Automatic Constraint Based Test Generation for Behavioral HDL Models

Siva Kumar Sastry Hari, Vishnu Vardhan Reddy Konda, Kamakoti V, Vivekananda M. Vedula, *Member, IEEE*, and Kailasnath S. Maneperambil

***Abstract*—With the emergence of complex high-performance microprocessors, functional test generation has become a crucial design step. Constraint-based test generation is a well-studied *directed* behavioral level functional test generation paradigm. The paradigm involves conversion of a given circuit model into a set of constraints and employing constraint solvers to generate tests for it. However, automatic extraction of constraints from a given behavioral hardware design language (HDL) model remained a challenging open problem. This paper proposes an approach for *automatic* extraction of word-level *model constraints* from the behavioral verilog HDL description. The *scenarios* to be tested are also expressed as constraints. The *model* and the *scenario* constraints are solved together using an integer solver to arrive at the necessary functional test. The effectiveness of the approach is demonstrated by automatically generating the constraint models for: 1) an exclusive-shared-invalid multiprocessor cache coherency model and 2) the 16-bit DLX-architecture, from their respective Verilog-based behavioral models. Experimental results that generate test vectors for high level scenarios like pipeline hazards, cache miss, etc., spanning over multiple time-frames are presented.**

***Index Terms*—Behavioral models, constraint solvers, functional test generation (FTG), hardware description languages (HDL), processor architectures.**

I. Introduction

THE EVER-GROWING demand for greater performance, complex functionality, and faster time to market, coupled with the exponential growth in hardware size has resulted in the functional test generation (FTG) being widely acknowledged as the bottleneck of the hardware design cycle. The key focus of the present day FTG approach is to generate test vectors that can verify the complex functionality and *importantly the interaction* between multiple design units. The current practice is to generate millions of random test vector sets. The random test generation does not guarantee the coverage of all the functionalities, especially in the case of complex designs. This necessitates *directed tests*, that shall cover the corner cases not covered by the random tests. The interesting problem here is that the corner cases are no more bit values on specified wires/nets, but are much more abstract *scenarios* involving multiple clock cycles. For example, in a reasonably complex microprocessor a typical corner case can be *to generate an instruction which accesses a memory and a register such that the register access results in a data hazard and the memory access results in a page miss*. Specifying such *higher-level* corner cases to the test generator becomes extremely cumbersome at lower levels of abstraction. In addition, the crucial bottleneck with existing test generation tools is their scalability with larger designs. It is well-studied and reported in the literature that for a tool to be scalable with larger designs, it is important to handle the design at higher levels of abstraction, typically at the behavioral level. This explains the need for a directed behavioral level functional test generation (DBFTG) tool. Unlike many design tasks like logic synthesis or place and route that have been automated with sophisticated tools, functional verification has remained largely as a manual process. Languages like Verilog [1] enable the designer to specify the model at the behavioral level. The hardware verification languages (HVLs) [2] are capable of working in synchrony with the behavioral level models. This can automate generation of test benches. However, configuring the HVL environment as a DBFTG requires a significant amount of manual effort. In order to reduce the manual intervention, there is a need for a tool that is capable of generating test vectors from a given behavioral level description of the design under test (DUT) and a *higher-level test specification*. This paper proposes a constraint-based DBFTG technique that addresses the previous issue.

II. Previous Work

Previous work reported in the literature for the DBFTG may be broadly classified into two, namely, the conventional automatic test pattern generation (ATPG) [3]-based approaches and the constraint-based approaches. Classical ATPG methods [3] work at gate-level representations of the design and hence exhibit less scalability with increasing design size. Some of the recent ATPG-based techniques consider behavioral level design models as inputs. DBFTG techniques using ATPG are reported in [4] and [5]. These papers also present a comprehensive survey of the techniques reported prior to them. The framework presented in [5] converts a given HDL representation to an assignment decision diagram (ADD) representation and uses a modified ATPG algorithm, called the RTL ATPG to create functional tests. However, no results were reported for processor level circuits. The other issue is to express the test scenario to the tool. It is not clear from [5] of how to specify a *scenario* that spans across multiple time frames. These types of scenarios arise specifically during verification of complex interactions between the modules of the processor.

Manuscript received October 7, 2006; revised July 7, 2007. This work was supported under the IIT Madras-Intel joint research initiative on functional test generation for processor architectures.

S. K. S. Hari, V. V. R. Konda, and K. V are with the Reconfigurable and Intelligent Systems Engineering Group, Department of Computer Science and Engineering, Indian Institute of Technology, Madras 600036, India (e-mail: hsiva@cs.iitm.ernet.in; vishnu_k@cs.iitm.ernet.in; kama@cs.iitm.ernet.in).

V. M. Vedula and K. S. Maneperambil are with the Validation and Test Solutions, Intel Corporation, Austin, TX 78746 USA (e-mail: vivekananda.vedula@intel.com; kailasnath.s.maneparambil@intel.com).

Digital Object Identifier 10.1109/TVLSI.2008.917424

Constraint-based FTG approaches reported in [6]–[14], model the given circuit description as constraints and use constraint solvers to generate the required functional tests. There are two approaches to constraint modeling, namely, 1) *to model using Boolean constraints* [that use a Boolean satisfiability (SAT) solver] and 2) *to model using integer constraints* (that uses an integer solver). A Boolean circuit can be encoded as a satisfiability equivalent conjunctive normal form (CNF) formula using the method of [15]. In SAT procedures, the CNF representation facilitates powerful methods to prune the search space based on conflict analysis. However, since practical gate-level circuits can be quite large, dealing with substantially large CNF formulas results in unacceptable CPU runtime. Another approach is to solve the Boolean satisfiability problems based on binary decision diagrams (BDDs) [16], [17]. The major limitation is that BDD-based methods like BSAT require excessive time/memory to create BDDs for the test circuits. A detailed survey of recent advances in SAT-based formal verification is presented in [18]. Regular expression algebra-based techniques for identifying relevant paths that can be sensitized by test vectors in a precomputed test set to test modules in a circuit model in which the control and the data paths are separated are presented in [19] and [20]. The previous technique is efficient only for circuit models in which: 1) the data and control paths are separate and 2) have design for testability (DFT) [3] support. A SAT-based ATPG technique for non-separable control-datapath circuits is presented in [21].

Techniques that employ word-level reasoning for FTG are reported in the literature. The technique presented in [22] proposes a functional vector generation method for RTL models using word-level constraint logic programming based on assertions. Constraint propagation techniques across different domains, that is, (both arithmetic and Boolean domains) have been explored to generate functional tests and high level ATPG vectors on HDL descriptions [23]–[25]. A hybrid ATPG-based modular arithmetic constraint solving technique for assertion checking is proposed in [26]. A hybrid satisfiability approach (HSAT) to generate functional test vectors for RTL design is proposed in [27]. This approach was unified in [28] that yields a single linear constraint problem instance. The approach presented in [28] is not scalable for SAT instances with large portions of sequential logic. The ideas discussed in [29]–[32] use constraint solvers to generate tests for functional verification of higher-level architectural features. One of the recent results reported in literature dealing with micro-architecture verification [33] concentrates on using a generic-test-plan (GTP) approach to generate tests. However, the paper does not explain how to generate directed tests that verify specific micro-architectural features. A comprehensive survey of many test generation techniques for processor verification is presented in [33]. Employing genetic algorithms to evolve efficient test benches for behavioral level circuit models is studied in [34] and [35]. A unified framework for functional verification of processors using constraint solvers was proposed in [36]. This framework did not address automatic constraint generation from circuit models.

This paper presents a fully-automated framework to generate directed tests for functional verification of any digital system, specifically microprocessors. The framework can be used to verify the functionality at different levels of the processor architecture, namely, the instruction set architecture to micro-architecture, and also combinations of them. The proposed methodology accepts as input a behavioral level Verilog model and converts it into an ADD [4], [37]-based data structure called the assign-always-module (A^2M) graph. The A^2M graph is further *optimized* and converted into a set of integer constraints. These constraints are called the *model constraints*. The salient feature of the methodology is that the previous steps are *fully automated*, reducing the human effort significantly (almost nil). The *scenario* for which a test has to be generated is also modeled as constraints, namely, the *scenario constraints*. Both the *model* and *scenario* constraints are together solved using an integer constraint solver to generate the required test. Importantly, the *scenario* may span across several clock cycles or time frames. The salient features of the proposed technique are outlined in the following.

A. Input to the Proposed Technique is a Behavioral Level HDL Model

The behavioral level models are not only lesser in size but more abstract than their corresponding gate level models. The behavioral model expresses the system functionality more explicitly than their corresponding gate level representation. This implies that the constraints generated from behavioral descriptions capture the design functionality more comprehensively than those generated from gate level descriptions. This is crucial for functional level test generation.

B. Automatic Generation of the Constraint Model From the Behavioral Model

Extraction of constraints from a behavioral or register transfer level (RTL) description is a challenging problem [21]. The proposed methodology does solve the problem comprehensively.

C. Word-Level Constraints in Contrast to Bit-Level Constraints

The proposed methodology deals with word-level constraints and employs an integer solver. This has the following advantages.

