Integrated Design Validation: Combining Simulation and Formal Verification for Digital Integrated Circuits

Lun Li, Mitchell A. Thornton, Stephen A. Szygenda
Dept. of Computer Science and Engineering
Southern Methodist University, Dallas, Texas 75206
{lli, mitch,szygenda}@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 stages 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 technique that integrates the best characteristics of both simulation and formal verification methods to provide an effective design validation tool, referred as *Integrated Design Validation* (IDV). The novelty in this approach consists of three components, circuit complexity analysis, partitioning based on design hierarchy, and coverage analysis. The circuit complexity analyzer and partitioning decompose a large design into sub-components 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 of design validation and improves the coverage by further partitioning. Various simulation and verification 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: Formal Verification, Simulation, Digital Circuit Design

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), and formal verification resulting in a practical verification engine with reasonable runtime called the Integrated Design Validation (IDV) system.

2. METHODOLOGY

The IDV system utilizes existing simulation and verification techniques in an efficient manner of 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 used to provide a design compliance tool that is extremely effective as is illustrated with the example described in a later section of this paper.

The novelty in this approach is in the use of circuit complexity analysis and partitioning to decompose a large design into sub-components and validate the sub-components using different tools based upon known existing strengths of modern verification and simulation tools, new coverage analysis methods that compute the degree of design validation, using the result of coverage analysis as an indication for further validation iterations, and integration of 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, to our knowledge, no overall verification/simulation engine with significant analysis before design validation occurs has been produced.

The overall structure of the Integrated Design Validation (IDV) system is shown in the block diagram of Figure 1. A primary focus is on the complexity analysis, partitioning, and coverage analysis blocks, that are used to determine the most effective use of simulation or verification tools.

2.1 Complexity Analyzer

The complexity analyzer estimates the complexity of an RTL or netlist design based on existing methods for controller/datapath extraction.

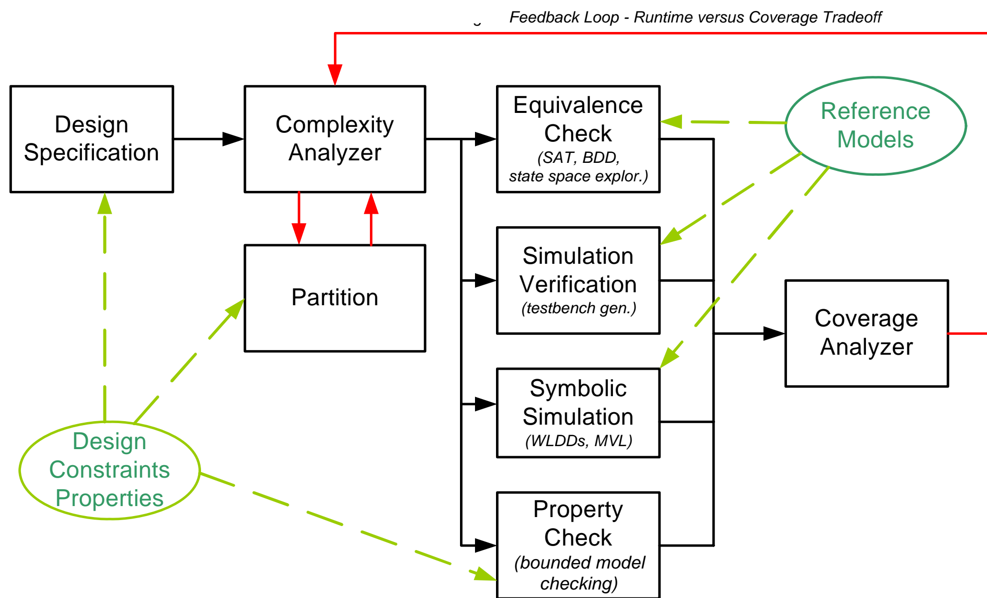

Figure 1 Architecture of the IDV

Integration with the partitioner is crucial for this function. The extracted control and datapath portions of the circuitry are analyzed for applicability of various techniques. One simple method is to send the same partitions of a circuit to more than one validation technique. As an example, a portion of a datapath is 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 an approach similar to this one in IDV.

2.2 Design Partitioning

One of the biggest hurdles in applying formal techniques is to correctly identify the target circuits.

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

Another technique allows for abstracting away large portions of the datapath circuitry leaving the controllers intact [13]. This technique uses 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 computations, 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.

