Combining Simulation and Formal Verification for Integrated Circuit Design Validation

Lun Li, Stephen A. Szygenda, Mitchell A. Thornton
Dept. of Computer Science and Engineering
Southern Methodist University, Dallas, Texas 75206
{lli, szygenda, mitch}@engr.smu.edu

ABSTRACT

The correct design of complex hardware continues to challenge engineers. Bugs in a design that are not uncovered in early design stage can be extremely expensive. Simulation is a predominantly used tool to validate a design in industry. Formal verification overcomes the weakness of exhaustive simulation by applying mathematical methodologies to validate a design. The work described here focuses upon a combinational technique that integrates the best characteristics of both simulation and formal verification methods to provide an effective design validation tool, referred as IDV. The novelty in this approach consists of three components, a circuit complexity analyzer, a partitioning tool and a coverage analysis unit. The circuit complexity analyzer and partitioning tool partition a large design and feed sub-components to different verification and/or simulation tools based upon known existing strengths of modern verification and simulation tools. The coverage analysis unit computes the coverage rate of design validation and improves the coverage by further partitioning. Various tools comprising IDV are evaluated and an example is used to illustrate the overall validation process. The overall process successfully validates the example to a high coverage rate within a short time. The experimental result shows that our approach is a very promising design validation method.

Keywords: Simulation, Formal Verification, Circuit Design, Validation

1. INTRODUCTION

Design validation is the process of finding design errors in a model of an electronic *integrated circuit* (IC) before it is manufactured. IC designers rely heavily upon simulation techniques; however, the size of ICs continues to increase in terms of the number of transistors per chip resulting in diminished validation effectiveness when using simulation only. More recently, formal verification methods have been developed that utilize specialized models of ICs and then mathematically reason about them to prove design correctness in an automated way. While some formal verification methods are beginning to appear in commercial tools, most formal methods are limited to relatively small ICs or small sub-circuits of large ICs. The work described here focuses upon the creation of a new technique that integrates the best characteristics of both simulation and formal verification methods to provide a new and effective IC design validation CAD tool.

We describe an integrated approach to design validation that takes advantage of current technology in the areas of simulation (for both critical timing and fault simulation), formal verification and *Automatic Test Pattern Generation* (ATPG) resulting in a practical verification engine with reasonable runtime called the *Integrated Design Validation* system (IDV). To accomplish this objective it is essential that researchers must have an understanding of these somewhat diverse approaches. This enables the development of IDV such that the benefits of both simulation and verification techniques are exploited and will ultimately result in a method that allows for a more effective technique for design validation.

2. METHODOLOGY

This research utilizes existing simulation and verification techniques to concentrate on their **efficient integration** to provide a comprehensive tool for design specification compliance. The latest results in all areas of verification [3, 6 11, 12] simulation [9, 2], and test [5] are being used to provide a design compliance tool that will be extremely effective and has the potential to "out-perform" the current "state-of-the-art" methods focused upon a single methodology.

The novelty in this approach is in the development of a circuit complexity analyzer and partitioning tool based upon known existing strengths of modern verification and simulation tools, to develop coverage analysis methods that compute the degree of design validation, to develop a method for intelligently updating the complexity analyzer for further validation iterations, and to integrate these techniques with existing simulation and formal verification techniques. There have been recent attempts to tightly combine two different verification tools [4, 7], most notably SAT solvers and BDD approaches for equivalence checking [8]; however, no overall verification/simulation engine with significant analysis before design validation occurs has been produced.

The overall structure of the prototype *Integrated Design Validation* (IDV) system is shown in the block diagram of Figure 1. A primary focus of this project is on the complexity analyzer, partition, and coverage analyzer blocks, to determine the most effective use of simulation on this problem. Each of these blocks is described in detail.