- RTL-based design methodologies are widely used to model the expected behavior of an integrated circuit before the actual circuit is fabricated. Depending on the nature of the intended circuit, an RTL model may contain variables of varying width. Some of these variables, such as those representing data operands, are used in arithmetic operations that have a regular structure and meaning in the integer domain. Test generation approaches that use CNF-SAT and ATPG operate at the Boolean level and require that the RTL model be flattened. This results in loss of the regularity in the arithmetic operations that could have been leveraged while reasoning about these operations for test generation. Thus, the *word-level reasoning* is best suited for the DBFTG.
- Integer solvers are better suited for word-level reasoning than the SAT solvers. While the control portion of the design lends itself well to Boolean-level reasoning, word-level reasoning can be used wherever possible to improve the efficiency of the overall test generation. The intuition behind improved performance is that the Boolean SAT and conventional ATPG techniques are *NP-complete in the number of bit-level variables*, whereas, word-level solving

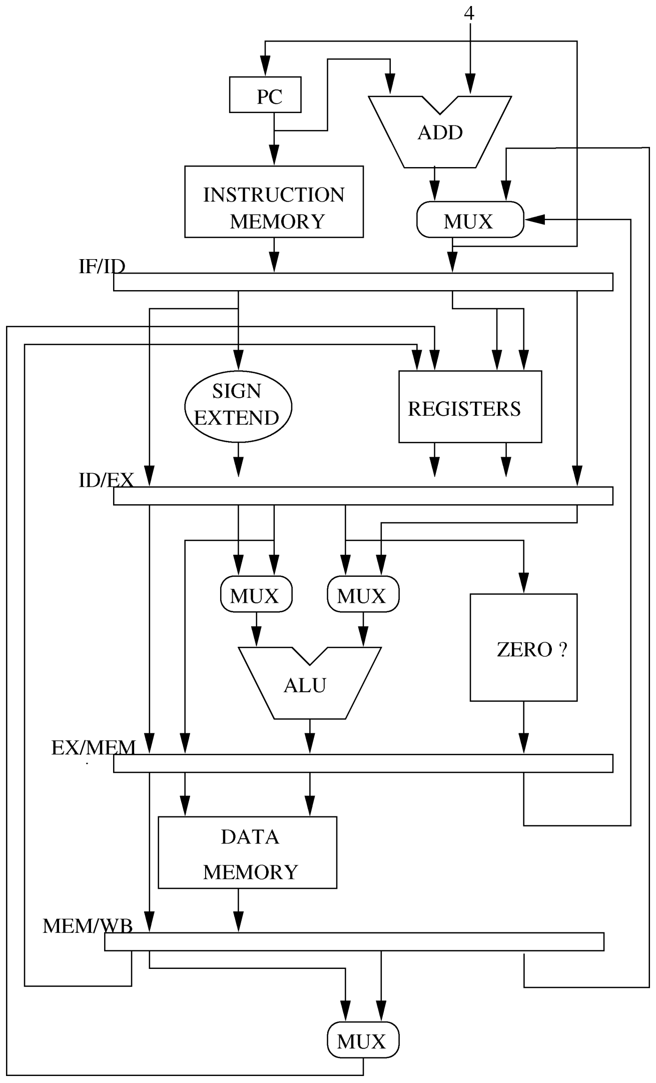

Fig. 1. DLX architecture.

complexity is *NP-complete in the number of word-level variables, which grow less dramatically with increasing design functionality.*

D. Single Constraint Model With Unified Control and Data Paths

Unlike some of the previous approaches, the proposed technique does not demand a separation between control and data paths.

E. Handling Sequential Designs

Most integer level constraint solvers are not built to handle sequential behavior of circuits. A set of techniques have been devised in this paper that converts sequential RTL description to constraint models.

F. Scalability With Increasing Design Size

The number of model constraints generated grows linearly with the size of the underlying A^2M graph.

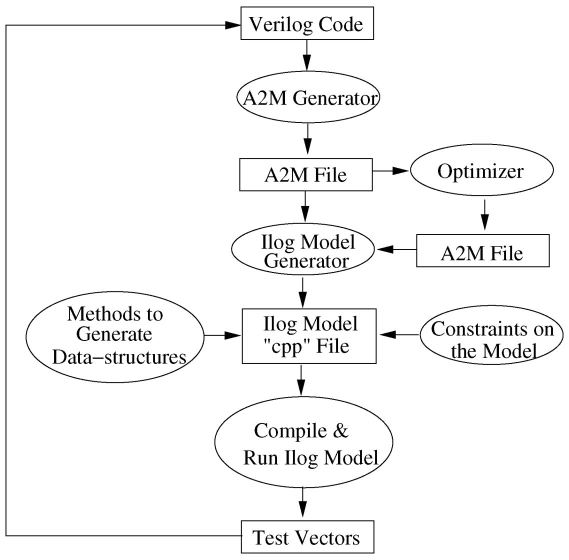

Fig. 2. Tool flow.

G. Verifying Complex Scenarios

The proposed technique generates test vectors automatically that can verify specific module interaction issues that can go over multiple time frames.

The proposed methodology is used to verify the behavioral description of the exclusive-shared-invalid (ESI) multiprocessor cache coherency protocol model [38]. The description of the previous experiment illustrates to the reader the different aspects of generation of both *model* and *scenario* constraints, specifically, for scenarios that *span across multiple time frames*. In addition, The proposed methodology is employed on the well-known DLX architecture [38]. The DLX architecture has a five-stage pipeline. These stages are instruction fetch, instruction decode/register fetch, execute/address calculation, memory access, and write back. The architecture provides a good platform to test the proposed methodology as it offers a variety of interesting and complex micro-architectural features and scenarios such as hazards, arithmetic logic unit (ALU) operations, etc. The architecture is shown in Fig. 1. The code snaps that are presented throughout this paper for explanation of the different concepts are taken from the Verilog description of the DLX architecture.

III. DBFTG Approach

Fig. 2 presents the complete design flow for the proposed approach. The approach has three major phases as listed in the following:

1) automatic generation of the A^2M graph from the given behavioral HDL model and its optimization;
2) automatic generation of the model constraints from the optimized A^2M graph;
3) modeling of scenario constraints and generation of the functional test.

The following subsections explain the previous three phases in detail.

A. A^2M Generation and Optimization

In Fig. 2 the A^2M generator parses the input Verilog code and hierarchically creates the A^2M graph. The synthesizable part of the Verilog code that describes the behavior of a circuit has three major structures, namely, the `Always block`, the `Assign (continuous) statements`, and the `Module instantiations` [1]. Hence, the acronym A^2M. The A^2M graph essentially has four entities, namely, `Module`, `Component`, `Signal`, and `ADD Node`. Every *module definition* in the Verilog code maps on to a `Module` entity of the A^2M graph. Every *always*, *assign*, and *module instantiation* statements in the Verilog code map on to a `Component` entity of the A^2M graph.

The `Module` entity contains the various fields necessary to encapsulate a module definition in Verilog. They include `Name` of the module, an array of all the `Components` (always, assign, and module instantiation statements) inside the module, an array of all the `Signals` (all `reg` and `wire` Verilog variables including the `input` and `output` variables) inside a module, an array storing all the indices of entries in the previous `Signal` array corresponding to the `input` to the module, and an array storing all the indices of entries in the previous `Signal` array corresponding to the `output` to the module.

The `Component` entity contains the various fields required to encapsulate the three types of structures, namely, a module instantiation, an assign statement, and an always block. Two additional `Component` types, namely, the `feedback component` and the `stem component` are defined to handle the *loops* and *stems*, respectively, inside the design. The *stems* are signals that drive multiple components. The various fields inside a `Component` are `name` and `type` of the Component, pointer to the `Parent Module` of the Component (the Verilog module inside which the Component is defined), `fan-ins` (input signals to the component), `fan-outs` (output signals of the component), and the `ADDNode` corresponding to the Component. The `ADDNode` captures the functionality of the `Component` and is defined only for the `Components` that corresponds to an `always` block or an `assign` statement. The ADD framework is an internal representation of the HDL description and has been shown to be *complete and efficient*. More details on ADD are available in [4], [5], [37], and [39].

The `Signal` entity contains the various fields required to encapsulate a signal inside a Verilog module. The signals connect the different `Component` entities in the A^2M graph. The different fields inside a `Signal` are `name` and `type` of the signal, the `source component` of the signal, the `destination component` of the signal, the `size` of the signal (if it is a vector), and the `constant` value (if the signal is assigned to a constant in the Verilog module).

The proposed approach is implemented for Verilog-based HDL models. However, the tool is extendable to other HDLs like VHDL. The code of the Verilog parser is written in the `yacc` language which captures the grammar of Verilog and uses the syntax directed translation to construct the A^2M graph.

The A^2M graph generated before is optimized further as follows. The optimization stage refines the A^2M graph by two levels so as to remove the `Component` entities that shall cause redundant constraints in the integer model to be generated in the next phase. The first level of refinement is the elimination of the `stem components` from the graph which is a redundant information for the constraint model, as all the branches driven by the *stem component* can be replaced by a single variable. Thus, the `stem components`, if kept in the A^2M input graph shall result in more numbers of constraints and in turn reduces the performance of the solver. However, if the test scenario assumes that a branch of a particular `stem component` s is faulty, then s may be retained. The second level of the refinement is the removal of the nodes in the A^2M graph corresponding to *redundant constants*, *redundant operators*, and *copy statements*. For example, the `ADDNode` corresponding to $b \leftarrow c \,\&\, 1$ can be replaced by the one corresponding to $b \leftarrow c$, thereby eliminating the redundant constant 1. Now, the statement $b \leftarrow c$ is a *copy statement*, where b is the `target` signal and c is the `original` signal. This copy statement can be further eliminated by replacing the `target` signal by the `original` signal. These refinements reduce the number of constraints and the variables in the subsequent constraint model, hence improving the performance of the solver.

