN NETWORK FOR A COMBINATIONAL LOGIC DESIGN

This chapter presents a series of experiments that were performed by applying the solution for Coverage Directed Test Generation (CDG) described in Chapter 4 to a basic combinational logic design. Section 5.1 describes a 4-bit Wallace Tree multiplier that was used as an example design under test (DUT). Section 5.2 describes the Bayesian network used along with basic examples of inference being performed to generate test input parameters. Section 5.3 gives descriptions of four experiments and the results of applying CDG to the multiplier. The experiments performed were the following:

- **Experiment 1:** Verify the Bayesian network correctly learned network parameters by generating all possible results randomly and then using the network to predict inputs to recreate the previously observed results.
- **Experiment 2:** Verify the Bayesian network can correctly relearn network parameters at simulation runtime by starting simulation with unlearned parameters and periodically relearning from new testbench outputs.
- **Experiment 3:** Generate all possible 2-result sequences for the multiplier using a Bayesian network with no previous knowledge of generating results.
- **Experiment 4:** Perform verification using a Bayesian network based on input properties and random distribution settings instead of input values.

- **Experiment 5:** Perform inference using a more complex Bayesian network to reach interesting internal states such as certain full adder carry bits being set.

All simulations were performed on a computer with an Intel Xeon E5345, 2.33 GHz processor. In most cases the state space of coverage goals was small enough that simulations only required 1 to 2 minutes to complete the coverage goal, with exception to experiment 3 which used more challenging coverage goal and took as long as 20 minutes to complete.

5.1 OVERVIEW OF THE WALLACE TREE MULTIPLIER

A Wallace Tree multiplier was chosen as an initial example DUT since it is a well known logic design with practical applications while being relatively easy to understand [8]. The Wallace-Tree design also allows for interesting coverage goals to be defined since there are many internal states. Figure 5-1 shows the architecture of a 4-bit Wallace Tree multiplier. Two signed 4-bit numbers are multiplied to produce a signed 8-bit result.

The Wallace Tree inputs are all the partial products of the multiplicand and multiplier. The partial products are formed by performing a logic AND between all pairs of bit positions between the multiplier and multiplicand. For a 4-bit multiplier, these partial products are then added through three levels of full adders (FA) and half adders (HA), as shown in Figure 5-1. The final stage of adders forms a carry-look-ahead adder, producing the final 8 bits of the result. Each FA and HA has an output and a carry out. Verification coverage goals were defined to exercise all possible results from the multiplier, as well as some special scenarios for generating all 2-pair sequences of results and all level 1 FA carry outs being 1.

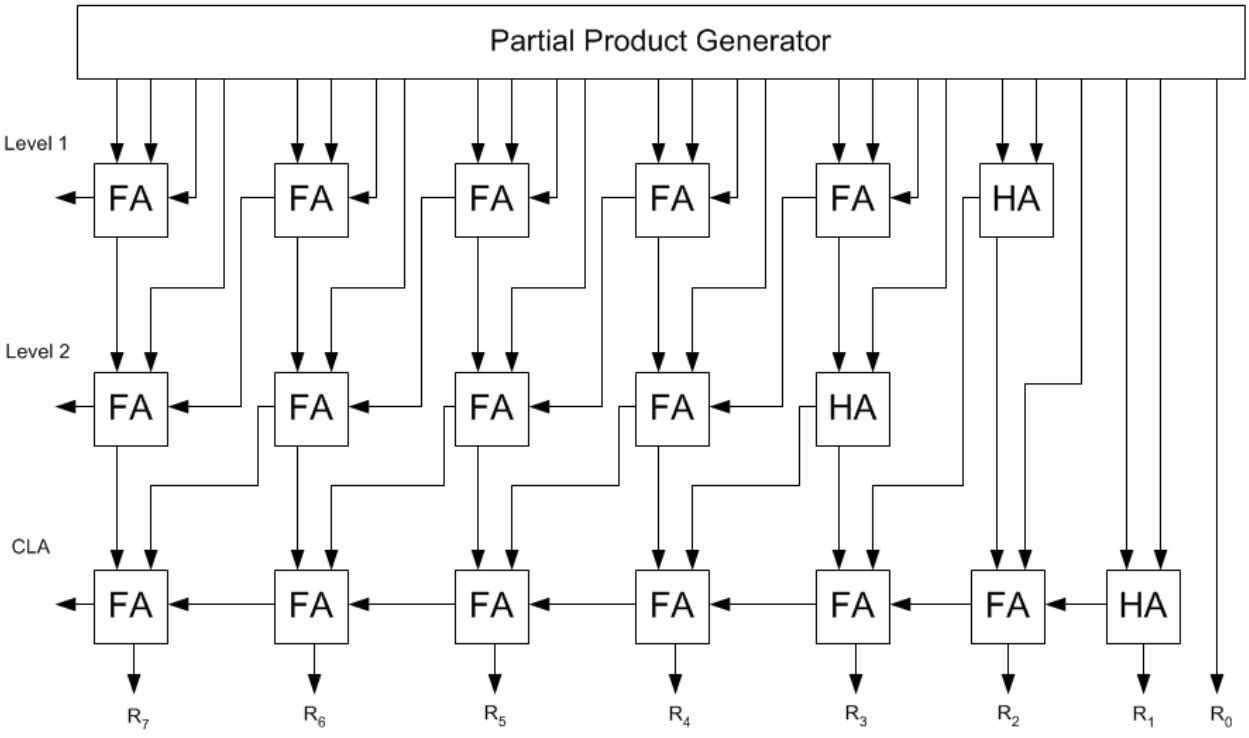


Figure 5-1: 4-bit Wallace Tree Multiplier (based on [18])

5.2 A SIMPLE BAYESIAN NETWORK

The simplest Bayesian network that can be used for the Wallace Tree is shown in Figure 5-2. This network has only three nodes: the value of the multiplier (MR), the value of the multiplicand (MD), and the result. In this scenario, the coverage goal is simplified to observing all possible results of the multiplier. At this high a level of observation, it is easy to see all direct data dependencies are accounted for since only the multiplier and multiplicand determine the result of the multiplier.

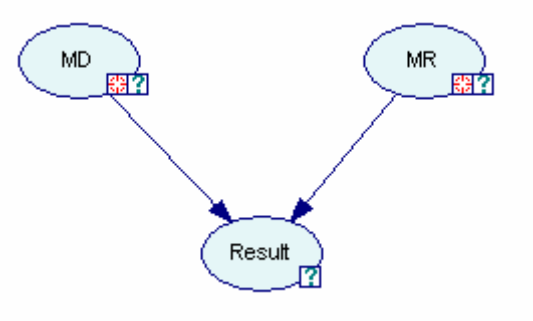


Figure 5-2: Basic 3-Node Bayesian Network

The MD and MR nodes each have 16 states corresponding to the multiplier input values -8 through 7. The result node has 256 states to model all possible 8-bit values, however many of these are impossible states since the only outputs can be the multiplication of two numbers between -8 and 7. Of the 256 states, only 60 are valid results that can be generated from the multiplier.

5.2.1 Inference with Simple Network

The Bayesian network for the Wallace Tree was first defined manually using GeNIe (Graphical Network Interface), which is a graphical front-end and inference tool for working with Bayesian networks [13]. A testbench was created in SystemVerilog to test a Wallace Tree multiplier implemented in Verilog. The testbench generated a series of random inputs to the DUT and collected all results and internal states of interest. This data was then used by GeNIe which applied the EM algorithm for learning network parameters. The initial scenario used 1000 test cases to ensure a large abundance of data covering all possible input scenarios.

After the parameters of the Bayesian network were learned, inference was performed by querying the network for different coverage goals. Figure 5-3 shows two instances of the

network with MD and MR nodes expanded to show bar graphs of the possible states. In both cases, the network was queried for which test inputs will most likely produce a result of five.

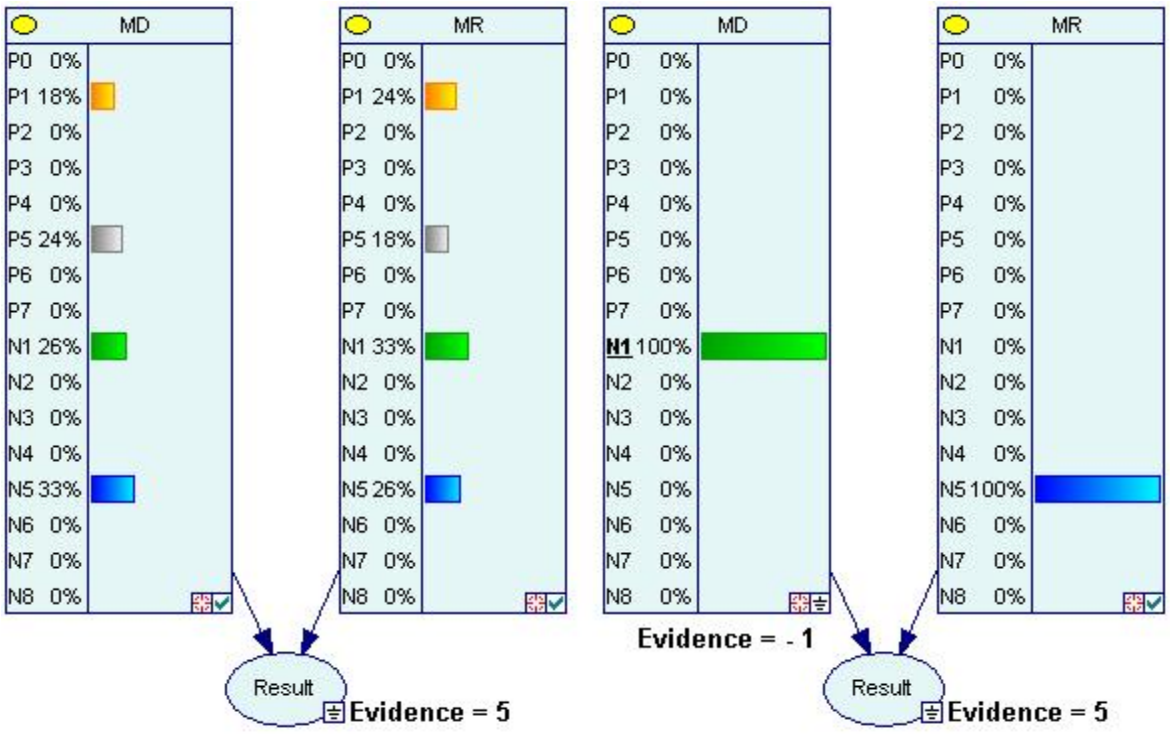


Figure 5-3: Inference with Basic Bayesian Network

The Bayesian network was queried by setting the desired output as evidence on the result node. The belief values were then updated by GeNIe to predict the most likely inputs that will lead to that result. The first example in the left of Figure 5-3 shows the network successfully determined that certain combinations of 1, 5, -1, and -5 on each input will likely produce a 5. With the goal of producing this test scenario, one can then use this information to change the testbench input generation parameters accordingly. In this case, the parameters can be modified so there is about 25% chance of producing positive and negative 1's and 5's, while other values have no chance of being generated.

While the first results will quickly lead to the desired test output of a 5, it may not occur immediately since the network may multiply two 1's or two 5's. There is no information about what combination of inputs will produce the desired output. Because the different ways of producing a 5 occurred with similar frequencies in the test data, the network distributed the certainty of producing this result across all the possible inputs that can lead to that result.

The second result in Figure 5-3 shows the result of setting additional evidence on the MD node to -1. Updating the beliefs, the network then determines the only possible way to produce a 5 with MD of -1 is with an MR of -5. This is an example of the network using a conditional dependency relationship to "explain away" the state of the other node, as previously described in Section 3.0.

Figure 5-4 shows another scenario with a coverage goal of producing a result of six. The network successfully determined the eight possible inputs on each node that most likely produce that value. However, since there are many more ways to produce a 6 than a 5, the certainty of producing that result is smaller. The inputs that occurred most often in the test data were given a slightly higher probability. One might update the testbench with 10% - 14% chance of generating those inputs and many undesired test cases may be generated before producing a 6. Similar to the previous test case, the second result shows applying an evidence of 2 to MD resulted in 100% certainty that an MR value of 3 would produce a 6.

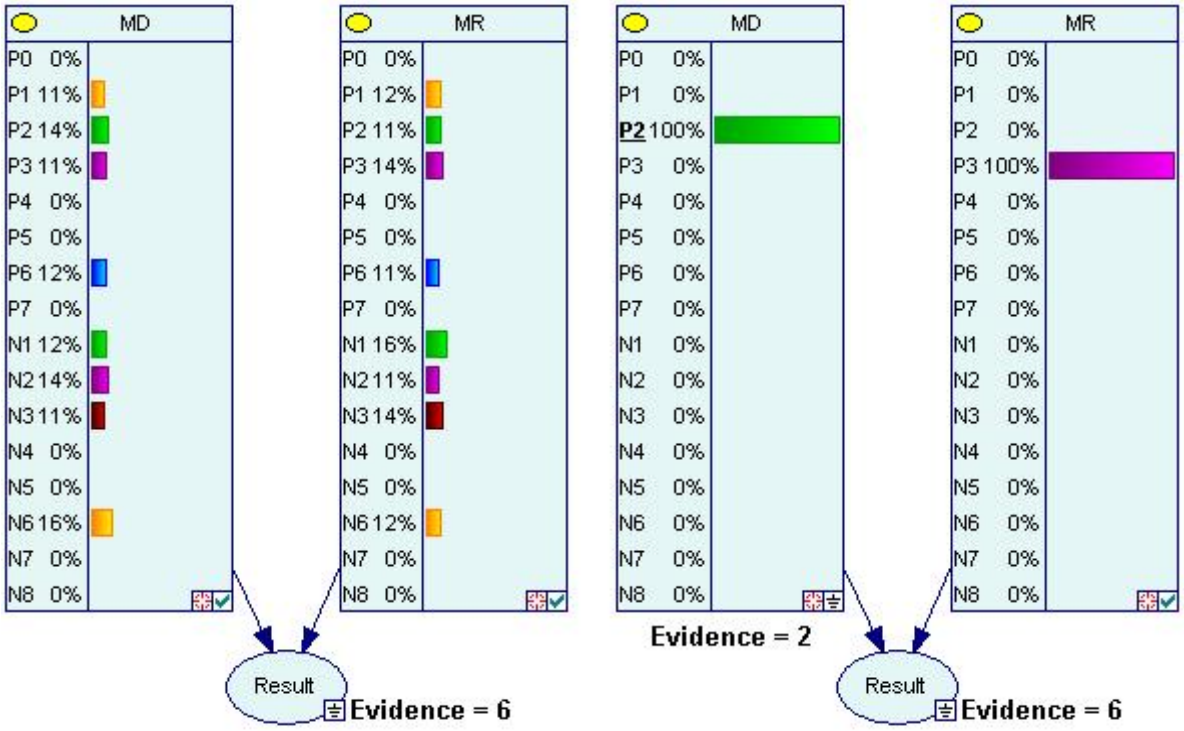


Figure 5-4: Inference Results with More Predictions

As discussed in Section 3.1.3, this network used actual input values rather then the test directives that generate these values, resulting in predictions that were deterministic. Querying the network for the inputs that would lead to a 5 would produce a Most Probable Explanation (MPE) result that would produce a -1 and a -5, guarantying a 5 would be generated every time with these settings. In this context, the Bayesian network is essentially generating a directed test to produce a desired result.

5.3 CDG APPLIED TO WALLACE TREE MULTIPLIER

This section presents the results of the five experiments applied to the Wallace tree multiplier. Experiment 1 was a more basic experiment performed without the automated feedback of new data from the testbench to the Bayesian network. This was done to verify the Bayesian network correctly learned from a testbench log and could generate inputs to produce all 60 outputs of the design under test (DUT). Experiment 2 built on the first experiment by adding the complete CDG automation to ensure the Bayesian network could repeatedly relearn parameters as the testbench generated data. Experiment 3 attempted to meet or more challenging coverage goal of generating all possible 2-result sequences. Experiment 4 attempted to use an alternate Bayesian network using input properties and random distribution settings instead of input values. The final experiment used a more complex Bayesian network to reach interesting internal states of the DUT.

Because of the random nature of the results in these experiments, all experiments were performed five times, averaging the total number of test cases needed to reach 100% coverage. For each execution, a random seed was used for random input generation in the Questa simulation tool.

5.3.1 Experiment 1: Generate all Outputs

The first coverage goal that was used was to simply generate each of the possible 60 results from the multiplier. Although this is a seemingly simple goal, applying unconstrained random test inputs required an average of 536 test cases across five attempts (standard deviation of 161) until all of the 60 results were generated at least once. The initial experiment attempted to reduce the

number of test cases required using the Bayesian network trained with data from the initial random tests.

At this point, the complete automated CDG solution was not exercised since the feedback from the testbench to the Bayesian network was not performed. All inference calculations were based on network parameters learned from a full data set previously produced by the testbench. The goal was simply to reduce the number of test cases needed to reach the 60 results given a full history of how the 60 results were previously generated. This was done to focus on verifying the Bayesian could correctly learn parameters to accurately model the design domain. For a real verification scenario this experiment may seem of little value since it is only reducing the time it takes to generate test outputs that have already been observed. However, this type of situation may be useful when regression testing is needed for a design for which changes have been made after previous simulations were executed. The time needed to perform regression testing would be improved compared to the initial testing since there is historic data for how to reach the coverage goals.

Since the network used input values as nodes that would lead to deterministic predictions for new test inputs, the simulation was executed by only constraining the MR operand. Leaving the multiplicand operand completely random allowed for richer test cases to be generated while constraining MR to a value more likely to fill the needed coverage hole. In this context, constraining both operands would only lead to a series of directed tests being generated that would directly fill all the coverage holes. While this may be useful in some verification scenarios, it goes against the goals of constrained random testing and doesn't provide much value over other methods for generating directed tests.

The simulation was performed with the testbench using completely random inputs for the first 100 test cases before looking for coverage holes to fill. Once a coverage hole was found, the testbench would update the generation parameters on the multiplier operand to generate the MPE for that variable. In the case of needing to reach an output of 64, the testbench deterministically determined (from previous testbench logs) the MR operand should be -8. The testbench would then keep the multiplier constrained to this value for 10 test cases, allowing the multiplicand to remain random.

After every 10 test cases, the testbench would look for another coverage hole to fill. In many cases the previous hole was found again since the multiplicand did not randomly hit the needed value in the previous 10 test cases. Eventually 100% coverage was met after an average of 282 test cases vs. the average of 536 fully random tests previously required. Multiple simulations were executed with different starting points for actively looking to fill holes. Table 5-1 shows the resulting number of test cases needed to reach 100% coverage for each of the different scenarios.

Table 5-1: Simulation Results for Generating all Outputs

Test Scenario (Performed 5 Times)	**Average Number of Test Cases**	**Standard Deviation**
Fully Random	536	161
100 random tests before targeting holes every 10 tests.	282	60
60 random tests before targeting holes every 10 tests.	300	38
150 random tests before targeting holes every 10 tests.	256	26
100 random tests before targeting holes every test.	225	47

Although the multiplier serves as a small design under test with a very small state space for verification, it reveals several aspects of constrained random testing that is consistent with

larger and more complex designs. The results in Table 5-1 show that coverage closure can be met more quickly using constrained settings. Choosing when to start applying constrained settings may impact how quickly coverage is met. In a state space this small, there is little variation between test scenarios, however the results show there is little value in attempting to fill holes earlier in the simulation or to look for new holes more frequently. In some cases, doing either caused simulation to take longer on average. This is generally because of two factors that are repeated in the other experiments described in following sections. The first is overly constraining the inputs early in the process limit the richness of test cases being generated, therefore limiting the diversity of coverage scenarios being met. The second factor is that the constraints to target a specific coverage hole are loose enough that other coverage goals are still being fulfilled. Therefore there is little value in targeting a hole and checking if it was filled after every single constrained test input is generated.

5.3.2 Experiment 2: Generate all Outputs with Relearning Parameters

The full CDG framework (Figure 4-2) was added to the previous experiment to completely close the feedback loops between the testbench and Bayesian network. The network was initialized to a state with no previous data for performing inference. The coverage goal of generating all possible outputs was modified slightly to generating all outputs at least twice, not necessarily consecutively. This was done since the network was starting with no historical data and the testbench would have little opportunity to benefit from predictions with no need to repeat a previous event.

The results using the CDG framework are shown in Table 5-2. The new coverage goal required 864 test cases on average with completely random inputs. This was reduced to an

average of 500 test cases when targeting coverage holes to be filled after an initial 100 completely random tests. In this scenario, the initial 100 tests provided enough data for the Bayesian network to make MPE calculations for many of the results in the following test inputs. Attempting to target holes at the start of simulation was not much better since the richness of test cases was limited early on, slowing the rate the Bayesian network could learn parameters.

Table 5-2: Simulation Results for Generating all Outputs with Relearning

Test Scenario (Performed 5 Times)	**Average Number of Test Cases**	**Standard Deviation**
Fully Random	864	107
100 random tests before targeting holes every 10 tests.	500	122
Target holes every 10 tests from start of simulation.	572	95

5.3.3 Experiment 3: Generating Result Sequences

Another coverage goal was defined to create a more difficult scenario to reach quickly. This goal was to generate all possible 2-result sequences from the multiplier. Such a goal would ensure that any possible result could follow any other result in sequence without error. Although such a goal may not be necessary for a more basic combinational design such as this, this is a very common goal for designs with state behavior that may depend on sequences of data such as pipelined architectures. For the 4-bit multiplier that generates 60 different results, there is a total of 3,600 different sequences (including repeated outputs). Therefore the best case scenario of meeting 100% coverage is at most 3,600 test cases. The reason the best case scenario is not exactly this number is because the nature of generating sequences will likely complete coverage

with fewer tests. For example, if using directed tests to generate each sequence, the act of simply transitioning from one directed test to another (i.e., test 5 followed by 6, then test 5 followed by 7) may cause another sequence (i.e. 6 followed by 5) to be generated that wasn't targeted. Although this example is using directed tests, the occurrence of hitting coverage goals not directly targeted is more common when using constrained random tests and demonstrated with the results of this experiment.

