Deriving a Simulation Input Generator and a Coverage Metric From a Formal Specification

Kanna Shimizu
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, USA
kanna.shimizu@stanfordalumni.org

David L. Dill
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, USA
dill@cs.stanford.edu

ABSTRACT

This paper presents novel uses of functional interface specifications for verifying RTL designs. We demonstrate how a simulation environment, a correctness checker, and a functional coverage metric are all created automatically from a single specification. Additionally, the process exploits the structure of a specification written with simple style rules. The methodology was used to verify a large-scale I/O design from the Stanford FLASH project.

Categories and Subject Descriptors

B.5.2 [**RTL Implementation**]: Design Aids

General Terms

Documentation, Performance, Design, Verification

Keywords

testbench, input generation, BDD minimization, coverage

1. INTRODUCTION

1.1 Motivation

Before a verification engineer can start simulating RTL designs, he must write three *verification aids*: input testbenches to stimulate the design, properties to verify the behavior, and a functional coverage metric to quantify simulation progress. It would be much easier if the three can be automatically derived from the interface protocol specification. Not only will this save a great deal of work, but it will also result in fewer bugs in the test inputs and the checking properties because they are mechanically derived from a verified specification. Motivated by these advantages, we developed a methodology where the three aids are automatically generated from a specification. Furthermore, we demonstrate how a specification structured by certain style rules allows for more memory efficient simulation runs. The primary contributions of this paper are:

- *a presentation of how a bus protocol specification can be used for simulation to automatically*
  - *generate the most general set of legal input sequences to a design*
  - *produce properties to check interface protocol conformance.*
  - *define a new functional coverage metric*
  - *bias inputs to increase the probability of hitting uncovered cases*
- *a new input generation scheme that greatly reduces the size of the BDD (binary decision diagram) [1] used*
  - *by dynamically selecting relevant constraints on a cycle-by-cycle basis and not translating the entire specification into a BDD*
  - *and by extracting out the pure "constraining" logic and discarding the conditional logic ("the guard") when building the BDD*
- *a report on the successful application of the methodology to verify a fabricated working design, and a description of the new bugs found*

1.2 Background

Many of today's digital circuit designs depend on the tight integration of multiple design components. These components are often designed by different engineers who may have divergent interpretations of the interface protocol that "glue" the components together. Consequently, designs may be incompatible and behave incorrectly when combined. Thus, functional interface specifications are pivotal for successful module integration and should be accordingly solid and precise. However, specifications widely in use today are still written informally, forfeiting an opportunity for automated analysis and logical clarity. In many cases, specifications such as those of standard bus protocols are buggy, ambiguous, and contradictory: all problems that can be resolved by formal specification development.

The advantages of formal specifications seem clear but they are nonetheless avoided due to a perceived cost-value problem; they are often considered too costly for the benefits they promise. Specifically, they are criticized for their lengthy development time and the need for formal verification training. For many, the value of a correct specification does not justify these costs, and precious resources are allocated for more pressing design needs. Thus, to counter these disincentives, we are developing a formal specification methodology that attacks this cost-value problem from two angles.

Cost Side The earlier papers [9, 8] for this project focus on the first half of the problem: minimizing the cost by making the specification process easier. We have directed our attention to signal-level

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

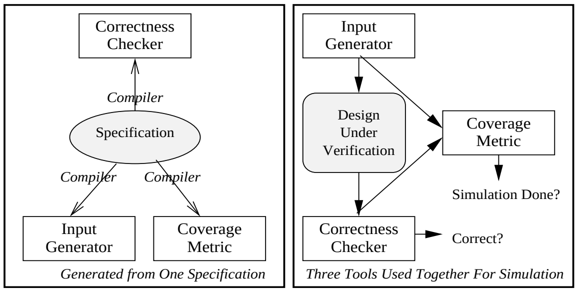

Figure 1: The Trio of Verification Aids

bus protocol descriptions for this aim since they are both important and challenging to specify. Using the PCI (Peripheral Component Interconnect) bus protocol and the Intel® Itanium™ Processor bus protocol as examples, we developed a specification style that produces correct, readable, and complete specifications with less effort than free-form, *ad hoc* methods. The syntactic structuring used in the methodology is language-independent and can be applied to many specification languages from SMV [6] to Verilog. This is a reflection of our belief that methodology, as opposed to tool or language development, is the key to achieving the stated goal.