B. Model Constraints Generation

This section deals with different Verilog constructs, their corresponding A^2M representations and their equivalent constraint models. This in turn, shall explain the *automatic* generation of the *model constraints* from the given behavioral Verilog model. The standard ILOG [40] constraint solver was used for experimentation. Hence, a ILOG-type syntax is used to illustrate the example model constraints in this paper.

1) `reg` *and* `wire` *Variables:* Two types of variables are commonly used in Verilog models, namely, the `reg` and `wire` variables [1]. These variables can either be *bits* or integers (*bit-vectors*). The integer domain deals more with *bit-vectors* rather than individual bits. This in turn, increases the modeling complexity.

Any *bit-vector* of size n in the input behavioral model is mapped on to a `Signal` entity in the A^2M graph with size $= n$. This is treated as an integer variable ($IloIntVar$ in ILOG) whose value ranges from 0 to $2^n - 1$. The *bit* variables are also treated as *bit-vectors* of size $n = 1$.

The `reg` and `wire` variables in a Verilog expression map on to one of the two types of nodes on the A^2M graph, namely, the *read nodes* and the *write nodes*. As the names suggest, the variables on the right-hand side (left-hand side) of a Verilog expression map on to read (write) nodes. The read and write nodes in the A^2M graph have two attributes attached to it, namely, the $range$ and the $index$. The need for these attributes arises from the fact that a vector can be referred in the following three ways in a Verilog code.

- *Bit Select*: For example, $ir[3]$, that selects the third bit of the bit-vector ir. The respective node in the A^2M graph has $index = 3$ and $range = [-1 : -1]$. Since, the integer solver do not directly deal with bits, the corresponding ILOG representation of $ir[3]$ is

$$2^3 * (IloDiv(ir, 2^3) - 2 * IloDiv(ir, 2^4))$$

where `IloDiv` stands for integer division in ILOG.
- *Part Select*: For example, $ir[16 : 13]$, that selects the four bits starting from bit 13–16 of the bit-vector ir. The re-

spective node in A^2M graph has $range = [16:13]$ and $index = -1$. The corresponding ILOG representation is

$$2^{13} * IloDiv(ir, 2^{13}) - 2^{17} * IloDiv(ir, 2^{17}).$$

- *Entire Vector*: For example ir. The respective node in A^2M graph has $index = -1$ and $range = [-1 : -1]$. The ILOG representation models the variable ir as a normal integer ($IloIntVar$).

The variables defined inside a module M carry different values for different instantiations of M. To encapsulate this, every variable in M is declared as an array of size equal to the number of instantiations of M. For example, given that the variable ir is defined in M and that M is instantiated twice, say, instantiation 0 and instantiation 1, $ir[0]$ ($ir[1]$) denotes the value of ir in instantiation 0 (1). On similar lines, the third bit of ir in instantiation 0 is modeled as

$$2^3 * (IloDiv(ir[0], 2^3) - 2 * IloDiv(ir[0], 2^4)).$$

The part select of ir comprising of the four bits (bits 13–16) in instantiation 1 is modeled as

$$2^{13} * IloDiv(ir[1], 2^{13}) - 2^{17} * IloDiv(ir[1], 2^{17}).$$

2) Operators: Integer solvers do not support certain operators that are provided by the HDLs like bit-wise operators, concatenation operation [1], etc. The proposed technique handles these operators by modeling them as functions described in the following.

- *Bitwise Operators*: Consider the following Verilog statement that uses the AND bitwise operation:

$$out = in\_1 \ \& \ in\_2$$

where out, in_1, and in_2 are all one bit Verilog variables. The corresponding A^2M graph is shown in Fig. 3(a). The nodes labelled in_1 and in_2 are the *read nodes*. The node labelled out is a *write node*. The node labelled & denotes the bitwise AND operator. The triangle is a *decision node* [37] that assigns the output of the operator node to the write node *if the input condition is true*. In this case, the input condition to the decision node is always set to be *true* (node labelled 1). The A^2M structure is converted to the following constraint by a function that is called on encountering a bitwise operator node

$$(out == ((in\_1 == 1) \&\& (in\_2 == 1)).$$

The previous method is extended to handle the case, wherein, in_1, in_2, and out are vectors. Similarly all other bitwise operators are modeled as constraints in the integer domain.

- *Logical Operators*: Logical operators are of two types, namely, Boolean and comparison operators.
  1) *Logical Boolean Operators*: This includes logical AND, logical OR, etc. These are handled similar to the bitwise operators.

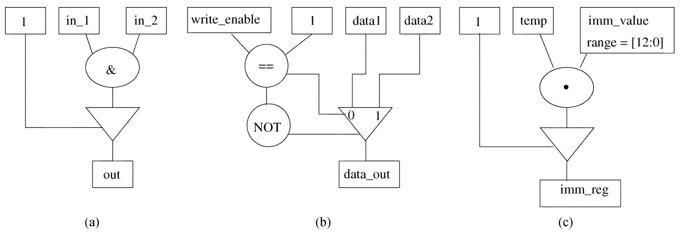

Fig. 3. ADD elements of A^2M graph. (a) Bit-wise operator. (b) Logical operator. (c) Concatenation operator.

  2) *Logical Comparison Operators*: Consider the following Verilog code:

$$\text{if}\,(write\_enable == 1)\ data\_out <= data1$$
$$\text{else}\ data\_out <= data2.$$

  The corresponding A^2M graph is shown in Fig. 3(b). The previous structure is modeled as the following constraint:

$$data\_out == (write\_enable == 1) * data1 + (write\_enable == 0) * data2.$$

  The previous constraint states that $data\_out$ is equal to $data1$, if $write\_enable == 1$ else it is $data2$.

- *Concatenation Operator*: The concatenation operator is modeled using simple arithmetic operations. For example, consider the Verilog statement

$$imm\_reg = \{4\{sign\}, imm\_value\}$$

where imm_reg is a 16-bit register, imm_value is a 12-bit wire, and sign is a 1-bit wire. The corresponding A^2M graph is shown in Fig. 3(c). The corresponding model constraints are as follows:

$$temp == 2^3 * sign + 2^2 * sign + 2^1 * sign + 2^0 * sign$$
$$imm\_reg == imm\_value + temp * 2^{13}.$$

3) Modeling Sequential Circuits: Every sequential circuit can be represented by the conventional Huffman model [3]. The combinational and the sequential parts are clearly distinguished in this representation. To model the circuit in the integer domain the following two basic principles are used:

- each sequential element is a variable in the integer domain;
- each combinational element produces a constraint on its inputs and outputs in the integer domain.

The behavior of a sequential circuit S over k time frames can be modeled as a combinational circuit using the conventional time frame expansion approach, which unrolls the combinational part of S, k times [3]. The previously mentioned approach for modeling in the integer domain is illustrated by using an example of a counter. The following Verilog code models a counter.

```
reg [4 : 0] counter;

always @ (posedge Clk)
counter = counter + 1;
```

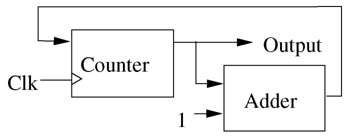

Fig. 4. `Counter`.

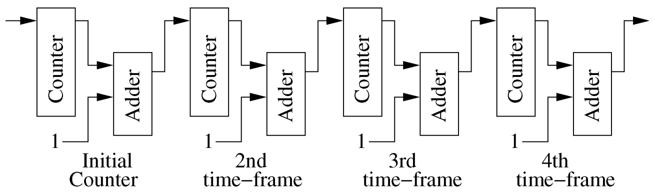

Fig. 5. Time frame unrolled counter.

The Huffman Model of the counter is shown in Fig. 4. The register labeled `Counter` forms the sequential part and the `Adder` is the combinational part. It is easy to infer that at every clock (`Clk`) pulse the value stored in the `Counter` gets incremented by 1.

The unrolled model of the counter is shown in Fig. 5, wherein, the circuit is unrolled over three time frames. The Counter blocks shown in Fig. 5 are just wires. Note that by assigning, say, five to the `Initial Counter` wires in Fig. 5, Figs. 7–9 are output in `Counter` blocks representing the 2–4 time frames, respectively. This indeed captures the functionality of the counter over three time frames.

The register `Counter` in Fig. 4 is a reg variable, $counter$, in the Verilog description. Without loss of generality, let there be only one instantiation of the module containing the $counter$ variable. Therefore, the variable shall be denoted by $counter[0]$ in the corresponding A^2M graph. To unroll a circuit over time frames, another dimension is added to the variables to represent the time frame. Thus, the variable $counter[0][j]$, denotes the variable $counter$ in the jth time frame. The combinational part of the Huffman Model is the one that sets constraints on the variable $counter[\,][\,]$ across time frames. The Verilog code and the underlying A^2M graph imply the following constraint:

$$counter[0][j] == counter[0][j-1] + 1.$$

Thus, the clock in Verilog is realized as a time frame in the corresponding constraint model.