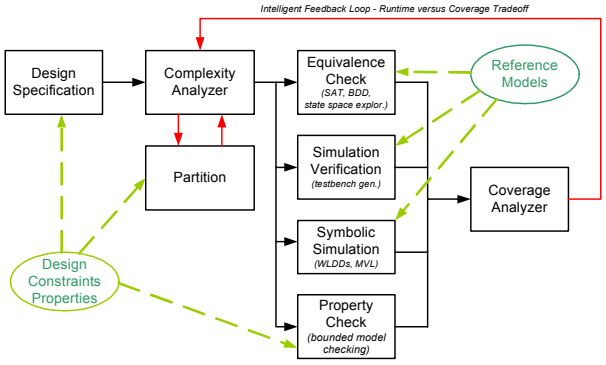

Figure 1 Proposed Architecture of the IDV System

2.1 Complexity Analyzer

The complexity analyzer estimates the complexity of an RTL or netlist design based on existing methods for controller/datapath extraction and new techniques are being developed for this purpose. Integration with the partitioner is crucial for this function. The extracted control and datapath portions of the circuitry are being analyzed for applicability of

various techniques. One method being investigated is to send the same partitions of a circuit to more than one validation technique. As an example, a portion of a datapath may be sent to both a SAT-based equivalence checker and a BDD-based equivalence checker. It is known that if a BDD-based approach is going to be effective, it will likely execute very rapidly. Thus, we can initiate multiple verification "threads" for a given portion of a circuit and let the thread producing a result first "win". If some amount of preset computational resource is reached with no result, then the sub-circuit is sent to the simulator.

In terms of automating the simulator, an interesting approach was presented in [14] where the focus was the automated generation of 1) an input generator, 2) a coverage metric and, 3) an output correctness checker. These three simulation "aids" were generated from a formal specification of module interface protocols. The input generator produced input sequences based on what the interface protocol allows and the output checker compared the output to what the protocol defined as correct, The coverage metric quantified coverage by exploiting the fact that the protocol defines the set of all possible interface events. We are utilizing this approach initially, making changes where needed to comply with the verification techniques.

2.2 Design Partitioning

One of the biggest hurdles in applying formal techniques is to correctly identify the target circuits. We propose to start with the designer defined hierarchy. Designers usually partition their design into multiple RTL blocks (these blocks usually mirror the floor-planned design). Methods that exploit such an inherent design hierarchy have been used in the past such as the jMocha tool [1].

Although a lot of work has been accomplished with respect to partitioning for logic and physical level synthesis, there is not as much for design validation and simulation. In [10] a methodology for automatically extracting controllers from an RTL-HDL specification is described. This work introduces an algorithm for automatically separating the datapath and controller described at the RTL level by locating general patterns of FSMs in a *Process-Module* (PM) graph representation of the design. A PM graph is defined to be a directed graph where each node represents a sequential process, a concurrent dataflow statement, or a module instantiation in both VHDL and Verilog. In such a representation, the hierarchy is preserved and each module contains its own PM graph. Because a FSM's next states always functionally depend on their current state, signals stemming from state-registers will loop back after some combinational paths. Finding the FSMs in the HDL is based on finding such loops in the PM graph. Some loops found, however, may have a valid pattern topologically but not be part of a FSM. To deal with such instances, the checking of functional dependency follows the loop search to determine if the loop is a valid part of the FSM. The extraction process is divided into 4 phases, most of which are traversal procedures resembling a Depth-First Search of the PM graph. All steps are of linear complexity. Using this approach as a starting point, this research demonstrates feasibility by has focusing on the separation of a design into separate controller and datapath circuits.

Another technique allows for abstracting away large portions of the datapath circuitry leaving the controllers intact [13]. The research presents a methodology for abstracting away portions of the datapath and reducing the bit-width size of some elements while preserving the control structure. This process, referred to as a spatial abstraction, reduces the state space of the system under consideration and allows for complete verification with model checking. The fundamental concept is to identify the data path storage elements that do not contribute to the control flow of the design and reduce their size to a single bit. Next, using interval computation, the range of values that can be assumed by all the storage elements is determined. The abstraction procedure consists of 1) partitioning the design into a *module call graph* -a collection of modules as a list, 2) classifying the variables as control, data, or mixed 3) initializing the variables with respect to their classification and size and 4) Interval Propagation. Experimental results show a drastic reduction in states and CPU time for verification.

2.3 Coverage Analysis

Some work has been done in terms of coverage analysis particularly with respect to evaluating the effectiveness of simulation-based validation. An overview of design validation coverage methods is given in [16] that classify existing metrics in terms of code coverage, metrics based on circuit structure, metrics defined on finite state machines, functional coverage, error models, observability, and metrics applied to specifications. These existing metrics will be used as a starting point for the development of the coverage analyzer. Not much work has been accomplished in terms of combining formal verification with simulation and computing the overall coverage. We plan to extend these methods.

