Coverage directed test generation for functional verification using Bayesian networks

Conference Paper · July 2003

DOI: 10.1145/775832.775907 · Source: IEEE Xplore

2 authors:

Shai Fine
Interdisciplinary Center (IDC) Herzliya
59 PUBLICATIONS **2,570** CITATIONS

SEE PROFILE

Avi Ziv
IBM
92 PUBLICATIONS **1,396** CITATIONS

SEE PROFILE

Coverage Directed Test Generation for Functional Verification using Bayesian Networks

Shai Fine Avi Ziv
IBM Research Laboratory in Haifa
Haifa, 31905, Israel
{fshai, aziv}@il.ibm.com

ABSTRACT

Functional verification is widely acknowledged as the bottleneck in the hardware design cycle. This paper addresses one of the main challenges of simulation based verification (or dynamic verification), by providing a new approach for *Coverage Directed Test Generation* (CDG). This approach is based on Bayesian networks and computer learning techniques. It provides an efficient way for closing a feedback loop from the coverage domain back to a generator that produces new stimuli to the tested design. In this paper, we show how to apply Bayesian networks to the CDG problem. Applying Bayesian networks to the CDG framework has been tested in several experiments, exhibiting encouraging results and indicating that the suggested approach can be used to achieve CDG goals.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Verification*

General Terms

Verification, Measurement, Algorithms, Experimentation

Keywords

Functional Verification, Coverage Analysis, Bayesian Networks

1. INTRODUCTION

Functional verification is widely acknowledged as the bottleneck in the hardware design cycle [1]. To date, up to 70% of the design development time and resources are spent on functional verification. The increasing complexity of hardware designs raises the need for the development of new techniques and methodologies that can provide the verification team with the means to achieve its goals quickly and with limited resources.

The current practice for functional verification of complex designs starts with a definition of a test plan, comprised of a large set of events that the verification team would like to observe during the verification process. The test plan is usually implemented using random test generators that produce a large number of test-cases, and coverage tools that detect the occurrence of events in the test plan, and provide information related to the progress of the test plan. Analysis of the coverage reports allows the verification team to modify the directives for the test generators and to better hit areas or specific tasks in the design that are not covered well [5].

The analysis of coverage reports, and their translation to a set of test generator directives to guide and enhance the implementation of the test plan, result in major manual bottlenecks in the otherwise highly automated verification process. Considerable effort is invested in finding ways to close the loop of coverage analysis and test generation. *Coverage directed test generation* (CDG) is a technique to automate the feedback from coverage analysis to test generation. The main goals of CDG are to improve the coverage progress rate, to help reaching uncovered tasks, and to provide many different ways to reach a given coverage task. Achieving these goals should increase the efficiency and quality of the verification process and reduce the time and effort needed to implement a test plan.

In this paper, we propose a new approach for coverage directed test generation. Our approach is to cast CDG in a statistical inference framework, and apply computer learning techniques to achieve the CDG goals. Specifically, our approach is based on modeling the relationship between the coverage information and the directives to the test generator using *Bayesian networks* [9]. A Bayesian network is a directed graph whose nodes are random variables and whose edges represent direct dependency between their sink and source nodes. Each node in the Bayesian network is associated with a set of parameters specifying its conditional probability given the state of its parents.

Simply stated, the CDG process is performed in two main steps. In the first step, a training set is used to learn the parameters of a Bayesian network that models the relationship between the coverage information and the test directives. In the second step, the Bayesian network is used to provide the most probable directives that would lead to a given coverage task (or set of tasks).

Bayesian networks are well suited to the kind of modeling required for CDG, because they offer a natural and compact representation of the rather complex relationship between the CDG ingredients, together with the ability to encode essential domain knowledge. Moreover, adaptive tuning of the Bayesian network parameters provides a mean to focus on the rare coverage cases.

We describe two experiments in which we tested the the ability of Bayesian networks to handle aspects of the CDG problem in various settings. The goals of the experiments were to increase the hitting rates in hard-to-reach coverage cases; design directives aimed at reaching uncovered tasks; and provide many different directives for a given coverage task. We used two settings for our experiments. In the first setting, we used a Bayesian network to generate instruction streams to an abstract model of the pipeline of

DAC 2003, June 2–6, 2003, Anaheim, California, USA.