Applying completely random inputs to the CDG testbench, aiming to generating all possible sequences required an average of 126,030 tests to reach 100% coverage. However 99% coverage was usually met as early as 52,000 tests. Therefore more than half of the simulation time was spent trying to complete the final 1% of coverage. This was due to certain coverage holes that were statistically very unlikely to be generated without tighter constraints. This commonly occurs with random testing of designs and is the primary motivation for applying CDG. The remaining holes all involved sequences with a result of 64, which was the rarest result likely to be generated randomly (since this is a 4-bit multiplier for numbers in 2's complement representation, the only way to generate a 64 is if both operands are -8 at the same time). The fully random simulation required as long as 20 minutes using a system with an Intel Xeon E5345 processor.

The testbench was modified to apply Bayesian network predictions using a different technique than the methods of the previous experiment. Since the goal was to generate sequences of data, both input operands were constrained in between completely random test inputs. Cover points were created in the testbench to track all 3,600 possible sequences partitioned into 60 cover groups based on common starting values. For example, all sequences starting with a 1 were tracked in one cover group, with 59 other cover groups for the other

possible starting values. When actively targeting a certain coverage hole, the driver of the testbench would alternate between completely random inputs and applying a deterministically driven constraint to generate the first value of the sequence group that is needed. For example, attempting to meet the cover group of all sequences that begin with a 64, the testbench would drive inputs of -8 alternating with random values, thus ensuring the output would alternate between 64 and a random result. In this sense, the CDG engine automatically created directed tests mixed with random ones in order to target specific coverage areas.

Four different test scenarios were used for reaching the result sequence goal. The four scenarios are described in Table 5-3, along with the number of test cases required for 100% coverage. The trend of coverage closure per number of test cases is shown for each scenario in Figure 5-5, along with the trend for fully random tests. In all results, the average number of tests needed to complete coverage was reduced significantly while reducing the typical simulation time from 20 minutes for fully random to about 8 minutes.

Table 5-3: Simulation Results for Generating all Output Sequences

Test Scenario (Performed 5 Times Each)	**Average Number of Test Cases**	**Standard Deviation**
Fully Random	126,030	8,254
Test 1: 10,000 random tests followed by targeting a new hole every 10 tests.	32,658	2,155
Test 2: Target a new hole very 10 test cases from the start of simulation.	39,423	5,035
Test 3: Same as Test 1, but with 20,000 random tests before targeting holes.	32,619	1,660
Test 4: Same as Test 1, but with 30,000 random tests before targeting holes.	40,162	897

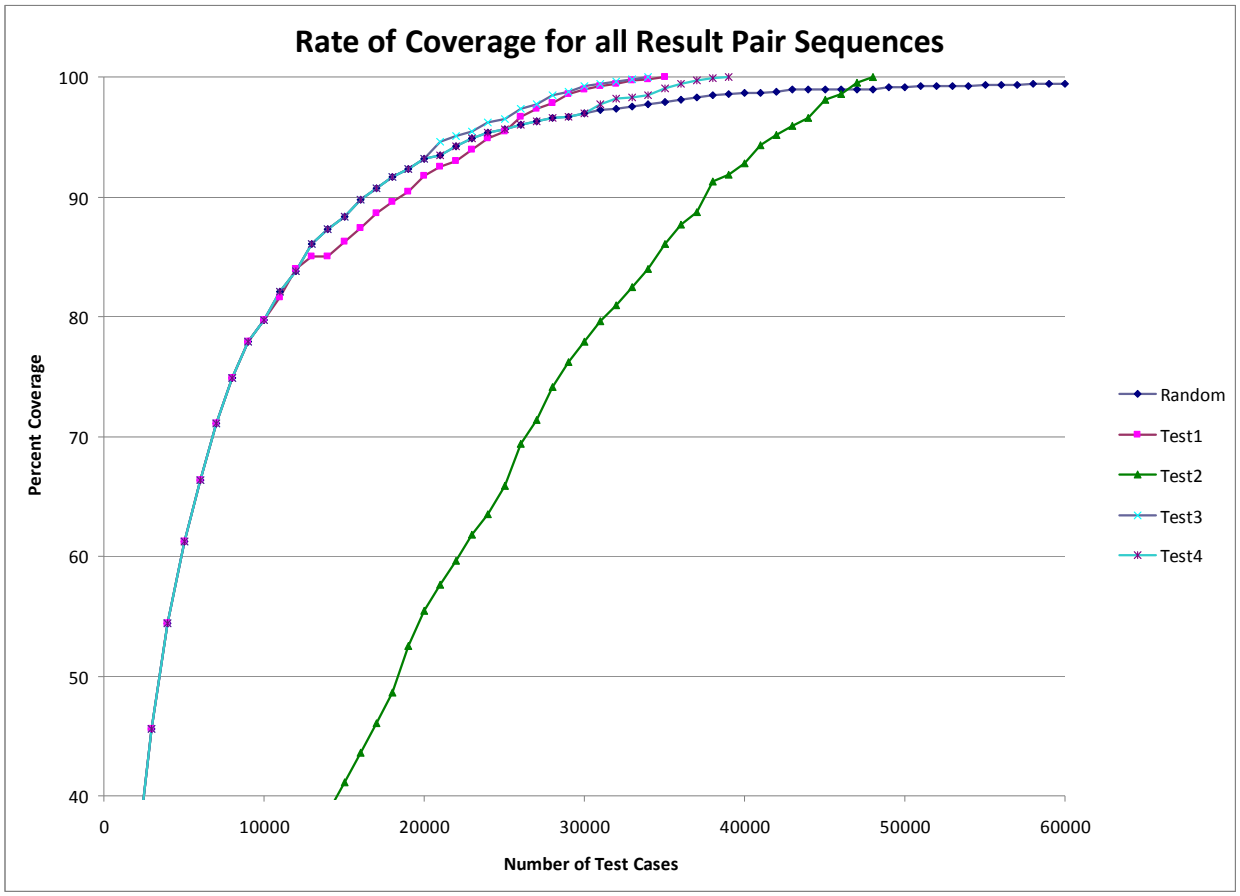


Figure 5-5: Trend of Coverage for Generating Sequences

The trend shows the benefits of not constraining test generation too early. The completely unconstrained inputs (fully random) created a rich set of test cases that hit many cover points very quickly. However, once the probability of hitting remaining holes was reduced, the trend eventually flattened to a rate that completed coverage very slowly. The alternate approach used in Test 2 of targeting holes from the start of simulation greatly reduced the richness of test cases and slowed the rate of coverage closure. In this way, the testing covered the state space of the problem by focusing on a small subset of the state space before moving on to another subset. Since each of these smaller state spaces could be covered more

easily with similar statistics, the trend progressed in a somewhat linear fashion. Although random testing initially performed much better than test 2, ultimately the constraints in test 2 allowed 100% coverage to be met much more efficiently.

Tests 1, 3, and 4 show the best results occur by taking advantage of the amount of state space that can be covered by the pure random tests before applying constraints to target holes. Tests 1 through 4 also improve on the number of tests needed by targeting the more difficult holes (rarest occurring) first. In this case, the most difficult cover points were sequence results starting with high magnitude values (64, 56, -56, etc). This is because high magnitude values have the fewest number of possible inputs that lead to these outputs. For test 1, this allowed coverage to be met with about 9,000 fewer tests compared to an initial simulation that targeted outputs in numerical order. This savings is because targeting difficult holes created a rich abundance of rarely occurring results which are also needed to hit other coverage holes. In the case of the rarest result, targeting the cover point of all sequences starting with 64 resulted in hitting other cover points of sequences that end in 64 due to the natural transition from one test case to another.

Throughout the simulations for generating all result sequences, the Bayesian network relearned network parameters on a periodic basis and provided an updated list of MPE calculations to the testbench. As previously shown, the network only requires a few hundred random test cases to have a model accurate enough for producing all 60 MPE calculations. Since most simulations for this scenario used fully random tests for at least the first 10,000 cases, the testbench had fully accurate and deterministic predictions from network by the time it needed them to start filling holes. This demonstrates a scenario where a relatively small state space of

cover points (the 60 different outputs) are used to reach a very large state space of an overall coverage goal (all possible sequences of the 60 outputs).

For the Wallace Tree multiplier, applying CDG to meet the coverage goals of all result sequences may seem unnecessary since it is very easy to write directed tests to finish testing. In the case of finding holes with 64 in the sequence, it is very trivial for someone to quickly create a few direct test cases with -8 as the inputs and finish coverage. However in more complex designs, it may not be a trivial task for a person to identify which inputs are necessary to reach the coverage goal. If the test was to generate a certain sequence of instructions in consecutive stages of a pipelined processor, certain exceptions such as data dependencies, structural dependencies, and invalid inputs may cause stalls in the pipeline, preventing the coverage event from being generated. The effort of analyzing the exact properties of inputs for reaching *all* of these types of cover points could be very high when considering a typical instruction set and all the possible instruction sequences that should be tested. The CDG framework helps automate the determination of what inputs should be used to reach these scenarios, allowing the verification to be completed more quickly and with less manual labor.

5.3.4 Experiment 4: Network with Input Properties

Section 3.1.3 discussed motivations for using test generation parameters as Bayesian network nodes instead of the values of the actual inputs. Figure 5-6 shows an alternate network for the Wallace Tree that uses higher level test generation directives with 5 possible percentage weights between 0% and 100%. Rather than deterministically determining the actual value of inputs to generate, it instead determines a constraint setting for a looser category of inputs. These include whether in inputs are positive or negative, equal operands, or large in magnitude. The possibility

of generating 100% equal operand directive was not created in the testbench to ensure contradictions would not occur with the other directives and cause simulation to halt at runtime.

The testbench was modified to initially use random weights for these constraints and capture the results. Figure 5-6 shows an example of inference performed to determine the best inputs to generate a 1. The results are generally what would be expected, with a 0% probability of large operands, a larger weighted probability of equal operands, and fairly even predictions for positive/negative operands.

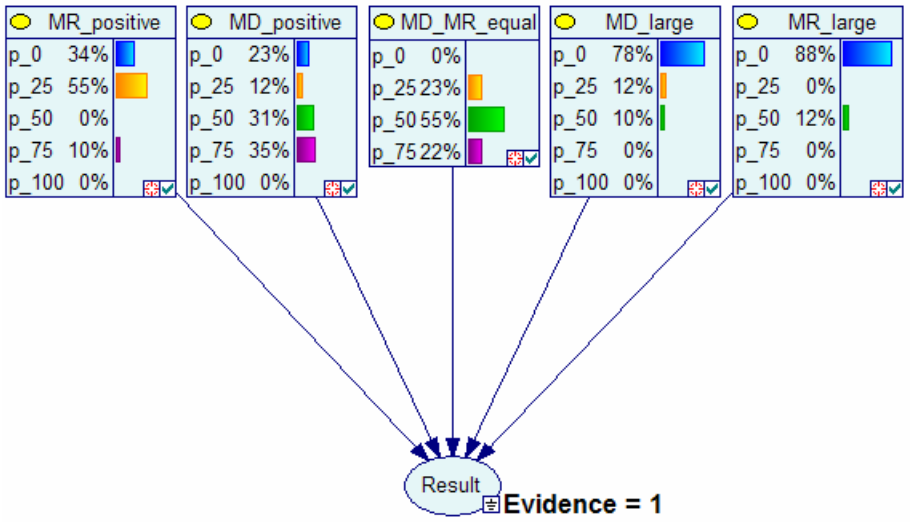


Figure 5-6: Bayesian Network using Test Directives

The network with input weights was used in the CDG framework repeating the previous coverage goals of generating all 60 results twice and generating all 2-result sequence. Despite the accurate learning of network parameters for input directives, the results turned out to be about the same as using true random inputs. This was most likely due to the constraints being too loose when targeting coverage holes. These constraints allow for a large amount of random variation when considering that some weights will only be applied a percentage of the time in addition to a variation of possible inputs within a single property. For example, setting

constraints of MD_large = 50% will only result in large operand values 50% of the time, during which the operand can be any value the ranges 4 to 7 and -8 to -5.

Another attempt at using test input property directives was performed by constraining the directives in Figure 5-6 to only 0% or 100% settings (0% and 75% for equal operand directive). This allowed for slightly tighter constraints that focus more accurately on the targeted cover points, while still allowing the random variation of values within the properties. The results for generating all outputs twice showed improved rate of coverage compared to using random weights, as shown in Table 5-4.

Table 5-4: Simulation Results using Input Properties to Generate all Results Twice

Test Scenario (Performed 5 Times Each)	**Average Number of Test Cases**	**Standard Deviation**
Random Weights assigned every 10 tests	910	261
200 random weights before targeting holes every 10 tests.	380	23
100 random weights before targeting holes every 10 tests.	420	49

The results for generating all 2-result sequences in this scenario allowed coverage to be met after an average of 89,854 tests (7,484 Standard Deviation). Although this not better than the previous results of alternating random and automatic directed tests (between 32,000 and 40,000 tests), it is still an improvement over the fully random scenario which required 126,030 tests on average. This demonstrates the importance of wisely choosing domain variables for the Bayesian network and the associated test directives that can be applied. While certain test directives may seem likely to target coverage holes, too much variation in the resulting settings may not be much of an improvement over unconstrained random tests.

5.3.5 Experiment 5: Reaching Internal States

A more complex Bayesian network is required for observing internal states and how these states influence outputs. The network in Figure 5-7 was created to capture the direct dependencies between key states in the Wallace Tree. Each arrow corresponds to a direct dependency between states and can be correlated to the hardware block diagram in Figure 5-1. Red arrows highlight the direct dependencies to the output result bits.

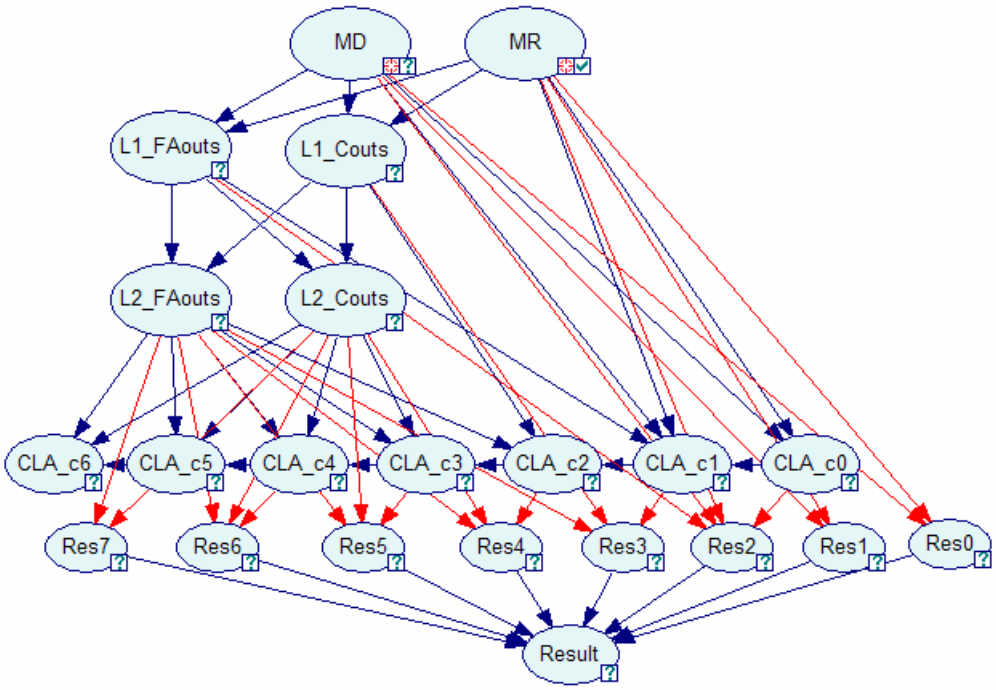


Figure 5-7: A More Complex Bayesian Network for the Multiplier

The same input states are used for multiplicand and multiplier inputs as the basic network described in Section 4.1. These nodes are followed by nodes for the full adder (FA) outputs and FA carry-outs in the first level of the Wallace Tree. The single bit outputs of each of the level 1 adders are packed into a single word representation for the node "L1_FA_outs." The same was done for the carry outs of the level 1 adders (node L1_Couts). This was done reduce the number

of nodes in the network. The alternative of using a node for the states of each FA output and the many additional arcs required to show dependencies resulted in a much larger network for which it was difficult to accurately learn parameters from data sets with as many as 1,000 test cases. As discussed in Section 3.2.4, the Expectation Maximization (EM) learning algorithm needs much larger data sets to accurately learn network parameters for large networks. Modeling the adder outputs in carry bits in this way allowed for fewer nodes, each with larger number of states, as opposed to large number of nodes with only two states each (0 and 1). This allowed network parameters to be learned accurately with data sets as small as 50 test cases.

The outputs and carry-outs of the level 2 adders are captured, as well as the carry-outs of the CLA adders. These influence each of the 8 bits that form the result (nodes Res0 – Res7). The node "Result" was created as a convenient means to specify test scenarios and is not required for the network to be used. Since every state of the Result correlates directly to an 8-bit representation in nodes Res0 – Res7, setting evidence to "Result" propagates directly as evidence on the Res nodes. This allows evidence to be applied to a single node, rather than eight.

The parameters for the more complex Bayesian network were learned using the same data sets as the basic network in Experiment 1. Inference was performed to predict inputs to reach interesting internal states, such as all carry bits in the level 2 adders being 1 (Figure 5-8). As shown in the figure, GeNIe was used to set the evidence L2_Couts node to all 1's, and inference was performed to produce the most likely inputs that lead to all the level 2 carry bits being set. The potential inputs that lead to this state are shown in the graph of results with non-zero percent likelihood predictions. The results with higher likelihood were the inputs that were seen most often in the data set used for learning parameters.

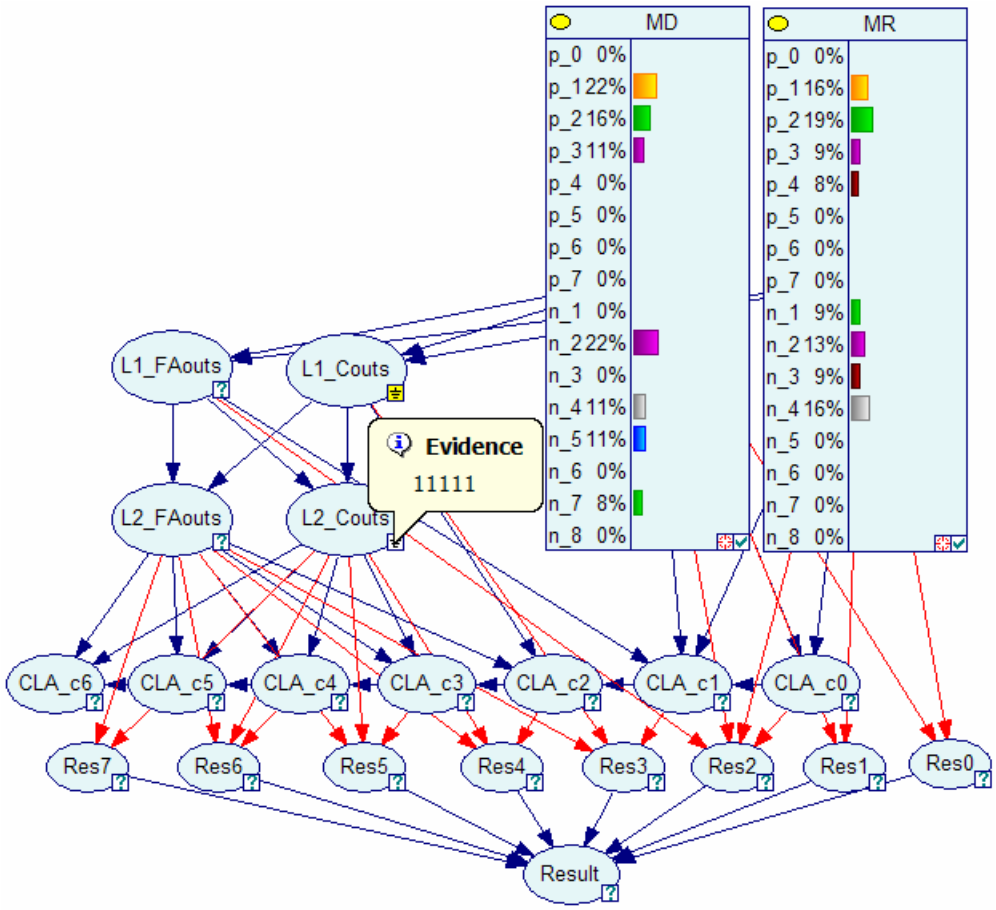


Figure 5-8: Inference to Determine Internal States

Additional inference can be performed by applying evidence to one of the input nodes in addition to evidence for internal state. Similar to the examples in Sections 3.1.3 and 5.2.1, this allows the network to "explain away" certain predictions. Figure 5-9 shows two examples of this. For the first example, setting evidence of MD equal to 1 results in MR predictions to either -1, -2, -3, or -4. Any of these predictions combined with MD of 1 will result in level 2 carry bits being 1. The second example shows evidence of MR equal to 1, leading to predictions that MD equal either -2, -4, -5, or -7.

Figure 5-9: Inference with Additional Evidence for Internal State

This experiment demonstrates several advantages for applying Bayesian networks to CDG. The first is that special scenarios of internal states can be targeted as a coverage goal and recreated more easily. Although these predictions can not be made without prior observation of the internal state, it is typical for coverage goals to be made out of complex sequences or cross coverage points of internal states. For example, this could be a goal of the level 2 carry bits all being 1 for a certain number of consecutive clock cycles. The results in Figures 5-8 and 5-9 can be provided to the constrained random generation parameters to ensure only these inputs get generated and increasing the likelihood of the goal being reached.