3 VERIFICATION AND SIMULATION TOOLS COMPRISING IDV

To integrate the tools and make them complement each other is our goal. Various tools have been developed for formal verification and simulation. Choosing the right tools will set up the base for the success of our system.

3.1 STE (Symbolic Trajectory Evaluation)

STE is a model checking approach designed to verify circuits with very large state spaces. It is more sensitive to the property being checked instead of the size of the circuit. The STE package we use is from the Intel Strategic Research Lab, *Forte*. It also supports a simple yet effective compositional theory beside STE. Two important properties of STE are:

a. It is suitable for verifying designs of circuits at the gate or switch level
b. STE provides accurate models of timing, which is reflected in the types of properties checked for.

STE originated the idea from multi-level simulation and ternary-value symbolic simulation. It is a formal verification method that is close to traditional simulation. One of the distinguishing features of STE is that the state space is represented as lattice. The partial order of the lattice represents an information ordering or abstraction relation between states. The higher up we go in the information ordering, the more information we have. The computational advantage of this is that, given the appropriate logical framework, if we prove a property holds of a state in the lattice, it holds of all states above it in the lattice. Another important fact is that circuits have natural representations as lattices, and the use of the information ordering allows us to easily abstract out the necessary information for property checking [17].

The properties be checked are represented as temporal logic (TL). TL is usually propositional or first-order logic augmented with temporal modal operators that allow reasoning about how the truth values of assertions change over time. TL can express safety and liveness properties, such as "property *p* holds at all times" or "if *p* holds at some instant in time, q must eventually

hold at some later time." Properties of this sort can be employed to specify desired properties of systems, i.e. in a traffic signal control system, "the signals at both directions should never be green at the same time" and "the signal at one direction will eventually be green".

The properties that STE focuses on are a restricted TL that offers only the next-time operator [19], which is called trajectory formula. A trajectory assertion has the form $A \rightarrow C$, where A and C are trajectory formulas, named as *antecedent* and *consequent* respectively. Informally, a trajectory assertion holds for a circuit M iff each sequence of states of M that satisfies the antecedent A also satisfies the consequent C. Typically, A specifies constraints on how the inputs of a circuit are driven, while C asserts the expected results on the output nodes [18]. For example, the formula ($read\_enable$=1 $\wedge$ $addr \rightarrow out = \textbf{\textit{Next}}(M[addr])$ asserts that if signal *read_enable* is 1 and *address* is specified, the output of memory is the value stored at *address* in the next cycle.

3.2 SMV(Symbolic Model Checking)

TL introduced in above section can be used as a framework for the specification of the temporal properties of a design. Computational TL (CTL) [22] is a propositional logic of branching time. It is based on propositional logic and uses a discrete model of time where, at each instant, time may split into more than one possible future event. Thus, it forms a tree structure. The algorithm to determine whether or not a given design satisfies a CTL is named as *model checking*. In model-checking techniques, the entire state transition graph need to be constructed either explicitly or using a symbolic representation.

Reduced, Ordered binary decision diagrams (ROBDDs) [23] provide a powerful symbolic representation for Boolean functions. A binary decision diagram (BDD) is a rooted, directed, acyclic graph (DAG). There are two types of nodes in the graph: terminal and non-terminal nodes. The terminal node is labeled with either the constant 0 or constant 1 and has no outgoing edges. Each non-terminal node is associated with one binary variable and has two outgoing edges labeled as T (Then) and E (Else) respectively, which correspond to the two possible valuations of the node's variable. ROBDD has the additional property that no variable appears more than once, and the variables appear in the same order on every path. ROBDD is a canonical representation for Boolean functions. The model checking algorithm utilize BDD are also referred as symbolic model checking or SMV. Since McMillan implemented a SMV algorithm based on BDD [24], SMV has attracted lots of attentions. However, BDD-based SMV can only middle-sized designs since BDD size usually exceed memory limit in large designs. The SMV tool we use in the IDV is VIS (Verification Interacting with Synthesis).

VIS is a verification package developed jointly at the University of California at Berkeley, the University of Colorado at Boulder, and more recently, at the University of Texas, Austin [21]. VIS is able to synthesize finite state systems and/or verify properties of such systems, which have been specified hierarchically as a collection of interacting finite state machines. VIS is build upon the BDD package developed by the University of Colorado at Boulder, named CUDD [20]. VIS and CUDD has been used extensively in academia for symbolic model checking (SMV).