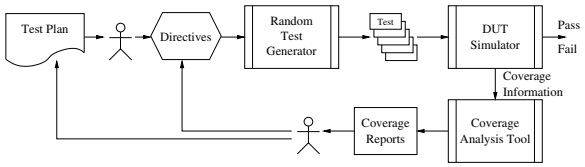

Figure 1: Verification process with automatic test generation

an advanced super-scalar PowerPC processor. In the second setting, we used a Bayesian network to generate directives to an existing test generator of a storage control unit of a mainframe with a goal to cover all possible transactions from the CPUs connected to this unit. In both experiments we reached our goals. The encouraging results suggests that Bayesian networks may well be used to achieve the primary goals of CDG.

The remainder of this paper is as follows. In Section 2, we briefly present the CDG framework and review related work. In Section 3, we describe Bayesian networks and their application to CDG. Sections 4 and 5 provide detailed descriptions of the experiments. We conclude with a few remarks and suggestions for future study.

2. COVERAGE DIRECTED TEST GENERATION (CDG)

In current industry practice, verification by simulation, or dynamic verification, is the leading technique for functional verification. Coverage is used to ensure that the verification of the design is thorough, and the definition of coverage events or *testing requirements* is a major part in the definition of the verification plan of the design. Often, a family of coverage events that share common properties are grouped together to form a *coverage model* [7]. Members of the coverage model are called *coverage tasks* and are considered part of the test plan. *Cross-product* coverage models [7] are of special interest. These models are defined by a basic event and a set of parameters or attributes, where the list of coverage tasks comprises all possible combinations of values for the attributes.

Figure 1 illustrates the verification process with an automatic random test generation. A test plan is translated by the verification team to a set of directives for the random test generator. Based on these directives and embedded domain knowledge, the test generator produces many test-cases. The *design under test* (DUT) is then simulated using the generated test-cases, and its behavior is monitored to make sure that it meets its specification. In addition, coverage tools are used to detect the occurrence of coverage tasks during simulation. Analysis of the reports provided by the coverage tools allows the verification team to modify the directives to the test generator to overcome weaknesses in the implementation of the test plan. This process is repeated until the exit criteria in the test plan are met.

The use of automatic test generators can dramatically reduce the amount of manual labor required to implement the test plan. Even so, the manual work needed for analyzing the coverage reports and translating them to directives for the test generator, can constitute a bottleneck in the verification process. Therefore, considerable effort is spent on finding ways to automate this procedure, and close the loop of coverage analysis and test generation. This automated feedback from coverage analysis to test generation, known as *Coverage Directed test Generation* (CDG), can reduce the manual work in the verification process and increase its efficiency.

In general, the goal of CDG is to automatically provide directives that are based on coverage analysis to the test generator. This can be further divided into two sub-goals: First, to provide directives to the test generator that help in reaching hard cases, namely uncovered or rarely covered tasks. Achieving this sub-goal can shorten the time needed to fulfill the test plan and reduce the number of manually written directives. Second, to provide directives that allow easier reach for any coverage task, using a different set of directives when possible. Achieving this sub-goal makes the verification process more robust, because it increases the number of times a task has been covered during verification. Moreover, if a coverage task is reached via different directions, the chances to discover hidden bugs related to this task are increased [8].

In the past, two general approaches for CDG have been proposed: feedback-based CDG and CDG by construction. Feedback-based CDG relies on feedback from the coverage analysis to automatically modify the directives to the test generator. For example, in [2], a genetic algorithm is used to select and modify test-cases to increase coverage. In [13], coverage analysis data is used to modify the parameters of a Markov Chain that represents the DUT. The Markov Chain is then used to generate test-cases for the design. In [11], the coverage analysis results trigger a set of generation rules that modify the testing directives. In contrast, in CDG by construction, an external model of the DUT is used to generate test directives designed to accurately hit the coverage tasks. For example, in [14] an FSM model of pipelines is used to generate tests that cover instruction interdependencies in the pipes.