Value Side As the latest work in this series, this paper focuses on the second angle of the cost-value problem: to increase the value of a formal specification beyond its role as a documentation. It is based on the idea that once a correct, well-structured specification is developed, it can be exploited in a way that a haphazard and incomplete specification cannot be. In particular, we investigate ways to use the specification *directly* to generate inputs, check behavior, and monitor coverage. Because our goal is to verify large designs common in industry, the methodology is specifically tailored for simulation-based verification.

1.3 The Problem and Our Approach

Given an HDL (hardware description language) component design to verify, an engineer needs various additional machinery (Figure 1, right).

1. Input Logic There must be logic to drive the inputs of the design. One method uses random sequences, which are not guaranteed to comply with the protocol, and consequently, it is difficult to gauge correctness of the design because its inputs may be incorrect. A more focused method is directed testing where input sequences are manually written, but they are time-consuming to write and difficult to get correct.

2. Output Check Logic to determine the correctness of the component's behavior is needed if manual scrutiny is too cumbersome. With the methodology presented here, the scope of correctness checking is limited to interface protocol conformance. It cannot check higher-level properties such as whether the output data from one port correctly corresponds to another port's input data.

3. Coverage Metric Because complete coverage with all possible input sequences tested is not possible, there must be some metric that quantifies the progress of verification coverage. The verification engineer would like to know whether the functionalities of the design have been thoroughly exercised and all interesting cases have been reached. This is an open area of research, and currently, most practitioners resort to methods with little theoretical backing.

Our Approach At the foundation of our methodology is a *unified framework approach* where the three tools are generated from a single source specification (Figure 1). This is possible because all three are fundamentally based on the interface protocol, and consequently, the interface specification can be used to automatically create the three. In current practice, the verification aids are each written from scratch; this requires a tremendous amount of time and effort to write and debug. By eliminating this step, our methodology enhances productivity and shortens development time.

Also, a thoroughly debugged, solid specification invariably leads to correct input sequences, checking properties, and coverage metrics. The correctness of the core document guarantees the correctness of the derivatives. In contrast, with current methods, each verification aid needs to be individually debugged. The advantages of this are most pronounced for standard interfaces where the correctness effort can be concentrated in the standards committee and not duplicated among the many implementors. Furthermore, when a change is made to the protocol (a frequent occurrence in industry), one change in the protocol specification is sufficient to reflect this because the verification aids can be regenerated from the revised document. Otherwise, the engineer would have to determine manually the effect of the change for each tool.

The derivation of a behavioral checker is the most straightforward of the three. The checker is on-the-fly; during simulations, it flags an error as soon as the component violates the protocol. As addressed in the first paper[9], the specification is written in a form very close to a checker. Furthermore, the specification is guaranteed to be executable by the style rules (described in section 2.1), and so the translation from it to a HDL checker requires minimal changes [9].

The bulk of the current work addresses the issue of automatically generating input sequences. Our method produces an input generator which is dynamic and reactive; the generated inputs depend on the previous cycle outputs of the design under verification. In addition, these inputs always obey the protocol, and the generation is a one-pass process. The mechanism relies on solving boolean constraints by building and traversing BDD structures on every clock cycle. Although input generation using constraint solvers is not by itself novel, our approach is the first to use and exploit a complete and structured specification.

Finally, a new simulation coverage metric is introduced, and the automatic input biasing based on this metric is also described. Although more experiments are needed to validate this metric's effectiveness, its main advantage (currently) is that it is specification-based and saves time: extra work is not needed to write out a metric or to pinpoint the interesting scenarios for they are gleaned directly from the specification document.

Previous Works Clarke et al. also researched the problem of specifications and generators in [3] but the methodology is most closely related to the *SimGen* project described in the 1999 paper [11] by Yuan et al. As with the SimGen work, we are focusing on practical methods that can be used for existing complex designs. Within that framework, there are mainly two features that differentiate our approach from SimGen.

