JOURNAL OF ELECTRONIC TESTING: Theory and Applications 15, 173–189 (1999)

On Design Validation Using Verification Technology*

DINOS MOUNDANOS
Advanced CAD Research Group, Fujitsu Labs of America, Sunnyvale, CA 94086, USA
dinos@fla.fujitsu.com

JACOB A. ABRAHAM
Computer Engineering Research Center, The University of Texas at Austin, Austin, TX 78712, USA
jaa@cerc.utexas.edu

Received July 8, 1998

Editors: M. Nicolaidis and R. Roy

Abstract. Despite great advances in the area of Formal Verification during the last ten years, simulation is currently the primary means for performing design verification. The definition of an accurate and pragmatic measure for the coverage achieved by a suite of simulation vectors and the related problem of coverage directed automatic test generation are of great importance. In this paper we introduce a new set of metrics, called the Event Sequence Coverage Metrics (ESCMs). Our approach is based on a simple and automatic method to extract the control flow of a circuit so that the resulting state space can be explored for validation coverage analysis and automatic test generation. During simulation we monitor, in addition to state and transition coverage, whether certain control event sequences take place or not. We then combine formal verification techniques, using BDDs as the underlying representation, with traditional ATPG and behavioral test generation techniques to automatically generate additional sequences which traverse uncovered parts of the control state graph, or exercise an uninstantiated control event sequence.

Keywords: OBDDs, verification, abstraction, extracted control flow machine, coverage analysis, test generation

1. Introduction

Design verification deals with checking the conformance of the design to its functional specification at any level of abstraction, but usually at higher levels of abstraction (i.e. behavioral or register transfer level). It is a process very critical in making sure that the design is bug-free before tapeout. However, design verification is a very complex task which becomes even more difficult in the case of modern, high performance circuits which employ a series of design and architectural techniques aiming at boosting their performance (i.e., pipelining, superscalar execution, speculative execution, dynamic scheduling, branch prediction, etc.). These techniques add significant complexity and make circuits susceptible to very subtle design errors. Detection of such errors requires the combination of a series of conditions to take place in a specified temporal sequence. Setting up these conditions is a very hard task. Currently both Formal Verification and Validation by Simulation are employed to attack the design verification problem.

While much progress has been made in automating the verification process and using formal verification tools, a major limitation of this technology is still the size of the circuits it can handle. As a result, validation by simulation is still the primary means of checking the

*This research was supported in part by the Semiconductor Research Corporation under contract 97-DP-388 and in part by the Texas Advanced Technology Program under Project 003658-268 at the University of Texas at Austin.

correctness of a design. Under this methodology the design is simulated for all the vectors in a functional test suite, in an environment that models the actual hardware system, and the simulation output is checked against expected results to determine whether the design is behaving as specified. The test vectors can be generated by random or pseudo-random (biased) test generators, constraint solvers, can be taken from typical workloads, or can be hand written by the designers based on the functional specification of the design. It has been observed that all these methods fail to provide a measurable degree of confidence that a complex design has been adequately tested.

Associated with validation through simulation are the problems of "coverage" and generation of simulation inputs. Coverage is a measure of the completeness of a test suite. The ideal metric for evaluating the functional coverage of a test suite would be the fraction of specified behaviors exercised by that suite. A behavior can be modeled as an execution of the design. Finding all possible execution paths has exponential complexity and attempting to exercise all of them would require enormous computational resources. The next best measure is the fraction of reachable states or transitions that have been exercised. This is the approach adopted in [1] when defining functional coverage metrics.