Another advantage the Bayesian network provides is insight to domain behavior that may not initially be targeted as a coverage goal. After simulating the design, one may need additional information for recreating a scenario that wasn't planned but resulted in a bug. In a complex design, it may not be easily apparent from the design specifications how to recreate the scenario.

In this situation, the Bayesian network would have captured the relationships between inputs and internal states needed to recreate the bug.

As previously discussed in Section 2.3.1, one of the goals of CDG is to improve the quality of verification by reaching the same coverage point through different test inputs. Bayesian networks can also help achieve this goal. As demonstrated in Figure 5-9, different inputs can be targeted to reach the same coverage goal by applying additional evidence to an input node and performing inference to update predictions on the remaining input nodes. Through "explaining away" other predications, the accuracy of the new likelihoods are greater and focus on more specific types of input combinations. Applying different evidence to an input node will focus predictions on a different set of input combinations, but leading to the same coverage goal. This is especially important in the example scenario of generating consecutive cycles with all level 2 carry bits set to 1. It would not be useful to meet this goal by using one prediction of MD = 1 and MR= -4 and leaving these inputs constant for several cycles. Instead, one can use the Bayesian network to generate diverse predications that all lead to the desired state of carry bits.

6.0 BAYESIAN NETOWORK FOR A SEQUENTIAL LOGIC DESIGN

This chapter describes a Bayesian network and experiments that were performed on a basic sequential logic design. A description for a vending machine state machine is given along with a method for tracking the order internal states are exercised. GeNIe was used to perform several examples of inference and demonstrate how the network can be used to generate input predictions to cover certain state machine paths. This is followed by results of using the Coverage Directed Test Generation (CDG) framework to cover all possible state path traversals more quickly.

6.1 BAYESIAN NETWORK FOR A VENDING MACHINE

A second design that was chosen as a target Design Under Test (DUT) was a sequential logic design. The state machine example of a vending machine was chosen (from [3]) with some variations and is shown in Figure 6-1. In this design, 30 cents is required to make a purchase, with the possible inputs being a nickel, dime, or quarter. In the event someone overpays, they are credited the correct amount towards another purchase. This design example had many interesting paths that may lead to a purchase.

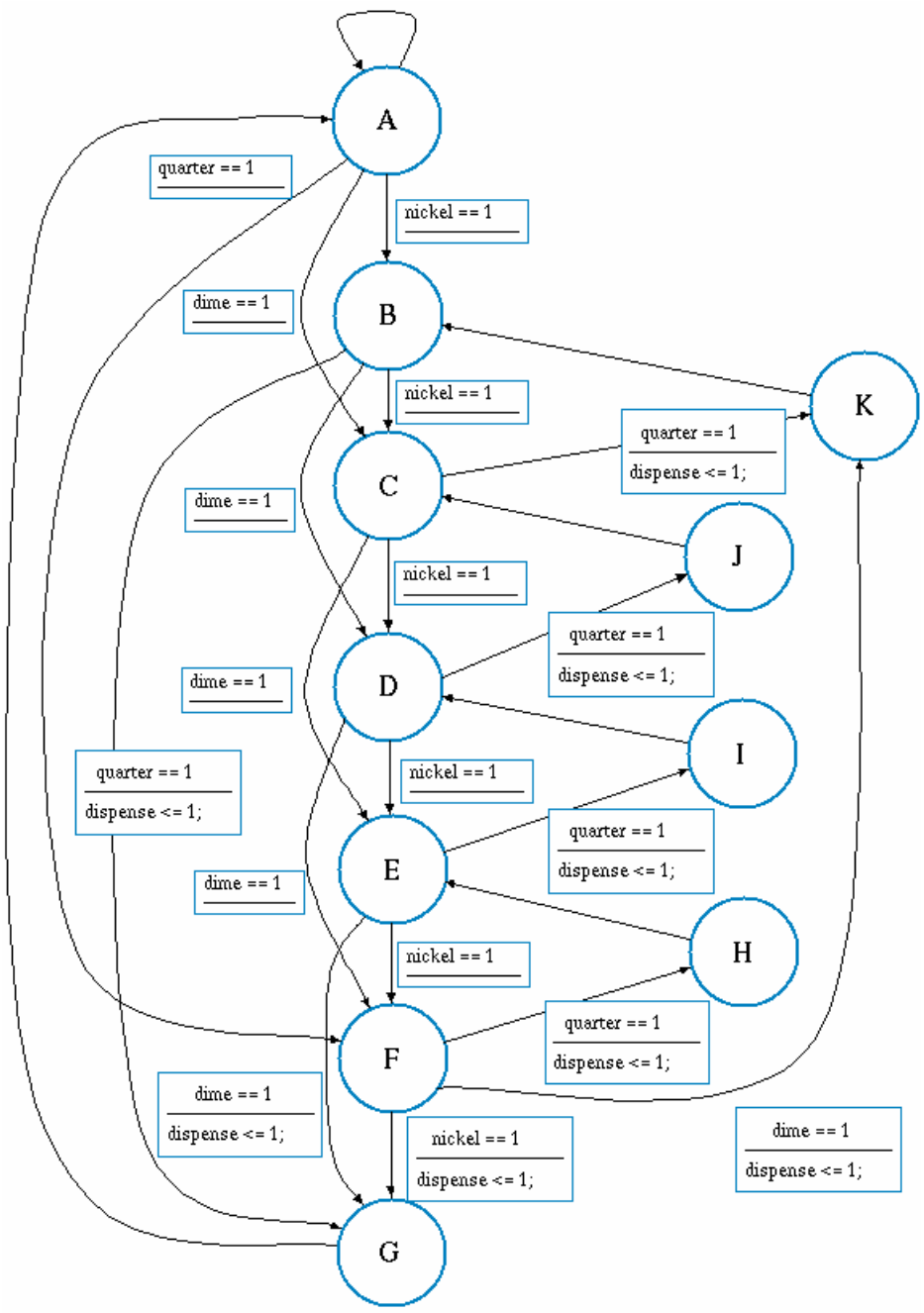


Figure 6-1: State Machine for a Vending Machine

Because the sequence of inputs is a factor for this design, coverage goals were created in terms of a sequence step number, starting from state A as step 1. For example, if one were to give 6 nickels, state G would be reached at step 7. For coverage analysis in this study, only the first 7 sequential steps (from state A) through the state machine were tracked. This was done to simplify coverage tracking since loops in the state machine could potentially allow states to be reached an indefinite number of steps from state A. Since the state machine may loop back to state A prior to 7 steps completing, state A could also occur on steps 4 through 7. However after completing 7 steps, the step count was reset to 1 upon returning to state A.

Coverage goals were defined to reach all possible states at each sequence step. For example at sequence step 2 the DUT could be in states B, D, or F. Cover points were defined for all possible sequence step/state combinations. In many cases, specifying a single step sequence/state pair implied a specific path that must be traversed. For example, to reach state I on step 4 implies the path A-C-E-I had to have been covered. Defining coverage this way therefore achieves verification that all arcs are traversed in every possible order (within 7 transitions).

The Bayesian network for the vending machine is shown in Figure 6-2. This network has nodes for the test generation weights used for generating a nickel, dime, or quarter input. The testbench was designed to ensure only one input would be active at a time. The children nodes represent the state reached at a given sequence step. Step 1 is not included since the DUT can only be in state A at the beginning. Each of the other step directly depend on the state of the previous step. For example, which state is reached on step 7 is directly dependent on which state was reached on step 6. Since paths exist that may return to state A before .

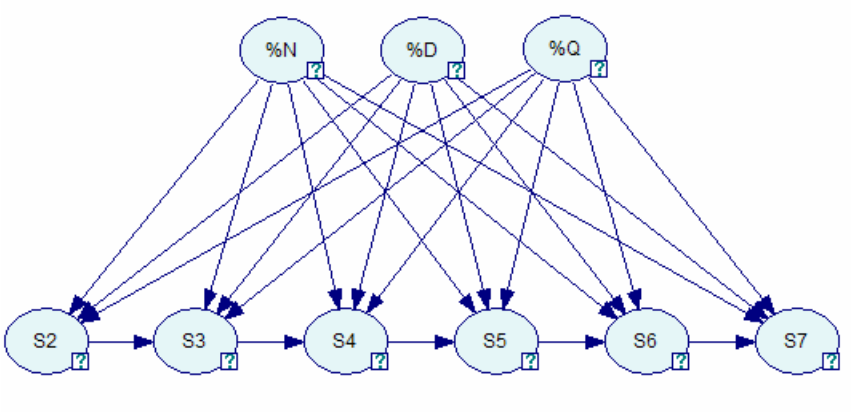


Figure 6-2: Bayesian Network for a Vending Machine

A testbench was created to generate test data and learn parameters for the Bayesian network. This was done by running simulations while randomizing the random distribution weights for inputting nickels, dimes, and quarters. The possible distribution weights were constrained to 0, 25, 50, 75, and 100. Although the order of inputs has an impact on the which states are reached on a certain step, no test directives were created to determine what the order should be. Test cases therefore relied only on having a sufficient proportion of certain inputs to reach the needed state, allowing the flexibility of random generation to determine the order. This led to scenarios where the same input directives may lead to different paths.

The results of various inferences are shown in Figures 6-3, 6-4, and 6-5. Figure 6-3 shows the result of setting evidence for reaching state I on step 4. The network determined with 100% certainty that the previous steps traversed through states C and E. Although not targeted for generating tests, the results also predict the most likely states to occur on the following steps. The network also determined weights of test directives to be used to reach that event. In this case a high probability of dimes and quarters with few nickels are the best inputs. This is consistent with the observation that one needs 2 dimes followed by a quarter to reach state I on step 4.

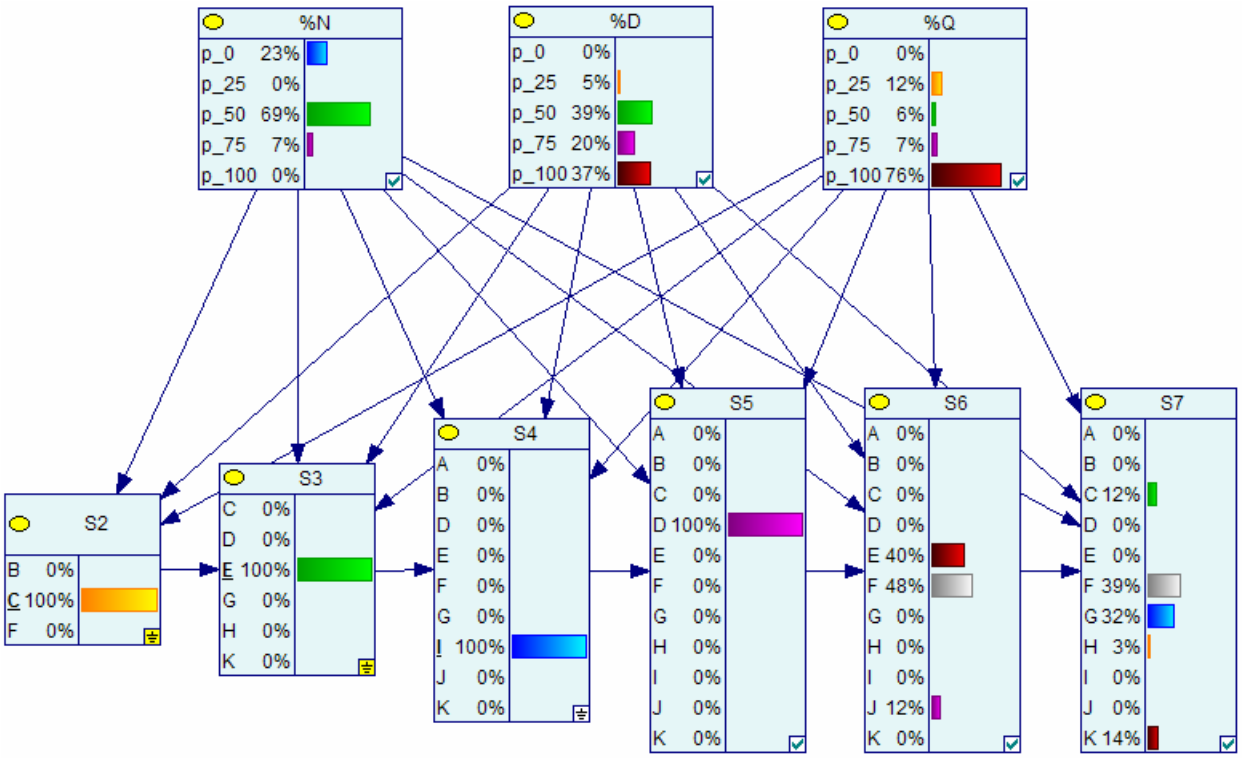


Figure 6-3: Inference to Reach State I on Step 4

Figure 6-4 shows inference for reaching state K on step 3. The state diagram shows there are two paths that could lead to this event and the network confirms this with the probability of step 2 split between states C and F. In either case a combination of a dime and a quarter are required inputs, with only the order of the inputs determining the path. As expected, the network inference results shows to use test directives with a higher probability of dimes and quarters and no nickels (0% weight).

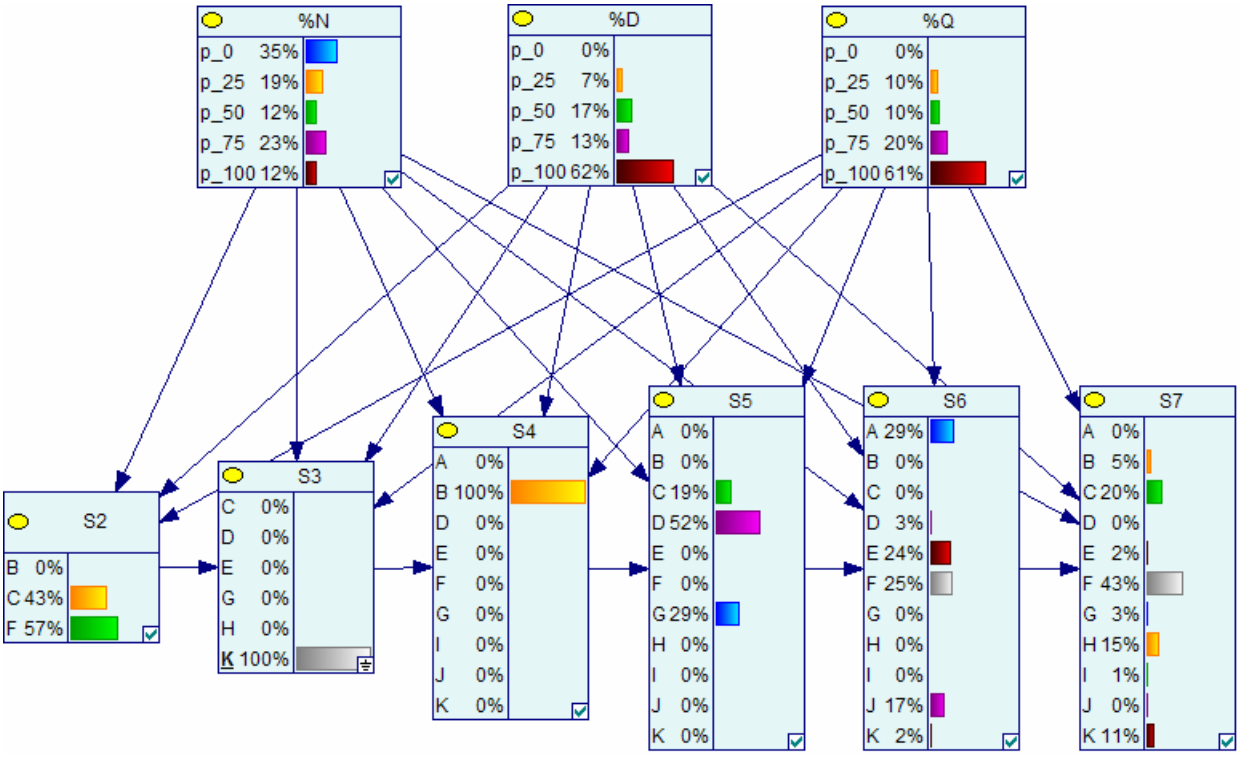


Figure 6-4: Inference to Reach State K on Step 3

Another example of inference using the state machine is shown in Figure 6-5 for reaching state H on step 7. The network accurately tracks several potential paths traversed through the state machine as well as the necessary input weights to get there. However, this is a case where two paths exist for reaching state H on step 7, each requiring very different inputs. One might expect a high probability of nickels is required with relatively few dimes and quarters since this state can be reached by inputting 5 nickels followed by a quarter. However another path that exists for this state is A-C-K-B-D-F-H. This path requires mostly dimes and quarters and is the one shown in the inference results. In this case the data used for learning network parameters did not include a scenario of reaching state H by inputting 5 nickels. Therefore the network was not able to provide inference calculations to predict these inputs.

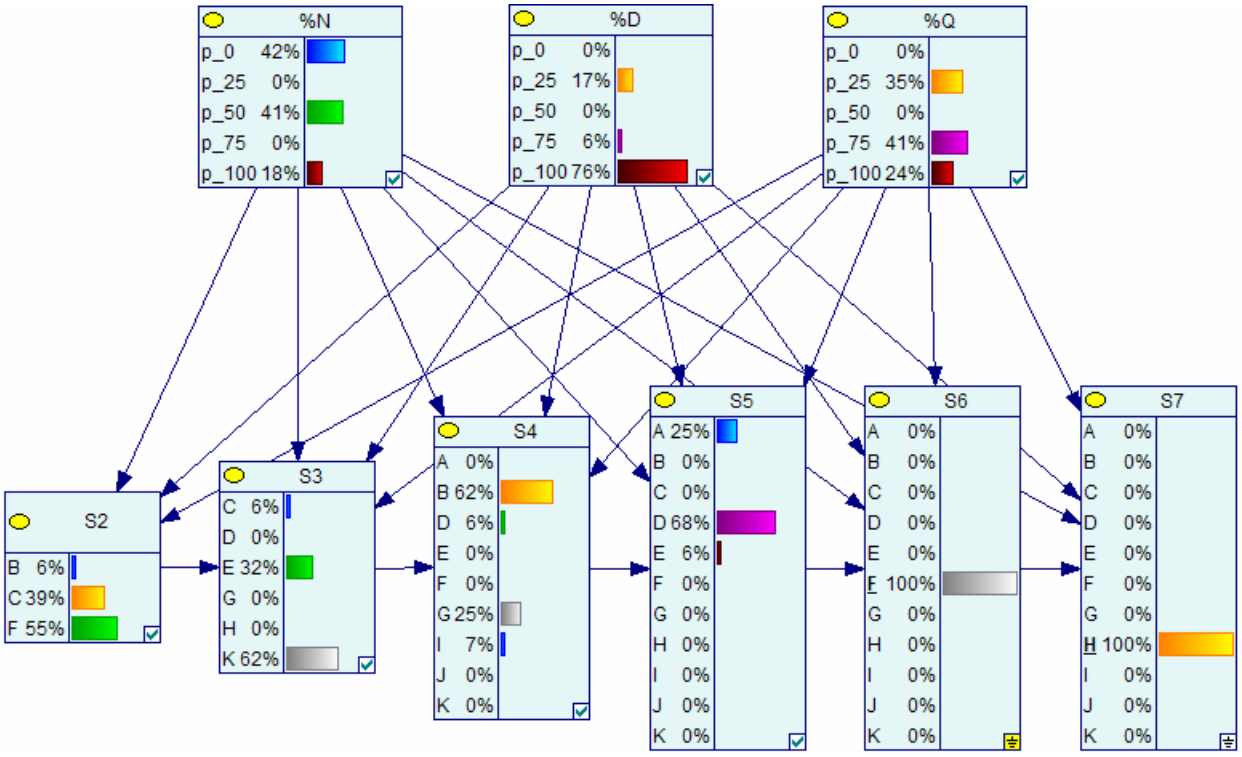


Figure 6-5: Inference to Reach State H on Step 7

The CDG solution was applied to the state machine with coverage goal of reaching each of the step/state combinations at least twice. The results are shown in Table 6-1. Randomizing the input parameters every 10 tests allowed coverage to be met after an average 4,278 tests. The CDG framework was able to improve on this by reaching coverage on an average of 1,364 tests. Similar to the approach used for the multiplier design in Chapter 5, fully random tests were used for the first 500 test cases before starting to target holes to cover. Attempting to target holes from the start of simulation typically required more tests to meet coverage, but not always.

Table 6-1: Results of Vending Machine Coverage

Test Scenario (Performed 5 Times Each)	Average Number of Test Cases	Standard Deviation
Random weights assigned every 10 tests	4,278	1,911
500 random weights before targeting holes every 10 tests (Using network with previously learned parameters)	1,364	140
Targeting holes every 10 tests from start of simulation. (Using network with previously learned parameters)	2,412	1,351

The state machine example demonstrates how Bayesian networks can be useful for sequential logic designs where internal state depends on sequence of inputs. The testbench can be simplified by not having to specify order of inputs and only be concerned with setting constraints to change the proportion of types of inputs. This is a similar type of Bayesian network structure to what is described in [7] for reaching coverage goals of certain instruction types in consecutive pipeline stages of a NorthStar IBM PowerPC processor. The primary difference between these examples is for the pipelined processor the network tracks the types of instructions at each pipeline stage, where this scenario tracks a certain state per sequence step. In both scenarios, a certain state at one time is dependent on a state at a previous time.

7.0 CONCLUSIONS AND FUTURE WORK

The experiments in this thesis have demonstrated a means of creating a Coverage Directed Test Generation (CDG) environment from existing tools. The methods from previous research on using Bayesian networks for CDG have been successfully applied to both a combinational logic design and a sequential logic design. The experiments demonstrate how one can construct a fully automated CDG framework for verification of digital hardware. Combining existing EDA simulation tools and testbench methodologies, one can integrate existing inference engines for Bayesian networks to both capture domain knowledge of a DUT and make predictions for reaching difficult coverage scenarios.