First, the SimGen software uses a statically-built BDD which represents the entire input constraint logic; in contrast, our framework dynamically builds the appropriate BDD constraint on every clock cycle. This results in a dramatically smaller BDD for two reasons. One, only a small percentage of the protocol logic is relevant on each cycle, and so the corresponding BDD is always much smaller than the static BDD representing the entire protocol. Two, the BDD *contains only the design's input variables and do not contain state variables or the design's outputs*. Thus, for the PCI example, instead of a BDD on 161 variables, we have a BDD on 15 variables, *an order of magnitude difference*. Consequently, our input generation uses exponentially less memory. This reduction is based on the observation that the sole role of the state variables and the design's outputs is to determine which parts of the protocol are relevant (and thus required in the BDD) for a particular cycle. Otherwise, these variables are not needed to calculate the inputs. We believe that these two reasons for smaller BDDs would hold for many interfaces and therefore allow input generation for a large interface that may otherwise be hindered by BDD blowup.

A second difference with SimGen is that, unlike our framework, it requires the users to provide the input biases. A unique contribution of our work is the automated process of determining biases. It is noted that all these advantages are possible because our method exploits the structure of a stylized specification whereas SimGen is applicable for more general specifications.

2. METHODOLOGY

2.1 Specification Style

The specification style was introduced in [9] and is summarized here for the reader. It is based on using multiple *constraints* to collectively define the signalling behavior at the interface. The constraints are short boolean formulas which follow certain syntactic rules. They are also independent of each other, rely on state variables for historic information, and when AND-ed together, define exactly the correct behavior. This is similar to using (linear or branching time) temporal logic for describing behavior. However, our methodology allows and requires only the most basic operators for writing the constraints, and it aims for a complete specification as opposed to an *ad hoc* list of properties that should hold true.

This decomposition of the protocol into multiple constraints has many advantages. For one, the specification is easier to maintain. Constraints can be added or removed and independently modified. It is also believed that it is easier to write and debug. Since most existing natural-language specifications are already written as a list of rules, the translation to this type of specification requires less effort and results in fewer opportunities for errors. For debugging, a symbolic model checker can be easily used to explore the states allowed by the constraints.

Style Rule 1 The first style rule requires the constraints to be written in the following form.

$$prev(signal_0 \ldots \wedge \neg signal_j \ldots \vee variable_0 \ldots \wedge \neg variable_k)$$
$$\rightarrow signal_i \vee \ldots \wedge \neg signal_n$$

where "→" is the logical symbol for "implies". The antecedent, the expression to the left of the "→", is a boolean expression containing the interface signal variables and auxiliary state variables, and the consequent, the expression to the right of the "→", contains *just the interface signal variables*. The allowed operators are AND, OR, and NEGATION. The *prev* construct allows the value of a signal (or the state of a state machine) a cycle before to be expressed. The constraints are written as an implication with the past expression as the antecedent and the current expression as the consequent. In essence, the past history, when it satisfies the antecedent expression, requires the current consequent expression to be true; otherwise, the constraint is not "activated" and the interface signals do not have to obey the consequent in the current cycle. In this way, the *activating* logic and the *constraining* logic are separated. For example, the PCI protocol constraint, $prev(trdy \wedge stop) \rightarrow stop$ means "if the signals $trdy$ and $stop$ were true in the previous cycle (the "activating" logic), then $stop$ must be true in the current cycle (the "constraining" logic)" where a "true" signal is asserted and a "false" signal is deasserted. This separation is what identifies the relevant (i.e."activated") constraints on a particular cycle. Also, it allows the BDD to be an expression purely of the "constraining" logic (as explained in the next section, *2.2.1*).

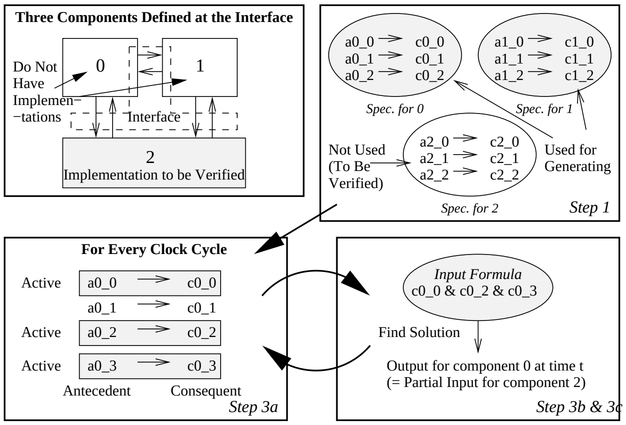

Figure 2: The Input Generation Algorithm

Style Rule 2 The second style rule, the *separability* rule, requires *each constraint to constrain only the behavior of one component.* Equivalently, because the constraining part is isolated from the activating part (due to the first style rule), the rule requires *the consequent to contain only outputs from one component.*