STE and VIS are both capable of model checking. They differ in the following aspects:

Properties: VIS can verify more properties since it uses CTL while the trajectory formula supported by STE is less expressive.

Capacity: STE can handle bigger circuits in terms of latches and bit cells (over 1000 latches). VIS usually exceeds memory capacity when there are more than 200 latches. STE trades expression power for capacity.

BDD Memory: The underlying engine for VIS is compact symbolic representation in term of BDDs representing circuit model was built on. The underlying engine for STE is symbolic simulation where the size of BDDs is related more to the properties instead of circuit model.

Application: Based on above differences, we can conclude that VIS is better in control dominated designs while STE is more suitable for memory dominated circuits. Actually, STE has been used extensively in property checking for memory.

3.3 Speed5

Speed5 is Tegas-like, 5-value multi-modal, assignable-delay, five-valued simulator [15]. It performs gate-level and functional-level simulation. Nominal and critical timing (min/max) delays are used in simulation. Speed has fault simulation ability by fault generation and insertion into the simulated circuit. Fault models that are provided are stuck-at, shorts, transient fault models, and multiple faults. Performance is improved by parallel simulation of faults where a specified number of faults are simulated in one pass. The number of faults per simulation is determined from indistinguishable fault classes, fault blocking characteristics and the desired diagnostic resolution.

3.4 SMU Equivalence Checker (SMU-EQ)

The equivalence checker developed in our group [11] performs quite well on large designs. The core part of the equivalence checking tools is image computation where conjunctive scheduling is very important to reduce BDD size of intermediate computations. In our approach, we developed a genetic-based approach to minimize total lifetime and active lifetime at the same time. Experimental result shows that SMU-EQ is very effective. We are incorporating a SAT engine into our equivalence checker to make it more robust and to handle larger designs.

3.5 SMU Functional Simulator

We are also working on a functional simulator which will be used for system level simulation. In the system level, we are more interested in the interconnection of modules instead of the internal function of separate modules. The functionalities of these modules are fully verified by VIS or STE or simulated by Speed before they are put into the functional simulator.

4 VERIFICATION PROCESS

In this section, we use an example to show the whole process of our IDV system. The example we use is quite simple but contains all the necessary modules for big designs. We only use this small example to show the idea of our methodology. Our design is targeted at much larger system designs. The example we use is an inverse circuit which calculate the inverse of an 8-bit unsigned integer, B, using Newton-Raphson iterative equation.

$$x_{i+1} = 2x_i - Bx_i^2$$

The block diagram of the system is shown in Figure 2. The system works as follows: the initial value of x_0 is an estimate that is stored in *RAM*. The 3-bit address is generated from the integer to get the initial value from RAM. Then the value will be sent to the circuit that implements the above equation, named as *newtraph*. The result of computation is feedback and loops 5

times before the result output. The loop and selection are handled by *controller*.

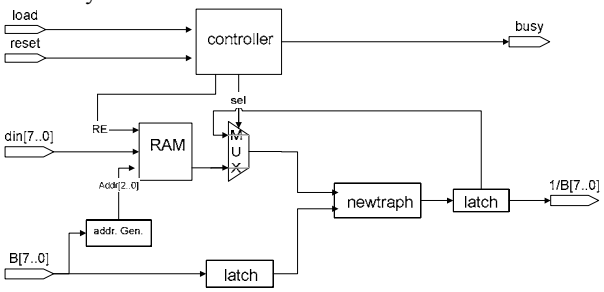

Figure 2 Inverse Circuit Block Diagram

The above design includes a memory unit (RAM), a datapath (2 multipliers and 1 adder), control logic (counter and FSM), and some small components, such as an *address generator*, *latch*, and *mux*. For such a design, the properties to be checked may be as follows:

a. Liveness property: *load*=1 → *AX*:2(*AF*(*busy*=0)): Along all the path in the future, there will be a state that busy=0 and it will last at least for next 2 states. The signal *load* is 1 means an unsigned integer is loaded for calculating the inverse. The signal *busy* is 1 while in the process of calculating the inverse and 0 when it is idle or the calculation is done. This property indicates that if an integer is loaded for inverse and it must finish calculation sometime in the future and won't get into endless loop (busy will never be 0). It's a liveness property since it indicates that the constraint that circuits will not be dead.