The experiments in this study demonstrate how a CDG solution can improve the efficiency of the verification process without reducing the quality of the verification. Although experiments were conducted on relatively simple designs compared to the complexities of typical designs in industry, they demonstrate the values of constrained random testing. One of the primary goals is to allow testing of unplanned scenarios that may have been missed in development of a verification plan. Applying CDG properly can allow this goal to be met while also reducing the number of test cases needed to meet 100% coverage. Simply automating the construction of directed tests is not always an ideal solution since it will limit the richness of scenarios to be verified. Experiments have also demonstrated how Bayesian networks can improve the quality of tests by predicting diverse inputs that may lead to the same coverage goal.

It has been shown that applying Bayesian networks to verification requires care to choose domain variables and test input directives that will most effectively meet the goals of CDG. Constraints that are too broad may not improve simulation time over using fully random tests. Constraints that are too specific will result in more direct tests and limit the quality of verification. Although applying CDG removes a lot of manual labor from the verification process, applying Bayesian networks in this fashion adds new areas of manual effort and complexity in choosing how to properly model the domain in the network. Other research has explored methods of automating the application of Bayesian networks further by allowing the network structure to be learned from data sets instead of being built manually. Although the accuracy of such models is expected to be lower, the tradeoff of reduced manual effort saved makes this desirable [1].

There are several improvements that could be made for implementing the test environment described in this thesis. The inference engine SMILE was used with a custom C++ program for generating all MPE queries on a repetitive basis at runtime. The SystemVerilog testbench would routinely execute the custom program and read results through log files. This process is relatively slow and hinders the simulation speed for executing tests. For the multiplier example, each program execution to learn parameters and calculate MPE values effectively paused simulation for about 3-5 seconds (running on an Intel Xeon E5345, 2.33 GHz processor). Over the course of long simulation runs, this could accumulate into a lot of time lost that could have been spent testing more scenarios. For larger designs this approach is likely to be impractical. Improvements can be made by integrating the inference engine directly into the testbench. One way this can likely be done is by using the SystemVerilog Direct Programming Interface (DPI) [19]. This interface allows foreign languages such as C/C++ to be used directly

in SystemVerilog code. This would allow faster simulation time as well as allow more detailed interaction between the test generation parameters and MPE calculations. For example, specific inference calculations could be made on demand as needed by the testbench to fill a coverage hole, as opposed to the experiments in this paper which pre-calculated inference for all anticipated cover points ahead of time.

Electronic Design Automation vendors continue to enhance the capabilities of simulation and verification tools. Eventually the technology may be commercially available to quickly implement powerful CDG solutions for complex designs with minimal effort. Until such a time, custom solutions are needed for achieving an automated CDG framework. This thesis has demonstrated that applying Bayesian network inference engines with existing verification tools can achieve this goal with some initial effort to develop the framework.

APPENDIX A

TESTBENCH CODE FOR WALLACE TREE MULTIPLIER

This section contains SystemVerilog code for the Multiplier testbench. The testbench was implemented using the Open Verification Methodology (OVM).

A.1 TOP LEVEL MODULE

```
/* -----------------------------------------------------------------
   File:  wallace_tb_top.sv
   Desc:  This file defines the top level module for the testbench
   -----------------------------------------------------------------*/
module wallace_tb_top;

   `include "ovm.svh"
   `include "mul.v"
   `include "wallace_interface.sv"
   `include "wallace_item.sv"
   `include "wallace_driver.sv"
   `include "wallace_sequencer.sv"
   `include "wallace_sequence.sv"
   `include "wallace_monitor.sv"
   `include "wallace_env.sv"
   `include "wallace_wrapper.sv"
   `include "wallace_base_test.sv"
   `include "wallace_seq_test.sv"

   wallace_interface DUT_if();
   wallace_internal_if int_if();

   wallace_wrapper dutw(DUT_if, int_if);
```

```
   bit clock = 1;

   initial begin
      //uncomment desired test to run

      run_test("wallace_base_test");
      //run_test("wallace_seq_test");

   end

   // generate clock
   always begin
      #10 clock = ~clock;
   end

endmodule
```

A.2 DESIGN INTERFACE

```
/* ---------------------------------
   File: wallace_interface.sv
   Desc: Physical interface to DUT
   ---------------------------------*/
interface wallace_interface();
   logic signed [7:0] result;
   logic signed [3:0] mr;
   logic signed [3:0] md;
endinterface

// Interface to internal DUT signals
interface wallace_internal_if();

   //output bits of each adder in Level 1
   bit L1_o1;
   bit L1_o2;
   bit L1_o3;
   bit L1_o4;
   bit L1_o5;
   bit L1_o17;

   //carry outs of each adder in Level 1
   bit L1_c1;
   bit L1_c2;
   bit L1_c3;
   bit L1_c4;
   bit L1_c5;
   bit L1_c17;

   //output bits of each adder in Level 2
```

```
   bit L2_o6;
   bit L2_o7;
   bit L2_o8;
   bit L2_o9;
   bit L2_o18;

   //carry outs of each adder in Level 2
   bit L2_c6;
   bit L2_c7;
   bit L2_c8;
   bit L2_c9;
   bit L2_c18;

   //carry bits of CLA adders:
   bit CLA_c0;
   bit CLA_c1;
   bit CLA_c2;
   bit CLA_c3;
   bit CLA_c4;
   bit CLA_c5;
   bit CLA_c6;

endinterface
```

A.3 DESIGN WRAPPER

```
/*  ---------------------------------------------------------------------
    File:  wallace_wrapper.sv
    Desc:  wrapper to assign testbench interface and internal signals
    ---------------------------------------------------------------------*/
module wallace_wrapper(wallace_interface DUT_if, wallace_internal_if int_if);

   mul_test1 DUT( DUT_if.result, DUT_if.mr, DUT_if.md );

   assign int_if.L1_o1 = DUT.tree.o1;
   assign int_if.L1_o2 = DUT.tree.o2;
   assign int_if.L1_o3 = DUT.tree.o3;
   assign int_if.L1_o4 = DUT.tree.o4;
   assign int_if.L1_o5 = DUT.tree.o5;
   assign int_if.L1_o17 = DUT.tree.o10;

   assign int_if.L1_c1 = DUT.tree.w1;
   assign int_if.L1_c2 = DUT.tree.w2;
   assign int_if.L1_c3 = DUT.tree.w3;
   assign int_if.L1_c4 = DUT.tree.w4;
   assign int_if.L1_c5 = DUT.tree.w5;
   assign int_if.L1_c17 = DUT.tree.w16;

   assign int_if.L2_o6 = DUT.tree.o6;
   assign int_if.L2_o7 = DUT.tree.o7;
   assign int_if.L2_o8 = DUT.tree.o8;
```

```
   assign int_if.L2_o9 = DUT.tree.o9;
   assign int_if.L2_o18 = DUT.tree.o11;

   assign int_if.L2_c6 = DUT.tree.w6;
   assign int_if.L2_c7 = DUT.tree.w7;
   assign int_if.L2_c8 = DUT.tree.w8;
   assign int_if.L2_c9 = DUT.tree.w9;
   assign int_if.L2_c18 = DUT.tree.w17;

   assign int_if.CLA_c0 = DUT.tree.w10;
   assign int_if.CLA_c1 = DUT.tree.w11;
   assign int_if.CLA_c2 = DUT.tree.w12;
   assign int_if.CLA_c3 = DUT.tree.w13;
   assign int_if.CLA_c4 = DUT.tree.w14;
   assign int_if.CLA_c5 = DUT.tree.w15;
   assign int_if.CLA_c6 = DUT.tree.w18;


endmodule
```

A.4 SEQUENCE ITEM

```
/*------------------------------------------------------------------------
  File: wallace_item.sv
  Desc:  This  file  implements  the  sequence  item  for  the  wallace  tree
         testbench.
        Sequence item consists of two 4-bit input values.
        Random distributions are set as control variables assigned by the
        sequencer.
------------------------------------------------------------------------*/

class wallace_item extends ovm_sequence_item;

  rand logic signed [3:0] mr;
  rand logic signed [3:0] md;

  //control variables
  rand int mr_dist_0, mr_dist_1, mr_dist_2, mr_dist_3, mr_dist_4, mr_dist_5,
mr_dist_6, mr_dist_7;
  rand  int  mr_dist_n1,  mr_dist_n2,  mr_dist_n3,  mr_dist_n4,  mr_dist_n5,
mr_dist_n6, mr_dist_n7, mr_dist_n8;

  rand int md_dist_0, md_dist_1, md_dist_2, md_dist_3, md_dist_4, md_dist_5,
md_dist_6, md_dist_7;
  rand  int  md_dist_n1,  md_dist_n2,  md_dist_n3,  md_dist_n4,  md_dist_n5,
md_dist_n6, md_dist_n7, md_dist_n8;

  int tmp = 0;

  constraint c_mr
  {
```

```
      mr dist
      {  0:= tmp + mr_dist_0,
         1:= tmp + mr_dist_1,
         2:= tmp + mr_dist_2,
         3:= tmp + mr_dist_3,
         4:= tmp + mr_dist_4,
         5:= tmp + mr_dist_5,
         6:= tmp + mr_dist_6,
         7:= tmp + mr_dist_7,
         -1:= tmp + mr_dist_n1,
         -2:= tmp + mr_dist_n2,
         -3:= tmp + mr_dist_n3,
         -4:= tmp + mr_dist_n4,
         -5:= tmp + mr_dist_n5,
         -6:= tmp + mr_dist_n6,
         -7:= tmp + mr_dist_n7,
         -8:= tmp + mr_dist_n8 };
   }

   constraint c_md
   {
      md dist
      {  0:= tmp + md_dist_0,
         1:= tmp + md_dist_1,
         2:= tmp + md_dist_2,
         3:= tmp + md_dist_3,
         4:= tmp + md_dist_4,
         5:= tmp + md_dist_5,
         6:= tmp + md_dist_6,
         7:= tmp + md_dist_7,
         -1:= tmp + md_dist_n1,
         -2:= tmp + md_dist_n2,
         -3:= tmp + md_dist_n3,
         -4:= tmp + md_dist_n4,
         -5:= tmp + md_dist_n5,
         -6:= tmp + md_dist_n6,
         -7:= tmp + md_dist_n7,
         -8:= tmp + md_dist_n8 };
   }

   `ovm_object_utils_begin(wallace_item)
      `ovm_field_int(mr, OVM_ALL_ON)
      `ovm_field_int(md, OVM_ALL_ON)
   `ovm_object_utils_end

   function new (string name = "wallace_item");
       super.new(name);
   endfunction : new

endclass
```

A.5 DRIVER

```
/* ------------------------------------
    File: wallace_driver.sv
    Desc: driver to input data to DUT
   ------------------------------------*/


class wallace_driver extends ovm_driver #(wallace_item);

  wallace_item item;
  virtual wallace_interface vif;

  //OVM macro for general components
  `ovm_component_utils(wallace_driver)

  function new (string name = "wallace_driver", ovm_component parent);
    super.new(name, parent);
  endfunction : new

  task run();

    forever begin

       //get next data item from sequencer (will block until received)
       seq_item_port.get_next_item(item);

       //drive item
       @(posedge wallace_tb_top.clock)
       vif.mr <= item.mr;
       vif.md <= item.md;

       //signal sequencer that execution of item is complete
       seq_item_port.item_done();

    end
  endtask : run

  // Assign virtual interface to physical DUT interface; Called by Agent
  function void assign_vi(virtual interface wallace_interface DUT_if);
    this.vif = DUT_if;
  endfunction : assign_vi

endclass
```

A.6 SEQUENCE

```
/*-------------------------------------------------------------------------
   File: wallace_sequence.sv
   Desc: Used to assign control variables of input distrubution settings
   -------------------------------------------------------------------------*/

class wallace_sequence extends ovm_sequence #(wallace_item);
   `ovm_sequence_utils(wallace_sequence, wallace_sequencer)

    wallace_item seq_item;

    function new(string name="");
      super.new(name);
    endfunction: new

    virtual task body();
      repeat(p_sequencer.item_count) begin
        `ovm_do_with( seq_item,{mr_dist_0 == p_sequencer.mr_dist_0;
                                mr_dist_1 == p_sequencer.mr_dist_1;
                                mr_dist_2 == p_sequencer.mr_dist_2;
                                mr_dist_3 == p_sequencer.mr_dist_3;
                                mr_dist_4 == p_sequencer.mr_dist_4;
                                mr_dist_5 == p_sequencer.mr_dist_5;
                                mr_dist_6 == p_sequencer.mr_dist_6;
                                mr_dist_7 == p_sequencer.mr_dist_7;
                                mr_dist_n1 == p_sequencer.mr_dist_n1;
                                mr_dist_n2 == p_sequencer.mr_dist_n2;
                                mr_dist_n3 == p_sequencer.mr_dist_n3;
                                mr_dist_n4 == p_sequencer.mr_dist_n4;
                                mr_dist_n5 == p_sequencer.mr_dist_n5;
                                mr_dist_n6 == p_sequencer.mr_dist_n6;
                                mr_dist_n7 == p_sequencer.mr_dist_n7;
                                mr_dist_n8 == p_sequencer.mr_dist_n8;
                                md_dist_0 == p_sequencer.md_dist_0;
                                md_dist_1 == p_sequencer.md_dist_1;
                                md_dist_2 == p_sequencer.md_dist_2;
                                md_dist_3 == p_sequencer.md_dist_3;
                                md_dist_4 == p_sequencer.md_dist_4;
                                md_dist_5 == p_sequencer.md_dist_5;
                                md_dist_6 == p_sequencer.md_dist_6;
                                md_dist_7 == p_sequencer.md_dist_7;
                                md_dist_n1 == p_sequencer.md_dist_n1;
                                md_dist_n2 == p_sequencer.md_dist_n2;
                                md_dist_n3 == p_sequencer.md_dist_n3;
                                md_dist_n4 == p_sequencer.md_dist_n4;
                                md_dist_n5 == p_sequencer.md_dist_n5;
                                md_dist_n6 == p_sequencer.md_dist_n6;
                                md_dist_n7 == p_sequencer.md_dist_n7;
                                md_dist_n8 == p_sequencer.md_dist_n8;
                               })
      end
    endtask
 endclass
```

A.7 SEQUENCER

```
/*-------------------------------------------------------------------------
   File:  wallace_sequencer.sv
   Desc:  Sequencer to generate items for tests.
          Also used to maintain new random distribution settings for inputs
-------------------------------------------------------------------------
*/

class wallace_sequencer extends ovm_sequencer #(wallace_item);

   //OVM macro for sequencers
   `ovm_sequencer_utils_begin(wallace_sequencer)
      `ovm_field_int(mr_dist_0, OVM_ALL_ON)
      `ovm_field_int(mr_dist_1, OVM_ALL_ON)
      `ovm_field_int(mr_dist_2, OVM_ALL_ON)
      `ovm_field_int(mr_dist_3, OVM_ALL_ON)
      `ovm_field_int(mr_dist_4, OVM_ALL_ON)
      `ovm_field_int(mr_dist_5, OVM_ALL_ON)
      `ovm_field_int(mr_dist_6, OVM_ALL_ON)
      `ovm_field_int(mr_dist_7, OVM_ALL_ON)
      `ovm_field_int(mr_dist_n1, OVM_ALL_ON)
      `ovm_field_int(mr_dist_n2, OVM_ALL_ON)
      `ovm_field_int(mr_dist_n3, OVM_ALL_ON)
      `ovm_field_int(mr_dist_n4, OVM_ALL_ON)
      `ovm_field_int(mr_dist_n5, OVM_ALL_ON)
      `ovm_field_int(mr_dist_n6, OVM_ALL_ON)
      `ovm_field_int(mr_dist_n7, OVM_ALL_ON)
      `ovm_field_int(mr_dist_n8, OVM_ALL_ON)
      `ovm_field_int(md_dist_0, OVM_ALL_ON)
      `ovm_field_int(md_dist_1, OVM_ALL_ON)
      `ovm_field_int(md_dist_2, OVM_ALL_ON)
      `ovm_field_int(md_dist_3, OVM_ALL_ON)
      `ovm_field_int(md_dist_4, OVM_ALL_ON)
      `ovm_field_int(md_dist_5, OVM_ALL_ON)
      `ovm_field_int(md_dist_6, OVM_ALL_ON)
      `ovm_field_int(md_dist_7, OVM_ALL_ON)
      `ovm_field_int(md_dist_n1, OVM_ALL_ON)
      `ovm_field_int(md_dist_n2, OVM_ALL_ON)
      `ovm_field_int(md_dist_n3, OVM_ALL_ON)
      `ovm_field_int(md_dist_n4, OVM_ALL_ON)
      `ovm_field_int(md_dist_n5, OVM_ALL_ON)
      `ovm_field_int(md_dist_n6, OVM_ALL_ON)
      `ovm_field_int(md_dist_n7, OVM_ALL_ON)
      `ovm_field_int(md_dist_n8, OVM_ALL_ON)

      `ovm_field_int(item_count,OVM_ALL_ON)
    `ovm_sequencer_utils_end

   function new(string name="wallace_sequencer", ovm_component parent);
      super.new(name,parent);
      `ovm_update_sequence_lib_and_item(wallace_item)
   endfunction : new
```

```
   //variables to control knobs of item
   // set defaults to all 100

   int item_count = 10;  //initial count of test cases to generate

   int mr_dist_0 = 100;
   int mr_dist_1 = 100;
   int mr_dist_2 = 100;
   int mr_dist_3 = 100;
   int mr_dist_4 = 100;
   int mr_dist_5 = 100;
   int mr_dist_6 = 100;
   int mr_dist_7 = 100;
   int mr_dist_n1 = 100;
   int mr_dist_n2 = 100;
   int mr_dist_n3 = 100;
   int mr_dist_n4 = 100;
   int mr_dist_n5 = 100;
   int mr_dist_n6 = 100;
   int mr_dist_n7 = 100;
   int mr_dist_n8 = 100;

   int md_dist_0 = 100;
   int md_dist_1 = 100;
   int md_dist_2 = 100;
   int md_dist_3 = 100;
   int md_dist_4 = 100;
   int md_dist_5 = 100;
   int md_dist_6 = 100;
   int md_dist_7 = 100;
   int md_dist_n1 = 100;
   int md_dist_n2 = 100;
   int md_dist_n3 = 100;
   int md_dist_n4 = 100;
   int md_dist_n5 = 100;
   int md_dist_n6 = 100;
   int md_dist_n7 = 100;
   int md_dist_n8 = 100;

endclass
```

A.8 MONITOR

The following is an excerpt from the code used for the testbench monitor. Many cover points were defined in order to monitor each cover point at runtime. This was done due to a limitation

of SystemVerilog preventing the ability to monitor specific bins within a cover point at runtime.

```
/*  File: wallace_monitor.sv */

class wallace_transaction extends ovm_transaction;
   logic signed [3:0] mr;
   logic signed [3:0] md;
   logic signed [7:0] result;

   //output bits of each adder in Level 1
   logic unsigned [5:0] L1_FAout_bits;
                              //[5:0]are wires o17, o5, o4, o3, o2, o1

   //carry outs of each adder in Level 1
   logic unsigned [5:0] L1_Cout_bits;
                              // [5:0] are wires c17, c5, c4, c3, c2, c1

   //output bits of each adder in Level 2
   logic unsigned [4:0] L2_FAout_bits;
                              // [4:0] are wires o18, o9, o8, o7, o6

   //carry outs of each adder in Level 2
   logic unsigned [4:0] L2_Cout_bits;
                              // [4:0] are wires c18, c9, c8, c7, c6

   //carry outs of CLA adders
   logic unsigned [6:0] CLA_Cout_bits;
                         // [6:0] are wires w18, w15, w14, w13, w12, w11,w10

endclass


class wallace_monitor extends ovm_monitor;

   virtual wallace_interface vif;

   protected wallace_transaction trans;
   bit coverage_enable = 1;
   int file;
   string s_result;
   string s_mr;
   string s_md;
   string r_sign;
   string mr_sign;
   string md_sign;
   shortint i_tmp;

   event cov_transaction;

   //declare fields that are adjustable parameters
   `ovm_component_utils_begin(wallace_monitor);
      `ovm_field_int(coverage_enable, OVM_ALL_ON)
   `ovm_component_utils_end
```

```
   covergroup covgrp_trans @cov_transaction;

   cp_result: coverpoint trans.result{

       bins result_val[] = {[$:127]};

       ignore_bins impPosRes =
{11,13,17,19,22,23,26,27,29,31,33,34,[37:39],41,[43:47],[50:55],[57:63],[65:$
]};
       ignore_bins impNegRes = {[$:-57],[-55:-50],[-47:-43],-41,[-39:-37],-
34,-33,-31,-29,-27,-26,-23,-22,-19,-17,-13,-11};

        option.at_least =2;
   }

   cp_result0: coverpoint trans.result{ bins val={0};
                                        option.at_least = 2; }
   cp_result1: coverpoint trans.result{ bins val={1};
   option.at_least = 2; }
   cp_result2: coverpoint trans.result{ bins val={2};
   option.at_least = 2;}
   cp_result3: coverpoint trans.result{ bins val={3};
   option.at_least = 2; }
   cp_result4: coverpoint trans.result{ bins val={4};
   option.at_least = 2;}
```