In this paper we will briefly review the ECFM model (Section 3), before introducing a new set of coverage metrics, called Event Sequence Coverage Metrics (ESCMs) in Section 4. These new metrics are designed to complement the metrics defined in [1]. In this methodology the designer specifies interesting control event sequences that need to be exercised during simulation. These sequences are specified in a specialized language that we provide. Alternatively they can be specified as non-deterministic Finite State Machines (FSMs) in a hardware description language like VHDL or Verilog. These control sequences can capture the complex interaction of control events, especially those that affect the datapath, and generating tests for them gives sufficient confidence that the majority of difficult corner cases will be exercised. The main motivation behind the introduction of these new coverage metrics is that full transition coverage even on the ECFM of a design may be neither possible nor desirable. The former is true due to the fact that the non control parts of the design are modeled non deterministically, and the latter is true because not all transitions may need to be exercised to fully test functionality [2], although the concept of equivalent transitions [1] eliminates this problem for the most part. We believe that a lot of these event sequences can be generated automatically (for example making sure that the design never enters an illegal state). However, if the behaviors being checked involve complicated parts of the design like the bus protocol, the designer needs to provide the sequences to be monitored. Alternatively a subset of Model Checking [3] properties that have not been formally verified can be utilized to generate interesting event sequences.

We also apply the same abstract model to automatic test generation for validation vectors. Our approach to test generation is twofold (Section 5). We first provide a technique for Coverage-Directed Test Generation. In this regard we can either supplement a given test suite that does not achieve satisfactory functional coverage according to our metrics, or generate test sequences that guarantee high coverage of the control behavior of the design. Additionally for event sequences not exercised by the functional test suite we automatically generate test vectors that would cause the machine to go through that sequence of events. We employ techniques widely used in formal verification to automatically generate tests to exercise all control transitions. Test sequence generation is performed on the ECFM model. This sequence may not be directly applicable to the original machine because of data conflicts [4]. In this case, we use traditional test generation techniques to expand the generated sequence and map it back to the original machine.

The main contributions of this paper are presented in Sections 4.1 where we introduce a new set of coverage metrics, and 5 where we show how these metrics can guide the test generation process. Particularly in Section 5.3.2 we introduce a new approach to solving the mapping back problem which is one of the main issues in all abstraction based test generation methodologies [4, 5, 2, 6]. An overview of the system is depicted in Fig. 1.

The system consists of a test model generation facility (extraction of the ECFM of the design which may be an iterative process), a test generation subsystem which includes the mapping-back facility and is driven by coverage metrics, and a coverage monitoring facility. The vectors that are generated can be used to compare two descriptions against each other. Additionally, during simulation, we can monitor the occurence of event sequences.

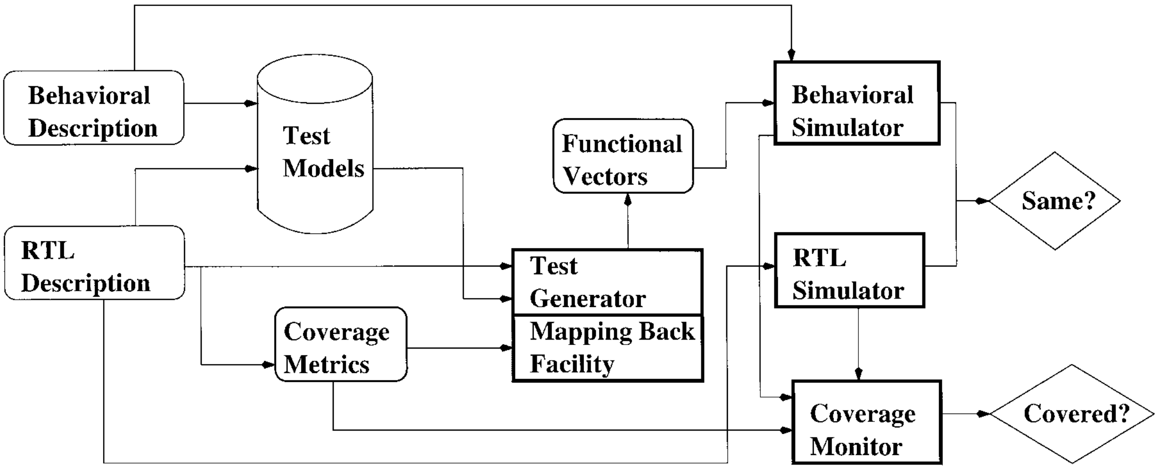

Fig. 1. System overview.