b. Safety properties: *load*=1 → ***Next***(*busy*=1): If signal *load* =1, the signal *busy* has to be 1 in the next cycle. As we indicated before, busy is 1 when the circuit is getting into the calculation process. So the property means that if an integer is load, the calculation should be started in next cycle.

c. Properties related to Memory: *RE*=1 ∧ *addr* → *RAM_out* = ***Next***(*M*[*addr*]): The signal RE indicates read enable for RAM. The property can be interpreted as: if read enable for RAM is on and a valid address is given, the output of RAM in next cycle should be the value stored in that address.

d. Also most importantly is overall functionality. This means that the multiplier, adder, counter, and all other components should work properly separately and together.

Based upon previous tool sets, we modified our verification process as Figure 3 shows. We will explain the whole process by showing how it works on the above example.

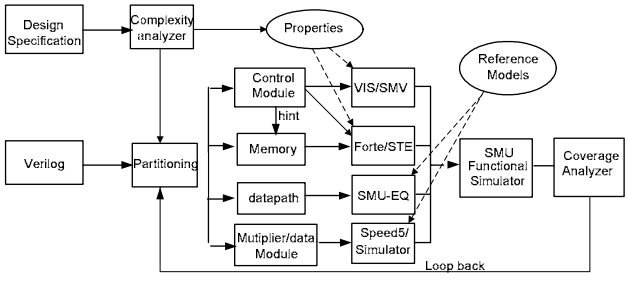

Figure 3 Modified Verification Flow

4.1 Complexity Analyzer

The complexity analyzer is mainly focused on:

a. Analyzing the properties checked and assigning them to different verification tools. Complex properties specified in CTL are fed into VIS while properties given in Trajectory Formula go to STE. Given the above example, property *a* is quite complicated and STE can not handle it. So we use VIS. While property *b* can be verified via either VIS or STE, in such a case, we prefer using STE since STE have larger capacity. Also, all properties related to memory go to STE for formal verification since STE performs better in such component and the related properties usually can be expressed as trajectory formulas.

b. Deciding to use a SAT-based approach or BDD-based approach for equivalence checking based on the number of variables. A SAT-based approach is more time consuming but can handle large designs while BDD-based approach are better for middle-sized designs and much faster. Thus, if the number of variables is over a threshold, we will use a SAT-based method, otherwise we use BDD-based approach. We are also investigating a hybrid method that could combine these two methods in the same framework that, given the time to build a BDD, if the BDD can be built, go ahead and use BDD; otherwise, SAT is used.

c. Extracting components/modules/processes and interconnections among them information. These information will be used for partitioning and system level functional simulation. We build a process-module graph which describes the hierarchy of the design. Each node in the graph represents a component/module/process and edge corresponding to the interconnections of these components. The process-module graph for the above example is shown in Figure 4.

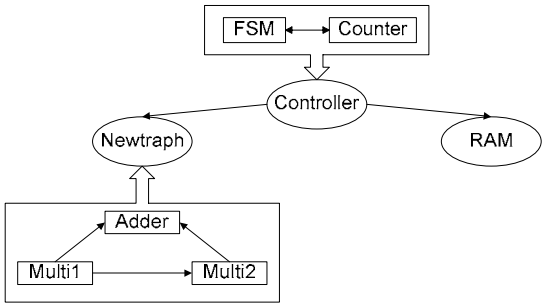

Figure 4 Graph Representation of Design Hierarchy

4.2 Partitioning

Design hierarchy explored in the previous stage is analyzed further in this stage. We initially explore coarse-grain partitions and continuously increase the granularity of the partitions until the desire goal is reached. Given the above example, the top-level consists of three parts, *controller*, *RAM*, and *datapath*. These three parts will be extracted and put into different tools for verification or simulation. The *controller* will go to property checking and reachability analysis, the *datapath* will go to equivalence checking or simulation, while the memory unit will go to STE for property checking.

We also allow the designer to specify some components that should be checked separately, especially for complex components.