3. COVERAGE DIRECTED TEST GENERATION USING BAYESIAN NETWORKS

The random nature of automatic test-case generators imposes a considerable amount of uncertainty in the relationship between test directives and coverage tasks, e.g., the same set of directives can be used to generate many different test-cases, each leading to different coverage tasks. This inherent uncertainty suggests to cast the CDG setup in a statistical inference framework. To this end, Bayesian networks offer an efficient modeling scheme by providing a compact representation of the complex (possibly stochastic) relationships among the CDG ingredients, together with the possibility to encode essential domain knowledge. It should be noted that we do not suggest modeling the behavior of the design, typically a large and complicated (deterministic) finite state machine. Rather, we model the CDG process itself, namely the trial-and-error procedure governed by the verification team, which controls the test generation at one end and traces the progress of covering the test plan at the other.

3.1 A Brief Introduction to Bayesian Networks

A Bayesian network is a graphical representation of the joint probability distribution for a set of variables. This representation was originally designed to encode the uncertain knowledge of an expert and can be dated back to the geneticist Sewall Wright [15]. Their initial development in the late 1970s was motivated by the need to model the top-down (semantic) and bottom-up (perceptual) combinations of evidence (observations/findings). Their capability for bidirectional inferences, combined with a rigorous probabilistic foundation, led to the rapid emergence of Bayesian networks as the method of choice for uncertain reasoning in AI and expert systems, replacing ad hoc rule-based schemes. Bayesian networks also play a crucial role in diagnosis and decision support systems [10].

Obviously, there's a computational problem in dealing with many sources of uncertainty, i.e. the ability to perform probabilistic manipulations in high dimensions (the "curse of dimensionality"). The main breakthrough emerged in the late 1980s and can be attributed to Judea Pearl [12], who introduced 'modularity', thus enabling

large and complex models and theirs associated calculations, to be split up into small manageable pieces. The best way to do this is via the imposition of meaningfully simplified conditional independence assumptions. These, in turn, can be expressed by means of a powerful and appealing graphical representation.

A Bayesian network consists of two components. The first is a directed acyclic graph in which each vertex corresponds to a random variable. This graph represents a set of conditional independence properties of the represented distribution: each variable is probabilistically independent of its non-descendants in the graph given the state of its parents. The graph captures the qualitative structure of the probability distribution, and is exploited for efficient inference and decision making. The second component is a collection of local interaction models that describe the conditional probability $p(X_i|Pa_i)$ of each variable X_i given its parents Pa_i. Together, these two components represent a unique joint probability distribution over the complete set of variables X [12]. The joint probability distribution is given by the following equation:

$$p(X) = \prod_{i=1}^{n} p(X_i|Pa_i) \quad (1)$$

It can be shown that this equation actually implies the conditional independence semantics of the graphical structure given earlier. Eq. 1 shows that the joint distribution specified by a Bayesian network has a factored representation as the product of individual local interaction models. Thus, while Bayesian networks can represent arbitrary probability distributions, they provide a computational advantage for those distributions that can be represented with a simple structure.

The characterization given by Eq. 1 is a purely formal characterization in terms of probabilities and conditional independence. An informal connection can be made between this characterization and the intuitive notion of direct causal influence. It has been noted that if the edges in the network structure correspond to causal relationships, where a variable's parents represent the direct causal influences on that variable, then resulting networks are often very concise and accurate descriptions of the domain. Thus it appears that in many practical situations, a Bayesian network provides a natural way to encode causal information. Nonetheless, it is often difficult and time consuming to construct Bayesian networks from expert knowledge alone, particularly because of the need to provide numerical parameters. This observation, together with the fact that data is becoming increasingly available and cheaper to acquire, has led to a growing interest in using data to learn both the structure and probabilities of a Bayesian network (cf. [3, 9, 12]).

Typical types of queries that can be efficiently answered by the Bayesian network model are derived from applying the Bayes rule to yield posterior probabilities for the values of a node (or set of nodes), X, given some evidence, E, i.e. assignment of specific values to other nodes:

$$p(X|E) = \frac{p(E|X) * p(X)}{p(E)}$$

Thus, a statistical inference can be made in the form of either selecting the *Maximal A Posteriori* (MAP) probability, $\max p(X|E)$, or obtaining the *Most Probable Explanation* (MPE), $\arg\max p(X|E)$.