4) Assign Statement: Given that a circuit is unrolled for max_tfs time frames, the assign statement in Verilog leads to constraints on the variable for all time frames. For example, the assign statement

$$assign\ sum = input\_1 + input\_2$$

inside the kth instantiation of a module M shall lead to the following set of constraints:

$$\begin{aligned}&\mathtt{for}\ (\mathtt{tf} = 0; \mathtt{tf} < \mathtt{max\_tfs}; \mathtt{tf}++)\\ &\qquad (sum[k][tf] == input\_1[k][tf] + input\_2[k][tf]).\end{aligned}$$

5) Always Statement: The constraint model for an always construct in Verilog depends on the *event* in its *sensitivity list.* The event can be either *clocked* or *non-clocked* and is represented by an event node in the corresponding A^2M graph. The following code is a clocked always structure:

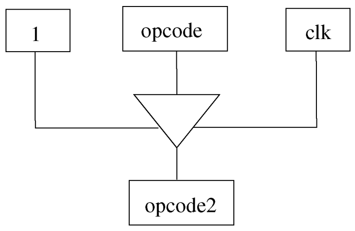

Fig. 6. Always statement with clk.

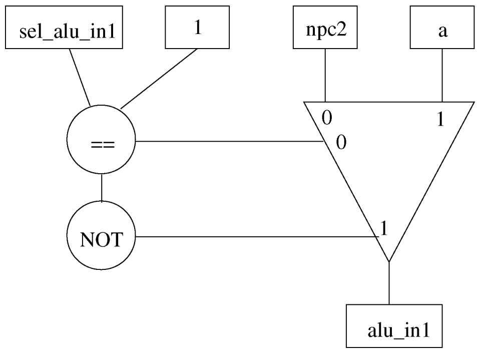

Fig. 7. Always statement without clk.

$$\begin{gathered}always\ @(posedge\ clk)\\ opcode2 <= opcode.\end{gathered}$$

The A^2M graph for the previous case is shown in Fig. 6. The model constraint for the previous A^2M graph for the kth *instantiation* of the module to which it belongs to will be

$$(opocde2[k][tf+1] == opocde[k][tf]).$$

The following code is a non-clocked always structure:

$$\begin{aligned}&always\ @(sel\_alu\_in1\ or\ npc2\ or\ a)\\ &\text{if}\,(sel\_alu\_in1 == 1)\ alu\_in1 <= npc2;\\ &\text{else}\ alu\_in1 <= a.\end{aligned}$$

The A^2M graph for the previous case is shown in Fig. 7. The model constraint for the previous A^2M graph for the kth *instantiation* of the module to which it belongs to will be

$$\begin{aligned}(alu\_in1[k][tf] == (sel\_alu\_in1[k][tf] == 1) * npc2[k][tf]\\ +(sel\_alu\_in1[k][tf] == 0) * a[k][tf]).\end{aligned}$$

6) Value Remembrance: The major challenge in modeling Verilog constructs is the modeling of the memory elements. The Verilog statements do not explicitly state that the value has to be remembered to the next time frame. But, the modeling of constraints should be in such a way that the values are carried forward appropriately. The basic assumption in Verilog is that if a variable is set under clock edge then it is a flip-flop. If a value is not written in the flip-flop in a particular clock cycle due to some condition C, then the flip-flop retains the old value in the next time frame. For example, consider the following code snap from the data memory description of the DLX model:

$$\begin{gathered} always\ @(posedge\ clk) \\ \text{if}\,(write\_enable == 1) \\ data0 <= data\_in \end{gathered}$$

where $write\_enable$ is the enable signal, $data0$ is the first address line, and $data\_in$ is the data input to the data memory. The corresponding model constraint is as follows:

$$\begin{gathered} data0[k][tf+1] == \\ (write\_enable[k][tf] == 1)*data\_in[k][tf] \\ +(write\_enable[k][tf] == 0)*data0[k][tf]). \end{gathered}$$

The previous constraint takes care of the fact that the old value of $data0$ is taken forward to the next time frame if $write\_enable$ is 0.

7) Module Instantiation: Two major points have to be addressed to handle module instantiations, namely, handling the variables and addition of interface constraints.

Handling Variables: The Verilog design description is hierarchical and hence different modules can have variables with the same name. While deriving the constraint model, the variables with same names in different modules have to be distinguished. To resolve this, a variable `var` inside a module `module_name` is referred to as `module_name_var` in the constraint model. Given that two module definitions do not have the same name solves the previous issue.

As mentioned earlier, every variable of the Verilog model is converted to an array in the constraint model. For example, the variable `out` in the *second* instantiation of the module `alu` in time frame tf is specified in the constraint model as `alu_out[1][tf]`. While generating the *scenario constraints* the user needs to specify variables corresponding to a *particular* instantiation of a module. This demands a mechanism that simplifies the specification process. Specifically, the user should easily *infer the index* into the variable array corresponding to a particular instantiation. This is achieved as follows.

Whenever a module M is instantiated with a name, say, M_inst, the size of the arrays corresponding to the *wire*, *reg*, *input*, and *output* variables defined inside M increases by one. This incremented size S also denotes the number of times M is instantiated till that point. The hierarchical name of M_inst is used as a #`define` constant with value $S-1$ in the constraint model. Given that the hierarchical name of every instantiated module is different [1] ensures that there are no conflicts. In the constraint model the user can use the hierarchical name of the instantiated module as an index to specify any variable inside it.

For example, in the DLX processor design there are module definitions by the name dlx and alu. The module definition of alu is as follows

$$module\ alu(out, clk, in1, in2, opc).$$

The dlx module instantiates the alu with the name $alu1$

$$alu\ alu1(o1, clk, i1, i2, op1).$$

The dlx module, in turn, is instantiated by the top module of the Verilog hierarchy with the name $Dlx1$. Note that the hierarchical name of $alu1$ is top_Dlx1_alu1, which is a #`define` constant in the constraint model. The variable out in the instantiation $alu1$ of alu in the time-frame tf is specified as

$$alu\_out[top\_Dlx1\_alu1][tf].$$

Interface Constraints: These constraints are used to establish connections between the instantiating and the instantiated modules. The number of interface constraints is equal to the number of input and output ports of the instantiated module. For example, given that the circuit is unrolled over max_tfs time frames, the model constraints generated to effect the interface between the previous alu instantiation inside dlx is as follows.

$$\begin{aligned} &\texttt{for } (\texttt{j} = 0; \texttt{j} < \texttt{max\_tfs}; \texttt{j}++)\{ \\ &\quad alu\_out[top\_Dlx1\_alu1][j] == dlx\_alu\_o1[top\_Dlx1][j]); \\ &\quad (alu\_in1[top\_Dlx1\_alu1][j] == dlx\_alu\_i1[top\_Dlx1][j]); \\ &\quad (alu\_in2[top\_Dlx1\_alu1][j] == dlx\_alu\_i2[top\_Dlx1][j]); \\ &\quad (alu\_opc[top\_Dlx1\_alu1][j] == dlx\_op1[top\_Dlx1][j]); \\ &\} \end{aligned}$$

As mentioned earlier, as the clock in Verilog corresponds to time frames in the constraint model, no constraint is generated for the `clk` port above.

C. Automatic Constraint Model Generation

The previous sections explained the mapping between the corresponding A^2M graph to the integer constraints for different Verilog constructs. As mentioned earlier, the conversion from Verilog to the equivalent A^2M graph is done automatically using a syntax-directed translation. This section presents the algorithm *i-Gen* (refer Algorithm 1) that automates the conversion from the A^2M graph to a constraint model. Before proceeding further, the following data arrays used by the algorithm are defined.

- $module\_declaration$ Array: An array of lists, one list per every module declaration, storing the module name, and input-output ports of the same.
- $list\_modules$ Array: An array of lists, one list per every module instantiation, storing the module name, instantiation name, parent module name, and list of input-output ports. This list is used for generating *interface constraints* due to a module instantiation, as described earlier.
- $module\_vars$ Array: An array of lists, one list per every module declaration, storing the name and size of every signal declared inside the module.

Algorithm 1 The i-Gen Algorithm

```
1: igen ($A^2M$)
2: begin
3: read_block()
4: **while** $block = ModuleDeclaration$ **do**
5:     $update\_module\_declarations(name, port\_list)$
6:     **if** $current\_module = top\_module$ **then**
7:         $update\_list\_modules(instantiation\_name,$
           $parent\_name, module\_name, port\_list)$
8:     **end if**
```

```
9:      read_block()
10:     while $block = Signals$ do
11:       $update\_module\_vars(name, size)$
12:       read_block()
13:     end while
14:     while $block = Component$ do
15:       if $Component = MODULE$ then
16:          $update\_list\_modules(instantiation\_name,$
             $parent\_name, module\_name, port\_list)$
17:       else if $Component = STEM$ then
18:          Add constraints such that fan-outs are equal to
             fan-in
19:       else
20:          $generate\_constraints\_always\_assign()$
21:       end if
22:       read_block()
23:     end while
24: end while
25: for all element of $list\_module$ do
26:     $generate\_connection\_constraints()$
27: end for
28: end
```

The working of the algorithm *i-Gen* is described as follows. The routine $ttread\_block$ used by *i-Gen*, reads a `block` from the input A^2M graph. This `block` is either a `module declaration`, `Signal`, or `Component`. The first `block` read (in step 3 of *i-Gen*) from the A^2M graph will always be a `module declaration` corresponding to the topmost module of the Verilog hierarchy. Therefore, the condition for the `while` loop in step 4 is true for the first time. The routine `update_module_declarations` in step 5 updates the $module\_declaration$ array described before. As the topmost module of the Verilog model is not explicitly instantiated, steps 6 and 7 update the $list\_modules$ array for the current top module declaration. Steps 10–13 update the $module\_vars$ array described before on encountering a `Signal` block. Steps 14–23 handle a `Component` block. As mentioned earlier, a `Component` in the A^2M graph corresponds to either a module instantiation, a stem or an always/assign statement of the input Verilog code. Steps 14–16 handle the case when the `Component` is a module instantiation by updating the $list\_modules$ array. Steps 17 and 18 handle the case when the `Component` is a stem by adding constraints such that the variables corresponding to the fan-out branches are equal to the variable corresponding to the stem. Step 20 is executed when the component is of type *always/assign*. In this case, the function `generate_constraints_always_assign()` will generate the necessary constraints for the corresponding always/assign statement as described in the previous section. Steps 25–27 generate the *interface* constraints for all module instantiations. It is interesting to note that the previous algorithm automatically handles the hierarchy of the behavioral HDL model.