. . . Additional cover points for other values defined similarly to above.

```
//coverpoint array for all potential transitions between results
   cp_Result_trans: coverpoint trans.result{

        bins result_trans[] = ( [-56:64] => [-56:64] );

        ignore_bins imp_trans1[] =
(11,13,17,19,22,23,26,27,29,31,33,34,[37:39],41,[43:47],[50:55],[57:63],
[-55:-50],[-47:-43],-41,[-39:-37],-34,-33,-31,-29,-27,-26,-23,-22,-19,-17,-
13,-11
     => [-56:64]);

        ignore_bins imp_trans2[] = ( [-56:64] =>

11,13,17,19,22,23,26,27,29,31,33,34,[37:39],41,[43:47],[50:55],[57:63],
[-55:-50],[-47:-43],-41,[-39:-37],-34,-33,-31,-29,-27,-26,-23,-22,-19,-17,-
13,-11);
    }

    cp_trans_p0: coverpoint trans.result{

      bins result_trans[] = ( 0 => [-56:64] );
      ignore_bins imp_trans[] = ( 0 =>

11,13,17,19,22,23,26,27,29,31,33,34,[37:39],41,[43:47],[50:55],[57:63],
                                    [-55:-50],[-47:-43],-41,[-39:-37],-34,-
33,-31,-29,-27,-26,-23,-22,-19,-17,-13,-11);
```

```
    }
    cp_trans_p1: coverpoint trans.result{

      bins result_trans[] = ( 1 => [-56:64] );
      ignore_bins imp_trans[] = ( 1 =>

11,13,17,19,22,23,26,27,29,31,33,34,[37:39],41,[43:47],[50:55],[57:63],
[-55:-50],[-47:-43],-41,[-39:-37],-34,-33,-31,-29,-27,-26,-23,-22,-19,-17,-
13,-11);
    }
```

. . . Additional cover points for transitions from other values similar to above.

```
function new (string name, ovm_component parent);
     super.new(name, parent);
     covgrp_trans = new();
     trans = new();
  endfunction

  virtual task run();

     file = $fopen("wallace_data.txt");

$fwrite(file,"MR\tMD\tResult\tL1_FAouts\tL1_Couts\tL2_FAouts\tL2_Couts\tCLA_c
6\tCLA_c5\tCLA_c4\tCLA_c3\tCLA_c2\tCLA_c1\tCLA_c0\tRes7\tRes6\tRes5\tRes4\tRe
s3\tRes2\tRes1\tRes0\n");
     $fclose(file);

     fork
        collect_transactions();
     join
  endtask

  virtual protected task collect_transactions();

     forever begin

       file = $fopen("wallace_data.txt", "a");

       @(negedge wallace_tb_top.clock);

       trans.mr = vif.mr;
       trans.md = vif.md;
       trans.result = vif.result;

      trans.L1_FAout_bits = { wallace_tb_top.int_if.L1_o17,
                              wallace_tb_top.int_if.L1_o5,
                              wallace_tb_top.int_if.L1_o4,
                              wallace_tb_top.int_if.L1_o3,
                              wallace_tb_top.int_if.L1_o2,
                              wallace_tb_top.int_if.L1_o1 };

      trans.L1_Cout_bits = { wallace_tb_top.int_if.L1_c17,
                             wallace_tb_top.int_if.L1_c5,
```

```
                        wallace_tb_top.int_if.L1_c4,
                        wallace_tb_top.int_if.L1_c3,
                        wallace_tb_top.int_if.L1_c2,
                        wallace_tb_top.int_if.L1_c1 };

trans.L2_FAout_bits = {wallace_tb_top.int_if.L2_o18,
                        wallace_tb_top.int_if.L2_o9,
                        wallace_tb_top.int_if.L2_o8,
                        wallace_tb_top.int_if.L2_o7,
                        wallace_tb_top.int_if.L2_o6 };

trans.L2_Cout_bits = { wallace_tb_top.int_if.L2_c18,
                        wallace_tb_top.int_if.L2_c9,
                        wallace_tb_top.int_if.L2_c8,
                        wallace_tb_top.int_if.L2_c7,
                        wallace_tb_top.int_if.L2_c6 };

            trans.CLA_Cout_bits = { wallace_tb_top.int_if.CLA_c6,
                         wallace_tb_top.int_if.CLA_c5,
                         wallace_tb_top.int_if.CLA_c4,
                         wallace_tb_top.int_if.CLA_c3,
                         wallace_tb_top.int_if.CLA_c2,
                         wallace_tb_top.int_if.CLA_c1,
                         wallace_tb_top.int_if.CLA_c0 };

            //convert Result to string
            $cast(i_tmp, trans.result);
            if (i_tmp < 0) begin
              i_tmp = -i_tmp;
              s_result.itoa(i_tmp);
              r_sign = "n_";
            end
            else begin
              s_result.itoa(i_tmp);
              r_sign = "p_";
            end

            //convert MR to string
            $cast(i_tmp, trans.mr);
            if (i_tmp < 0) begin
              i_tmp = -i_tmp;
              s_mr.itoa(i_tmp);
              mr_sign = "n_";
            end
            else begin
              s_mr.itoa(i_tmp);
              mr_sign = "p_";
            end

            //convert MD to string
            $cast(i_tmp, trans.md);
            if (i_tmp < 0) begin
              i_tmp = -i_tmp;
              s_md.itoa(i_tmp);
              md_sign = "n_";
            end
            else begin
```

```
                    s_md.itoa(i_tmp);
                    md_sign = "p_";
                  end

                  $fwrite(file,
"%s%s\t%s%s\t%s%s\tb_%b\tb_%b\tb_%b\tb_%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b\t%b
\t%b\t%b\t%b\t%b\t%b\t%b\n",
                mr_sign,
                s_mr,
                md_sign,
                s_md,
                r_sign,
                s_result,
                trans.L1_FAout_bits,
                trans.L1_Cout_bits,
                trans.L2_FAout_bits,
                trans.L2_Cout_bits,
                trans.CLA_Cout_bits[6], trans.CLA_Cout_bits[5],
trans.CLA_Cout_bits[4],trans.CLA_Cout_bits[3],trans.CLA_Cout_bits[2],trans.CL
A_Cout_bits[1],trans.CLA_Cout_bits[0],
                trans.result[7], trans.result[6], trans.result[5],
trans.result[4], trans.result[3], trans.result[2], trans.result[1],
trans.result[0]
                );

       $fclose(file);

       if (coverage_enable)
         -> cov_transaction;
     end
  endtask

  // Assign virtual interface to physical DUT interface;
  function void assign_vi(virtual interface wallace_interface DUT_if );
    this.vif = DUT_if;
  endfunction

endclass
```

A.9 TESTBENCH ENVIRONMENT

```
class wallace_env extends ovm_env;

  `ovm_component_utils(wallace_env)

  wallace_sequencer         sequencer;
  wallace_driver            driver;
  wallace_monitor           monitor;

  function new(string name, ovm_component parent);
    super.new(name, parent);
```

```
  endfunction

  virtual function void build();
    super.build();

    sequencer = wallace_sequencer::type_id::create("sequencer",this);
    driver = wallace_driver::type_id::create("driver", this);
    monitor = wallace_monitor::type_id::create("monitor", this);

  endfunction

  virtual function void connect();

  // Connect driver's port to sequencer's export
  driver.seq_item_port.connect(sequencer.seq_item_export);
     driver.assign_vi(wallace_tb_top.DUT_if);
     monitor.assign_vi(wallace_tb_top.DUT_if);
  endfunction
endclass
```

A.10 BASE TEST

```
/* Test for generating all result outputs */

class wallace_base_test extends ovm_test;

   `ovm_component_utils(wallace_base_test)

   int CLOCK_PERIOD = 20;

   wallace_sequence base_sequence;

   integer input_file, c, r;
   string label, line, val, val2;

   //bit input_dist[string];
   int  MR_MPE[string];      //used  to  assign  current  dist  parameters  to
sequencer
   int MD_MPE[string];
   string MR_key;
   string MD_key;

   string MPE_line_ref[string];
   string MPE_key;

   real cvr_pt_coverage;
   real old_coverage;
   int same_coverage_count = 0;

   wallace_env tb;      //will instantiate the testbench
```

```
   function new (string name = "wallace_base_test", ovm_component parent =
null);
      super.new(name, parent);
   endfunction

   //build testbench
   virtual function void build();
      super.build();

      // set number sequence items for default sequencer to 0
      set_config_int("*.sequencer","count", 0);

      base_sequence = new();

      tb = wallace_env::type_id::create("tb", this);

      //set up array of dist values for MR
      MR_MPE["p_0"] = 100;
      MR_MPE["p_1"] = 100;
      MR_MPE["p_2"]= 100;
      MR_MPE["p_3"]= 100;
      MR_MPE["p_4"]= 100;
      MR_MPE["p_5"]= 100;
      MR_MPE["p_6"]= 100;
      MR_MPE["p_7"]= 100;
      MR_MPE["n_1"]= 100;
      MR_MPE["n_2"]= 100;
      MR_MPE["n_3"]= 100;
      MR_MPE["n_4"]= 100;
      MR_MPE["n_5"]= 100;
      MR_MPE["n_6"]= 100;
      MR_MPE["n_7"]= 100;
      MR_MPE["n_8"]= 100;

      MD_MPE["p_0"] = 100;
      MD_MPE["p_1"] = 100;
      MD_MPE["p_2"]= 100;
      MD_MPE["p_3"]= 100;
      MD_MPE["p_4"]= 100;
      MD_MPE["p_5"]= 100;
      MD_MPE["p_6"]= 100;
      MD_MPE["p_7"]= 100;
      MD_MPE["n_1"]= 100;
      MD_MPE["n_2"]= 100;
      MD_MPE["n_3"]= 100;
      MD_MPE["n_4"]= 100;
      MD_MPE["n_5"]= 100;
      MD_MPE["n_6"]= 100;
      MD_MPE["n_7"]= 100;
      MD_MPE["n_8"]= 100;

   endfunction

   function void read_MPE_file();
     //open MPE Reference file and store each line in an array indexed by
result string
      input_file = $fopen("Wallace_MPE.txt","r");
```

```
   while(!$feof(input_file)) begin

     c = $fgetc(input_file);

     if(c == "/")
       r = $fgets(line, input_file);
     else begin
       //push char back tofile to read line
       r = $ungetc(c, input_file);
       r = $fgets(line, input_file);
       r = $sscanf(line, "%s\t%s\t%s", MPE_key, val, val2);
       MPE_line_ref[MPE_key] = line;
     end
   end

   $fclose(input_file);

endfunction


//define runtime behavior
virtual task run();

     // Start simulation with full random tests
   base_sequence.start(tb.sequencer);
   base_sequence.wait_for_sequence_state(FINISHED);

   cvr_pt_coverage = tb.monitor.covgrp_trans.cp_result.get_coverage();
   `message(OVM_LOW, ("Coverage is %f", cvr_pt_coverage))


   while (cvr_pt_coverage < 100) begin

     //Relearn parameters for Bayesian Network
     $system("wall_learn_mpe >tmp.txt");

     //Read new most probable expectation for all results
     read_MPE_file();

     //Find next hole and bias MR for 10 test cases
     set_dist(1,10);


     base_sequence.start(tb.sequencer);
     base_sequence.wait_for_sequence_state(FINISHED);

     cvr_pt_coverage = tb.monitor.covgrp_trans.cp_result.get_coverage();
     `message(OVM_LOW, ("Coverage is %f", cvr_pt_coverage))

   end

   #(CLOCK_PERIOD);
   global_stop_request();

endtask
```

```
task set_dist(int mode, int count);

 tb.sequencer.item_count = count;

 if(mode == 0) begin
   //`message(OVM_LOW, ("Starting Flat Sequence"))

   tb.sequencer.mr_dist_0 = 100;
   tb.sequencer.mr_dist_1 = 100;
   tb.sequencer.mr_dist_2 = 100;
   tb.sequencer.mr_dist_3 = 100;
   tb.sequencer.mr_dist_4 = 100;
   tb.sequencer.mr_dist_5 = 100;
   tb.sequencer.mr_dist_6 = 100;
   tb.sequencer.mr_dist_7 = 100;
   tb.sequencer.mr_dist_n1 = 100;
   tb.sequencer.mr_dist_n2 = 100;
   tb.sequencer.mr_dist_n3 = 100;
   tb.sequencer.mr_dist_n4 = 100;
   tb.sequencer.mr_dist_n5 = 100;
   tb.sequencer.mr_dist_n6 = 100;
   tb.sequencer.mr_dist_n7 = 100;
   tb.sequencer.mr_dist_n8 = 100;

   tb.sequencer.md_dist_0 = 100;
   tb.sequencer.md_dist_1 = 100;
   tb.sequencer.md_dist_2 = 100;
   tb.sequencer.md_dist_3 = 100;
   tb.sequencer.md_dist_4 = 100;
   tb.sequencer.md_dist_5 = 100;
   tb.sequencer.md_dist_6 = 100;
   tb.sequencer.md_dist_7 = 100;
   tb.sequencer.md_dist_n1 = 100;
   tb.sequencer.md_dist_n2 = 100;
   tb.sequencer.md_dist_n3 = 100;
   tb.sequencer.md_dist_n4 = 100;
   tb.sequencer.md_dist_n5 = 100;
   tb.sequencer.md_dist_n6 = 100;
   tb.sequencer.md_dist_n7 = 100;
   tb.sequencer.md_dist_n8 = 100;
 end
 else begin //ONLY CONSTRAINING MR

   // FIND FIRST HOLE IN COVERAGE
   find_first_hole(); //will set MPE_key to result not generated yet

   if(MPE_key == "") begin
     `message(OVM_LOW,("No coverage holes found"))

      foreach(MR_MPE[i])
        MR_MPE[i] = 100;
   end
   else if( 0 == MPE_line_ref.exists(MPE_key) ) begin

     //Desired result has not been observed by Bayesian Network
```

```
      foreach(MR_MPE[i])
         MR_MPE[i] = 100;
    end

    else begin

      `message(OVM_LOW,("Starting Sequence with MR biased distribution"))

      //Clear all previous entries in MR and MD dists.
      foreach(MR_MPE[i])
         MR_MPE[i] = 0;

      r = $sscanf(MPE_line_ref[MPE_key], "%s\t%s\t%s", val,MR_key,MD_key);

      //Set probability dist of MR MPE to 100
      MR_MPE[MR_key] = 100;

    end

      tb.sequencer.mr_dist_0 = MR_MPE["p_0"];
      tb.sequencer.mr_dist_1 = MR_MPE["p_1"];
      tb.sequencer.mr_dist_2 = MR_MPE["p_2"];
      tb.sequencer.mr_dist_3 = MR_MPE["p_3"];
      tb.sequencer.mr_dist_4 = MR_MPE["p_4"];
      tb.sequencer.mr_dist_5 = MR_MPE["p_5"];
      tb.sequencer.mr_dist_6 = MR_MPE["p_6"];
      tb.sequencer.mr_dist_7 = MR_MPE["p_7"];
      tb.sequencer.mr_dist_n1 = MR_MPE["n_1"];
      tb.sequencer.mr_dist_n2 = MR_MPE["n_2"];
      tb.sequencer.mr_dist_n3 = MR_MPE["n_3"];
      tb.sequencer.mr_dist_n4 = MR_MPE["n_4"];
      tb.sequencer.mr_dist_n5 = MR_MPE["n_5"];
      tb.sequencer.mr_dist_n6 = MR_MPE["n_6"];
      tb.sequencer.mr_dist_n7 = MR_MPE["n_7"];
      tb.sequencer.mr_dist_n8 = MR_MPE["n_8"];

  end
endtask

task find_first_hole();
  if( tb.monitor.covgrp_trans.cp_result0.get_coverage() == 0 )
    MPE_key = "p_0";
  else if( tb.monitor.covgrp_trans.cp_result1.get_coverage() == 0 )
    MPE_key = "p_1";
  else if( tb.monitor.covgrp_trans.cp_result2.get_coverage() == 0 )
    MPE_key = "p_2";
  else if( tb.monitor.covgrp_trans.cp_result3.get_coverage() == 0 )
    MPE_key = "p_3";
  else if( tb.monitor.covgrp_trans.cp_result4.get_coverage() == 0 )
    MPE_key = "p_4";
  else if( tb.monitor.covgrp_trans.cp_result5.get_coverage() == 0 )
    MPE_key = "p_5";
  else if( tb.monitor.covgrp_trans.cp_result6.get_coverage() == 0 )
    MPE_key = "p_6";
  else if( tb.monitor.covgrp_trans.cp_result7.get_coverage() == 0 )
    MPE_key = "p_7";
  else if( tb.monitor.covgrp_trans.cp_result8.get_coverage() == 0 )
```

```
  MPE_key = "p_8";
else if( tb.monitor.covgrp_trans.cp_result9.get_coverage() == 0 )
  MPE_key = "p_9";
else if( tb.monitor.covgrp_trans.cp_result10.get_coverage() == 0 )
  MPE_key = "p_10";
else if( tb.monitor.covgrp_trans.cp_result12.get_coverage() == 0 )
  MPE_key = "p_12";
else if( tb.monitor.covgrp_trans.cp_result14.get_coverage() == 0 )
  MPE_key = "p_14";
else if( tb.monitor.covgrp_trans.cp_result15.get_coverage() == 0 )
  MPE_key = "p_15";
else if( tb.monitor.covgrp_trans.cp_result16.get_coverage() == 0 )
  MPE_key = "p_16";
else if( tb.monitor.covgrp_trans.cp_result18.get_coverage() == 0 )
  MPE_key = "p_18";
else if( tb.monitor.covgrp_trans.cp_result20.get_coverage() == 0 )
  MPE_key = "p_20";
else if( tb.monitor.covgrp_trans.cp_result21.get_coverage() == 0 )
  MPE_key = "p_21";
else if( tb.monitor.covgrp_trans.cp_result24.get_coverage() == 0 )
  MPE_key = "p_24";
else if( tb.monitor.covgrp_trans.cp_result25.get_coverage() == 0 )
  MPE_key = "p_25";
else if( tb.monitor.covgrp_trans.cp_result28.get_coverage() == 0 )
  MPE_key = "p_28";
else if( tb.monitor.covgrp_trans.cp_result30.get_coverage() == 0 )
  MPE_key = "p_30";
else if( tb.monitor.covgrp_trans.cp_result32.get_coverage() == 0 )
  MPE_key = "p_32";
else if( tb.monitor.covgrp_trans.cp_result35.get_coverage() == 0 )
  MPE_key = "p_35";
else if( tb.monitor.covgrp_trans.cp_result36.get_coverage() == 0 )
  MPE_key = "p_36";
else if( tb.monitor.covgrp_trans.cp_result40.get_coverage() == 0 )
  MPE_key = "p_40";
else if( tb.monitor.covgrp_trans.cp_result42.get_coverage() == 0 )
  MPE_key = "p_42";
else if( tb.monitor.covgrp_trans.cp_result48.get_coverage() == 0 )
  MPE_key = "p_48";
else if( tb.monitor.covgrp_trans.cp_result49.get_coverage() == 0 )
  MPE_key = "p_49";
else if( tb.monitor.covgrp_trans.cp_result56.get_coverage() == 0 )
  MPE_key = "p_56";
else if( tb.monitor.covgrp_trans.cp_result64.get_coverage() == 0 )
  MPE_key = "p_64";
else if( tb.monitor.covgrp_trans.cp_result_n1.get_coverage() == 0 )
  MPE_key = "n_1";
else if( tb.monitor.covgrp_trans.cp_result_n2.get_coverage() == 0 )
  MPE_key = "n_2";
else if( tb.monitor.covgrp_trans.cp_result_n3.get_coverage() == 0 )
  MPE_key = "n_3";
else if( tb.monitor.covgrp_trans.cp_result_n4.get_coverage() == 0 )
  MPE_key = "n_4";
else if( tb.monitor.covgrp_trans.cp_result_n5.get_coverage() == 0 )
  MPE_key = "n_5";
else if( tb.monitor.covgrp_trans.cp_result_n6.get_coverage() == 0 )
  MPE_key = "n_6";
```

```
    else if( tb.monitor.covgrp_trans.cp_result_n7.get_coverage() == 0 )
      MPE_key = "n_7";
    else if( tb.monitor.covgrp_trans.cp_result_n8.get_coverage() == 0 )
      MPE_key = "n_8";
    else if( tb.monitor.covgrp_trans.cp_result_n9.get_coverage() == 0 )
      MPE_key = "n_9";
    else if( tb.monitor.covgrp_trans.cp_result_n10.get_coverage() == 0 )
      MPE_key = "n_10";
    else if( tb.monitor.covgrp_trans.cp_result_n12.get_coverage() == 0 )
      MPE_key = "n_12";
    else if( tb.monitor.covgrp_trans.cp_result_n14.get_coverage() == 0 )
      MPE_key = "n_14";
    else if( tb.monitor.covgrp_trans.cp_result_n15.get_coverage() == 0 )
      MPE_key = "n_15";
    else if( tb.monitor.covgrp_trans.cp_result_n16.get_coverage() == 0 )
      MPE_key = "n_16";
    else if( tb.monitor.covgrp_trans.cp_result_n18.get_coverage() == 0 )
      MPE_key = "n_18";
    else if( tb.monitor.covgrp_trans.cp_result_n20.get_coverage() == 0 )
      MPE_key = "n_20";
    else if( tb.monitor.covgrp_trans.cp_result_n21.get_coverage() == 0 )
      MPE_key = "n_21";
    else if( tb.monitor.covgrp_trans.cp_result_n24.get_coverage() == 0 )
      MPE_key = "n_24";
    else if( tb.monitor.covgrp_trans.cp_result_n25.get_coverage() == 0 )
      MPE_key = "n_25";
    else if( tb.monitor.covgrp_trans.cp_result_n28.get_coverage() == 0 )
      MPE_key = "n_28";
    else if( tb.monitor.covgrp_trans.cp_result_n30.get_coverage() == 0 )
      MPE_key = "n_30";
    else if( tb.monitor.covgrp_trans.cp_result_n32.get_coverage() == 0 )
      MPE_key = "n_32";
    else if( tb.monitor.covgrp_trans.cp_result_n35.get_coverage() == 0 )
      MPE_key = "n_35";
    else if( tb.monitor.covgrp_trans.cp_result_n36.get_coverage() == 0 )
      MPE_key = "n_36";
    else if( tb.monitor.covgrp_trans.cp_result_n40.get_coverage() == 0 )
      MPE_key = "n_40";
    else if( tb.monitor.covgrp_trans.cp_result_n42.get_coverage() == 0 )
      MPE_key = "n_42";
    else if( tb.monitor.covgrp_trans.cp_result_n48.get_coverage() == 0 )
      MPE_key = "n_48";
    else if( tb.monitor.covgrp_trans.cp_result_n49.get_coverage() == 0 )
      MPE_key = "n_49";
    else if( tb.monitor.covgrp_trans.cp_result_n56.get_coverage() == 0 )
      MPE_key = "n_56";
    else MPE_key = "";
  endtask

endclass
```