Based on the factor that 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].

In IDV, design partitioning is mainly done manually with assistant of PM graph mentioned previously and design hierarchy. IDV Uses PM graph as a starting point and further partitions are accomplished manually by using finer grained designer-defined partitions that occur naturally in the hierarchy of the HDL code describing the system under validation.

2.3 Coverage Analysis

Some work has been accomplished 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.

Currently, we are focusing on vector coverage based on three factors: a) the coverage rate of the components related to the output when they are simulated or verified at the block level, b) the contributions or importance of each component related to the output, c) the vector coverage of the system-level simulation and the interconnection error. The first two points are easy to understand. The third point is related to the system level simulation and interconnection errors. Even if all related components are fully verified, the coverage for an output may not reach a perfect level of 100% since the interconnections may cause errors. System level simulation can be used to detect the presence of possible interconnection errors. The more interconnections, the more errors are possible. Also, the more system level simulation that is accomplished, the less possible interconnection errors present in a design.

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 Symbolic Trajectory Evaluation (STE)

STE is an approach similar to model checking that verifies circuits with very large state spaces. It is more sensitive to the property being checked instead of the size of the circuit when compared to model checking. The STE package we use is from the Intel Strategic Research Laboratory called *Forte*. It also supports a simple yet effective compositional theory besides 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 from the idea of multi-level and ternary-valued logic 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 it goes in the information ordering, the more information it has. The computational advantage of this is that, given the appropriate logical framework, if a property is proved to hold for a state in the lattice, it holds for 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 to be checked are represented using 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. As an example, consider a traffic signal control system with properties “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 asserted and *address* is specified, the output of memory is the value stored at *address* in the next cycle.

3.2 Symbolic Model Checking

TL as introduced in the above section can be used as a framework for the specification of the temporal properties of a design in forms other than a trajectory assertion. Computational TL (CTL) 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. A powerful algorithm to determine whether or not a given design satisfies a CTL is referred to as *model checking* [22]. In model-checking techniques, the entire state transition graph needs to be constructed either explicitly or implicitly 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. 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. A popular form of the model checking algorithm utilizes BDDs for state and transition function representation and is referred to as symbolic model checking. McMillan first formulated this approach and implemented the SMV method [24]. However, BDD-based model checkers can cause memory explosion since BDD sizes may exceed memory limits in large designs. The symbolic model checking tool used in 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 utilizes 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.

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 formulas supported by STE are 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 mechanism for VIS is a compact symbolic representation in term of BDDs representing the circuit model. The underlying engine for STE is symbolic simulation where the size of the internal BDDs are related more to the properties being verified instead of the circuit model.

Application: Based on the 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 Digital Logic Simulation

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 the simulation. Speed5 also has a fault simulation ability by fault model generation and insertion of those models 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 Automatic Equivalence Checking

Equivalence checking methods have led to significant success in industry. Two designs are functionally equivalent if they produce identical output sequences for all valid input sequences (i.e. a gate-level design matches its desired behavior as specified at the *Register Transfer Language* (RTL) level). Because of the computational complexity of formal equivalence checking, a design methodology typically adopts specific rules to make the problem tractable for large designs. In practice, the specification and implementation of a design often have a large degree of structural similarity in terms of internal nets that implement the same function. For example, equivalence checking can check if the designs have corresponding latches. Once the correspondence between latches of a reference design and an implementation has been discovered, equivalence checking is just a matter of showing that corresponding latches have the same next-state function. This has proven to be very valuable in validating that an implemented gate-level design matches its desired behavior as specified at the RTL level.

The equivalence checker developed in our group, SMU-EQ [11] performs quite well on large designs. The core part of the equivalence checking tool is image computation during state space traversal where conjunctive scheduling is very important in order to reduce BDD size during intermediate computations. In our approach, a genetic-based approach is developed to minimize total lifetime and active lifetime at the same time. Experimental results show that SMU-EQ is very effective. We have also incorporated a SAT engine into our equivalence checker to make it more robust and to handle larger designs. This allows SMU-EQ to operate using SAT when it is impractical to represent the entire transition relations with collections of BDDs.

3.5 SMU Functional Simulator

Our group also developed a functional simulator that is used for system level simulation. At 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, STE, SMU-EQ, or simulated by Speed5 before the functional simulator is invoked.