Style Rule 3 The third rule requires that the specification is dead state free. This rule effectively guarantees that an output satisfying all of the constraints always exists as long as the output sequence so far has not violated the constraints. There is a universal test that can verify this property for a specification. Using a model checker, the following CTL (computation tree logic)[2] property can be checked against the constraints, and any violations will pinpoint the dead state: $AG(all\ constraints\ have\ been\ true\ so\ far \rightarrow$
$EX(all\ constraints\ are\ true))$

Although abiding by the style rules may seem restrictive, it promises many benefits. Furthermore, the style is still powerful enough to specify the signal-level PCI and Intel® Itanium™ Processor bus protocols.

2.2 Deriving an Input Generator

2.2.1 Basic Algorithm

Based on the following algorithm, input vectors are generated from the structured specification (Figure 2).

1. Group the constraints according to which interface component they specify. (This is possible because of style rule 2, the separability rule.) If there are n interface components, there will be n groups.
2. Remove the group whose constraints are for the component under verification. These will not be needed. Now, there are $n-1$ groups of constraints.
3. For each group of constraints, do the following on every clock cycle of the simulation run. The goal is to choose an input assignment for the next cycle.
   (a) For each constraint, evaluate *just the antecedent half.* The antecedent values are determined by internal state variables and observed interface signal values. For antecedents which evaluate to *true*, the corresponding constraints are marked as *activated*.
   (b) Within each group, AND together *just the consequent halves* of the activated constraints to form the *input formula*. As a result, there is one input formula for each

interface component. The formulas have disjoint support (because of rule 2), which greatly reduces the complexity of finding a satisfying assignment.

(c) A boolean satisfiability solver is used to determine a solution to each of the input formulas. A BDD-based solver is used instead of a SAT-based one in order to control the biasing of the input variables. Since the specification is nondeterministic and allows a range of behaviors, there will most likely be multiple solutions. (In section 2.3, we discuss how a solution is chosen so that interesting simulation runs are generated.) The chosen solutions form the input vector for this cycle.

(d) Go back to step 3(a) on the next clock cycle.

The significance of the style rules become clear from this algrotihm. The "activating" – "constraining" division is key to allowing for a dramatically smaller expression (just the consequent halves) to solve (rule 1). The separability rule also allows for smaller expressions by enforcing strict orthogonality of the specification along the interface components (rule 2). Finally, the lack of dead states guarantees the existence of a correct input vector assignment for every clock cycle (rule 3).

2.2.2 *Implementation*

A compiler tool, which reads in a specification and outputs the corresponding input generation module, has been designed and implemented. There are two parts to the input generator: the Verilog module which acts as the frontend and the C module as the backend (Figure 3).

The Verilog module contains all the antecedents of the constraints, and based on its inputs (the component's outputs) and its internal state variables, determines which constraints are activated for that clock cycle. Then, the indices of the activated constraints are passed to the backend C module. The C module will return an input assignment that satisfies all the activated constraints, and the Verilog module will output this to the component under verification. The choice of Verilog as a frontend allows many designs to be used with this framework.

The C module contains the consequent halves of the constraints. It forms conjunctions (ANDs) of the activated consequents, solves the resulting formula, and returns an assignment to the Verilog module. It is initialized with an array of BDDs where each BDD corresponds to a constraint consequent. On every clock cycle, after the activation information is passed to it, it forms one BDD per interface component by performing repeated BDD AND operations on activated consequents in the same group. The resulting BDD represents an (aggregated) constraint on the next state inputs from one component, and by traversing the BDD until the "1(TRUE)" terminal node is reached, an assignment can be found. Once an assignment is determined for each interface component, the complete input assignment to the component under verification has been established. The CUDD (Colorado University Decision Diagram) package [10] version 2.3.1 was used for BDD representation and manipulation, and Verilog-XL was used to simulate the setup.

2.3 Biasing the Inputs

2.3.1 *Coverage Metric*

We use the specification to define *corner cases*, scenarios where the required actions are complex. These states are more problematic for component implementations, and thus, simulations should drive the component through these scenarios. Consequently, whether a corner case has been reached or not can be used to measure simulation progress, and missed corner cases can be used to determine the direction of further simulations.