A.11 SEQUENCE TEST

```
class wallace_seq_test extends wallace_base_test;

   `ovm_component_utils(wallace_seq_test)

   function new (string name = "wallace_seq_test", ovm_component parent =
null);
      super.new(name, parent);
   endfunction

   //build testbench
   virtual function void build();
      super.build();
   endfunction

   //define runtime behavior
   virtual task run();

      int count;

      //Start full random

      for(int i=0; i<10; i++) begin

        base_sequence.start(tb.sequencer);
        base_sequence.wait_for_sequence_state(FINISHED);

        cvr_pt_coverage=
            tb.monitor.covgrp_trans.cp_Result_trans.get_coverage();
        `message(OVM_LOW, ("Coverage is %f", cvr_pt_coverage))
      end

      count= 1;

      while (cvr_pt_coverage < 100) begin

        //Relearn parameters for Bayesian Network
        $system("wall_learn_mpe >tmp.txt");

        //Read new most probable expectation for all results
        read_MPE_file();

        for(int i=0; i<10; i++) begin

              //create one input with predicted input value
          set_dist(1,1);
          base_sequence.start(tb.sequencer);
          base_sequence.wait_for_sequence_state(FINISHED);

              //create one random value
            set_dist(0,1);
          base_sequence.start(tb.sequencer);
```

```
          base_sequence.wait_for_sequence_state(FINISHED);

          cvr_pt_coverage=
tb.monitor.covgrp_trans.cp_Result_trans.get_coverage();
          if(cvr_pt_coverage == 100)
            break;
            end

        if( count++ == 50 ) begin
          `message(OVM_LOW, ("Coverage is %f", cvr_pt_coverage))
          count = 1;
        end

      end

      #(CLOCK_PERIOD);
      global_stop_request();

   endtask

   task set_dist(int mode, int count);

    tb.sequencer.item_count = count;

    if(mode == 0) begin

      tb.sequencer.mr_dist_0 = 100;
      tb.sequencer.mr_dist_1 = 100;
      tb.sequencer.mr_dist_2 = 100;
      tb.sequencer.mr_dist_3 = 100;
      tb.sequencer.mr_dist_4 = 100;
      tb.sequencer.mr_dist_5 = 100;
      tb.sequencer.mr_dist_6 = 100;
      tb.sequencer.mr_dist_7 = 100;
      tb.sequencer.mr_dist_n1 = 100;
      tb.sequencer.mr_dist_n2 = 100;
      tb.sequencer.mr_dist_n3 = 100;
      tb.sequencer.mr_dist_n4 = 100;
      tb.sequencer.mr_dist_n5 = 100;
      tb.sequencer.mr_dist_n6 = 100;
      tb.sequencer.mr_dist_n7 = 100;
      tb.sequencer.mr_dist_n8 = 100;

      tb.sequencer.md_dist_0 = 100;
      tb.sequencer.md_dist_1 = 100;
      tb.sequencer.md_dist_2 = 100;
      tb.sequencer.md_dist_3 = 100;
      tb.sequencer.md_dist_4 = 100;
      tb.sequencer.md_dist_5 = 100;
      tb.sequencer.md_dist_6 = 100;
      tb.sequencer.md_dist_7 = 100;
      tb.sequencer.md_dist_n1 = 100;
      tb.sequencer.md_dist_n2 = 100;
      tb.sequencer.md_dist_n3 = 100;
      tb.sequencer.md_dist_n4 = 100;
      tb.sequencer.md_dist_n5 = 100;
      tb.sequencer.md_dist_n6 = 100;
```

```
  tb.sequencer.md_dist_n7 = 100;
  tb.sequencer.md_dist_n8 = 100;
end
else begin


  // FIND FIRST HOLE IN COVERAGE
  find_trans_hole();

  if(MPE_key == "") begin
    `message(OVM_LOW,("No coverage holes found"))

     foreach(MR_MPE[i])
       MR_MPE[i] = 100;

    foreach(MD_MPE[i])
       MD_MPE[i] = 100;

  end
  else if( 0 == MPE_line_ref.exists(MPE_key) ) begin

    //Desired result has not been observed by Bayesian Network
    `message(OVM_LOW,("Desired output not observed by Bayesian Network"))

    foreach(MR_MPE[i])
       MR_MPE[i] = 100;

     foreach(MD_MPE[i])
       MD_MPE[i] = 100;

  end

  else begin

    //`message(OVM_LOW,("Starting Sequence with MR biased distribution"))

    foreach(MR_MPE[i])
       MR_MPE[i] = 0;

    foreach(MD_MPE[i])
       MD_MPE[i] = 0;

    r = $sscanf(MPE_line_ref[MPE_key], "%s\t%s\t%s", val,MR_key,MD_key);

    //Set probability dist of MR and MD MPEs to 100
    MR_MPE[MR_key] = 100;
    MD_MPE[MD_key] = 100;

  end

    tb.sequencer.mr_dist_0 = MR_MPE["p_0"];
    tb.sequencer.mr_dist_1 = MR_MPE["p_1"];
    tb.sequencer.mr_dist_2 = MR_MPE["p_2"];
    tb.sequencer.mr_dist_3 = MR_MPE["p_3"];
    tb.sequencer.mr_dist_4 = MR_MPE["p_4"];
    tb.sequencer.mr_dist_5 = MR_MPE["p_5"];
    tb.sequencer.mr_dist_6 = MR_MPE["p_6"];
```

```
      tb.sequencer.mr_dist_7 = MR_MPE["p_7"];
      tb.sequencer.mr_dist_n1 = MR_MPE["n_1"];
      tb.sequencer.mr_dist_n2 = MR_MPE["n_2"];
      tb.sequencer.mr_dist_n3 = MR_MPE["n_3"];
      tb.sequencer.mr_dist_n4 = MR_MPE["n_4"];
      tb.sequencer.mr_dist_n5 = MR_MPE["n_5"];
      tb.sequencer.mr_dist_n6 = MR_MPE["n_6"];
      tb.sequencer.mr_dist_n7 = MR_MPE["n_7"];
      tb.sequencer.mr_dist_n8 = MR_MPE["n_8"];

      tb.sequencer.md_dist_0 = MD_MPE["p_0"];
      tb.sequencer.md_dist_1 = MD_MPE["p_1"];
      tb.sequencer.md_dist_2 = MD_MPE["p_2"];
      tb.sequencer.md_dist_3 = MD_MPE["p_3"];
      tb.sequencer.md_dist_4 = MD_MPE["p_4"];
      tb.sequencer.md_dist_5 = MD_MPE["p_5"];
      tb.sequencer.md_dist_6 = MD_MPE["p_6"];
      tb.sequencer.md_dist_7 = MD_MPE["p_7"];
      tb.sequencer.md_dist_n1 = MD_MPE["n_1"];
      tb.sequencer.md_dist_n2 = MD_MPE["n_2"];
      tb.sequencer.md_dist_n3 = MD_MPE["n_3"];
      tb.sequencer.md_dist_n4 = MD_MPE["n_4"];
      tb.sequencer.md_dist_n5 = MD_MPE["n_5"];
      tb.sequencer.md_dist_n6 = MD_MPE["n_6"];
      tb.sequencer.md_dist_n7 = MD_MPE["n_7"];
      tb.sequencer.md_dist_n8 = MD_MPE["n_8"];
  end
endtask

task find_trans_hole();
  if( tb.monitor.covgrp_trans.cp_trans_p64.get_coverage() < 100)
    MPE_key = "p_64";
  else if( tb.monitor.covgrp_trans.cp_trans_p56.get_coverage() < 100)
    MPE_key = "p_56";
  else if( tb.monitor.covgrp_trans.cp_trans_n56.get_coverage() < 100)
    MPE_key = "n_56";
  else if( tb.monitor.covgrp_trans.cp_trans_p49.get_coverage() < 100)
    MPE_key = "p_49";
  else if( tb.monitor.covgrp_trans.cp_trans_n49.get_coverage() < 100)
    MPE_key = "n_49";
  else if( tb.monitor.covgrp_trans.cp_trans_p48.get_coverage() < 100)
    MPE_key = "p_48";
  else if( tb.monitor.covgrp_trans.cp_trans_n48.get_coverage() < 100)
    MPE_key = "n_48";
  else if( tb.monitor.covgrp_trans.cp_trans_p42.get_coverage() < 100 )
    MPE_key = "p_42";
  else if( tb.monitor.covgrp_trans.cp_trans_n42.get_coverage() < 100)
    MPE_key = "n_42";
  else if( tb.monitor.covgrp_trans.cp_trans_p40.get_coverage()< 100 )
    MPE_key = "p_40";
  else if( tb.monitor.covgrp_trans.cp_trans_n40.get_coverage() < 100)
    MPE_key = "n_40";
  else if( tb.monitor.covgrp_trans.cp_trans_p36.get_coverage()< 100 )
    MPE_key = "p_36";
  else if( tb.monitor.covgrp_trans.cp_trans_n36.get_coverage() < 100)
    MPE_key = "n_36";
  else if( tb.monitor.covgrp_trans.cp_trans_p35.get_coverage()< 100 )
```

```
  MPE_key = "p_35";
else if( tb.monitor.covgrp_trans.cp_trans_n35.get_coverage() < 100)
  MPE_key = "n_35";
else if( tb.monitor.covgrp_trans.cp_trans_p32.get_coverage() < 100)
  MPE_key = "p_32";
else if( tb.monitor.covgrp_trans.cp_trans_n32.get_coverage() < 100)
  MPE_key = "n_32";
else if( tb.monitor.covgrp_trans.cp_trans_p30.get_coverage() < 100)
  MPE_key = "p_30";
else if( tb.monitor.covgrp_trans.cp_trans_n30.get_coverage() < 100)
  MPE_key = "n_30";
else if( tb.monitor.covgrp_trans.cp_trans_p28.get_coverage() < 100)
  MPE_key = "p_28";
else if( tb.monitor.covgrp_trans.cp_trans_n28.get_coverage() < 100)
  MPE_key = "n_28";
else if( tb.monitor.covgrp_trans.cp_trans_p25.get_coverage() < 100)
  MPE_key = "p_25";
else if( tb.monitor.covgrp_trans.cp_trans_n25.get_coverage() < 100)
  MPE_key = "n_25";
else if( tb.monitor.covgrp_trans.cp_trans_p24.get_coverage() < 100)
  MPE_key = "p_24";
else if( tb.monitor.covgrp_trans.cp_trans_n24.get_coverage() < 100)
  MPE_key = "n_24";
else if( tb.monitor.covgrp_trans.cp_trans_p21.get_coverage() < 100)
  MPE_key = "p_21";
else if( tb.monitor.covgrp_trans.cp_trans_n21.get_coverage() < 100)
  MPE_key = "n_21";
else if( tb.monitor.covgrp_trans.cp_trans_p20.get_coverage() < 100)
  MPE_key = "p_20";
else if( tb.monitor.covgrp_trans.cp_trans_n20.get_coverage() < 100)
  MPE_key = "n_20";
else if( tb.monitor.covgrp_trans.cp_trans_p18.get_coverage() < 100)
  MPE_key = "p_18";
else if( tb.monitor.covgrp_trans.cp_trans_n18.get_coverage() < 100)
  MPE_key = "n_18";
else if( tb.monitor.covgrp_trans.cp_trans_p16.get_coverage() < 100)
  MPE_key = "p_16";
else if( tb.monitor.covgrp_trans.cp_trans_n16.get_coverage() < 100)
  MPE_key = "n_16";

else if( tb.monitor.covgrp_trans.cp_trans_p1.get_coverage()< 100 )
  MPE_key = "p_1";
else if( tb.monitor.covgrp_trans.cp_trans_p2.get_coverage() < 100 )
  MPE_key = "p_2";
else if( tb.monitor.covgrp_trans.cp_trans_p3.get_coverage() < 100 )
  MPE_key = "p_3";
else if( tb.monitor.covgrp_trans.cp_trans_p4.get_coverage() < 100 )
  MPE_key = "p_4";
else if( tb.monitor.covgrp_trans.cp_trans_p5.get_coverage() < 100 )
  MPE_key = "p_5";
else if( tb.monitor.covgrp_trans.cp_trans_p6.get_coverage() < 100)
  MPE_key = "p_6";
else if( tb.monitor.covgrp_trans.cp_trans_p7.get_coverage() < 100)
  MPE_key = "p_7";
else if( tb.monitor.covgrp_trans.cp_trans_p8.get_coverage() < 100)
  MPE_key = "p_8";
else if( tb.monitor.covgrp_trans.cp_trans_p9.get_coverage() < 100)
```

```
      MPE_key = "p_9";
    else if( tb.monitor.covgrp_trans.cp_trans_p10.get_coverage() < 100 )
      MPE_key = "p_10";
    else if( tb.monitor.covgrp_trans.cp_trans_p12.get_coverage() < 100)
      MPE_key = "p_12";
    else if( tb.monitor.covgrp_trans.cp_trans_p14.get_coverage() < 100)
      MPE_key = "p_14";
    else if( tb.monitor.covgrp_trans.cp_trans_p15.get_coverage() < 100)
      MPE_key = "p_15";

    else if( tb.monitor.covgrp_trans.cp_trans_n1.get_coverage() < 100)
      MPE_key = "n_1";
    else if( tb.monitor.covgrp_trans.cp_trans_n2.get_coverage() < 100)
      MPE_key = "n_2";
    else if( tb.monitor.covgrp_trans.cp_trans_n3.get_coverage() < 100)
      MPE_key = "n_3";
    else if( tb.monitor.covgrp_trans.cp_trans_n4.get_coverage() < 100)
      MPE_key = "n_4";
    else if( tb.monitor.covgrp_trans.cp_trans_n5.get_coverage() < 100)
      MPE_key = "n_5";
    else if( tb.monitor.covgrp_trans.cp_trans_n6.get_coverage() < 100)
      MPE_key = "n_6";
    else if( tb.monitor.covgrp_trans.cp_trans_n7.get_coverage() < 100)
      MPE_key = "n_7";
    else if( tb.monitor.covgrp_trans.cp_trans_n8.get_coverage() < 100)
      MPE_key = "n_8";
    else if( tb.monitor.covgrp_trans.cp_trans_n9.get_coverage() < 100)
      MPE_key = "n_9";
    else if( tb.monitor.covgrp_trans.cp_trans_n10.get_coverage() < 100 )
      MPE_key = "n_10";
    else if( tb.monitor.covgrp_trans.cp_trans_n12.get_coverage() < 100)
      MPE_key = "n_12";
    else if( tb.monitor.covgrp_trans.cp_trans_n14.get_coverage() < 100)
      MPE_key = "n_14";
    else if( tb.monitor.covgrp_trans.cp_trans_n15.get_coverage() < 100)
      MPE_key = "n_15";

    else if( tb.monitor.covgrp_trans.cp_trans_p0.get_coverage() < 100 )
      MPE_key = "p_0";
    else MPE_key = "";
  endtask

endclass
```

APPENDIX B

CODE FOR MULTIPLIER BAYESIAN INFERENCE

The following is C++ code used for performing inference on the Bayesian network. The SMILE API is used for the core functionality [13].

```
// Program to perform inference on Wallace Tree Bayesian network

#include "../smile.h"
#include "../smilearn.h"
#include <fstream>
#include <iostream>
#include <sys/stat.h>
using namespace std;


int main(int argc, char *argv[])
{

   // Open Reference files

   DSL_network theNet;
   DSL_dataset dataset;

   ofstream results_file("Wallace_MPE.txt");


   // First check if data file is present
   //-------------------------------------
   struct stat stFileInfo;
   int intStat;

   // Attempt to get file attributes
   intStat = stat("wallace_data.txt", &stFileInfo);
   if(intStat != 0)
   {
      //file doesn't exist, or don't have permissions to access
```

```
      results_file << "//No data to predict MPE\n";
      results_file.close();

      cout<<"Data Set not available to predict MPE"<<endl;

      return(DSL_OKAY);
   }


   if( DSL_OKAY != theNet.ReadFile("Wallace_Basic_orig.xdsl"))
   {
      cout << "Failed to open xdsl file."<<endl;
      return -1;
   }

   if( dataset.ReadFile("wallace_data.txt") != DSL_OKAY )
   {
      cout << "Parsing data set failed."<<endl;
      return -1;
   }

   vector<DSL_datasetMatch> matching;
   string err;

   if( DSL_OKAY != dataset.MatchNetwork(theNet, matching, err) )
   {
      cout << "Matching Network nodes failure" <<endl;
      return -1;
   }

   for (unsigned i = 0; i < matching.size(); i ++)
   {
      const DSL_datasetMatch &m = matching[i];
      printf("%d col=%d slice=%d h=%d %s\n", i, m.column, m.slice, m.node,
theNet.GetNode(m.node)->GetId());
   }

   DSL_em em;
   em.SetEquivalentSampleSize(0);
   em.SetRandomizeParameters(false);

   //Learn new parameters for network
   if (DSL_OKAY != em.Learn(dataset, theNet, matching))
   {
      cout << "Learning failed."  <<endl;
      return -1;
   }

   //Write network back to file
   if (theNet.WriteFile("Wallace_Basic_new.xdsl") != DSL_OKAY)
   {
      cout << "Failed to write network file." <<endl;
      return -1;
   }
```

```
   //--------------------------------
   // Generate MPE's for all results
   //--------------------------------

   int P_MR_value;
   int P_MD_value;
   int MR_state_index;
   int MD_state_index;
   char *R_name;

   char *MR_MPE_name;
   int MR_MPE_value = 0;
   char *MD_MPE_name;
   int MD_MPE_value = 0;

   std::vector<int> mapStates(2);
   double probM1E;
   double probE;

   // use clustering algorithm
   theNet.SetDefaultBNAlgorithm(DSL_ALG_BN_LAURITZEN);

   // get the handle of input nodes and Result Node
   int MR = theNet.FindNode("MR");
   int MD = theNet.FindNode("MD");
   int Result = theNet.FindNode("Result");

   const int mNodes[2] = {MR, MD};
   const std::vector<int> mapNodes(mNodes, mNodes + 2);

   // Get names of states for result node
   DSL_idArray  *Result_Names;
   Result_Names = theNet.GetNode(Result)->Definition()->GetOutcomesNames();

   // Get names of states for MR node
   DSL_idArray *MR_Names;
   MR_Names = theNet.GetNode(MR)->Definition()->GetOutcomesNames();

   // Get names of states for MD node
   DSL_idArray *MD_Names;
   MD_Names = theNet.GetNode(MD)->Definition()->GetOutcomesNames();

   // Get coordinates to the probability table of MR and MD nodes
   DSL_sysCoordinates MR_coords(*theNet.GetNode(MR)->Value());
   DSL_sysCoordinates MD_coords(*theNet.GetNode(MD)->Value());

   // Print column headers in results file
   results_file << "//MPE Reference file\n";

   // Loop through all possible results
   for (int j = 0; j < 185; j++) //stop after last possible state is reached
to avoid predicting impossible states.
   {
      if (theNet.GetNode(Result)->Value()->SetEvidence( j ) != DSL_OKAY)
      {
         continue;  //skip if evidence is an impossible state
      }
```

```
      theNet.ClearAllEvidence();

      const std::pair<int,int> ev_node(Result, j);
      const std::vector<std::pair<int,int> > evidNodes(1, ev_node);


      if( DSL_OKAY != theNet.AnnealedMAP(evidNodes, mapNodes, mapStates,
probM1E, probE, 1))
      {
         printf("\nError doing AnnealedMap Calculation\n");
         return -1;
      }

      R_name = (*Result_Names)[j];

      MR_state_index = mapStates[0];
      MD_state_index = mapStates[1];
      MR_MPE_name = (*MR_Names)[MR_state_index];
      MD_MPE_name = (*MD_Names)[MD_state_index];

      results_file <<R_name <<"\t"<<MR_MPE_name <<"\t"<<MD_MPE_name<<"\n";
   }

   results_file <<"//end of file";
   results_file.close();

   return(DSL_OKAY);
}
```

APPENDIX C

CODE FOR VENDING MACHINE TESTBENCH

C.1 TESTBENCH TOP MODULE

```
module vend_tb_top;

   `include "ovm.svh"
   `include "vending_fsm.v"
   `include "vend_interface.sv"
   `include "vend_item.sv"
   `include "vend_driver.sv"
   `include "vend_sequencer.sv"
   `include "vend_sequence.sv"
   `include "vend_monitor.sv"
   `include "vend_env.sv"
   `include "vend_wrapper.sv"
   `include "vend_base_test.sv"

   vend_interface DUT_if();
   vend_internal_if int_if();
   vend_wrapper dutw(DUT_if, int_if);

   bit clock = 1;

   initial begin
      run_test("vend_base_test");
   end

   // generate clock
   always begin
      dutw.DUT_if.rst <= 0;
      #10 clock = ~clock;
      dutw.DUT_if.clk <= clock;
   end

endmodule
```

C.2 DESIGN INTERFACE AND WRAPPER

```
// Physical interface to DUT
interface vend_interface();
   bit clk;
   bit rst;
   bit nickel;
   bit dime;
   bit quarter;
   bit dispense;

   int n_dist, d_dist, q_dist;
endinterface

// Interface to internal DUT signals
interface vend_internal_if();

   logic [3:0] current_state;

endinterface



module vend_wrapper(vend_interface DUT_if, vend_internal_if int_if);

   vending DUT( DUT_if.clk, DUT_if.dime, DUT_if.nickel, DUT_if.quarter,
DUT_if.rst, DUT_if.dispense );

   assign int_if.current_state = DUT.current_state;
endmodule
```