Based upon the tool sets just described, the prototype version of IDV being described here is given by the diagram in Figure 2.

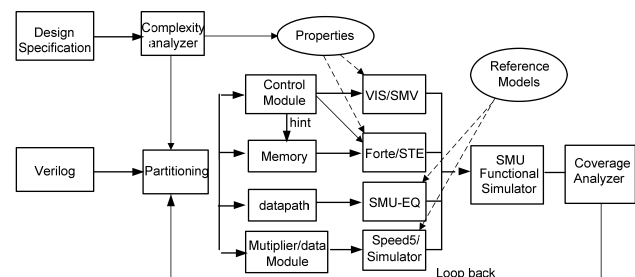

Figure 2 Prototype IDV Tool

4. VERIFICATION PROCESS

The IDV system is a constraint-based system. Constraints specify the system's operation such as what validation method will be used for each design module, the properties to be verified, and so on.

Figure 3 shows a validation flow chart for the IDV system. The circuit is parsed as a netlist either in blif or structural RTL format. The next block is the partitioning portion. The design hierarchy information is utilized for partitioning. This is reasonable since most current designs are created in a hierarchical format. After partitioning, based on the result of complexity analysis, all modules and corresponding constraints are supplied as input to appropriate validation engines for verification and/or simulation. The general rules are listed as 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 that will not cause memory explosion problems.

d. The Speed5 simulator is used to validate multipliers or other complex components not suitable for formal methods.

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

After the sub-modules are validated separately, the functional simulator is applied to simulate the system with the main focus on system interconnections. The coverage analyzer provides a degree of confidence metric for the design validation. When the coverage value is low, the components that are simulated in previous stages are further partitioned into smaller modules. The coverage for further partitioned component can be improved with formal verification methods or more simulation. The increased coverage on such components improves the overall coverage for the entire design.

In the following, we use an example to show the entire processing flow of the IDV system. The example we use is quite simple but contains all the necessary modules exercise the various verification and simulation methods incorporated into IDV. The example used here is an arithmetic inverse circuit that produces the inverse of an 8-bit unsigned integer, *B*, using the Newton-Raphson iteration equation:

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

A block diagram of the example digital system is shown in Figure 4. The system operates as follows: the initial value of x_0 is an estimate that is stored in a *ROM* lookup table for the arithmetic inverse of 1/*B* where *B* is the input unsigned 8-bit value. The 3-bit address is generated from the most significant bits of the integer *B* to get the initial value from the ROM. Then the value is input to the portion of the circuit that implements the above Newton-Raphson iteration equation, named *newtraph* in order to refine the approximation to the desired accuracy. The result of computation is fedback and iterates 5 times to produce an estimate of 1/B accurate to 6-bits before the result is output. The iteration and initial approximation selection operations are handled by the block labeled *controller*.

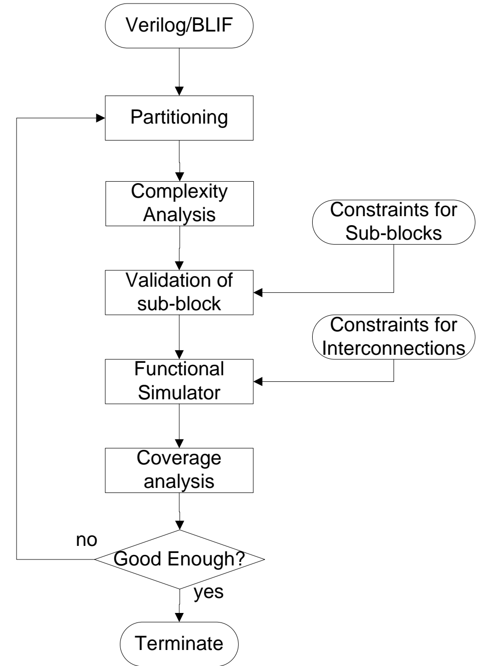

Figure 3 Processing Flow of IDV System

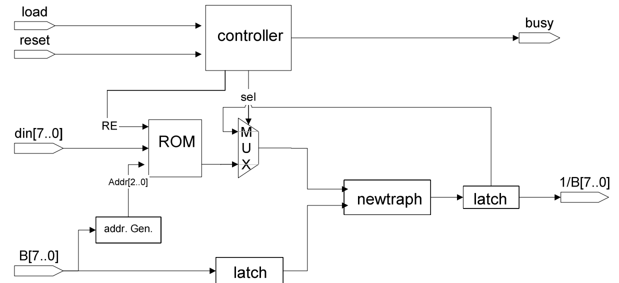

Figure 4 Inverse Circuit Block Diagram

The example circuit design includes a memory unit (ROM), a datapath (2 multipliers and 1 adder), control logic (a counter and synchronous sequential circuit controller), and some other datapath components such as an *address generator*, *latch*, and *multiplexer*. For such a design, the properties to be checked are as follows:

a. Liveness property: *load*=1 → *AX*:2(*AF*(*busy*=0)): Along all controller state-space paths in the future, there will be a state that busy=0 and it will be asserted for at least the next 2 states. The signal *load* =1 means an unsigned integer *B* is to be loaded for calculating the arithmetic inverse. The signal *busy* is asserted while the circuit is in the process of calculating the inverse and it is deasserted when the circuit is idle or the previous calculation is completed. This property indicates that if an integer is loaded for inverse calculation, it must finish the calculation sometime in the future and will not loop endlessly (busy will never be 0). This is a liveness property since it

indicates the constraint that circuit will finish a computation in a finite amount of time.

b. Safety property: *load*=1 → ***Next***(*busy*=1): If the signal *load* is asserted, then the signal *busy* has to be asserted in the next cycle. As we indicated previously, busy is asserted when the circuit is performing a calculation for the prior request. So this property ensures that if an integer is loaded, the calculation will be started in the next cycle.

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

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

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 assigned to VIS while properties specified as Trajectory Formulas are assigned to STE. Given the above example, property *a* is quite complicated and is not appropriate for an STE approach thus VIS is used. While property *b* can be verified via either VIS or STE, in such cases, we prefer using STE since STE can be more efficient. Also, all properties related to memory operations are assigned to the STE tool for formal verification since STE usually performs better for such components and the related properties can generally be expressed as trajectory formulas.

Deciding to use a SAT-based approach or BDD-based approach for equivalence checking based on the number of variables. A SAT-based approach can be more time consuming but can handle large designs where BDD-based approaches may lead to a memory explosion problem. BDD-based approaches are used when possible since they usually result in faster runtimes. Thus, if the number of variables is over a user-defined threshold, the IDV tool uses a SAT-based EQ method, otherwise the BDD-based approach is used.

b. Extracting components/modules/processes and information about their interconnection topology. This information is used for partitioning and system level functional simulation. A *Process-Module Graph* (PMG) is constructed that describes the hierarchy of the design under validation. Each node in the PMG represents a component/module/process and each edge corresponds to the interconnections of these components. The process-module graph for the above example is shown in Figure 5.

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 desired coverage goal is reached. This step is completed with the assistance of a PMG and the design hierarchy. Given the above example, the top-level consists of three parts, *controller*, *ROM*, and *datapath*. These three parts are extracted and assigned to different tools for verification or simulation. The *controller* is assigned to a property checking and reachability analysis tool, the *datapath* is assigned to an equivalence checker or simulator, and the memory unit is assigned to STE for property checking.

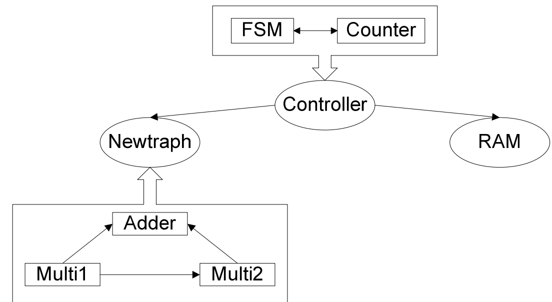

Figure 5 PMG Representation of the Example Design Hierarchy

4.3 Verification or Simulation process

After complexity analysis and partitioning are completed, the subcircuits undergo verification and/or simulation according to the process as depicted in the flowchart in Figure 3.

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 the required computational time for tools. These results were obtained using a Pentium-4 PC with 512MB of Memory running Microsoft WindowsXP under the cygwin UNIX emulation environment.