D. Dynamic Unrolling

As mentioned earlier, each variable in the ILOG constraint model is a 2-D array. The first dimension refers to the module instantiation number and the second refers to the time frame. The proposed method facilitates automatic unrolling of the model for any number of time frames. An interesting case study for dynamic unrolling is as follows. *Given the initial context of a translation lookaside buffer (TLB), to find a test that shall evict a particular entry of the TLB in least number of access cycles.* The maximum number of time frames MAX_TFS for which the model can be unrolled is specified by the user. The tool dynamically unrolls for k time frames, for different values of k, $1 \leq k \leq MAX\_TFS$, in a binary search like fashion to find the optimum number of time frames leading to a solution. This enables the user to find the optimum number of time frames needed to come up with a solution for a given scenario.

IV. Scenario Constraint Generation

This section explains the generation of scenario constraints through examples from the DLX architecture.

1) Scenario 1: Generate a sequence of instructions that will not create RAW data hazards [38] in the next five cycles starting at time frame tf. The *scenario constraints* are as follows:

$$\begin{aligned}&\text{for}\,(i = 0; i <= 4; i++)\\ &\text{for}\,(j = i + 1; j <= 5; j++)\{\\ &\quad (dlx\_rd[top\_dlx][tf + i]! = dlx\_r1\_id[top\_dlx][tf + j])\\ &\quad (dlx\_rd[top\_dlx][tf + i]! = dlx\_r2\_id[top\_dlx][tf + j])\\ &\}.\end{aligned}$$

The previous constraints ensure that the *instruction destination register* rd selected during the time frame $tf + i$ is *not used* as the instruction source registers $r1\_id$ and $r2\_id$ during the time frames $tf + i + 1$ to $tf + 5$, $0 \leq i < 4$.

2) Scenario 2: Generate a sequence of instructions that will not create WAW data hazards [38] in the next three cycles starting at time frame tf. The *scenario constraints* are as follows:

$$\begin{aligned}&\text{for}\,(i = 0; i <= 2; i++)\\ &\text{for}\,(j = i + 1; j <= 3; j++)\{\\ &\quad (dlx\_rd[top\_dlx][tf + i]! = dlx\_rd[top\_dlx][tf + j])\\ &\}.\end{aligned}$$

3) Scenario 3: Generate a sequence of instructions that will generate an ALU output to be $value1$ in time frame $tf1$ and $value2$ in time frame $tf2$. The scenario constraints are as follows:

$$\begin{aligned}&(dlx\_alu\_out[top\_dlx][tf1] == value1)\\ &(dlx\_alu\_out[top\_dlx][tf2] == value2).\end{aligned}$$

4) Scenario 4: Generate a sequence of instructions that will access consecutive memory address starting from addr in the next five consecutive clock cycles. The *scenario constraints* are as follows:

$$\begin{aligned}&\text{for}\,(i = 0; i < 5; i++)\{\\ &\quad (data\_memory\_addr\_in[top\_dlx][tf + i] == addr + i)\\ &\}.\end{aligned}$$

The previous constraints ensure that the input address of the data memory is set to $address + i$ in time frames $tf + i$, $0 \leq i < 5$.

5) Scenario 5: Generate a sequence of instructions that will access the same memory address, addr, in the next five consecutive clock cycles. The *scenario constraints* are as follows:

$$\begin{array}{l} for(i = 0; i < 5; i++)\{ \\ \quad (data\_memory\_addr\_in[top\_dlx][tf + i] == addr) \\ \}. \end{array}$$

Hybrid Scenarios can be easily created by adding the desired constraints together. For example, a hybrid scenario, wherein, the scenarios 1 and 4 mentioned previously should occur together is modeled as follows:

$$\begin{array}{l} \text{for}\,(i = 1; i <= 5; i++)\{ \\ \quad (dlx\_rd[top\_dlx][tf]! = dlx\_r1\_id[top\_dlx][tf + i]) \\ \quad (dlx\_rd[top\_dlx][tf]! = dlx\_r2\_id[top\_dlx][tf + i]) \\ \quad (data\_memory\_addr\_in[top\_dlx][tf + i] == addr + i) \\ \}. \end{array}$$

6) Constraints for Invariant Properties: The constraints are very effective to model certain invariant properties of the design. These invariant properties are crucial for generating valid tests. A very interesting example is in the case of the TLB, wherein, *all the valid TLB entries in the TLB should have different linear addresses stored in them.* This is an invariant property on the key element of any associative/content-addressable memory (CAM). Consider the TLB with N locations. The array $LA[1, \ldots, N]$ store the linear address and the bit-array $V[1, \ldots, N]$ specifies whether the entries are valid or not. In other words, if $V[i] = 1$, then $LA[i]$ is a valid linear address, else it is not. The following constraints model the invariant property:

$$\begin{array}{l} \text{for}\,(i = 0; i \le N; i++) \\ \quad (ARR[i] == V[i] * LA[i] + (V[i] - 1) * (i + 1)); \\ (IloAllDiff(ARR)). \end{array}$$

The previous constraints ensure that $ARR[i]$ stores the linear Address $LA[i]$, if $V[i] = 1$. Note that, if $V[i] = 0$, then $ARR[i] = -(i + 1)$. These are negative numbers and so they are different from the positive valid linear addresses. The previous assignment also ensures that no two negative numbers stored in *ARR* are the same. Note that $-(i + 1)$ is used, because $(-0 == +0)$ and that 0 can be a valid linear address. The ILOG construct `IloAllDiff(ARR)` constraints every entry of the array *ARR* to be different. As all negative entries are different, the previous constraints imply that all positive entries that correspond to valid linear addresses should be different. Hence, the invariant property of the TLB mentioned before is modeled. The invariant constraints are used along with the model and scenario constraints to generate valid tests.

V. Experimental Results

A. ESI Cache Coherency Protocol Model

The ESI is a standard multiprocessor cache coherency protocol used in the context of several processors, each having their own cache and sharing a common global memory through a *shared bus* as shown in Fig. 8. Every cache line/block in each of these caches can be in any one of the three states, namely, exclusive (E), shared (S), or invalid (I). The state of a cache line can be modified either due to a memory access by the processor owning it (*processor-initiated*) or due to some other processor (non-owner) accessing the same address stored in the cache line (*bus-initiated*, as it is inferred by *snooping* on the shared bus) [38]. The change of the state of a cache line, both processor- and bus-initiated, are modeled as state machines [38] and shown in Figs. 9 and 10, respectively.

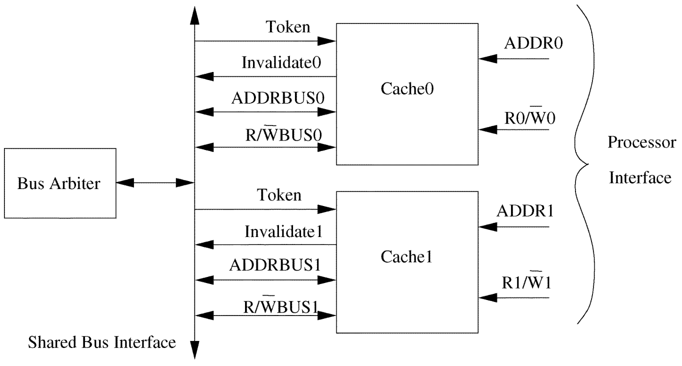

Fig. 8. Multiprocessor cache model.

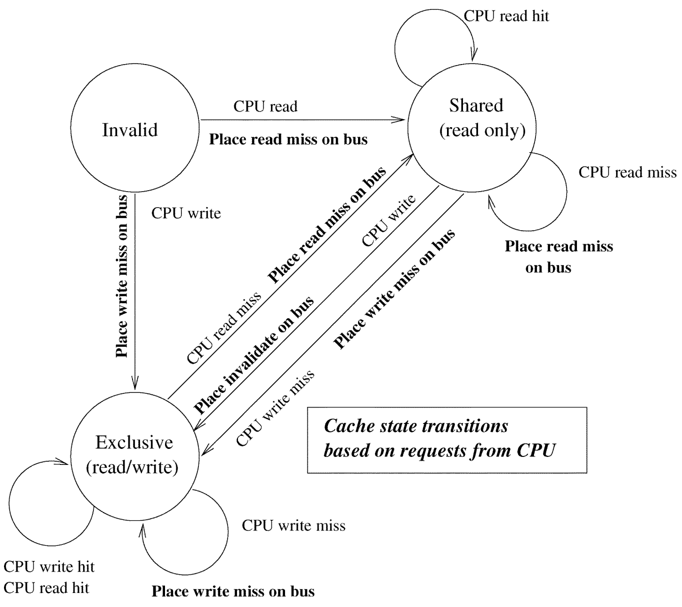

Fig. 9. Processor-initiated state change.