4.3 Verification or Simulation process

After the complexity analysis and partitioning are completed, we send the subcircuits to different tools for verification and/or simulation as shown in Figure 3. The general rules are listed follows:

a. VIS deals with complex properties presented in CTL and control logic

b. STE deals with simple TL and control logic, and also all properties related to memory

c. SMU-EQ deals with datapaths. The number of variables is used to choose BDD-based or SAT-based approaches

d. Speed5 simulator deals with multipliers or other complex components specified by the designers

e. Functional simulation/system-level simulation is used at the last step for the interconnections of the components

Given the 3 components of the inverse circuit example, we show results that use different tools in Table 1. The results give a sense of time the required computational time for tools. These results were obtained using a Pentium 4 PC with 512MB of Memory.

Table 1 Verification/Simulation result

Component	Properties	Tools	Result	Time
Controller	a	VIS	T	0.1s
	b	STE	T	0.08s
Memory	c	STE	T	0.3s
Newtraph	10%	Speed	work	0.9s
Functional	1%	Func. Sim	work	5s

4.4 Coverage Analysis

Generally, the formula for calculating coverage of an output is based on

a. the contributions or importance that each component provides for the output

b. the coverage rate of the components related to the output when they are tested separately

c. the rate that inputs/interconnections related to the component have been tested in the system-level simulation

Detailed descriptions for the above three points are given as follows. For each output, not all components contribute to it, such as the output signal *busy* which is only related to the component *Controller*. Also, not all components related to an output have the same contribution. For example, the output *1/B* is related to the all three components and it is the direct output of the component *NewtRaph* which in turn relates to the other two components. By examining the design, we can find that the component *Controller* controls selection of data from RAM or loopback while the other two only provide data. So we will assign a higher contribution rate to the component *Controller* than the other two components. The contribution of the component C_i to the output is represented as w_i. Currently, we put the load of assigning the contribution rate to designers and hope to automate it in the near future.

The second point is easy to understand. For example, if an output is related to the n components $C_1 \ldots C_n$, each component has a corresponding coverage rate $R_1 \ldots R_n$ when they are verified or simulated separately. R_i is 1 if the component has been fully verified or a value $R_i \in (0,1)$ that represents the percentage of vectors tested via simulation.

The third point is related to the system-level simulation and interconnection of components. We use $I_1 \ldots I_n$ to represent coverage rates of inputs at a system-level simulation. When the component is simulated in a system, the input of one component may be the output of another component. The interconnection between them will change the coverage rate of components, especially for data bus between them. We study the coverage rate for component *B* for such cases shown as Figure 5.

Assume that m percent of input vectors of component *A* are tested at a system-level and its' output accounts for n percent of vectors while p percent input vectors for component *B* are tested separately. So the new coverage rate for the component *B* is in the range $[\max(n, p), n + p]$. The left-end situation happens when the output vectors of the component *A* are totally covered by or cover tested the input vector of the component *B*. In such a case, a maximal number of test vectors can not exceed n or p, thus they can be written as $\max(n, p)$. The right-end situation happens when there is no overlap between output vectors of the component *A* and tested input vectors of the component *B*. In general, the coverage rate for interconnected components can be calculated as range:

$$R' = [\max(I_{out}^A, I_{in}^B), (I_{out}^A + I_{in}^B)] \quad (1)$$

where I_{out}^A, I_{in}^B represents the output coverage rate of component *A* and the tested input coverage rate of component *B*. The output coverage rate of component A is related to the system-level test input coverage rate. This equation is only applied to simulated components that have data bus interconnections. For verified components, $R' = 1$

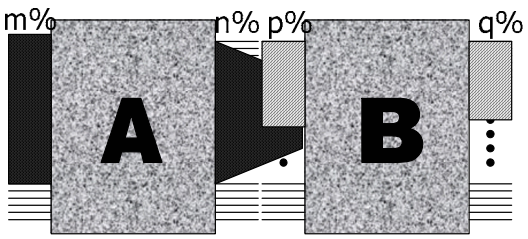

Figure 5 Sequential Model of Coverage Analysis

So the total coverage rate for the output can be written as:

$$P_o = w_1 R_1' + w_2 R_2' + \ldots + w_n R_n' \quad (2)$$

We will show how to use the above equation to calculate the coverage rate of the given example, there are two outputs, *busy* and *1/B*. We are going to calculate the coverage rate for both of them. Among three components, *Controller* and *RAM* are fully verified and their coverage rates are $R_c = 1$ and $R_m = 1$ respectively. While component, *NewtRaph*, is simulated with 50% of total being tested yielding $R_d = 0.5$. In the system-level simulation, input *load* is tested in both phases ($I_{load} = 1$) and 20% of vectors for *B* has been tested ($I_B = 0.2$).