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

Different coverage metrics have been proposed in different tool sets. We are currently focusing on vector coverage and plan to expand to other coverage metrics. The vector coverage of an output is based on:

a. the coverage rate of the components related to the output when they are simulated or verified at the block level

b. the contributions or importance of each component related to the output

c. the vector coverage of the system-level simulation and the interconnection error

Detailed descriptions for the above three points are demonstrated as follows. The first point can be described as

follows. For each output of the design, not all components contribute such as the output signal *busy* which is only related to the *controller* component. If an output is related to the n components C_1 … C_n, each component has a corresponding normalized coverage value R_1 … R_n when they are verified or simulated at the block level. R_i is 1 (i.e. 100%) if the component has been fully verified or a value $R_i \in (0,1)$ that corresponds to the percentage of all possible vectors simulated. The coverage for an output will increase as the coverage for each related component increases. However, even if all related components are fully verified ($R_i = 1$), the coverage for the output may not reach a perfect level of 100% since the interconnection may cause an error such as the case where two interconnections are reversely connected.

Also, not all components related to an output have the same contribution. For example, the output z is related to all three components and it is the direct output of the datapath which in turn relates to the other two components. The contribution or "weight" of the component C_i to the output is denoted as w_i. Currently, IDV allows for two ways to determine the value w_i: (1) designers manually assign the weight value, (2) an automatic method based on the input distribution of the directly related component is computed by IDV. An example is used to demonstrate the automated method. For the inverse circuit example, the directly related component datapath has three inputs. One of the three inputs is provided by the circuit inputs, one of them is from the *controller* component, and one of them is obtained from the ROM component. The contribution of each component is proportional to the input distribution. The contribution of the component datapath is $1/3$, the contribution of the component controller is $1/3$, and the contribution of the component controller is $1/3$.

The third point is related to the system level simulation and interconnection errors. Even if all related components are fully verified, the coverage for an output may not reach a perfect level of 100% since the interconnections may cause errors. System level simulation can be used to detect the presence of possible interconnection errors. The number of possible interconnection errors is related to the number of interconnections. The more interconnections, the more errors are possible. Also, the more system level simulation that is accomplished, the less possible interconnection errors present in a design. Based on this description, a graph of the relationship between possible interconnection errors and the coverage at the system level is shown in Figure 6 where the parameter p indicates the dropping rate of interconnection errors with respect to the coverage at system level simulation.

Figure 6 shows the change in coverage of the system level simulation and presence of possible interconnection errors. When no system level simulation is performed, all possible interconnection errors are normalized to one. The possible interconnection errors presented in the system decrease as the coverage of system level simulation increases. Various slopes in Figure 6 show the different decreasing rates of the possible interconnection errors which is referred to as the dropping rate, p. No interconnection error is present in a design once 100% coverage is reached at the system level simulation. The dropping rate p is related to the number of interconnections and present in the interconnection architecture. The smaller p is, the faster possible interconnection errors are detected via system-level or functional simulation. The function in Figure 6 is referred to as the dropping function and denoted as $d(s,p)$ where the parameter s is the vector coverage at the system level simulation. Currently $d(s,p)$ is calculated as

$$d = (1 - s^p)^{1/p} \quad (1)$$

where p is determined by $\frac{\textit{number of interconnections}}{\textit{total inputs}}$.

Here only the number of interconnections is considered and the interconnection architecture is ignored.

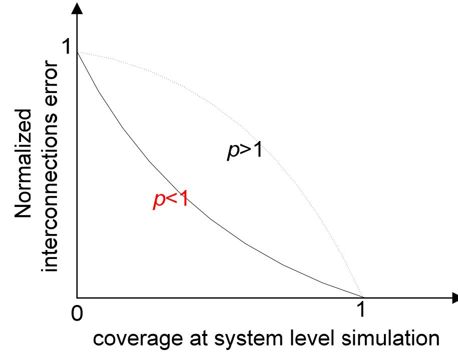

Figure 6 Interconnection Errors versus Coverage of System Level Simulation

An alternative way of describing $d(s,p)$ is that it defines the distance between the perfect situation (no interconnection error) and the imperfect situation due to interconnection errors. $1 - d(s,p)$ represents the confidence an output gains for interconnections with s system level simulations. If $d(s,p)$ is given, the coverage of an output can be calculated as $R \times (1 - d(s,p))$ considering interconnection errors. In an optimistic analysis, interconnection error is ignored. Thus, the coverage for components is adjusted with and without considering interconnection errors. The adjusted coverage rate is referred to as R' and is calculated as an interval value

$$R' = [P_{low}, P_{high}] \quad (2)$$

where $P_{low} = R \times (1 - d(s,p))$, $P_{high} = R$.