The sophisticated yet efficient methods that have been developed for using Bayesian networks provide the means for predictive and diagnostic inference[1]. A *diagnostic* query is such that the evidence

[1]This is in contrast to standard regression and classification methods (e.g., feed forward neural networks and decision trees) that encode only the probability distribution of a target variable given several input variables.

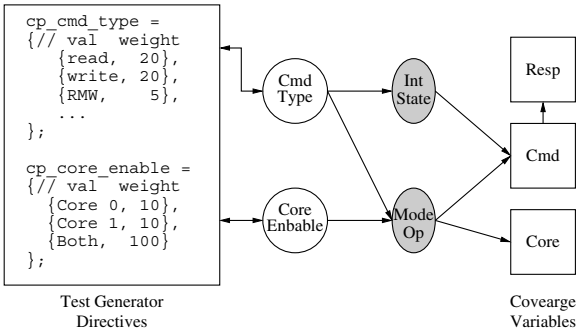

Figure 2: Bayesian Network of CDG

nodes E represent a cause, while the queried nodes, X, represent an effect. The reversed direction, i.e. evidence on the effect nodes which serves to determine the possible cause, is called *abductive*. These methods also allow Bayesian networks to reason efficiently with missing values, by computing the marginal probability of the query given the observed values.

There are two important extensions of Bayesian networks: Dynamic Bayesian networks and influence diagrams. The first extension (see [6]) enables the incorporation of time, thus modeling temporal dependencies in a stochastic process. The second extension (see [3]) enriches the Bayesian network paradigm with decision making and utility considerations which create a powerful mechanism for dealing with decisions under uncertainty constraints.

3.2 A Bayesian Network for CDG

The CDG process begins with the construction of a Bayesian network model that describes the relations between the test directives and the coverage space. Figure 2 illustrates a simple, yet typical, Bayesian network, which models a small excerpt of the CDG setup. The network describes the relationship between the directives that influence the type of command that is generated (`cp_cmd_type`) and the active cores inside a CPU (`cp_core_enable`), and the coverage attributes of a generated command (`cmd`), its response (`resp`), and the core that generated it (`core`). The network is comprised of input nodes (the white circles on the left) that relate to test directives that appear to their left and coverage nodes (the white squares on the right) that define the coverage space. In addition to these nodes, for which we have physical observations, the network may also contain *hidden* nodes, namely variables for which we don't have any physical evidence (observations) for their interactions. These variables are represented as shaded ovals in the figure. Hidden nodes are added to the Bayesian network structure primarily to reflect expert domain knowledge regarding hidden causes and functionalities which impose some structure on the interaction between the interface (observed) nodes[2].

The Bayesian network at Fig. 2 describes the causal relationships from the test generation directives (causes) to the coverage model space (effects). For example, it encodes the expert knowledge that indicates that there is an internal mode of operation for which we do not have any direct physical observation, yet it is determined by the combined values of the test generation attributes. On the other hand, the (hidden) mode of operation directly influences the choice of the resulting command and core, which are attributes of

[2]Introducing hidden nodes to the network structure has the secondary impact of reducing the computational complexity by dimensionality reduction, and as a means for capturing non-trivial (higher order) correlations between observed events.

the coverage model. Note the absence of a direct link between the requested core (via the directive `cp_core_enable`) and the observed one (at `Core`), which captures our understanding that there is no direct influence between the directives and the coverage attribute. Another assumption encoded in the CDG Bayesian network structure at Fig. 2, is that the only information that governs the response for the command is the generated command itself, and this is encoded via the direct link from `Cmd` to `Resp`.

In a nutshell, the design of the Bayesian network starts with identifying the ingredients (attributes) that will constitute the directives to the test generator on one hand, and to the coverage model on the other. These attributes are dictated by the interface to the simulation environment, to the coverage analysis tool, and by the specification of the coverage model in the test plan. These ingredients are used as the first guess about the nodes in the graph structure. Connecting these nodes with edges is our technique for expert knowledge encoding, as demonstrated in Fig. 2. Obviously, using a fully connected graph, i.e. with an edge between every pair of nodes, represents *absolutely no knowledge* about the possible dependencies and functionalities within the model. Hence, as the graph structure becomes sparser, it represents *deeper domain knowledge*. We discovered that a good practice in specifying a dependency graph is to remove edges for which we have strong belief that the detached nodes are not directly influencing one another. At this point, hidden nodes can be added to the structure, either to represent hidden causes, which contribute to a better description of the functionalities of the model, or to take on a role from the complexity stand point, by breaking the barges cliques in the graph (see [4]).