2. Previous Work

There exist ad-hoc techniques for coverage estimation in functional validation used in practice. These include toggle coverage on signals in the HDL program and HDL statement coverage. It is generally accepted that these are not accurate measures of functional (behavioral) coverage. For a more detailed discussion please refer to [1]. Other approaches to evaluate coverage check that all states are visited [7]. The coverage metric presented in [1] and reviewed briefly here is also based on a similar metric which measures transitions traversed not in the state machine representing the original circuit but in an abstracted machine capturing the control flow of the design.

More recently, approaches to coverage measurement and test generation presented in [5, 2] and [6] use techniques similar to the ones presented in this paper. State machines for part of the circuit, usually the control part, are extracted from the HDL description by various means and used as the target of the test generation. The approach in [5] uses graph traversal of the state graph for the extracted machine to traverse all its transitions. The main difference with the work presented here lies in the mapping of the tests generated on the extracted machine back to the original circuit. Their approach is to artificially inject the required signal values during simulation, while we propose the use of conventional ATPG and behavioral test generation techniques to expand the test sequence generated on the extracted model so as to make it valid on the original circuit. Controlled simulation limits the exploration of the design space. For example assume that a test contains a floating point exception. Using the approach in [5] the actual data on the datapath is ignored during the test. The exception is simply forced by taking control of the signal that specifies the occurence of the exception. Bugs related to the actual data that coused the exception won't be detected. The work in [5] is extended in [2] with the concept of control events which basically represent a set of commands to the datapath. A control event is a set of control state variable values observed by the datapath. Coverage is then defined as the fraction of control events covered over the total number of reachable control events. We believe that the concept of equivalent transitions in [4] achieves the same objective. Furthermore, we employ a more generic form of control events in this paper. The approach in [6] also targets part of the state space in a manner very similar to ours and uses the counterexample facility in the SMV model checker [3] to generate a test for each transition in the targeted state space. However, mapping this test to the actual machine is a textual translation process by means of pattern matching. A high level operation is viewed as a long sequence of low level state transitions that drive the unit along the high level execution path. A parser follows the state transitions and generates a high level operation whenever it recognizes a pattern indicating that such a transaction is taking place. A random biased test generator is then used to fill in blanks. Finally [8] introduces an observability-based code coverage metric. The approach is based on injecting tags in variables in the HDL and observe the "activation" and "propagation" of these tags to the outputs during simulation. It is an approach tied to the HDL syntactic style and based on the principle of activating every statement in the HDL which has been shown to not be an accurate coverage estimation method.

There have been other efforts to use information obtained from the high-level description in order to produce either functional tests or stuck-at fault test suites or to assist a lower level test generation tool. In [9], an extended finite state machine (EFSM) model is extracted from the behavioral description and is exhaustively traversed to generate functional tests. Exhaustive traversal of this machine ensures that all statements in the original code are executed. This approach analyzes the syntactic structure of the HDL program and identifies equivalence relations among parts of the data space. Such equivalence classes can be reencoded using fewer bits thus leading to a smaller FSM. The final FSM is equivalent to the original FSM. Obviously this does not always result in dramatic reduction in size. The dependence on syntactic style enables handling of larger machines than classical equivalence partitioning could handle. But it is also a drawback since it enforces hard restrictions on the coding style. In [10, 11] two more systems with the same objective as our approach, namely exercise the design for interesting bugs, are described. However, they operate at the instruction set architecture level or even higher, at the operational interface level and use techniques like symbolic simulation and constraint solving to generate effective tests.

Some researchers have used test suites generated from conventional ATPG tools used for structural testing to do functional verification. For example, the approach followed in [12–14] introduces the concept of "design errors" to model possible functional faults. Coverage of the test suite is the fraction of possible design errors detected by the suite. While this approach is effective at the gate or lower levels in detecting common designer mistakes, it is primarily a localized structural approach and does not give an indication of the extent to which the behavior of the design has been exercised.