The output *Busy*, is only related to the component *Controller* and also input *load*. So the component *Controller* totally controls the functionally of *Busy*, thus we have $w_c = 1$. Applying Eq. (1), we obtain:

$$P_{busy} = w_c R_c' = w_c R_c = 1 \times 1 = 1 = 100\%$$

which indicates that output *Busy* has been fully verified.

The output *1/B* related to all three components. The output is directly the output of the component *NewtRaph* which accepts 2 inputs; one input is directly coming from system input while the other one is produced by the component *RAM* or feedback which is controlled by the component *Controller*. Let's first determine the contribution of each component to the output. By examining the design, we can find that the component *Controller* controls selection of data from *RAM* or loopback. So we will assign a higher contribution rate to the component *Controller*, $w_c = 0.5$. The other two components are assigned equal contribution with $w_m = 0.25$ and $w_d = 0.25$ each. Next, we will determine the coverage rate for interconnection component, *NewtRaph*.

In above example, the interconnection data bus is from the component *RAM* (The *Mux* between them is fully verified). The coverage rate of the output vector of the component *RAM* is the same as the coverage rate of the tested input of the component *RAM* which is 0.1. The tested vector rate of the component

Newtraph is 50% as specified before. So the new coverage rate for *NewtRaph* is $I_{m\leftrightarrow d} = [50\%, 60\%]$.

We now calculate the coverage rate for the output ***1/B***.

$$\begin{aligned} P_{1/B} &= w_c R_c + w_m R_m + w_n R_d (I^d_{m\leftrightarrow d} + I^d_B) \\ &= 0.5\times 1 + 0.25\times 1\times + 0.25\times [0.6, 0.7] \\ &= [0.9, 0.925] \end{aligned}$$

We provide the coverage rate of the system with the initial partition. The rate can be further improved by refinement.

4.5 Loop Back

When the coverage is too low, IDV loops back to do more partitions on previously simulated components. In the above examples, *Newtraph* is the only component simulated. When IDV loops back and examines the component, *Newtraph*, it is found that it contains three small modules, two multipliers and one adder. We can further partition, *Newtraph*, into three small modules and formally verify some of the small modules. The tools get into another hierarchy of the design. The equivalence checking tool is used formally verify the adder and the two multipliers are simulated. After simulation, the coverage of this component is calculated using the method described above. Equal weight is assigned to the three components, $w_{mult1} = w_{mult2} = w_{adder} = 1/3$. Since the adder is fully verified ($R_{adder} = 1$) and the other two components have been tested at the same rate as before, $R_{mult1} = R_{mult2} = 0.5$. Thus applying yields:

$$\begin{aligned} R_d &= w_{mult1} R_{mult1} + w_{mult2} R_{mult2} + w_{adder} R_{adder} \\ &= 1/3\times (0.5 + 0.5 + 1) = 0.67 \end{aligned}$$

The coverage of the component, *Newtraph*, has been improved from 50% to 67% by refining partitioning. After that, the coverage of the output, ***1/B***, is calculated again. All other parameters stay the same. The coverage rate of component, *Newtraph*, when put into the system is improved since the coverage of I^B_{in} is increased by refinement.

$$R' = [\max(I^A_{out}, I^B_{in}), (I^A_{out} + I^B_{in})] = [0.67, 0.77]$$

and the coverage rate of the output *1/B*, is

$$\begin{aligned} P_{1/B} &= w_c R_c + w_m R_m + w_n R_d (I^d_{m\leftrightarrow d} + I^d_B) \\ &= 0.5\times 1 + 0.25\times 1\times + 0.25\times [0.77, 0.87] \\ &= [0.92, 0.96] \end{aligned}$$

We can see that the coverage rate of the output is improved by refinement.

5 FUTURE WORK

At the current stage, the entire IDV system has not been fully automated. We are working to automate the whole process. We are trying to develop IDV such that a minimum of manual involvement is required. We are also continuing to refine the component of IDV such as the functional simulator.