After the Bayesian network structure is specified, it is trained using a sample of directives and the respective coverage tasks. To this end, we activate the simulation environment and construct a training set out of the directives used and the resulting coverage tasks. We then use one of the many known learning algorithms (cf. [3]) to estimate the Bayesian network's parameters (i.e. the set of conditional probability distributions). This completes the design and training of the Bayesian network model.

In the evaluation phase, the trained Bayesian network can be used to determine directives for a desired coverage task, via posterior probabilities, MAP and MPE queries, which use the coverage task attributes as evidence. For example, in a model for which the directives are weights of possible outcomes for internal draws in the test generator (e.g. the directive `cp_cmd_type` in Fig. 2 specifies a preference to `read` commands, `write` commands, etc.), we can specify a desired coverage task assignment (evidence) for the coverage nodes (e.g. `Resp = ACK`) and calculate the posterior probability distribution for directive nodes (e.g. p(Cmd Type|Resp = ACK)), which directly translates to the set of weights to be written in the test generator's parameter file. Note, as the example demonstrates, we can specify partial evidence and/or determine a partial set of directives.

4. INSTRUCTION STREAM GENERATION USING A DYNAMIC NETWORK

To evaluate the feasibility of the suggested modeling approach to the CDG problem, we designed a controlled study that acts in a simple domain (small state space), where we have a deep understanding of the DUT's logic, direct control on the input, and a 'ground truth' reference to evaluate performance.

We conducted the experiment on a model of the pipeline of NorthStar, an advanced PowerPC processor. The pipeline of NorthStar contains four execution units and a dispatch unit that dispatches instructions to the execution units. Figure 3 illustrates the general

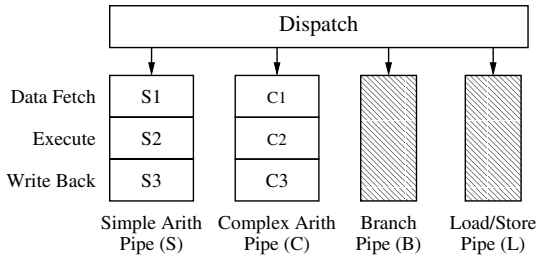

Figure 3: The structure of the NorthStar pipeline

structure of the NorthStar pipeline. For reasons of simplicity, our model contains only the simple arithmetic unit that executes simple arithmetic instructions such as add, and the complex arithmetic unit that can execute both simple and complex arithmetic instructions. Each execution unit consists of three pipeline stages: (1) Data fetch stage, in which the data of the instruction is fetched; (2) Execute stage, in which the instruction is executed; (3) Write back stage, where the result is written back to the target register. The flow of instructions in the pipeline is governed by a simple set of rules. For example, in-order dispatching of instructions to the execution units, and rules for stalling because of data dependency. Note, the complete set of rules is omitted to simplify the description.

We developed a simple abstract model of the dispatch unit and two pipelines and used it to simulate the behavior of the pipeline. The input to our NorthStar model is a simplified subset of the PowerPC instruction set. Each instruction is modeled by four input variables. The first variable indicates the type of the instruction. There are five possible types: S - simple arithmetic; C1, C2, C3 - complex arithmetic; and NOP - instructions that are executed in other execution units. The second and third input variables constitute the source and target register of the instructions. For simplicity and in order to increase the possibility of register interdependency, we used only eight registers instead of the 32 registers available in PowerPC. The last input variable indicates whether the instruction uses the condition register. Due to restrictions on the legal combinations of the input variables (e.g., NOP instruction is not using registers), there are 449 possible instructions.