The Verilog model for the ESI protocol used for experimentation in this paper consists of two caches, each having two interfaces, namely, the *processor interface* and the *shared-bus interface*, as shown in Fig. 8. The *processor interface* for each of the caches has two inputs to the model, namely, the `ADDR` (address lines corresponding to the address to be accessed by the procssor) and the $\mathrm{R}/\overline{\mathrm{W}}$ $(\mathrm{Read}/\overline{Write})$ line specifying whether the processor wants to *Read* or *Write* to the memory location specified by the `ADDR` lines. The caches share the bus based on a *token ring protocol*. Since there are only two caches in the current model, the `Token` is modeled as a 1-bit *Toggle* flip-flop. When the `Token` takes the value 0, the *Cache0* has control over

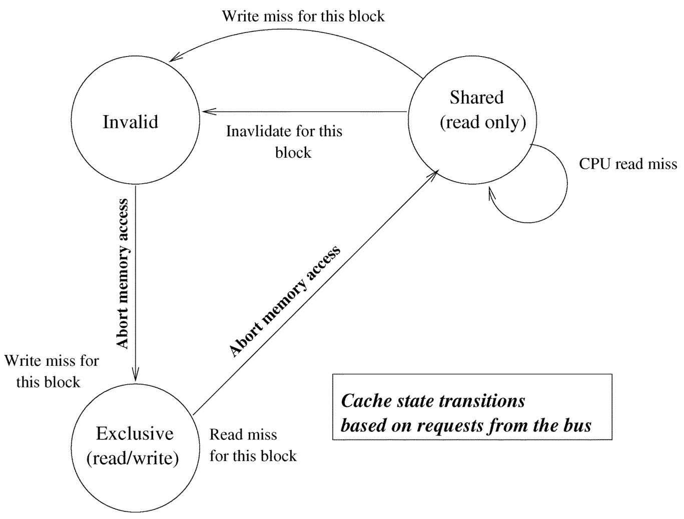
Write miss for this block
Invalid
Shared (read only)
Inavlidate for this block
CPU read miss
Abort memory access
Abort memory access
Write miss for this block
Exclusive (read/write)
Read miss for this block
Cache state transitions based on requests from the bus

Fig. 10. Bus-initiated state change.

TABLE I
EXPERIMENTAL RESULTS FOR THE ESI MODEL

S No	TF	Scenario Constraints			Output	
		To-ken	States	Tags	<ADDR0, ADDR1>	Action
1	0	1	<inv,inv>		<0,16>	P1: Write 16
	1		<inv,exc>			
2	0	0	<inv,exc>	<0,2>	<144,64>	P0: Read 144
	1		<sha,exc>		<144,64>	P1: Read 64
	2		<sha,exc>		<144,64>	P0: Read 144
	3		<sha,exc>			
3	0	0	<inv,exc>	<0,13>	<0,208>	P0: Write 208
	1		<exc,inv>		<208,208>	P1: Write 208
	2		<inv,exc>		<208,208>	P0: Write 208
	3		<exc,inv>			
4	0	0	<inv,exc>	<0,13>	<208,208>	P0: Read 208
	1		<sha,sha>		<208,208>	P1: Read 208
	2		<sha,sha>			
5	0	0	<sha,exc>	<13,9>	<208,144>	P0: Write 208
	1		<exc,exc>		<208,144>	P1: Read 144
	2		<exc,exc>		<208,144>	P0: Read 208
	3		<exc,exc>			

the bus, while the *Cache1 snoops* the bus. Similarly, when the `Token` takes the value 1, the *Cache1* has control over the bus, while the *Cache0 snoops* the bus. The *shared-bus interface* for each of the caches has the following signals.

- `ADDRBUS`: It is a set of `inout` (bidirectional) lines used by the cache to: 1) output the address generated by the processor owning it to the shared bus, while it controls the bus and 2) to snoop the address generated by the other processor (non-owner) while not in control of the shared bus.
- $\mathtt{R}/\overline{\mathtt{W}}\mathtt{BUS}$: It is an `inout` (bidirectional) line used by the cache to: 1) output to the shared bus, whether the processor owning it wants to Read or Write to the global memory, while it controls the bus and 2) to snoop whether the other processor (non-owner) is reading or writing to the global memory, while not in control of the shared bus.
- `Token`: A 1-bit input indicating the value of the `Token` to the cache.
- `Invalidate`: A 1-bit output line used by the cache to invalidate a transaction on the bus. More details on the need for invalidation is available in [38].

In the current model, the address lines are 8-bit wide. The caches are directly mapped and can store 16 words. Hence, the first four least significant bits of the address are used for mapping on to the cache lines and the remaining four bits are used as tag bits that are stored in the cache line. To make the model simple, in case a memory access initiated by a processor *P* leads to a conflict (`Invalidate` signal raised by the cache of the other processor), it is assumed that the conflict is resolved and the access initiated by *P* is completed within the same cycle. The cache model was *automatically* converted into a set of *model constraints*. The objective was to generate functional tests to verify the state machines shown in Figs. 9 and 10. The zeroth cache line of both the caches that map the addresses $(0, 16, 32, 48, \ldots)$ was used for this purpose. Table I shows the results of the experiment conducted on the previous model. The third column in Table I shows the initial value of the `Token`. The fourth column shows the desired states of the zeroth cache lines of both caches (Cache0 and Cache1 in order), represented as an ordered pair, one pair for every time frame, over consecutive set of time frames. The corresponding time frame number (*TF*) is shown in the second column of Table I. The fifth column of Table I shows the desired initial values of the tag bits of the zeroth lines of *Cache0* and *Cache1* in order, stored as an ordered pair. The values in the third, fourth, and fifth columns of Table I are input to the solver as *scenario constraints*. The sixth and seventh columns of Table I show the output of the solver, which is the required functional test. The sixth column shows the inputs to be applied to the address lines ADDR0 and ADDR1 in order, represented as an ordered pair, one pair for every time frame, over consecutive set of time frames. The seventh column states the *action (read or write)* taken by the processors.

The first experiment in Table I involved *two time frames*. First, the model was unrolled two times (for two time frames). Then, the following *scenario constraints* were added:

$$(\mathtt{Token}[0] == 1)$$

which states that the value of `Token` in the *zeroth* time frame is 1, indicating that Cache1 controls the bus in the *zeroth* time frame. Note that the variable `Token` is implemented as a toggle flip-flop in the design. Hence, the following *automatically* generated *model constraint*

$$(\mathtt{Token}[1] == \mathtt{not}(\mathtt{Token}[0]))$$

would ensure that the value of `Token` is 0 in time frame 1, thereby transferring control of the shared bus to Cache0. The other scenario constraints added were as follows:

$$(\mathtt{cache\_line}[0]\,[0]\,[0] == \mathtt{"inv"}).$$

The previous constraint ensures that the state of the *zeroth* cache line of the *zeroth* instantiation of the cache (Cache0) in the *zeroth* time frame is *invalid*

$$(\mathtt{cache\_line}[0]\,[1]\,[0] == \mathtt{"inv"}).$$

The previous constraint ensures that the state of the *zeroth* cache line of the *first* instantiation of the cache (Cache1) in the *zeroth* time frame is *invalid*

$$(\texttt{cache\_line}[0][0][1] == \texttt{"inv"}).$$

The above constraint ensures that the state of the *zeroth* cache line of the *zeroth* instantiation of the cache (Cache0) in the *first* time-frame is *invalid*.

$$(\texttt{cache\_line}[0][1][1] == \texttt{"exc"}).$$

The previous constraint ensures that the state of the *zeroth* cache line of the *first* instantiation of the cache (Cache1) in the *first* time frame is *exclusive*. The sixth and seventh columns of Table I shows the output of the constraint solver, which suggests that processor 1 should write to address 16 to cause the required change in the state of the cache line 0. From the definitions of the cache model and the state machines shown in Figs. 9 and 10, it can be inferred that the previous *functional test* generated by the solver, indeed causes the desired change in the state of the cache line.

Experiment 3 in Table I starts with the initial state of the zeroth cache line of Cache0 and Cache1 as *invalid* and *exclusive*, respectively. The required scenario was that in the subsequent three time frames the states of the cache lines should alternate between invalid and exclusive. Initially, the tag bits in the *zeroth* cache line of processor 1 was set to the value 13, which maps the address 208. From the definition of the caches, it could be inferred that the address 208 output by the constraint solver indeed maps on to cache line 0. From the definitions of the state machines shown in Figs. 9 and 10, it is seen that the functional test generated by the constraint solver indeed causes the desired scenario. In a similar manner, functional tests can be generated to cover every desired path of the state machines shown in Figs. 9 and 10.

B. 16-bit DLX Processor Model

This section presents the results from employing the proposed methodology on a 16-bit DLX processor model. All the results reported are measured on an HP workstation xw4200. The time and memory utilized are as reported by the ILOG constraint solver.

1) Scalability of the Constraint Model: Generation of a constraint model for multiple time frames consumes time and memory that grows linearly in the number of time frames. Table II shows the time and memory required to generate the constraint model for a full adder unrolled for different number of time frames. The results suggest that the solver performance is linear with the number of time frames unrolled. The important point to note is that in each of the cases shown in Table II a single constraint model is generated that represents the complete unrolled circuit.