C.3 SEQUENCE ITEM

```
class vend_item extends ovm_sequence_item;

  rand bit nickel;
  rand bit dime;
  rand bit quarter;

  rand int select;
  rand int n_dist, d_dist, q_dist;

  constraint c_perc{ n_dist inside {0,25,50,75,100};
                     d_dist inside {0,25,50,75,100};
                     q_dist inside {0,25,50,75,100}; }

  constraint c_sel{ select inside{1,2,3}; }

  constraint c_dist{ select dist{1:= n_dist, 2:=d_dist, 3:=q_dist}; }

  constraint c1{  (select==1)->( (nickel==1) && (dime==0) && (quarter==0) );}
  constraint c2{  (select==2)->( (nickel==0) && (dime==1) && (quarter==0) );}
  constraint c3{  (select==3)->( (nickel==0) && (dime==0) && (quarter==1) );}


  `ovm_object_utils_begin(vend_item)
     `ovm_field_int(select, OVM_ALL_ON)
     `ovm_field_int(n_dist, OVM_ALL_ON)
     `ovm_field_int(d_dist, OVM_ALL_ON)
     `ovm_field_int(q_dist, OVM_ALL_ON)
  `ovm_object_utils_end

  function new (string name = "vend_item");
      super.new(name);
  endfunction : new

endclass
```

C.4 DRIVER

```
class vend_driver extends ovm_driver #(vend_item);

  vend_item item;
  virtual vend_interface vif;

  //OVM macro for general components
  `ovm_component_utils(vend_driver)
```

```
  function new (string name = "vend_driver", ovm_component parent);
    super.new(name, parent);
  endfunction : new

  task run();

    forever begin

       //get next data item from sequencer (will block until received)
       seq_item_port.get_next_item(item);

       //drive item
       @(posedge vif.clk)
       vif.nickel <= item.nickel;
       vif.dime <= item.dime;
       vif.quarter <= item.quarter;

       vif.n_dist = item.n_dist;
       vif.d_dist = item.d_dist;
       vif.q_dist = item.q_dist;

       //signal sequencer that execution of item is complete
       seq_item_port.item_done();
    end
  endtask : run

  task reset();
    @(posedge vif.clk)
    vif.rst <= 1;
    @(posedge vif.clk)
    vif.rst <=0;
  endtask


  // Assign virtual interface to physical DUT interface
  function void assign_vi(virtual interface vend_interface DUT_if);
    this.vif = DUT_if;
  endfunction : assign_vi

endclass
```

C.5 SEQUENCE

```
class vend_sequence extends ovm_sequence #(vend_item);

   `ovm_sequence_utils(vend_sequence, vend_sequencer)

    vend_item seq_item;

    function new(string name="");
      super.new(name);
    endfunction: new

    virtual task body();
      repeat(p_sequencer.item_count) begin
        `ovm_do_with( seq_item,{n_dist == p_sequencer.n_dist;
                                d_dist == p_sequencer.d_dist;
                                q_dist == p_sequencer.q_dist;
                               })
      end

    endtask

 endclass
```

C.6 SEQUENCER

```
class vend_sequencer extends ovm_sequencer #(vend_item);

   //OVM macro for sequencers
   `ovm_sequencer_utils_begin(vend_sequencer)
      `ovm_field_int(n_dist, OVM_ALL_ON)
      `ovm_field_int(d_dist, OVM_ALL_ON)
      `ovm_field_int(q_dist, OVM_ALL_ON)

      `ovm_field_int(item_count,OVM_ALL_ON)
    `ovm_sequencer_utils_end


   //constructor
   function new(string name="vend_sequencer", ovm_component parent);
      super.new(name,parent);

      `ovm_update_sequence_lib_and_item(vend_item)

   endfunction : new
```

```
   //variables to control knobs of item

   int item_count = 10;

   int n_dist = 100;
   int d_dist = 100;
   int q_dist = 100;
endclass
```

C.7 MONITOR

```
// File: vend_monitor.sv

class vend_transaction extends ovm_transaction;
   bit nickel;
   bit dime;
   bit quarter;

   logic [3:0] l_state;
   int state;

endclass


class vend_monitor extends ovm_monitor;

   virtual vend_interface vif;

   protected vend_transaction trans;

   ovm_event state_A;

   bit coverage_enable = 1;
   int file;

   const int A = 0;
   const int B = 1;
   const int C = 2;
   const int D = 3;
   const int E = 4;
   const int F = 5;
   const int G = 6;
   const int H = 7;
   const int I = 8;
   const int J = 9;
   const int K = 10;

   string s_state;

   int step;
   string s_step;
```

```
string s_prefix = "s_";
string p_prefix = "p_";

string n_dist, d_dist, q_dist;

int mpe_count = 0;

event cov_transaction;

//declare fields that are adjustable parameters
`ovm_component_utils_begin(vend_monitor);
   `ovm_field_int(coverage_enable, OVM_ALL_ON)
`ovm_component_utils_end

// Define cover points
covergroup covgrp_trans @cov_transaction;

  option.at_least = 2;

  cp_A: coverpoint trans.state{ bins cA={A}; }
  cp_B: coverpoint trans.state{ bins cB={B}; }
  cp_C: coverpoint trans.state{ bins cC={C}; }
  cp_D: coverpoint trans.state{ bins cD={D}; }
  cp_E: coverpoint trans.state{ bins cE={E}; }
  cp_F: coverpoint trans.state{ bins cF={F}; }
  cp_G: coverpoint trans.state{ bins cG={G}; }
  cp_H: coverpoint trans.state{ bins cH={H}; }
  cp_I: coverpoint trans.state{ bins cI={I}; }
  cp_J: coverpoint trans.state{ bins cJ={J};}
  cp_K: coverpoint trans.state{ bins cK={K}; }

  cp_s2: coverpoint step{ bins s2={2}; }
  cp_s3: coverpoint step{ bins s3={3}; }
  cp_s4: coverpoint step{ bins s4={4}; }
  cp_s5: coverpoint step{ bins s5={5}; }
  cp_s6: coverpoint step{ bins s6={6}; }
  cp_s7: coverpoint step{ bins s7={7}; }

  cp_2B: cross cp_s2, cp_B;
  cp_2C: cross cp_s2, cp_C;
  cp_2F: cross cp_s2, cp_F;

  cp_3C: cross cp_s3, cp_C;
  cp_3D: cross cp_s3, cp_D;
  cp_3G: cross cp_s3, cp_G;
  cp_3E: cross cp_s3, cp_E;
  cp_3K: cross cp_s3, cp_K;
  cp_3H: cross cp_s3, cp_H;

  cp_4A: cross cp_s4, cp_A;
  cp_4B: cross cp_s4, cp_B;
  cp_4D: cross cp_s4, cp_D;
  cp_4E: cross cp_s4, cp_E;
  cp_4F: cross cp_s4, cp_F;
  cp_4G: cross cp_s4, cp_G;
  cp_4I: cross cp_s4, cp_I;
  cp_4J: cross cp_s4, cp_J;
```

```
   cp_4K: cross cp_s4, cp_K;

   cp_5A: cross cp_s5, cp_A;
   cp_5B: cross cp_s5, cp_B;
   cp_5C: cross cp_s5, cp_C;
   cp_5D: cross cp_s5, cp_D;
   cp_5E: cross cp_s5, cp_E;
   cp_5F: cross cp_s5, cp_F;
   cp_5G: cross cp_s5, cp_G;
   cp_5H: cross cp_s5, cp_H;
   cp_5I: cross cp_s5, cp_I;
   cp_5J: cross cp_s5, cp_J;
   cp_5K: cross cp_s5, cp_K;

   cp_6A: cross cp_s6, cp_A;
   cp_6B: cross cp_s6, cp_B;
   cp_6C: cross cp_s6, cp_C;
   cp_6D: cross cp_s6, cp_D;
   cp_6E: cross cp_s6, cp_E;
   cp_6F: cross cp_s6, cp_F;
   cp_6G: cross cp_s6, cp_G;
   cp_6H: cross cp_s6, cp_H;
   cp_6I: cross cp_s6, cp_I;
   cp_6J: cross cp_s6, cp_J;
   cp_6K: cross cp_s6, cp_K;

   cp_7A: cross cp_s7, cp_A;
   cp_7B: cross cp_s7, cp_B;
   cp_7C: cross cp_s7, cp_C;
   cp_7D: cross cp_s7, cp_D;
   cp_7E: cross cp_s7, cp_E;
   cp_7F: cross cp_s7, cp_F;
   cp_7G: cross cp_s7, cp_G;
   cp_7H: cross cp_s7, cp_H;
   cp_7I: cross cp_s7, cp_I;
   cp_7J: cross cp_s7, cp_J;
   cp_7K: cross cp_s7, cp_K;

endgroup

function new (string name, ovm_component parent);
   super.new(name, parent);
   covgrp_trans = new();
   trans = new();
endfunction

virtual task run();

   file = $fopen("vend_data.txt");
   $fwrite(file, "p_N\tp_D\tp_Q\tS2\tS3\tS4\tS5\tS6\tS7\n");
   $fclose(file);

   fork
      collect_transactions();
   join
endtask
```

```
virtual protected task collect_transactions();

   step = 100;

   forever begin

     @(negedge vif.clk);

     trans.l_state = vend_tb_top.int_if.current_state;

     $cast(trans.state, trans.l_state);

     //convert Result to string
    if (trans.state == A) s_state = "A";
    else if(trans.state == B) s_state = "B";
    else if(trans.state == C) s_state = "C";
    else if(trans.state == D) s_state = "D";
    else if(trans.state == E) s_state = "E";
    else if(trans.state == F) s_state = "F";
    else if(trans.state == G) s_state = "G";
    else if(trans.state == H) s_state = "H";
    else if(trans.state == I) s_state = "I";
    else if(trans.state == J) s_state = "J";
    else if(trans.state == K) s_state = "K";
    else `message( OVM_LOW, ("ILLEGAL STATE"));


    //start counting steps through state machine

    if(step > 7 ) begin

       if(trans.state == A) begin
          step = 0;

          n_dist.itoa(vif.n_dist);
          d_dist.itoa(vif.d_dist);
          q_dist.itoa(vif.q_dist);

          file = $fopen("vend_data.txt", "a");

          $fwrite(file, "%s%s\t%s%s\t%s%s\t",
              p_prefix, n_dist,
              p_prefix, d_dist,
              p_prefix, q_dist
            );

          $fclose(file);
        end
    end

    step++;

    if ( step > 1 && step < 7 ) begin

       file = $fopen("vend_data.txt", "a");
       $fwrite(file, "%s\t", s_state );
       $fclose(file);
```

```
          if (coverage_enable)
             -> cov_transaction;
      end

      else if ( step == 7 ) begin
            file = $fopen("vend_data.txt", "a");
            $fwrite(file, "%s\n", s_state );
        $fclose(file);

          if (coverage_enable)
             -> cov_transaction;

      end
     end
  endtask

  // Assign virtual interface to physical DUT interface;
  function void assign_vi(virtual interface vend_interface DUT_if );
    this.vif = DUT_if;
  endfunction

endclass
```

C.8 TESTBENCH ENVIRONMENT

```
/* verification environment */

class vend_env extends ovm_env;

  `ovm_component_utils(vend_env)

  vend_sequencer         sequencer;
  vend_driver            driver;
  vend_monitor           monitor;

  function new(string name, ovm_component parent);
    super.new(name, parent);
  endfunction

  virtual function void build();
    super.build();

    sequencer = vend_sequencer::type_id::create("sequencer",this);
    driver = vend_driver::type_id::create("driver", this);
    monitor = vend_monitor::type_id::create("monitor", this);

  endfunction

  virtual function void connect();
```

```
  // Connect driver's port to sequencer's export
  driver.seq_item_port.connect(sequencer.seq_item_export);

     driver.assign_vi(vend_tb_top.DUT_if);
     monitor.assign_vi(vend_tb_top.DUT_if);
  endfunction
endclass
```

C.9 BASE TEST

```
class vend_base_test extends ovm_test;

   `ovm_component_utils(vend_base_test)

   int CLOCK_PERIOD = 20;

   vend_sequence base_sequence;

   integer input_file, c, r;
   string label, line, val, val2, val3;

   rand int n_dist, d_dist, q_dist;

   constraint c_1{ n_dist inside {0,25,50,75,100};
                   d_dist inside {0,25,50,75,100};
                   q_dist inside {0,25,50,75,100};

                   n_dist || d_dist || q_dist != 0;
                  }

   string MPE_line_ref[string];
   string MPE_key;
   int N_MPE, D_MPE, Q_MPE;

   real cvr_pt_coverage;
   int counter;

   vend_env tb;   //will instantiate the testbench

   function new (string name = "vend_base_test", ovm_component parent =
null);
      super.new(name, parent);
   endfunction

   //build testbench
   virtual function void build();
      super.build();

      // set number sequence items for default sequencer to 0
      set_config_int("*.sequencer","count", 0);
```

```
      base_sequence = new();
      tb = vend_env::type_id::create("tb", this);

   endfunction

   function void read_MPE_file();
     //open MPE Reference file and store each line in an array indexed by
result string
      input_file = $fopen("vend_MPE.txt","r");

      while(!$feof(input_file)) begin

        c = $fgetc(input_file);

        if(c == "/")
          r = $fgets(line, input_file);
        else begin
          //push char back tofile to read line
          r = $ungetc(c, input_file);
          r = $fgets(line, input_file);
          r = $sscanf(line, "%s\t%s\t%s\t%s", MPE_key, val, val2, val3);
          MPE_line_ref[MPE_key] = line;
        end
      end

      $fclose(input_file);

   endfunction

   virtual task run();

      //Start simulation with random input paramaters every 20 cycles
      for(int k=0; k<50; k++) begin

        set_dist(1,10);  //randomize input parameters

        base_sequence.start(tb.sequencer);
        base_sequence.wait_for_sequence_state(FINISHED);

      end

      cvr_pt_coverage = tb.monitor.covgrp_trans.get_coverage();
      `message(OVM_LOW, ("Coverage is %f", cvr_pt_coverage))

      counter=0;

      while (cvr_pt_coverage < 100) begin

        //Relearn parameters for Bayesian Network
        $system("vend_mpe");

        //Read new most probable expectation for all results
        read_MPE_file();

        //reset
        tb.driver.reset();
```

```
      //Find next hole and bias for 10 test cases
      set_dist(1,10);
      base_sequence.start(tb.sequencer);
      base_sequence.wait_for_sequence_state(FINISHED);

      if( counter++ == 10) begin

         cvr_pt_coverage = tb.monitor.covgrp_trans.get_coverage();
         `message(OVM_LOW, ("Coverage is %f", cvr_pt_coverage))
         counter = 0;
      end
    end

    #(CLOCK_PERIOD);
    global_stop_request();

endtask

task set_dist(int mode, int count);

 tb.sequencer.item_count = count;

 if(mode == 0) begin  //use random input settings

    assert( randomize(n_dist, d_dist, q_dist) );

    tb.sequencer.n_dist = n_dist;
    tb.sequencer.d_dist = d_dist;
    tb.sequencer.q_dist = q_dist;

 end
 else begin

    // FIND FIRST HOLE IN COVERAGE
    find_first_hole(); //will set MPE_key to result not generated yet

    if(MPE_key == "") begin
      `message(OVM_LOW,("No coverage holes found"))

       assert( randomize(n_dist, d_dist,q_dist) );
       tb.sequencer.n_dist = n_dist;
       tb.sequencer.d_dist = d_dist;
       tb.sequencer.q_dist = q_dist;

    end
    else if( 0 == MPE_line_ref.exists(MPE_key) ) begin

      //Desired result has not been observed by Bayesian Network

       assert( randomize(n_dist, d_dist,q_dist) );
       tb.sequencer.n_dist = n_dist;
       tb.sequencer.d_dist = d_dist;
       tb.sequencer.q_dist = q_dist;

    end

    else begin
```

```
      `message(OVM_LOW,("Starting Sequence with biased distribution"))

      r = $sscanf(MPE_line_ref[MPE_key], "%s\tp_%d\tp_%d\tp_%d",
                  val, N_MPE, D_MPE, Q_MPE);

      tb.sequencer.n_dist = N_MPE;
      tb.sequencer.d_dist = D_MPE;
      tb.sequencer.q_dist = Q_MPE;
    end
  end
endtask

task find_first_hole();
  if( tb.monitor.covgrp_trans.cp_2B.get_coverage() == 0 )
    MPE_key = "2B";
  else if( tb.monitor.covgrp_trans.cp_2C.get_coverage() == 0 )
    MPE_key = "2C";
  else if( tb.monitor.covgrp_trans.cp_2F.get_coverage() == 0 )
    MPE_key = "2F";

  else if( tb.monitor.covgrp_trans.cp_3C.get_coverage() == 0 )
    MPE_key = "3C";
  else if( tb.monitor.covgrp_trans.cp_3D.get_coverage() == 0 )
    MPE_key = "3D";
  else if( tb.monitor.covgrp_trans.cp_3G.get_coverage() == 0 )
    MPE_key = "3G";
  else if( tb.monitor.covgrp_trans.cp_3E.get_coverage() == 0 )
    MPE_key = "3E";
  else if( tb.monitor.covgrp_trans.cp_3K.get_coverage() == 0 )
    MPE_key = "3K";
  else if( tb.monitor.covgrp_trans.cp_3H.get_coverage() == 0 )
    MPE_key = "3H";
  else if( tb.monitor.covgrp_trans.cp_4A.get_coverage() == 0 )
    MPE_key = "4A";
  else if( tb.monitor.covgrp_trans.cp_4B.get_coverage() == 0 )
    MPE_key = "4B";
  else if( tb.monitor.covgrp_trans.cp_4D.get_coverage() == 0 )
    MPE_key = "4D";
  else if( tb.monitor.covgrp_trans.cp_4E.get_coverage() == 0 )
    MPE_key = "4E";
  else if( tb.monitor.covgrp_trans.cp_4F.get_coverage() == 0 )
    MPE_key = "4F";
  else if( tb.monitor.covgrp_trans.cp_4G.get_coverage() == 0 )
    MPE_key = "4G";
  else if( tb.monitor.covgrp_trans.cp_4I.get_coverage() == 0 )
    MPE_key = "4I";
  else if( tb.monitor.covgrp_trans.cp_4J.get_coverage() == 0 )
    MPE_key = "4J";
  else if( tb.monitor.covgrp_trans.cp_4K.get_coverage() == 0 )
    MPE_key = "4K";
  else if( tb.monitor.covgrp_trans.cp_5A.get_coverage() == 0 )
    MPE_key = "5A";
  else if( tb.monitor.covgrp_trans.cp_5B.get_coverage() == 0 )
    MPE_key = "5B";
  else if( tb.monitor.covgrp_trans.cp_5C.get_coverage() == 0 )
    MPE_key = "5C";
```

```
else if( tb.monitor.covgrp_trans.cp_5D.get_coverage() == 0 )
  MPE_key = "5D";
else if( tb.monitor.covgrp_trans.cp_5E.get_coverage() == 0 )
  MPE_key = "5E";
else if( tb.monitor.covgrp_trans.cp_5F.get_coverage() == 0 )
  MPE_key = "5F";
else if( tb.monitor.covgrp_trans.cp_5G.get_coverage() == 0 )
  MPE_key = "5G";
else if( tb.monitor.covgrp_trans.cp_5H.get_coverage() == 0 )
  MPE_key = "5H";
else if( tb.monitor.covgrp_trans.cp_5I.get_coverage() == 0 )
  MPE_key = "5I";
else if( tb.monitor.covgrp_trans.cp_5J.get_coverage() == 0 )
  MPE_key = "5J";
else if( tb.monitor.covgrp_trans.cp_5K.get_coverage() == 0 )
  MPE_key = "5K";

else if( tb.monitor.covgrp_trans.cp_6A.get_coverage() == 0 )
  MPE_key = "6A";
else if( tb.monitor.covgrp_trans.cp_6B.get_coverage() == 0 )
  MPE_key = "6B";
else if( tb.monitor.covgrp_trans.cp_6C.get_coverage() == 0 )
  MPE_key = "6C";
else if( tb.monitor.covgrp_trans.cp_6D.get_coverage() == 0 )
  MPE_key = "6D";
else if( tb.monitor.covgrp_trans.cp_6E.get_coverage() == 0 )
  MPE_key = "6E";
else if( tb.monitor.covgrp_trans.cp_6F.get_coverage() == 0 )
  MPE_key = "6F";
else if( tb.monitor.covgrp_trans.cp_6G.get_coverage() == 0 )
  MPE_key = "6G";
else if( tb.monitor.covgrp_trans.cp_6H.get_coverage() == 0 )
  MPE_key = "6H";
else if( tb.monitor.covgrp_trans.cp_6I.get_coverage() == 0 )
  MPE_key = "6I";
else if( tb.monitor.covgrp_trans.cp_6J.get_coverage() == 0 )
  MPE_key = "6J";
else if( tb.monitor.covgrp_trans.cp_6K.get_coverage() == 0 )
  MPE_key = "6K";

else if( tb.monitor.covgrp_trans.cp_7A.get_coverage() == 0 )
  MPE_key = "7A";
else if( tb.monitor.covgrp_trans.cp_7B.get_coverage() == 0 )
  MPE_key = "7B";
else if( tb.monitor.covgrp_trans.cp_7C.get_coverage() == 0 )
  MPE_key = "7C";
else if( tb.monitor.covgrp_trans.cp_7D.get_coverage() == 0 )
  MPE_key = "7D";
else if( tb.monitor.covgrp_trans.cp_7E.get_coverage() == 0 )
  MPE_key = "7E";
else if( tb.monitor.covgrp_trans.cp_7F.get_coverage() == 0 )
  MPE_key = "7F";
else if( tb.monitor.covgrp_trans.cp_7G.get_coverage() == 0 )
  MPE_key = "7G";
else if( tb.monitor.covgrp_trans.cp_7H.get_coverage() == 0 )
  MPE_key = "7H";
else if( tb.monitor.covgrp_trans.cp_7I.get_coverage() == 0 )
```

```
      MPE_key = "7I";
    else if( tb.monitor.covgrp_trans.cp_7J.get_coverage() == 0 )
      MPE_key = "7J";
    else if( tb.monitor.covgrp_trans.cp_7K.get_coverage() == 0 )
      MPE_key = "7K";

    else MPE_key = "";
  endtask


endclass
```