We used a coverage model that examines the state of the two pipelines, and properties of the instructions in them. The coverage model consists of five attributes, the type of instruction at stage 1 of the simple and complex arithmetic pipelines (S1Type and C1Type, resp.), flags indicating whether stage 2 of the pipelines are occupied (S2Valid and C2Valid, resp.), and a flag indicating whether the instruction at stage 2 of the simple arithmetic pipeline uses the condition register (S2CR). The total number of legal coverage tasks in the model is 54 (out of 80 possible cases).

The goal of the experiment was to generate instruction streams that cover the coverage model described above. Specifically, we concentrated on the ability to reach the desired coverage cases with many, yet relatively short, instruction sequences.

We modeled the temporal dependencies between the instructions and coverage tasks and among the instructions using a two-slice *Dynamic Bayesian Network* (DBN) [6]. Rather than an accurate mapping of the specific state machine structure, the DBN encoded the general knowledge of an expert on the modus operandi of this type of DUT. Using an expert's domain knowledge proved to be vital in this setup because it provided essential information needed for the generation of instruction streams. Moreover, it enabled the use of hidden nodes, which effectively reduced the complexity through dimensionality reduction. The resulting DBN has 19

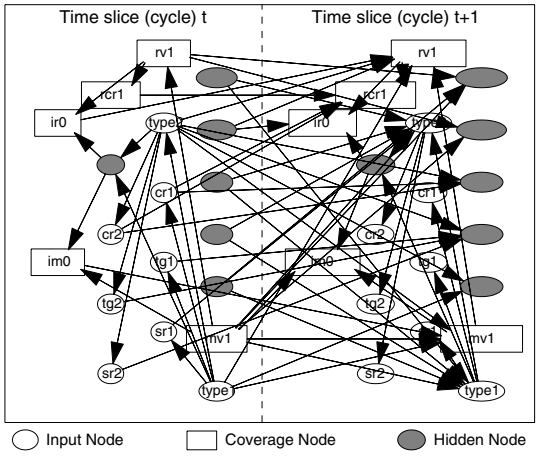

Figure 4: two-slice DBN for the NorthStar experiment

	Rare		Uncovered	
	Instructions	Cycles	Instructions	Cycles
Training Set	6	7	-	-
DBN	4	5	4	5
Text Book	3	4	3	4

Table 1: NorthStar experiment results

nodes per slice, 13 of which are observed, 15 intra (within a slice) edges, and 37 inter (between slices) edges (see Fig 4).

The training set is composed of 1000 sequences of random instructions. The length of each sequence is 10 cycles. Note, the model the we used for the Bayesian network made it easier to measure length in terms of cycles instead of instructions. The training set contained 385 different instructions. During its simulation, 49 (out of 54) coverage cases were observed. The average number of instructions per sequence in the training set was 9.7 out of the 20 possible dispatches in 10 cycles (i.e., more than half of the dispatch slots in the sequence are empty).

After training the Bayesian network, we tried to generate instruction sequences for all 54 coverage tasks in the coverage model. Each sequence was generated using the DBN, by solving the *Most Probable Explanation* (MPE) problem for the requested coverage task. All 49 coverage cases of the training set plus three additional uncovered cases were reached using instruction sequences designed by the DBN. In addition, we generated many different instruction sequences for each coverage task that was covered by the Bayesian network. The average number of cycles in a generated sequence dropped to 2.9, while the average number of instructions in a sequence reduced to 3.7. This reflects the fact that the generated instruction sequences cause less stall states en-route to reaching the desired coverage cases. Table 1 illustrates the details of reaching two difficult coverage cases—the rarest coverage task, which was seen only once in the training set, and an uncovered task. The table shows the number of cycles and instructions required to reach these tasks in the training set, the instruction sequences generated by the trained DBN, and the 'text book' solution—the best possible sequence. The table indicates that the instruction sequences generated by the DBN are shorter, both in instructions and cycles, than the sequences in the training set. Overall, the results indicate that the trained DBN is able to generate many compact instruction sequences that are not far from the best possible solution.