REFERENCES

[1] R. Alur, et al., "Mocha: A Model Checking Tool that Exploits Design Structure," **Proc. of the IEEE Inter.ational Conference on Software Engineering**, 2001

[2] A. Aziz, et al., "Hybrid Verification Using Saturated Simulation," **Proc. of the DAC**, 1998, pp. 615-618

[3] R. Bloem, et al., "Symbolic Guided Search for CTL Model Checking," **Proc. of the DAC,** 2000

[4] J. Burch, et al. "Tight Integration of Combinational Verification Methods," **Proc. of the ICCAD,** pp. 570-576, 2000

[5] A. Chandra et. al., "AVPGEN – A Test Generator for Architecture Validation," **IEEE Tran. VLSI,** Vol 3, No 2, June 1995

[6] S. G. Govindaraju, D. L. Dill, and J. P. Bergmann, "Improved Approximate Reachability using Auxiliary State Variables," **Proc. of the DAC**, June 1999, pp. 312-316

[7] S. Hazelhurst, et al.. "A Hybrid Verification Approach : Getting Deep into the Design," **Proc. of the DAC,** 2002

[8] A. Kuehlmann, M. Ganai and V. Paruthi, "Circuit-based Boolean Reasoning," **Proc. of the DAC,** pp. 232-237, 2001

[9] K. Kang and S.A. Szygenda, "Accurate Logic Simulation by Overcoming the Unknown Value Propagation Problem", **Simulation Journal**, Vol. 79, Issue2, February 2003

[10] C.-N. J. Liu and J.-Y. Jou, "An Automatic Controller Extractor for HDL Descriptions at the RTL," **IEEE Design & Test of Computers,** pp. 72-77, July-September 2000

[11] L. Li, M.A. Thornton, and S.A. Szygenda, "A Genetic Approach for Conjunction Scheduling in Symbolic Equivalence Checking," **IEEE Computer Society Annual Symposium on VLSI,** pp. 32-36, February 2004

[12] R. Marcyzynski, M.A. Thornton, and S.A. Szygenda, "Test Vector Generation and Classification Using Symbolic FSM Traversals," **International Symposium on Circuits and Systems,** pp. V-309 – V-312, May 2004

[13] V. Paruthi, N. Mansouri and R. Vemuri, "Automatic Data Path Abstraction for Verification of Large Scale Designs," **Proc. of the ICCD**, pp. 192-194, 1998

[14] K. Shimizu and D. L. Dill, "Using Formal Specifications for Functional Validation of Hardware Designs," **IEEE Design & Test of Computers,** pp. 96-106, July-August 2002

[15] S. Szygenda, "The Simulation Automation System, Using Automatic Program generation, for Hierarchical Digital Simulation Systems," **Proc. of the European Simulation Conference,** 1990

[16] S. Tasiran and K. Keutzer, "Coverage Metrics for Functional Validation of Hardware Designs," **IEEE Design & Test of Computers,** pp. 36-45, July-August 2001

[17] S. Hazelhurst and C-J Seger, "Symbolic Trajectory Evaluation." **In T. Kropf, editor, Formal Hardware Verification**, ch. 1, pp 3-78, Springer Verlag; New York, 1997

[18] Christoph Kern and Mark Greenstreet "Formal verification in hardware design: a survey," **ACM Trans. on Design Automation of Electronic Systems,** Vol. 4, Iss. 2, pp: 123-193, 1999

[19] Carl Seger, And Randy Bryant, "Formal verification by symbolic evaluation of partially- ordered trajectories," **Formal Methods System Design**, Vol. 6, Iss. 2, pp: 147-189, 1995

[20] F. Somenzi et al. CUDD: University of Colorado Decision Diagram Package. http://vlsi.colorado.edu/~fabio/CUDD/

[21] R. Brayton et al. VIS: A system for verification and synthesis. http://vlsi.colorado.edu/vis/

[22] E., Clarke, E., Emerson, and A. Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications." **ACM Trans. Program. Language System**, Vol. 8, Iss. 2, pp: 244–263, 1986

[23] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," **IEEE Trans. Computers**, vol. 35, pp. 677–691, Aug. 1986

[24] K. Mcmillan, "Symbolic model checking—an approach to the state explosion problem," **Ph.D. Dissertation,** Carnegie Mellon University, 1992