APPENDIX D

CODE FOR VENDING MACHINE BAYESIAN INFERENCE

The following is C++ code used for performing inference on the vending machine Bayesian network. The SMILE API was used for the core functionality [13].

```
#include "../smile.h"
#include "../smilearn.h"
#include <fstream>
#include <iostream>
#include <sys/stat.h>
using namespace std;


int main(int argc, char *argv[])
{

   // Open Reference files

   DSL_network theNet;
   DSL_dataset dataset;

   ofstream results_file("vend_MPE.txt");


   // First check if data file is present
   //-------------------------------------
   struct stat stFileInfo;
   int intStat;

   // Attempt to get file attributes
   intStat = stat("vend_data.txt", &stFileInfo);
   if(intStat != 0)
   {
      //file doesn't exist, or don't have permissions to access
```

```
      results_file << "//No data to predict MPE\n";
      results_file.close();

      cout<<"Data Set not available to predict MPE"<<endl;

      return(DSL_OKAY);
   }


   if( DSL_OKAY != theNet.ReadFile("vending_orig.xdsl"))
   {
      cout << "Failed to open xdsl file."<<endl;
      return -1;
   }

   if( dataset.ReadFile("vend_data.txt") != DSL_OKAY )
   {
      cout << "Parsing data set failed."<<endl;
      return -1;
   }

   vector<DSL_datasetMatch> matching;
   string err;

   if( DSL_OKAY != dataset.MatchNetwork(theNet, matching, err) )
   {
      cout << "Matching Network nodes failure" <<endl;
      return -1;
   }

   for (unsigned i = 0; i < matching.size(); i ++)
   {
      const DSL_datasetMatch &m = matching[i];
      printf("%d col=%d slice=%d h=%d %s\n", i, m.column, m.slice, m.node,
theNet.GetNode(m.node)->GetId());
   }

   DSL_em em;
   em.SetEquivalentSampleSize(0);
   em.SetRandomizeParameters(false);

   //Learn new parameters for network
   if (DSL_OKAY != em.Learn(dataset, theNet, matching))
   {
      cout << "Learning failed."  <<endl;
      return -1;
   }

   //Write network back to file
   if (theNet.WriteFile("vending_new.xdsl") != DSL_OKAY)
   {
      cout << "Failed to write network file." <<endl;
      return -1;
   }

   //---------------------------------
   // Generate MPE's for all results
```

```
   //--------------------------------

   int P_N_value;
   int P_D_value;
   int P_Q_value;
   int N_state_index;
   int D_state_index;
   int Q_state_index;
   char *R_name;

   char *N_MPE_name;
   int N_MPE_value = 0;
   char *D_MPE_name;
   int D_MPE_value = 0;
   char *Q_MPE_name;
   int Q_MPE_value = 0;

   std::vector<int> mapStates(2);
   double probM1E;
   double probE;

   // use clustering algorithm
   theNet.SetDefaultBNAlgorithm(DSL_ALG_BN_LAURITZEN);

   // get the handle of input nodes and Result Node
   int p_N = theNet.FindNode("p_N");
   int p_D = theNet.FindNode("p_D");
   int p_Q = theNet.FindNode("p_Q");
   int s2 = theNet.FindNode("S2");
   int s3 = theNet.FindNode("S3");
   int s4 = theNet.FindNode("S4");
   int s5 = theNet.FindNode("S5");
   int s6 = theNet.FindNode("S6");
   int s7 = theNet.FindNode("S7");

   const int mNodes[3] = {p_N, p_D, p_Q};
   const std::vector<int> mapNodes(mNodes, mNodes + 3);

   // Get names of states for result nodes
   DSL_idArray  *s2_Names, *s3_Names, *s4_Names,
*s5_Names,*s6_Names,*s7_Names;
   s2_Names = theNet.GetNode(s2)->Definition()->GetOutcomesNames();
   s3_Names = theNet.GetNode(s3)->Definition()->GetOutcomesNames();
   s4_Names = theNet.GetNode(s4)->Definition()->GetOutcomesNames();
   s5_Names = theNet.GetNode(s5)->Definition()->GetOutcomesNames();
   s6_Names = theNet.GetNode(s6)->Definition()->GetOutcomesNames();
   s7_Names = theNet.GetNode(s7)->Definition()->GetOutcomesNames();

   // Get names of states for nickel node
   DSL_idArray *N_Names;
   N_Names = theNet.GetNode(p_N)->Definition()->GetOutcomesNames();

   // Get names of states for dime node
   DSL_idArray *D_Names;
   D_Names = theNet.GetNode(p_D)->Definition()->GetOutcomesNames();

   // Get names of states for Quarter node
```

```
   DSL_idArray *Q_Names;
   Q_Names = theNet.GetNode(p_Q)->Definition()->GetOutcomesNames();

   // Get coordinates to the probability table of nickel, dime,and quarter
nodes
   DSL_sysCoordinates N_coords(*theNet.GetNode(p_N)->Value());
   DSL_sysCoordinates D_coords(*theNet.GetNode(p_D)->Value());
   DSL_sysCoordinates Q_coords(*theNet.GetNode(p_Q)->Value());

   // Print column headers in results file
   results_file << "//MPE Reference file\n";

   // Loop through all possible Step 2 results
   for (int j = 0; j < 3; j++) //stop after last possible state is reached to
avoid predicting impossible states.
   {
      if (theNet.GetNode(s2)->Value()->SetEvidence( j ) != DSL_OKAY)
        continue;  //skip if evidence is an impossible state


      theNet.ClearAllEvidence();

      const std::pair<int,int> ev_node(s2, j);
      const std::vector<std::pair<int,int> > evidNodes(1, ev_node);

      if( DSL_OKAY != theNet.AnnealedMAP(evidNodes, mapNodes, mapStates,
probM1E, probE, 1))
      {
         printf("\nError doing AnnealedMap Calculation\n");
         return -1;
      }

      R_name = (*s2_Names)[j];

      N_state_index = mapStates[0];
      D_state_index = mapStates[1];
      Q_state_index = mapStates[2];
      N_MPE_name = (*N_Names)[N_state_index];
      D_MPE_name = (*D_Names)[D_state_index];
      Q_MPE_name = (*Q_Names)[Q_state_index];

      results_file <<"2"<<R_name <<"\t"<<N_MPE_name
<<"\t"<<D_MPE_name<<"\t"<<Q_MPE_name<<"\n";
   }

   // Loop through all possible Step 3 results
   for (int j = 0; j < 6; j++) //stop after last possible state is reached to
avoid predicting impossible states.
   {
      if (theNet.GetNode(s3)->Value()->SetEvidence( j ) != DSL_OKAY)
       continue;  //skip if evidence is an impossible state     }

      theNet.ClearAllEvidence();

      const std::pair<int,int> ev_node(s3, j);
      const std::vector<std::pair<int,int> > evidNodes(1, ev_node);
```

```
      if( DSL_OKAY != theNet.AnnealedMAP(evidNodes, mapNodes, mapStates,
probM1E, probE, 1))
      {
         printf("\nError doing AnnealedMap Calculation\n");
         return -1;
      }

      R_name = (*s3_Names)[j];

      N_state_index = mapStates[0];
      D_state_index = mapStates[1];
      Q_state_index = mapStates[2];
      N_MPE_name = (*N_Names)[N_state_index];
      D_MPE_name = (*D_Names)[D_state_index];
      Q_MPE_name = (*Q_Names)[Q_state_index];

      results_file <<"3"<<R_name <<"\t"<<N_MPE_name
<<"\t"<<D_MPE_name<<"\t"<<Q_MPE_name<<"\n";
   }

   // Loop through all possible Step 4 results
   for (int j = 0; j < 9; j++) //stop after last possible state is reached to
avoid predicting impossible states.
   {
      if (theNet.GetNode(s4)->Value()->SetEvidence( j ) != DSL_OKAY)
       continue;  //skip if evidence is an impossible state     }

      theNet.ClearAllEvidence();

      const std::pair<int,int> ev_node(s4, j);
      const std::vector<std::pair<int,int> > evidNodes(1, ev_node);

      if( DSL_OKAY != theNet.AnnealedMAP(evidNodes, mapNodes, mapStates,
probM1E, probE, 1))
      {
         printf("\nError doing AnnealedMap Calculation\n");
         return -1;
      }

      R_name = (*s4_Names)[j];

      N_state_index = mapStates[0];
      D_state_index = mapStates[1];
      Q_state_index = mapStates[2];
      N_MPE_name = (*N_Names)[N_state_index];
      D_MPE_name = (*D_Names)[D_state_index];
      Q_MPE_name = (*Q_Names)[Q_state_index];

      results_file <<"4"<<R_name <<"\t"<<N_MPE_name
<<"\t"<<D_MPE_name<<"\t"<<Q_MPE_name<<"\n";
   }

   // Loop through all possible Step 5 results
   for (int j = 0; j < 11; j++) //stop after last possible state is reached
to avoid predicting impossible states.
   {
      if (theNet.GetNode(s5)->Value()->SetEvidence( j ) != DSL_OKAY)
```

```
       continue;  //skip if evidence is an impossible state     }

      theNet.ClearAllEvidence();

      const std::pair<int,int> ev_node(s5, j);
      const std::vector<std::pair<int,int> > evidNodes(1, ev_node);

      if( DSL_OKAY != theNet.AnnealedMAP(evidNodes, mapNodes, mapStates,
probM1E, probE, 1))
      {
         printf("\nError doing AnnealedMap Calculation\n");
         return -1;
      }

      R_name = (*s5_Names)[j];

      N_state_index = mapStates[0];
      D_state_index = mapStates[1];
      Q_state_index = mapStates[2];
      N_MPE_name = (*N_Names)[N_state_index];
      D_MPE_name = (*D_Names)[D_state_index];
      Q_MPE_name = (*Q_Names)[Q_state_index];

      results_file <<"5"<<R_name <<"\t"<<N_MPE_name
<<"\t"<<D_MPE_name<<"\t"<<Q_MPE_name<<"\n";
   }

   // Loop through all possible Step 6 results
   for (int j = 0; j < 11; j++) //stop after last possible state is reached
to avoid predicting impossible states.
   {
      if (theNet.GetNode(s6)->Value()->SetEvidence( j ) != DSL_OKAY)
       continue;  //skip if evidence is an impossible state     }

      theNet.ClearAllEvidence();

      const std::pair<int,int> ev_node(s6, j);
      const std::vector<std::pair<int,int> > evidNodes(1, ev_node);

      if( DSL_OKAY != theNet.AnnealedMAP(evidNodes, mapNodes, mapStates,
probM1E, probE, 1))
      {
         printf("\nError doing AnnealedMap Calculation\n");
         return -1;
      }

      R_name = (*s6_Names)[j];

      N_state_index = mapStates[0];
      D_state_index = mapStates[1];
      Q_state_index = mapStates[2];
      N_MPE_name = (*N_Names)[N_state_index];
      D_MPE_name = (*D_Names)[D_state_index];
      Q_MPE_name = (*Q_Names)[Q_state_index];

      results_file <<"6"<<R_name <<"\t"<<N_MPE_name
<<"\t"<<D_MPE_name<<"\t"<<Q_MPE_name<<"\n";
```

```
   }
   // Loop through all possible Step 7results
   for (int j = 0; j < 11; j++) //stop after last possible state is reached
to avoid predicting impossible states.
   {
      if (theNet.GetNode(s7)->Value()->SetEvidence( j ) != DSL_OKAY)
       continue;  //skip if evidence is an impossible state     }

      theNet.ClearAllEvidence();

      const std::pair<int,int> ev_node(s7, j);
      const std::vector<std::pair<int,int> > evidNodes(1, ev_node);

      if( DSL_OKAY != theNet.AnnealedMAP(evidNodes, mapNodes, mapStates,
probM1E, probE, 1))
      {
         printf("\nError doing AnnealedMap Calculation\n");
         return -1;
      }

      R_name = (*s7_Names)[j];

      N_state_index = mapStates[0];
      D_state_index = mapStates[1];
      Q_state_index = mapStates[2];
      N_MPE_name = (*N_Names)[N_state_index];
      D_MPE_name = (*D_Names)[D_state_index];
      Q_MPE_name = (*Q_Names)[Q_state_index];

      results_file <<"7"<<R_name <<"\t"<<N_MPE_name
<<"\t"<<D_MPE_name<<"\t"<<Q_MPE_name<<"\n";
   }
   results_file <<"//end of file";
   results_file.close();

   return(DSL_OKAY);
}
```

APPENDIX E

EXAMPLE LOG FILES

This Section shows several example log files that were created as part of the coverage directed test generation testbench. These logs were to transfer domain data between the testbench and Bayesian network inference engine. Inference results logged to files were read by the testbench when targeting specific coverage holes to fill. Testbench simulation data was logged to files to be parsed by the Bayesian inference engine for learning network parameters.

E.1 WALLACE TREE INFERENCE RESULTS

The following is an example log file for calculating Most Probable Explanation (MPE) predictions. The contents of this file varied from different simulations, with predictions for the inputs that occurred most often for each result. The first column identifies the result (multiplier output) and the second columns identify the corresoponding likely inputs to produce that result.

```
//MPE Reference file
p_0   n_1   p_0
p_1   n_1   n_1
p_2   n_1   n_2
p_3   n_1   n_3
p_4   n_1   n_4
```

```
p_5   n_1   n_5
p_6   n_6   n_1
p_7   n_1   n_7
p_8   n_1   n_8
p_9   n_3   n_3
p_10  n_5   n_2
p_12  n_6   n_2
p_14  p_2   p_7
p_15  n_3   n_5
p_16  n_8   n_2
p_18  n_6   n_3
p_20  p_4   p_5
p_21  n_3   n_7
p_24  n_6   n_4
p_25  n_5   n_5
p_28  p_4   p_7
p_30  n_6   n_5
p_32  n_8   n_4
p_35  n_5   n_7
p_36  n_6   n_6
p_40  n_5   n_8
p_42  n_6   n_7
p_48  n_8   n_6
p_49  p_7   p_7
p_56  n_8   n_7
p_64  n_8   n_8
n_1   n_1   p_1
n_2   n_1   p_2
n_3   n_1   p_3
n_4   n_1   p_4
n_5   n_1   p_5
n_6   n_1   p_6
n_7   n_1   p_7
n_8   p_4   n_2
n_9   n_3   p_3
n_10  p_2   n_5
n_12  n_3   p_4
n_14  p_7   n_2
n_15  n_3   p_5
n_16  p_4   n_4
n_18  n_3   p_6
n_20  p_4   n_5
n_21  n_3   p_7
n_24  n_6   p_4
n_25  n_5   p_5
n_28  p_4   n_7
n_30  n_6   p_5
n_32  p_4   n_8
n_35  p_7   n_5
n_36  n_6   p_6
n_40  n_8   p_5
n_42  n_6   p_7
n_48  n_8   p_6
n_49  p_7   n_7
n_56  p_7   n_8
//end of file
```

E.2 VENDING MACHINE TESTBENCH DATA

The following is an example data set generated by the testbench for learning Bayesian network parameters. The first three columns correspond to the probability distributions used as test generation parameters for inputting a nickel, dime, and quarter. The following columns capture the resulting states that occurred for the first 7 clock cycles after state A.

p_N	p_D	p_Q	S2	S3	S4	S5	S6	S7
p_75	p_25	p_100	F	K	B	G	A	B
p_50	p_50	p_25	B	D	E	I	D	J
p_100	p_25	p_50	B	G	A	F	G	A
p_50	p_0	p_100	F	H	E	I	D	J
p_25	p_100	p_0	C	E	G	A	C	E
p_0	p_0	p_100	F	H	E	G	A	C
p_25	p_100	p_0	C	E	G	A	C	E
p_50	p_50	p_25	F	G	A	C	E	F
p_50	p_50	p_75	F	G	A	B	D	E
p_0	p_75	p_0	C	E	G	A	C	E
p_50	p_50	p_25	B	C	K	B	D	F
p_75	p_0	p_25	B	G	A	F	H	E
p_50	p_75	p_25	B	D	F	H	E	G
p_0	p_0	p_75	F	H	E	I	D	E
p_50	p_100	p_50	C	E	I	D	E	F
p_50	p_100	p_75	C	E	G	A	C	E
p_0	p_50	p_25	C	K	B	D	F	K
p_100	p_75	p_75	C	E	I	D	J	C
p_75	p_100	p_50	C	D	E	F	H	E
p_0	p_100	p_75	C	K	B	G	A	F
p_25	p_25	p_0	B	D	E	G	A	B
p_50	p_25	p_0	B	C	D	E	F	G
p_25	p_75	p_0	C	E	G	A	C	D
p_50	p_50	p_0	C	D	E	F	G	A
p_50	p_25	p_100	F	H	E	I	D	F
p_0	p_0	p_75	F	H	E	I	D	J
p_100	p_100	p_75	B	C	D	F	K	B
p_100	p_75	p_0	B	D	E	F	K	B
p_25	p_25	p_0	C	D	F	K	B	D
p_0	p_100	p_50	C	D	J	C	D	J
p_75	p_50	p_75	B	D	J	C	K	B
p_100	p_25	p_50	C	D	E	G	A	F
p_50	p_75	p_50	F	K	B	D	F	K
p_0	p_25	p_50	C	K	B	G	A	C
p_25	p_25	p_75	A	F	H	E	E	E

BIBLIOGRAPHY

[1] D. Baras, L. Fournier, and A. Ziv, "Automatic Boosting of Cross-Product Coverage Using Bayesian Networks," *Proceedings of the 4th International Haifa Verification Conference*, *HVC 2008*, LNCS 5394, pp. 53-67, Springer-Verlag Berlin Heidelberg 2009.

[2] "Bayesian Network," Website: http://en.wikipedia.org/wiki/Bayesian_networks, Wikipedia, 2009.

[3] S. Brown, Z. Vranesic, "Fundamentals of Digital Logic with VHDL Design," McGaw-Hill Companies, 2000.

[4] "e Reuse Methodology Developer Manual," Verisity Design, 2002-2004.

[5] S. Fine, A. Ziv, "Coverage Directed Test Generation for Functional Verification Using Bayesian Networks," *Proceedings of the 40th Design Automation Conference*, June 2003, pp. 286-291.

[6] S. Fine, A. Freund, I. Jaeger, Y. Mansour, Y. Naveh, A. Ziv, "Harnessing Machine Learning to Improve the Success Rate of Stimuli Generation," IEEE *Transactions on Computers*, vol. 55, no. 11, pp. 1344-1355, Nov. 2006.

[7] S. Fine, L. Fournier, and A. Ziv, "Using Bayesian Networks and Virtual Coverage to Hit Hard-To-Reach Events," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 4, pp. 291-305, Springer Berlin Heidelberg, 2009.

[8] J. Hennessy, D. Patterson, "Computer Architecture: A Quantitative Approach," 4th Edition, Morgan Kaufmann, 2006.

[9] "IEEE Standard for the Functional Verification Language e," *IEEE STD 1647-2008*, 2008.

[10] S. Iman, "Step-by-Step Functional Verification with SystemVerilog and OVM," Hansen Brown Publishing Company, 2008.

[11] F. Jensen, "An Introduction to Bayesian Networks," Springer-Verlag New York, Inc, 1996.

[12] F. Jensen, and T. Nielsen, "Bayesian Networks and Decision Graphs," Second Edition, Springer Science + Business Media, LLC, 2007.

[13] T. Loboda, M. Voortman, Website “GeNIe & SMILE,” http://genie.sis.pitt.edu, Decision Systems Laboratory, University of Pittsburgh, 2009.

[14] “OVM World: Open Verification Methodology,” Website: http://www.ovmworld.org, Mentor Graphics Corporation and Cadence Design Systems, Inc, 2009.

[15] J. Pearl, “Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference,” Morgan Kaufmann Publishers, Inc, 1988.

[16] A. Piziali, “Functional Verification Coverage Measurement and Analysis,” Kluwer Academic Publishers, 2004.

[17] “Questa Advanced Verification and Debug Technologies,” Website: http://www.mentor.com/products/fv/questa, Mentor Graphics, 2009.

[18] N. Rahman, R. Yun, Website “8x8 Booth Encoded Wallace Tree Multiplier,” Website: http://www.eecs.tufts.edu/~ryun01/vlsi/verilog_simulation.htm, 2009.

[19] “Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language,” *IEC 62530:2007*, 2007.

[20] P. Wilcox, “Professional Verification: A Guide to Advanced Functional Verification,” Kluwer Academic Publishers, 2004.

[21] B. Wile, J.C. Goss, and W. Roesner, “Comprehensive Functional Verification – The Complete Industry Cycle,” Elsevier, 2005.

[22] S. Ur, Y. Yadin, “Micro Architecture Coverage Directed Generation of Test Programs,” *Proceedings of 40th Design Automation Conference*, 2003, pp. 286-291.

[23] O. Guzey, L. Wang, “Coverage-directed test generation through automatic constraint extraction,” *High Level Design Validation and Test Workshop, 2007.* HVDT 2007. *IEEE International*, pp. 151-158, Nov. 2007.

[24] H. Hsuch, K. Eder, “Test Directive Generation for Functional Coverage Closure Using Inductive Logic Programming,” *High-Level Design Validation and Test Workshop, 2006. Eleventh Annual IEEE International*, pp. 11-18, Nov. 2006.