2) Size of the Constraint Model: Table III (a)–(c) compares the behavioral Verilog model of the DLX processor with its corresponding A^2M graph and the constraint model. It is evident from Table III (a) and (c) that the number of variables ($= 746$) and constraints ($= 530$) in the optimized constraint model is far less than the number of gates ($= 13104$) needed to realize the model. This justifies the claim that a tool dealing with higher level models of abstraction handles data structures of lesser size than a tool dealing with lower levels. This contributes significantly towards the better scalability of the former in comparison to the latter with respect to the design size. The benefits of optimizing the A^2M graph can be inferred from Table III (b) and (c). As seen in Table III (b) there is a 46.7% (35.3%) decrease in the number of signals (operators) of the A^2M graph due to the optimization. As an effect of this optimization, as seen in Table III (c), there is a corresponding decrease of 46.94% (61.62%) in the number of variables (constraints) in the underlying ILOG-based constraint model.

TABLE II
RESULTS OF FULL ADDER

No of Time-frames Unrolled	Time (in sec)	Memory (in MB)
10	0.02	1.01
20	0.04	1.95
50	0.1	4.72
100	0.21	9.33
500	1.65	46.6
1000	4.48	92.13
10000	260.94	922.98

TABLE III
COMPARISON OF VERILOG, A^2M AND CONSTRAINT MODEL FOR A 16-BIT DLX PROCESSOR

(a) Verilog Model

No of lines verilog code	No of Gates
804	13104

(b) A^2M Graph

	No of Signals	No of Operators	No of Nodes
Before Optimization	1411	561	214
After Optimization	751	363	214

(c) Ilog

	No of Variables	No of Constraints
Before Optimization	1406	1381
After Optimization	746	530

TABLE IV
TIMING ANALYSIS

Phase	Timing
A^2M generation	$0.022s$
A^2M optimization	$0.145s$
Constraint Model Generation	$0.198s$

3) Generation of the Constraint Model: Table IV shows the time required by various phases in generating the ILOG model for the 16-bit DLX processor model. The total time consumed for the entire three phases was around 0.35 s. This illustrates the efficiency of the proposed methodology in handling large and complex designs.

4) FTG for User-Specified Scenarios: This section deals with time and memory requirements for test generation under various scenario constraints.

Table V shows the time taken for unrolling the DLX model to 10 time frames and generating test vectors to access two specified memory addresses under three different scenarios. In

TABLE V
INSTRUCTION GENERATION FOR MEMORY ACCESSES

Scenario	No of Time Frames	Time (in sec)	Memory (in MB)
Access two different Memory Addresses in two different Time-frames	10	14.8	134.53
Access same Memory Address in time-frames 7 & 8	10	66.74	380.66
Access consecutive Memory Addresses in time-frames 7 & 8	10	68.3	380.69

TABLE VI
INSTRUCTION GENERATION WITHOUT HAZARDS

Scenario	No of Time Frames	Time (in sec)	Memory (in MB)
No RAW No WAW for 5 time-frames	6	32.96	182.42
No RAW No WAW for 5 time-frames	10	84.98	379.92
No RAW No WAW for 3 time-frames	6	31.86	182.16
No RAW No WAW for 3 time-frames	10	81.97	379.31

TABLE VII
INSTRUCTION GENERATION FOR ALU OPERATIONS

Scenario	No of Time Frames	Time (in sec)	Memory (in MB)
ALU Out = 13 in time-frame 3 ALU Out = 5 in time-frame 2	6	11.08	88.52
ALU Out = 13 in time-frame 3 ALU Out = 5 in time-frame 2 ALU Out = 10 in time-frame 4	6	10.36	88.77
ALU Out = 13 in time-frame 3 ALU Out = 5 in time-frame 2 ALU Out = 10 in time-frame 4 ALU Out = 20 in time-frame 5	6	10.87	89.1

TABLE VIII
INSTRUCTION GENERATION FOR ALU OPERATIONS WITHOUT HAZARDS

Scenario	No of Time Frames	Time (in sec)	Memory (in MB)
Scenarios 1 & 3 from Tables VII & VI resp.	6	10.47	45.62
Scenarios 2 & 3 from Tables VII & VI resp.	6	959.98	242.34
Scenarios 3 & 3 from Table VII & VI resp.	6	959.95	242.63

the first scenario, the time frame numbers for the memory access and the memory addresses were different and independent of each other, while, in the remaining two scenarios, the two memory addresses were dependent, leading to constraints that were dependent on each other.

5) Observation 1: From Table V, it is seen that the constraint solver consumes less time and memory to solve *independent constraints* in comparison to *dependent constraints*. The dependency is measured *across* time frames. Table VI shows the time taken for unrolling the DLX model to 6 or 10 time frames and generating test vectors that ensure no RAW hazard over all the time frames and no WAW hazard within any three or five consecutive time frames as stated in the *Scenario* column of the same.

6) Observation 2: From Table V, it is seen that the constraint solver consumes more time and memory with increasing number of time frames for which the model is unrolled.

The scenarios stated in Tables VII and VIII had additional constraints that initialized the 16 registers of the *DLX* processor R_i to i, $0 \leq i \leq 15$. Table VII shows the time taken for unrolling the DLX model to six time frames and generating test vectors that ensure that the output of the ALU (`ALU Out`) carry the specified values in the specified time frames as stated in the *Scenario* column of the same. The generated test vector may have hazards.

7) Observation 3: From Table VII, it is seen that the addition of more number of independent (across time frames) constraints does not significantly impact the performance of the constraint solver.

The Table VIII shows the time taken for unrolling the DLX model to six time frames and generating test vectors that ensure the scenarios specified in Table VII without RAW hazards over all the time frames and no WAW hazard within any three consecutive time frames. The assembly code generated for Scenario 3 of Table VIII is as follows.

ADD R1 R2 R3; ADD R3 R13 R0; MUL R2 R5 R2; ADD R1 R11 R9.

As mentioned earlier, given that registers R_i initially store the value i, $0 \leq i \leq 15$, implies that the previous assembly code indeed generates the specified scenario.

8) Observation 4: Comparing Tables VII and VIII reveals interesting issues. The solver took *less* time to solve the first scenario in Table VIII when compared to the first one in Table VII. This is in spite of the fact that the former is a combination of the latter and additional constraints. The reason for the same would be that the additional constraints in the former scenario reduced the search space, essentially directing the solver towards the goal. Such a trend is not seen in the subsequent two scenarios in Tables VII and VIII.

C. Constraint Model for Logic Simulation

It is interesting to note that the constraint model of a given HDL behavioral code can be used for *logic simulation*. Table IX presents details on simulation of the constraint model of the DLX processor. The model was unrolled for the number of time frames as stated in Table IX. The instructions in the `Scenario` column of Table IX were applied to the initial time frame as constraints. Solving the constraints is equivalent to simulating the DLX processor over the next k cycles, where k is the number of times the model is unrolled. In other words, for the first scenario in Table IX, the DLX constraint model was unrolled for seven time frames and the variables corresponding to the instruction memory of the first frame was constrained to the binary equivalent of the instruction `LD R1, [R2]`. The initialization constraints included constraints for the variables corresponding to the registers of the processors. These variables were constrained to some known values. Solving the constraint model resulted in the variable corresponding to register *R1* being assigned the value stored in the memory address pointed to by

TABLE IX
LOGIC SIMULATION OF DLX CONSTRAINT MODEL

Scenario	No of Time-frames unrolled	Time (in sec)	Memory (in MB)
LD R1, [R2]	7	0.09	5.51
LD R1, [R2] ST [R6], R2 LD R12, [R1]	7	0.13	7.61
LD R1, [R2] ST [R6], R2 LD R12, [R1] ADD R3, R1, R2	12	0.23	12.90

register *R2*. Similar is the case for the other two scenarios presented in the Table IX. These type of scenarios yield constraints that represent the forward flow of data and hence easy to solve. This is reflected by the lesser amount of time and memory consumed by the ILOG solver as reported in Table IX.

VI. Conclusion and Future Work

This paper proposed a methodology for automatic conversion of a given behavioral HDL model into a set of integer constraints. The constraint model was further used for generation of directed functional tests. The conversion and optimization procedures are similar to that of *logic synthesis*. Integer (word-level) constraint solvers were employed by the proposed methodology in contrast to the widely reported SAT-solvers. The effectiveness of the proposed approach was illustrated by employing the same on a 16-bit DLX processor behavioral model and using the same to generate functional tests for different interesting scenarios. The methodology enabled the user to express the corner case to be tested as an higher-level constraint-based test specification spreading across multiple time frames. This paper presents examples of such test specifications. To further enhance the versatility of the proposed methodology, the following features can be easily incorporated.

A. Multi-Clock Designs and Latches

The concept of time-frames discussed so far assumed that all the storage elements are flip-flops, and, they are driven by a single clock. This can be extended for Multi-clock and Latch based designs by employing a verilog type technique, wherein, a basic simulator time-unit is assumed and every event (clocks and changes in wires) in the design is sampled at this time-unit [1]. A similar approach may be employed here, wherein, the time-frame is determined by the basic time-unit rather than a single global clock period.

B. Handling Large Vectors

As the size of vectors increases, the search space increases exponentially. One approach to reduce the search space will be to reduce the range of the variables. For example, none of the current integer solvers can efficiently support a 64-bit vector. Even assuming that solvers have such large integer support, it increases the search space enormously, thereby decreasing the performance of the solver. A solution to this problem is to split a large vector into small variables. For Example, a 32-bit vector ir can be split into 4 different variables, say, ir_1, ir_2, ir_3 and ir_4, each of size 8-bits. Each new variable has a range of 256 thereby decreasing the per variable search space for the solver. This can boost up the performance of the solver.