3. The Extracted Control Flow Machine Model

Most CAD algorithms depend on an implicit or explicit exploration of the design state space which, for the majority of modern circuits, is huge mainly because of a large data path component. This phenomenon makes many CAD problems intractable by present techniques. One area of hope is the use of abstraction. Another point is that in most applications, including design verification and test generation, it is the flow of control that is of prime interest. The ECFM of a sequential circuit is a model of the control flow in the design. The difficulty in identifying the control circuitry lies in defining the interface of the control units with the rest of the circuit, and not in differentiating the control registers from registers holding pure data. In the ECFM methodology, it is the designer's choice which registers are to be considered as contributing to the control state space and which make up the data. Consequently, we abstract the data registers from the circuit and group the data into "equivalence" classes with respect to their effect on the control.

More formally, let us assume that the sequential behavior of a circuit is represented as a Finite State Machine (FSM). This Mealy type FSM is a 6-tuple $(\Sigma, O, S, s^0, \Delta, \Lambda)$, where

$\Sigma = \{0, 1\}^n$ is the input space (n is the number of input bits)
$O = \{0, 1\}^l$ is the output space (l is the number of output bits)
$S = \{0, 1\}^{c+d}$ is the finite state space (c is the number of control bits and d is the number of data bits)
$s^0 = \{\langle \vec{s_c}, \vec{s_d} \rangle\}$ is the set of initial states, $\vec{s_c} \in \{0, 1\}^c$ and $\vec{s_d} \in \{0, 1\}^d$
$\Delta : \Sigma \times S \to S$ is the next-state functional vector, $\Delta = [\delta_1..\delta_{c+d}]$, and
$\Lambda : \Sigma \times S \to O$ is the output functional vector, $\Lambda = [\lambda_1..\lambda_l]$.

The ECFM is also represented as a Mealy type FSM $(\Sigma', O', S', s^{0'}, \Delta', \Lambda')$, where

$\Sigma' = \{0, 1\}^{n'+d'}$ is the input space ($n' \leq n, d' \leq d$)
$O' = \{0, 1\}^{l'}$ is the output space ($l' \leq l$)
$S' = \{0, 1\}^c$ is the finite state space
$s^{0'} = \{\langle s_c \rangle\}$ is the set of initial states, $s_c \in S'$
$\Delta' : \Sigma' \times S' \to S'$ is the next-state functional vector, $\Delta' = [\delta_1..\delta_c]$, and
$\Lambda' : \Sigma' \times S' \to O'$ is the output functional vector, $\Lambda' = [\lambda_1..\lambda_{l'}]$.

The input space of the ECFM shows the presence of data registers which are now primary inputs. The ECFM input space is smaller than $n + d$ because only those inputs and data registers that have an effect on control flow appear in the ECFM, the rest are dropped. Some outputs of the original circuit also may be dropped in the ECFM if they are not of interest to the application at hand. A transition in the original circuit can be represented as a 4-tuple $(\vec{i}, \vec{s_1}, \vec{s_2}, \vec{o})$ where $\vec{i} \in \Sigma$, $\vec{s_1}, \vec{s_2} \in S$ and $\vec{o} \in O$ and $\vec{o} = \Lambda(\vec{i}, \vec{s_1})$ and $\vec{s_2} = \Delta(\vec{i}, \vec{s_1})$. If we represent $\vec{s_1}$ and $\vec{s_2}$ as $\vec{s_1} = \langle \vec{s_{1c}}, \vec{s_{1d}} \rangle$ where $\vec{s_{1c}} \in \{0,1\}^c$ and $\vec{s_{1d}} \in \{0,1\}^d$, $\vec{s_2} = \langle \vec{s_{2c}}, \vec{s_{2d}} \rangle$ where $\vec{s_{2c}} \in \{0,1\}^c$ and $\vec{s_{2d}} \in \{0,1\}^d$, then this transition maps to the corresponding transition $(\langle \vec{i}, \vec{s_{1d}} \rangle, \vec{s_{1c}}, \vec{s_{2c}}, \vec{o})$ in the ECFM. Thus, every transition in the original circuit maps to at least one transition in the ECFM. Furthermore, $\vec{o} = \Lambda'(\langle \vec{i}, \vec{s_{1d}} \rangle, \vec{s_{1c}})$ and $\vec{s_{2c}} = \Delta'(\langle \vec{i}, \vec{s_{1d}} \rangle, \vec{s_{1c}})$.