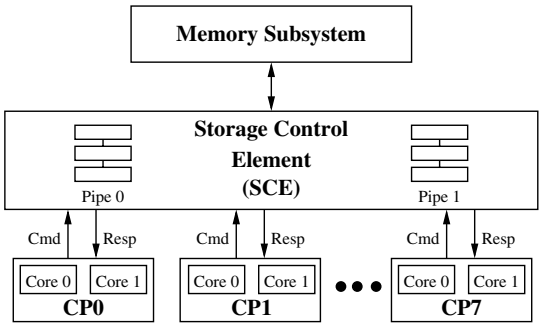

Figure 5: The structure of SCE simulation environment

5. STORAGE CONTROL EXPERIMENT USING A STATIC NETWORK

The second experiment was conducted in a real-life setting. The design under test in the experiment is the *Storage Control Element* (SCE) of an IBM z-series system. Figure 5 shows the structure of the SCE and its simulation environment. The SCE handles commands from eight CPUs (CP0 – CP7). Each CPU consists of two cores that generate commands to the SCE independently. The SCE handles incoming commands using two internal pipelines. When the SCE finishes handling a command, it sends a response to the commanding CPU.

The simulation environment for the SCE contains, in addition to the SCE itself, behavioral models for the eight CPUs that it services, and a behavioral model for the memory subsystem. The behavioral models of the CPUs generate commands to the SCE based on their internal state and a directive file provided by the user. The directive file contains a set of parameters that affect the behavior of the system. Some of these parameters control the entire system while others are specific to certain components of the system, such as a specific CPU. Figure 2 shows an example of some parameters that are used in the simulation environment of the SCE. Each parameter contains a set of possible values that the parameter can receive. Each value has a weight associated with it. When the value of a parameter is needed, it is randomly chosen from the set of possible values according the weights of these values. For example, when a CPU generates a new command, it first uses the `cp_cmd_type` parameter to determine the type of command to generate, and then a specific parameter for that command type to determine the exact command to be used.

In the experiment, we tried to cover all the possible transactions between the CPUs and the SCE. The coverage model contained five attributes: The CPU (8 possible values) and the core (2 values) in it that initiated the command, the command itself (31 values), its response (14 values), and the pipeline in the SCE that handled it (2 values). Overall, the cross product contains 13,888 cases and the coverage model contains 1968 legal coverage tasks.

This experiment added many new challenges over the controlled experiment described in the previous section. First, our knowledge about the DUT in this experiment was very limited compared to the full understanding of the design in the first experiment. In addition, we were less able to observe and control the input and output nodes of the Bayesian network. For the test parameters, we could only specify the distribution of each parameter and we could not observe the values that were actually used, only their distribution. Moreover, in some cases the behavioral models ignored the parameters and generated commands based on their internal state. Thus, the actual distribution used was not exactly the provided distribu-

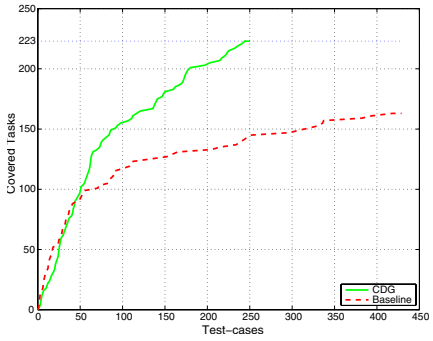

Figure 6: Coverage progress of the CDG process

tion of the parameters. This type of observation (distribution instead of specific value) is known as a *soft evidence*. The coverage data that we got out of the simulation environment was a summary of all the coverage tasks that occurred during the simulation of a test-case. Therefore, it was hard to correlate between the observed coverage tasks and the parameters' values that caused them and between the different observed coverage tasks.

Because we had limited knowledge about the DUT and the correlation between the parameters in the test directives and the coverage tasks, the first Bayesian network we constructed contained arcs between each of the coverage variables and each of the test parameters. We trained this network with 160 test-cases (each taking more than 30 minutes to execute). After the initial training, we analyzed the Bayesian network and found out that most of the test parameters were strongly correlated either to the command and response coverage variables or the pipe and core variables, but only a single variable was strongly correlated to all coverage variables. Therefore, we partitioned the Bayesian network into two networks, one for command and response and the other for core and pipe. The result of the inference on the common parameter from the first network was used as input for the second one. We trained the second network with the same training set of 160 test-cases. During the training, 1745 out of the 1968 tasks in the model were covered, while 223 remained uncovered.