As observed in the experimental section, the constraint solver performs much efficiently when the underlying constraints are more *independent*. Future work shall concentrate on concretizing this notion of *independent constraints*, specifically trying to arrive at constraint models with higher degree of independence among its constraints.

References

[1] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*. Singapore: Pearson Education, 2004.

[2] S. Palnitkar, *Design Verification with E*. Singapore: Pearson Education, 2005.

[3] M. Abromovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. Piscataway, NJ: IEEE Press, 2001.

[4] I. Ghosh and M. Fujita, "Automatic test pattern generation for functional register transfer level circuits using assignment decision diagrams," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 20, no. 3, pp. 402–415, Mar. 2001.

[5] I. Ghosh, L. Zhang, and M. Hsiao, "Automatic design validation framework for HDL descriptions via RTL ATPG," in *Proc. Asian Test Symp.*, 2003, pp. 148–153.

[6] P. Vishakantaiah, J. A. Abraham, and D. G. Saab, "Cheetaa: Composition of hierarchical sequential tests using atket," in *Proc. Int. Test Conf.*, Oct. 1993, pp. 606–615.

[7] R. S. Tupuri, A. Krishnamachary, and J. A. Abraham, "Test generation for gigahertz processors using an automatic functional constraint extractor," in *Proc. Int. Test Conf.*, Jun. 1999, pp. 646–652.

[8] J. Lee and J. H. Patel, "Architectural level test generation for microprocessors," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 13, no. 10, pp. 1288–1300, Oct. 1994.

[9] A. Ghosh, S. Devadas, and A. R. Newton, "Sequential test generation and synthesis for testability at the register transfer and logic levels," *IEEE Trans. Computer-Aided Des. Integr. Circuits Syst.*, vol. 12, no. 5, pp. 579–598, May 1993.

[10] V. M. Vedula, J. A. Abraham, and T. Larrabee, "Program slicing for hierarchical test generation," in *Proc. VLSI Test Symp. (VTS)*, 2002, pp. 237–243.

[11] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Instruction based self testing of processor cores," in *Proc. VLSI Test Symp. (VTS)*, Apr. 2002, pp. 223–228.

[12] L. Chen, S. Ravi, A. Raghunathan, and S. Dey, "A scalable software based self-test methodology for programmable processors," in *Proc. Des. Autom. Conf. (DAC)*, 2003, pp. 548–553.

[13] L. Chen and S. Dey, "Software-based self testing methodology for processor cores," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 20, no. 3, pp. 369–380, Mar. 2001.

[14] W. C. Lai, A. Krstic, and K. T. Cheng, "On testing the path delay faults of a microprocessor using its instruction set," in *Proc. VLSI Test Symp. (VTS)*, May 2000, pp. 15–20.

[15] T. Larrabee, "Test pattern generation using Boolean satisfiablilty," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 11, no. 1, pp. 4–15, Jan. 1992.

[16] R. E. Bryant, "Graph based algorithms for Boolean function manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, pp. 677–691, Aug. 1986.

[17] K. S. Brace, R. Rudell, and R. E. Bryant, "Efficient implementation of the BDD package," in *Proc. Des. Autom. Conf.*, 1990, pp. 40–45.

[18] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in sat-based formal verification," *Int. J. Softw. Tools Technol. Transfer*, vol. 7, no. 2, pp. 156–173, 2005.

[19] S. Ravi, G. Lakshminarayana, and N. K. Jha, "Tao: Regular expression based register-transfer level testability analysis and optimization," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 9, no. 6, pp. 824–832, Dec. 2001.

[20] S. Ravi and N. K. Jha, "Fast test generation for circuits with RTL and gate-level views," in *Proc. Int. Test Conf. (ITC)*, 2001, pp. 1068–1077.

[21] L. Lingappan, S. Ravi, and N. K. Jha, "Satisfiability-based test generation for nonseparable RTL controller-datapath circuits," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 25, pp. 544–557, Mar. 2006.

[22] T. Li, Y. Guo, and S. K. Li, "Assertion-based automated functional vectors generation using constraint logic programming," in *Proc. 14th ACM Great Lakes Symp. VLSI (GLSVLSI)*, 2004, pp. 288–291.

[23] R. Vemuri and R. Kalyanaraman, "Generation of design verification tests from behavioral VHDL programs using path enumeration and constraint programming," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 3, no. 2, pp. 201–214, Jun. 1995.

[24] A. K. Chandra and V. S. Iyengar, "Constraint solving for test case generation: A technique for high-level design verification," in *Proc. Int. Conf. Comput. Des.*, 1992, pp. 245–248.

[25] F. Fallah, S. Devadas, and K. Keutzer, "Functional vector generation for HDL models using linear programming and Boolean satisfiability," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 20, no. 8, pp. 994–1002, Aug. 2001.

[26] C. Y. Huang and K. T. Cheng, "Assertion checking by combinated word-level ATPG and modular arithmetic constraint solving techniques," in *Proc. Des. Autom. Conf.*, 2000, pp. 118–123.

[27] F. Fallah, S. Devadas, and K. Keutzer, "Functional vector generation for HDL models using linear programming and 3-satisfiability," in *Proc. Des. Autom. Conf.*, 1998, pp. 528–533.

[28] Z. Zeng, P. Kalla, and M. Ciesielski, "Lpsat: A unified approach to RTL satisfiability," in *Proc. DATE Conf.*, 2001, pp. 398–4002.

[29] D. Lewin, L. Fournier, and M. Levinger, "Constraint satisfaction for test program generation," in *Proc. IEEE 14th Phoenix Conf. Comput. Commun.*, 1995, pp. 45–48.

[30] A. Aaharon, D. Goodman, and M. Levinger, "Test program generation for functional verification of powerpc processors in IBM," in *Proc. 32nd Des. Autom. Conf.*, 1995, pp. 279–285.

[31] A. Adir, E. Almog, L. Fournier, and E. Marcus, "Genesys-Pro: Innovations in test program generation for functional processor verification," *IEEE Des. Test Comput.*, vol. 21, no. 2, pp. 84–93, Mar. 2004.

[32] J. Yuan, C. Pixley, A. Aziz, and K. Albin, "A framework for constrained functional verification," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD)*, 2003, p. 142.

[33] A. Adir, H. Azatchi, E. Bin, O. Peled, and K. Shoikhet, "A generic micro-architectural test plan approach for microprocessor verification," in *Proc. 42nd Ann. Conf. Des. Autom. (DAC)*, 2005, pp. 769–774.

[34] F. Corno, P. Prinetto, and M. S. Reorda, "Testability analysis and ATPG on behavioral RT-level VHDL," in *Proc. Int. Test Conf.*, 1997, pp. 753–759.

[35] S. Chuisano, F. Corno, and P. Prinetto, "RT-level TPG exploiting highlevel synthesis information," in *Proc. VLSI Test Symp.*, 1999, pp. 341–346.

[36] D. Prasad, R. Archana, V. Karthik, Senthilkumar, V. Kamakoti, S. M. Kailasnath, and V. M. Vedula, "A novel unified framework for functional verification of processors using constraint solvers," in *Proc. VLSI Des. Test Symp. (VDAT)*, 2006, pp. 418–426.

[37] V. Chaiyakul and D. D. Gajski, "Assignment decision diagram for high-level synthesis," University of California, Irvine, CA, Tech. Rep. 92-103, 1992.

[38] J. L. Hennessy and D. A. Patterson*, Computer Architecture – A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 2003.

[39] V. Chaiyakul, D. D. Gajski, and L. Ramachandran, "High-level transformations for minimizing syntatic variances," in *Proc. Des. Autom. Conf.*, Jun. 1993, pp. 413–418.

[40] ILOG Inc., Sunnyvale, CA, "Ilog technology web site," 1987 [Online]. Available: http://www.ilog.com

Siva Kumar Sastry Hari received the B.Tech. degree in computer science and engineering from Indian Institute of Technology, Madras, India, in 2007.

His research interests include VLSI design and test, computer architecture, and systems.

Vishnu Vardhan Reddy Konda received the B.Tech. degree in computer science and engineering from Indian Institute of Technology, Madras, India, in 2007.

His research interests include VLSI design and test, computer architecture, and systems.

Kamakoti V received the B.E. degree from University of Madras, Madras, India, in 1989, and the M.S. and Ph.D. degrees from the Indian Institute of Technology, Madras, India, in 1991 and 1995, respectively.

He is currently an Associate Professor with the Department of Computer Science and Engineering, Indian Institute of Technology. His research interests focus on software for VLSI and reconfigurable systems design.

Vivekananda M. Vedula (M'03) received the B.S. degree in chemical engineering from the Indian Institute of Technology, Madras, India, in 1996, and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Texas at Austin, Austin, in 1998 and 2003, respectively.

He is currently a staff Research Scientist with the Design and Technology Solutions Group, Intel Corporation, Austin, TX. His research interests include VLSI-CAD algorithms and formal methods for manufacturing test generation and design validation.

Kailasnath S. Maneperambil received the B.S. degree in electrical engineering from the Indian Institute of Technology, Mumbai, India, in 1986, and the M.S. degree in computer science from Pennsylvania State University, Philadelphia, in 1993.

He is currently a staff Research Scientist with the Design and Technology Solutions Group, Intel Corporation, Austin, TX. His research interests include VLSI-CAD algorithms, DFT, and formal methods for manufacturing test generation and design validation.