Based on the previous analysis, the total coverage rate for the output can be written as:

$$P_o = w_1 R_1 + w_2 R_2 + ... + w_n R'_n \quad (3)$$

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*. Among three components, *Controller* and *ROM* are fully verified and their coverage rates are $R_c = 1$ and $R_m = 1$ respectively. While the component *newtraph*, is simulated with 10% of total vectors being tested yields $R_d = 0.1$.

The output *Busy* is only related to the component *controller* and also the input *load*. Since the component *controller* totally controls the functionally of *Busy*, we have $w_c = 1$. Applying Eq. (3), we obtain:

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

indicating that the output *Busy* has been fully verified.

The output *1/B* is related to all three components. *1/B* is the direct output of the *datapath* component which accepts inputs from the system level input B, the output of the *ROM component* and the outputs of the *controller* component. Using this automatic method based on the input distribution, the contribution of the component controller is the contribution of the component datapath is $w_d = 1/3 = 0.33$, and the contribution of the ROM component is also $w_m = 1/3 = 0.33$. Next, the coverage for the *datapath* component is updated.

In the system-level simulation, 10% of the input values (vectors) for B have been simulated ($I_B = 0.1$). Among the three inputs for the *datapath* component, two of them come from the interconnected components yielding $p = 2/3$. According to Eq. (1), the dropping function $d(s,p)$ is written as $d = (1 - s^{2/3})^{3/2}$. Substituting I_B with S in the formula yields $d = 0.70$. This shows that with 10% of system simulation values of *B*, a 30% overall coverage confidence is obtained with the system interconnections accounted for.

The simulated vectors of the *datapath* component at the block level are 20%. Thus the adjusted coverage for the *datapath* is in the interval $R'_d = [0.06, 0.2]$ according to Eq. (2).

Then the coverage for the output *1/B* is calculated as

$$\begin{aligned} P_{1/B} &= w_c R_c + w_m R_m + w_n R'_d \\ &= 0.33 \times 1 + 0.33 \times 1 \times + 0.33 \times [0.06, 0.2] \\ &= [0.69, 0.74] \end{aligned}$$

The coverage of the output *1/B* with the initial partition is in the interval [0.69 0.74]. This coverage can be further improved by partitioning refinement and iteration in the IDV system.

4.5 IDV Iterative Refinement

When the coverage is too low, IDV iterates over more fine-grained partitions of the previously simulated components. In the above example, *newtraph* is the only component simulated. When IDV iterates and the component *newtraph* is repartitioned, it is found that it hierarchically consists of three smaller modules; two multipliers and one adder. The IDV tool formally verifies one of these three smaller modules. The equivalence checking tool is used to formally verify the adder and the two multipliers are simulated. After simulation, the coverage of the multiplier components 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 simulated at the same rate as before yielding $R_{mult1} = R_{mult2} = 0.3$. Thus applying Eq. 3 yields:

$$\begin{aligned} R_d &= w_{mult1} R_{mult1} + w_{mult2} R_{mult2} + w_{adder} R_{adder} \\ &= 1/3 \times (0.3 + 0.3 + 1) = 0.53 \end{aligned}$$

The result is that the coverage of the component, *newtraph*, has been improved from 20% to 53% by refining the partitioning. The coverage rate of component, *newtraph*, is increased by refinement.

$$R' = [0.16, 0.53]$$

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

$$\begin{aligned} P_{1/B} &= w_c R_c + w_m R_m + w_n R'_d \\ &= 0.33 \times 1 + 0.33 \times 1 \times + 0.33 \times [0.16, 0.53] \\ &= [0.72, 0.84] \end{aligned}$$

We can see that the coverage rate of the output is improved by partitioning refinement and IDV iteration.

5. CONCLUSION

The prototype IDV tool described here integrates various formal verification and simulation methodologies into a single design validation framework. The architecture of IDV is such that new verification and simulation tools can be easily integrated allowing for more choices of component validation. IDV has been applied to an example circuit design consisting of components that are well suited for both simulation and formal verification and the process of computing overall system coverage has been described with improvements obtained through iteration. This approach allows designers to trade-off IDV runtime for overall system coverage. This is one of the first approaches for loose coupling of validation techniques and the example has shown that this approach is successful.

6. 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