We checked the performance of the trained network and its ability to increase the coverage rate for the uncovered tasks in the training set. The baseline for comparison was the progress achieved by the best test directive file created by an expert user.

We tried to maximize the coverage progress rate using a large number of test directive files aimed at specific sets of uncovered tasks. This approach is not realistic for a human user due the effort needed to create each set of directives. However, it is useful for the automatic creation of directives, because the inference time from the trained network is negligible. Our method to maximize the coverage progress rate was to randomly partition the uncovered tasks, use the trained network to create a test directive file for each partition, and simulate a single test-case for each directive file. This process was repeated until all the tasks were covered. The CDG process was able to cover all uncovered tasks after 250 test-cases, while the baseline case of the user defined test directives file covered only two thirds of them after over 400 test-cases (see Figure 6).

6. CONCLUSIONS AND FUTURE WORK

In this paper we demonstrated how Bayesian networks can be used to close the loop between coverage data and directives to test generators. The experiments described in the paper show that this modeling technique can be efficiently used to achieve the CDG goals of easier reach for hard coverage cases, diverse reach for average cases, and improved coverage progress rate. It should be noted that the suggested CDG method is not limited to the types of simulation environments handled in this paper (i.e., parameters-based test generation and direct stimuli generation). It can be used in other types of environments, such as test generators in which the control on the stimuli is embedded in the generator itself.

Our future work has two distinct aspects: enhancing the learning capabilities and effectively applying the suggested framework to the verification process. From the learning perspective, we plan to explore other techniques that may increase our capabilities. For example, incremental structure learning as a means for encoding richer domain knowledge, and the efficient construction of good queries to boost targeting rare cases using selective sampling. To effectively deploy the CDG framework, we need to gain a better understanding of the type of knowledge that should be encoded in the model, and to identify in which areas the suggested approach may prove most beneficial to the verification process.

7. REFERENCES

[1] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, January 2000.

[2] M. Bose, J. Shin, E. M. Rudnick, T. Dukes, and M. Abadir. A genetic approach to automatic bias generation for biased random instruction generation. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 442–448, May 2001.

[3] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer-Verlag, 1999.

[4] G. Elidan, N. Lotner, N. Friedman, and D. Koller. Discovering hidden variables: A structure-based approach. In *Proceedings of the 13th Annual Conference on Neural Information Processing Systems*, pages 479–485, 2000.

[5] L. Fournier, Y. Arbetman, and M. Levinger. Functional verification methodology for microprocessors using the Genesys test-program generator. In *Proceedings of the 1999 Design, Automation and Test in Europe Conference (DATE)*, pages 434–441, March 1999.

[6] Z. Ghahramani. Learning dynamic Bayesian networks. In *Adaptive Processing of Sequences and Data Structures*, Lecture Notes in Artificial Intelligence, pages 168–197. Springer-Verlag, 1998.

[7] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. User defined coverage - a tool supported methodology for design verification. In *Proceedings of the 35th Design Automation Conference*, pages 158–165, June 1998.

[8] A. Hartman, S. Ur, and A. Ziv. Short vs long size does make a difference. In *Proceedings of the High-Level Design Validation and Test Workshop*, pages 23–28, November 1999.

[9] D. Heckerman. A tutorial on learning with Bayesian networks. Technical report, Microsoft Research, 1996.

[10] D. Heckerman, A. Mamdani, and M. Wellman. Real-world applications of Bayesian networks. *Communications of the ACM*, 38(3):24–30, 1995.

[11] G. Nativ, S. Mittermaier, S. Ur, and A. Ziv. Cost evaluation of coverage directed test generation for the IBM mainframe. In *Proceedings of the 2001 International Test Conference*, pages 793–802, October 2001.

[12] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Network of Plausible Inference*. Morgan Kaufmann, 1988.

[13] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer. A functional validation technique: biased-random simulation guided by observability-based coverage. In *Proceedings of the International Conference on Computer Design*, pages 82–88, September 2001.

[14] S. Ur and Y. Yadin. Micro-architecture coverage directed generation of test programs. In *Proceedings of the 36th Design Automation Conference*, pages 175–180, June 1999.

[15] S. Wright. Correlation and causation. *Journal of Agricultural Research*, 1921.