Definition 1. Two transitions $(\vec{i}, \vec{s_1}, \vec{s_2}, \vec{o})$ and $(\vec{j}, \vec{s_3}, \vec{s_4}, \vec{p})$ in the ECFM of a circuit are **equivalent** iff $\vec{s_1} = \vec{s_3}$ and $\vec{s_2} = \vec{s_4}$ and $\vec{o} = \vec{p}$.

We thus define an equivalence relation among transitions in the ECFM such that transitions are grouped into equivalence classes.

Lemma 1. *The equivalence partitions on the transitions of the ECFM of a circuit define corresponding equivalence partitions on the transitions of the original circuit.*

As we have seen, each transition in the original circuit maps to a corresponding transition in the ECFM and it is an onto mapping. Thus every transition in the original circuit can be placed in the same equivalence class of its corresponding transition in the ECFM such that all transitions in a class affect the flow of control in the same manner. In essence the process of grouping equivalent transitions in the ECFM of a circuit partitions the state space of the original circuit in terms of its effect on the control flow of the circuit. This is true since data register values in the original design have moved on the transitions as PIs in the ECFM. Those ECFM transitions are grouped together (if they are equivalent according to Definition 1) and therefore the corresponding states (in terms of data values) in the original circuit are also grouped together.

It should be noted that computing the reachable states in the ECFM does not directly correspond to the control states that may be reachable in the actual machine. Rather, it is an over-estimation of the reachable control state space because some data registers of the original circuit are unconstrained in the ECFM and may assume values not possible in the actual circuit. However, it is intuitively a close approximation because, in general, data assumes any value. Thus the ECFM is not equivalent to the original machine, as is the case for the EFSM [9].

4. Functional Coverage Metrics

We now describe how the ECFM can be used to derive a pragmatic estimate of the functional coverage provided by a sequence of input vectors.

4.1. Previously Defined Coverage Metrics

In order to compute a quantitative measure that reflects the quality of a test suite, we use two metrics, a state coverage metric (SCM) and a more accurate transition coverage metric (TCM). Given an initial state or a set of possible initial states, we first compute the reachable states in the ECFM. The two metrics are then given as,

$$\text{SCM} = \frac{\text{Number of states visited in the ECFM}}{\text{Total number of reachable states in the ECFM}} \tag{1}$$

$$\text{TCM} = \frac{\text{Number of transitions traversed in the ECFM}}{\text{Total number of reachable transitions in the ECFM}} \tag{2}$$

Obviously, TCM is the most comprehensive metric of the two. However, we have found that it helps to consider 100% SCM as the first target. Providing pointers to the part of the state space that is not covered by the given tests is a useful feature in case the designer wishes to address the coverage holes by adding tests. The coverage metrics reflect the amount of control behavior exercised during the simulation.

There are two points that need to be made about this approach. First, 100% transition coverage may not be possible. This is the case because some transitions in the ECFM may require data values that are not possible in the original machine. Hence, it is reasonable to be satisfied with a relatively high value of TCM. Completely characterizing the input space of the ECFM to avoid such cases is an unsolved problem. Second, the user has some control over the ECFM generation and can iterate over the extraction process by changing the designation of registers as control or data. Recall that data registers will not appear in the ECFM if they are not in any dependency set (no control registers or primary outputs of the design functionally depend on them).

4.2. Extending the Coverage Metrics

The coverage metrics (TCM and SCM) that were described in the previous section provide a realistic and meaningful measure of the control behavior of a circuit exercised by a set of verification vectors. However, they cannot capture the control behavior described by the interaction of several subparts of the design which is expressed by activation of state transitions in a specific order. This is why a more accurate coverage metric would be Execution Path Coverage. However, this is unrealistic since it has exponential complexity. We propose, as a solution to this problem, a new approach which is called Event Sequence Coverage.