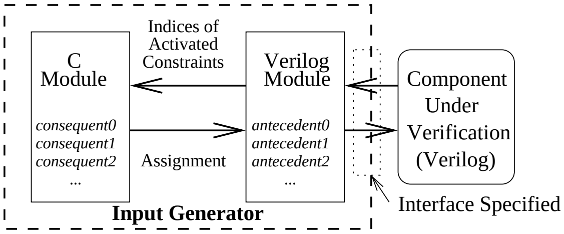

Figure 3: Implementation Details of the Input Generator

As a first order approximation of corner cases, the antecedents of the constraints are used. This is because only when the antecedent clause is true does the implementation have to comply with the constraint clause. As an example, consider the PCI constraint, "master must raise irdy within 8 cycles of the assertion of frame." The antecedent is "the counter that starts counting from the assertion of frame has reached 7 and irdy still has not been asserted" and the consequent is "irdy is asserted." Unless this antecedent condition happens during the simulation, compliance with this constraint cannot be completely known. For a simulation run which has triggered only 10% of the antecedents, only 10% of the constraints have been checked for the implementation. In this sense, the number of antecedents fired during a simulation run is a rough coverage metric.

There is one major drawback to using this metric for coverage. The problem is intimately related to the general relationship between implementation and specification. By the process of design, for every state, a designer chooses an action from the choices offered by the nondeterministic specification to create a deterministic implementation. As a result, the implementation will not cover the full range of behavior allowed by the specification. Thus, some of the antecedents in the specification will never be true because the implementation precludes any paths to such a state. Unless the verification engineer is familiar with the implementation design, he cannot know whether an antecedent has been missed because of the lack of appropriate simulation vectors or because it is structurally impossible.

2.3.2 *Deriving Biases for Missed Corner Cases*

To reach interesting corner cases, verification engineers often apply biasing to input generation. If problematic states are caused by certain inputs being true often, the engineer programs the random input generator to set the variable true *n*% instead of the neutral 50% of the time. For example, to verify how a component reacts to an environment which delays its response, *env_response*, the engineer can set the biasing so that the input, *env_response*, is true only 5% of the time. 0% is not used because it may cause the interface to deadlock. With prevailing methods, the user needs to provide the biasing numbers to the random input generator. This requires expert knowledge of the design, and the biases must be determined by hand. In contrast, by targeting antecedents, interesting biasing can be derived automatically. The algorithm works as follows:

1. Gather the constraints that specify the outputs of the component to be verified. The goal: the antecedents of these constraints should all become true during the simulation runs.
2. Set biases for all input signals to neutral (50% true) in the input generator described in section 2.2. (Exactly how this is done will be explained in the following subsection.)

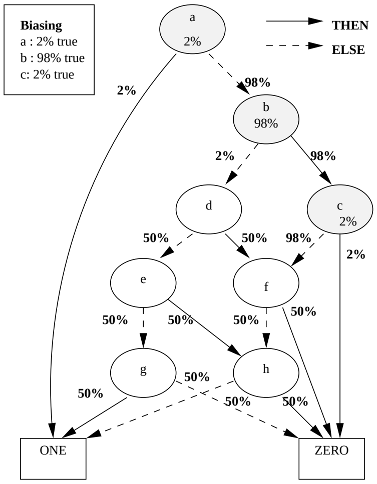

Figure 4: The Biased BDD Traversal

3. Run the simulation for some number of cycles.
4. Determine which antecedents have not fired so far.
5. Pick one missed antecedent, and use it to determine the variable biasing. If, for example, antecedent $\neg a \wedge b \wedge \neg c$ has not been true, set the following biases: a is true 2% of the time, b for 98%, c for 2%.
6. Re-run the simulation and repeat from step 4. Continue until all antecedents have been considered.

There are a number of interesting conclusions. First, although effort was invested in determining optimal bias numbers exactly, biases that simply allowed a signal to be true (or false) "often" was sufficient. Empirically, interpreting "often" as 49 out of 50 times (98%) seems to work well. Second, an antecedent expression contains not only interface signal variables but also counter values and other variables that cannot be skewed directly. Just skewing the input variables in the antecedent is primary biasing, and a more refined, secondary biasing can be done by dependency analysis. This was done manually. For example, many hard-to-reach cases are states where a counter has reached a high value, and by dependency analysis, biases that will allow a counter to increment frequently without resetting were determined.

2.3.3 Implementing Biasing

The actual skewing of the input variables is done during the BDD traversal stage of the input generation. After the input formula BDD for a component has been built, the structure is traversed according to the biases. If variable b is biased to be true 49 out of 50 times, the THEN branch is taken 49 out of 50 times (Figure 4). If this choice of branching forces the expression to evaluate to false (i.e. the traversal inevitably leads to the "ZERO" leaf), the algorithm will backtrack and the ELSE branch will be taken. As a result, even if b is biased to be true 49 out of 50 occurrences, the protocol logic can force b to be false most of the time. What is guaranteed by the biasing scheme is that whenever b is allowed to be true by the constraint, it will most likely be true.

An extra step is added to the input generation algorithm to accommodate the biasing. The variables need to be re-ordered so that the biased variables are at the top of the BDD, and their truth value are not determined by the other variables. In Figure 5, variable c is intended to be true most of the time. However, since c is buried towards the bottom of the BDD, if $\{a = 0,\ b = 1\}$ is chosen, c is forced to be false to satisfy the constraint. In contrast, if c is at the top of the BDD, the true branch can be taken as long as the other variables are set accordingly (for example, $a = 1$). Fortunately, since the number of BDD variables is kept small, reordering for this purpose does not lead to BDD blowup problems.

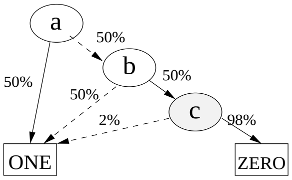

Figure 5: Incorrect Ordering

Compared to the biasing technique used in SimGen, the biasing used in this framework is coarse. With SimGen, branching probabilities, which take into account variable ordering, are calculated from the desired biases. In contrast, this method directly uses the biasing as the branching probabilities; it requires no calculations and compensates for possible distortions by reordering. Although implementing the SimGen calculations is not difficult, the advantages of achieving more precise biasing are not clear from the examples attempted.

3. EXPERIMENTAL RESULTS

To demonstrate the methodology on a meaningful design, we chose the I/O component from the Stanford FLASH [5] project for verification. The I/O unit, along with the rest of the project, had been extensively debugged, fabricated, and tested and is part of a working system in operation. The methods are evaluated on the PCI interface of the component.

The design is described by 8000 lines of Verilog and contains 283 variables which range from 1-bit to 32-bit variables: a complexity which renders straightforward model checking unsuitable. Approximate model checking was used by Govindaraju et al [4] to verify this design but no bugs were found because the design inputs were overly constrained and only a small state space was explored. Our simpler and more flexible simulation-based checking proved to be more effective by finding new bugs.

The Setup A formal PCI specification was used to constrain the inputs and check the outputs at the PCI interface of the design. A simulation checker that flags PCI protocol violations was generated from the specification using a compiler tool written in OCAML [7]. The same compiler tool was modified to output the constrained random simulation generator which controls the PCI interface inputs of the I/O unit. The I/O unit (the design under verification), checker, and input generator are connected and simulated together, and results are viewed using the VCD (Value Change Dump) file. The inputs were skewed in different configurations for each simulation run in order to produce various extreme environments and stress the I/O unit.

Verification Results Using the 70 assertions provided by the interface specification, nine previously unreported bugs have been found in the I/O unit. Most are due to incorrect state machine design. For example, one bug manifested itself by violating the protocol constraint, "once *trdy* has been asserted, it must stay asserted until the completion of a data phase." Because of an incorrect path in the state machine, in some cases, the design would assert *trdy* and then, before the completion of the data phase, deassert *trdy*. This can deadlock the bus if the counterparty infinitely waits for the assertion of *trdy*. The bug was easily corrected by removing the problematic and most likely unintended path. The setup makes

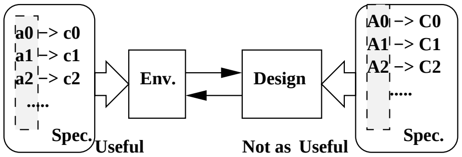

Figure 6: The Two Types of Simulation Coverage Metric and their Effectiveness

the verification process much easier; the process of *finding* signal-level bugs is now nearly automated, and so, most of the effort can focus on *reasoning* about the bug once it is found.

Coverage Results Unfortunately, the original intended use of the coverage metric proved to be fruitless for this experiment. Using antecedents of the constraints that *specify the component* was meaningless because the FLASH PCI design is conservative and implements a very small subset of the specification. For example, the design only initiates single data phase transactions, and never initiates multiple data phase transactions. Thus, most of the antecedents remained false because it was structurally impossible for them to become true.