In this respect we concentrate on monitoring Control Event Sequences that are specified by the user. More specifically the user (designer) comes up with a test plan for a specific design. This test plan includes the critical event sequences that he or she wants monitored during simulation. By an event sequence we simply mean a series of events that have to take place in a specific order and according to some specific timing requirements. In our framework we define two types of event sequences: "good" sequences which indicate the presence of desirable behavior (this is a type of liveness requirement) and "bad" sequences which are monitored in order to ensure the absence of undesirable behavior (safety requirement). Along these lines we define two additional coverage metrics:

$$\mathrm{ESCM}_{\mathrm{incl}} = \frac{\text{Number of good Event Sequences covered}}{\text{Total number of good Event Sequences monitored}} \tag{3}$$

$$\mathrm{ESCM}_{\mathrm{excl}} = \frac{\text{Number of bad Event Sequences covered}}{\text{Total number of bad Event Sequences monitored}} \tag{4}$$

As is obvious from the above discussion, TCM, SCM, $\mathrm{ESCM}_{\mathrm{incl}} \rightarrow 100\%$ while, $\mathrm{ESCM}_{\mathrm{excl}} \rightarrow 0\%$. Particularly in the case of $\mathrm{ESCM}_{\mathrm{excl}}$, what we are interested in is making sure that for the test suite unter monitoring (before or after enhancement) none of the bad sequences are being observed. In the case that a bad sequence is observed we can provide a witness to the designer to facilitate the debugging process. In that sense test generation is only directed by the other three metrics.

The Test Plan goals are specified by the user in a specialized Input Language, and are eventually translated into a special form of State Machines, the Event Sequence Finite State Machines (ESFSMs), which are characterized by the presence of non deterministic states. We take advantage of non determinism to describe the timing requirements of the event sequences. The Input Language is described in the following:

```
INCLUDE | EXCLUDE
TE <triggering_event> IMPLIES
CA <consequent_action>
```

The Triggering Event is simply described as a series of Boolean expressions which involve signals and Boolean connectives.

```
{<bool_expr1>,...,<bool_exprN>}
```

The Consequent Action is described in the following form:

```
[Comb_Op]
{<set_of_signals>}{<tw1>,...,<twN>}
[{<set_of_signals>}{<tw1>,...,<twN>},...]
[FOLLOWED]
[{<set_of_signals>}{<tw1>,...,<twN>},...]
```

where Comb_Op can be AND, OR, MUTEX (mutual exclusion).

In the above description tw stands for timing window, a construct used to describe the timing requirements of the event sequences. A timing window (TW) $[t_1, t_2]$ specifies when and for how long a signal is to be asserted, where t_1, t_2 are natural constants with $t_1 \leq t_2$. If $t_1 = t_2$ then $[t_1, t_2] = [t_1]$. The start point is relative to the Triggering Event (TE), infinity is equivalent to the end of simulation and endpoints are inclusive. [0] means that the signal preceding the TW must be asserted at the same cycle as the triggering event. The following list describes the different forms of Timing Windows:

- $[..t_1]$: signal has to be asserted from 0 to t_1 cycles continuously.
- $[t_1..]$: signal has to be asserted from t_1 to ∞ continuously.
- $[t_1..t_2]$: signal has to be asserted from t_1 to t_2 continuously.
- $[*t_1]$: signal has to be asserted at least once from 0 to t_1.
- $[t_1*]$: signal has to be asserted at least once from t_1 to ∞.
- $[t_1, t_2]$: signal has to be asserted at least once from t_1 to t_2.