However, using the metric to ensure that the environment is maximally flexible proved to be much more powerful. The motivation is to ensure that the design is compatible with any component that complies with the interface protocol. The design should be stimulated with the most general set of inputs, and so, using the missed antecedents from the constraints that *specify the environment* (in Figure 6, “a0, a1, ...”) to determine biases was extremely fruitful; most of the design bugs were unearthed with these biasings.

Performance Results Performance issues, such as speed and memory usage, did not pose to be problems, and so, we were free to focus on generating interesting simulation inputs. However, to demonstrate the scalability of the method for larger designs, performance results were tabulated. The simulations were run on a 4-Processor Sun Ultra SPARC-II 296 MHz System with 1.28Gbytes of main memory. The specification provided 63 constraints to model the environment. These constraints required 161 boolean variables, but because of the “activating” – “constraining” logic separation technique, only 15 were needed in the BDDs. Consequently, the BDDs used were very small; the peak number of nodes during simulation was 193, and the peak amount of memory used was 4Mbytes.

Furthermore, speed was only slightly sacrificed in order to achieve this space efficiency. The execution times for different settings are listed in Table 2. With no constraint solving, where inputs are randomly set, the simulation takes 0.64s for 12,000 simulator time steps. If the input generator is used, the execution time increased by 57% to 1.00s; this is not a debilitating increase, and now the inputs are guaranteed to be correct. The table also indicates how progressively adding signal value dumps, a correctness checker module, and coverage monitor modules, adds to the execution time.

4. FUTURE WORK

Better coverage metrics can probably be deduced from the specification. A straightforward extension would be to see whether pairs of antecedents become true during simulations. Exploiting a structured formal specification for other uses is also of interest. Perhaps incomplete designs can be automatically augmented by specification constraints for simulation purposes. Or, useful synthesis information can be extracted from the specification. Also, experiments to determine whether designs that are too big for SimGen-type algorithms can be handled by ours would further validate the methodology. Furthermore, more extensive experiments to quantify the speed penalty for the dynamic BDD building should be done.

	Number
Boolean Vars in Spec	161
Boolean Vars in BDD	15
Constraints on Env	63
Assertions on Design	70
Peak Nodes in BDD	193
BDD Memory Use	4 Mbytes
Bugs Found in Design	9

Table 1: Interface Specification Based Generation Details for the FLASH Example

Settings	User Time	System Time	Total
Random	0.53s	0.11s	0.64s
Constrained	0.77s	0.23s	1.00s
with Dump	0.77s	0.26s	1.03s
with Monitor	1.33s	0.29s	1.62s
with Coverage	1.54s	0.25s	1.79s

Table 2: Time Performance of the Methodology on FLASH Example (for 12000 simulator time steps)

Acknowledgement This research was supported by GSRC contract SA2206-23106PG-2.

5. REFERENCES

[1] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, volume C-35, pages 677–691, August 1986.

[2] E. Clarke and E. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, May 1981.

[3] E. Clarke, S. German, Y.Lu, H.Veith, and D.Wang. Executable Protocol Specification in ESL. In *Proceedings of FMCAD*, November 2000.

[4] S. G. Govindaraju and D. L. Dill. Counterexample-guided choice of projections in approximate symbolic model checking. In *Proceedings of ICCAD*, November 2000. San Jose, CA.

[5] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, and et al. The Stanford FLASH Multiprocessor. In *Proceedings of ISCA*, pages 302–313, April 1994.

[6] K. McMillan. http://www-cad.eecs.berkeley.edu/˜kenmcmil/smv/.

[7] OCAML. http://caml.inria.fr.

[8] K. Shimizu, D. L. Dill, and C.-T. Chou. A Specification Methodology by a Collection of Compact Properties as Applied to the Intel Itanium Processor Bus Protocol. In *Proceedings of CHARME*, September 2001.

[9] K. Shimizu, D. L. Dill, and A. J. Hu. Monitor-Based Formal Specification of PCI. In *Proceedings of FMCAD*, November 2000.

[10] F. Somenzi. http://vlsi.colorado.edu/˜fabio/CUDD/cuddIntro.html.

[11] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling Design Constraints and Biasing in Simulation Using BDDs. In *Proceedings of ICCAD*, November 1999.