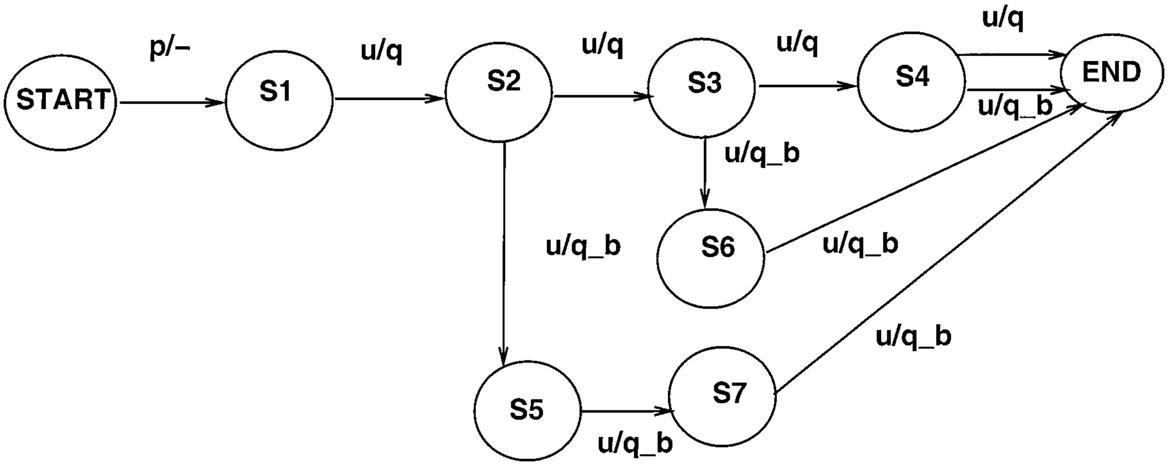

Fig. 2. An FSM for Event Sequence 1.

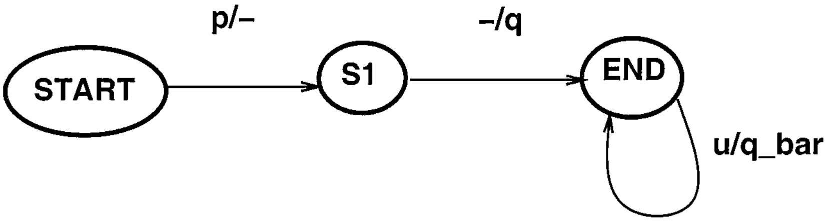

Fig. 3. An FSM for Event Sequence 2.

- [$*t_1$, t_2*]: signal has to be asserted at least once from 0 to t_1 or t_2 to ∞.
- [$t_1*..t_2$]: signal has to be asserted sometime between t_1 and t_2 and remain asserted until t_2.

Figures 2 and 3 illustrate the translation of two example event sequences into ESFSMs. The first sequence indicates that signal q will be asserted 1 cycle after p is asserted and will remain asserted for up to 3 cycles:

```
{INCLUDE TE {p} CA {q}{[1..*4]}
```

The second event sequence indicates that signal q will eventually be asserted 2 cycles after p is asserted:

```
INCLUDE TE {p} CA {q}{[2*]}
```

The timing window in the second event sequence has the semantics of eventuality, thus the semantics of infinity. Therefore, it cannot be represented as an FSM. In cases like these we internally complement the event sequence description and translate this new sequence into an ESFSM. We then check for absence (presence) of bad (good) behavior depending on whether the original event sequence was a “good” (“bad”) sequence.

Our Event Sequence coverage system is depicted in Fig. 4. First the ECFM of the design is extracted and is given as input, along with the functional test suite, to the core of the Event Sequence Coverage System which consists of a 2-value logic simulator and a symbolic engine based on BDDs. Event Sequences are translated into ESFSMs and for each one of them a coverage monitor structure is created. The Coverage Monitor contains both an explicit and a symbolic description of the ESFSM. When the simulation starts for each occurrence of a Triggering Event a Coverage Monitor is instantiated. This facilitates reporting on multiple occurrences of the same event sequence. If a “good” event sequence is not observed during simulation, then the facility is provided for the generation of a test sequence that would activate that event sequence. If a “bad” event sequence is observed during simulation, a

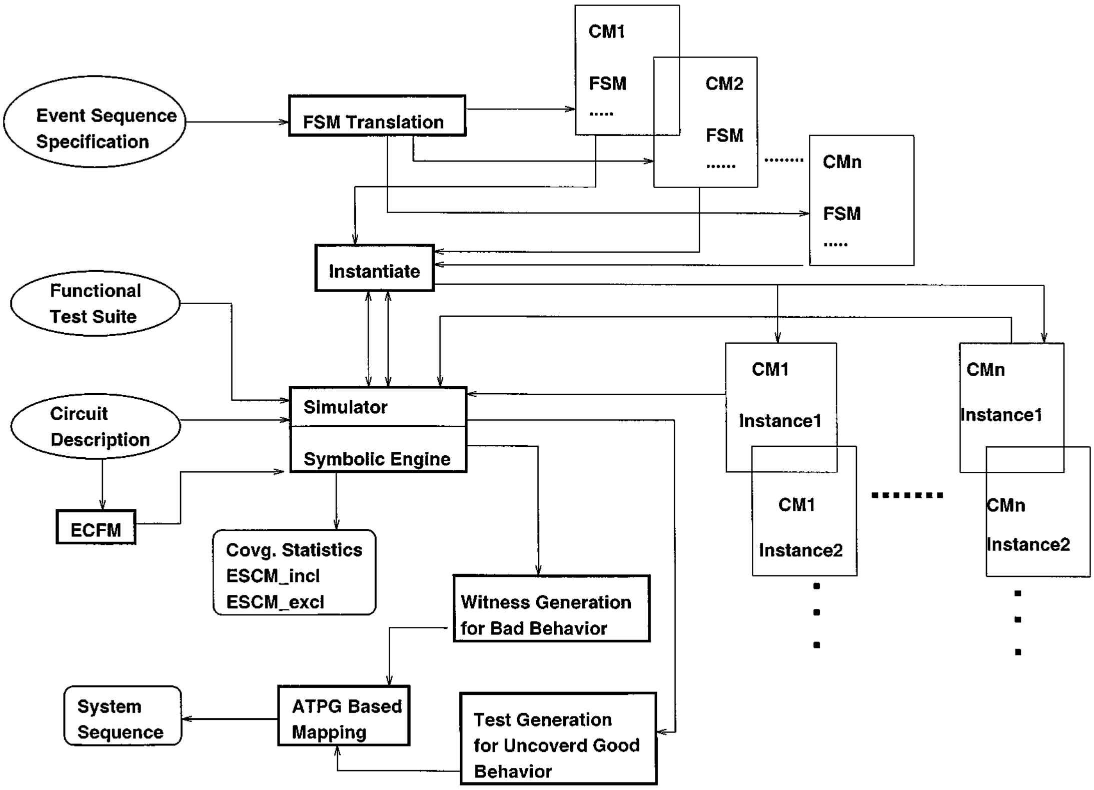

Fig. 4. Event Coverage System.

witness can be generated to help the designer pinpoint what caused the undesirable behavior. The test generation system is described in more detail in the following section.

5. Test Generation

5.1. Coverage Directed Test Generation

Figure 5 describes the Coverage-Directed Test Generation System. In this case SCM or TCM drive the test generation process. There are three main subsystems; A logic simulator, a coverage estimation system based on a symbolic engine that utilizes BDDs and a justification subsystem. The simulator simulates each produced vector on a model of the original circuit, while coverage estimation is done on the ECFM of the circuit. As can been seen the input vectors to the ECFM may consist of both Primary Inputs (PIs) and data registers. The role of the simulator is to provide the values for those ECFM inputs that used to be data registers. Test generation is terminated if a user-specified level of coverage is achieved or if a user-specified time limit is reached. The objective is to come up with a set of transition sequences starting from the initial state whose union will include all transitions in the ECFM graph. The traversal of the graph initially proceeds in a depth-first manner, selecting untraversed transition groups out of each visited state until no more untraversed groups exist. We mark control states out of which untraversed transition groups exist. These states are placed in a set and graded based on two criteria: the number of untraversed transition groups out of them and the number of unvisited next states that they have. We give higher priority to the second factor and break ties based on the first factor. When we reach a state where no more choices exist, we go back to the set of marked states and pick one state based on the discussion above. This introduces a breadth-first flavor in our search and allows us to visit a more diverse part of the ECFM state graph given the time and memory limits. The transition sequence generated starting from a marked state is prepended with a prefix sequence leading to it from the reset state. This sort of “backtracking” sometimes can

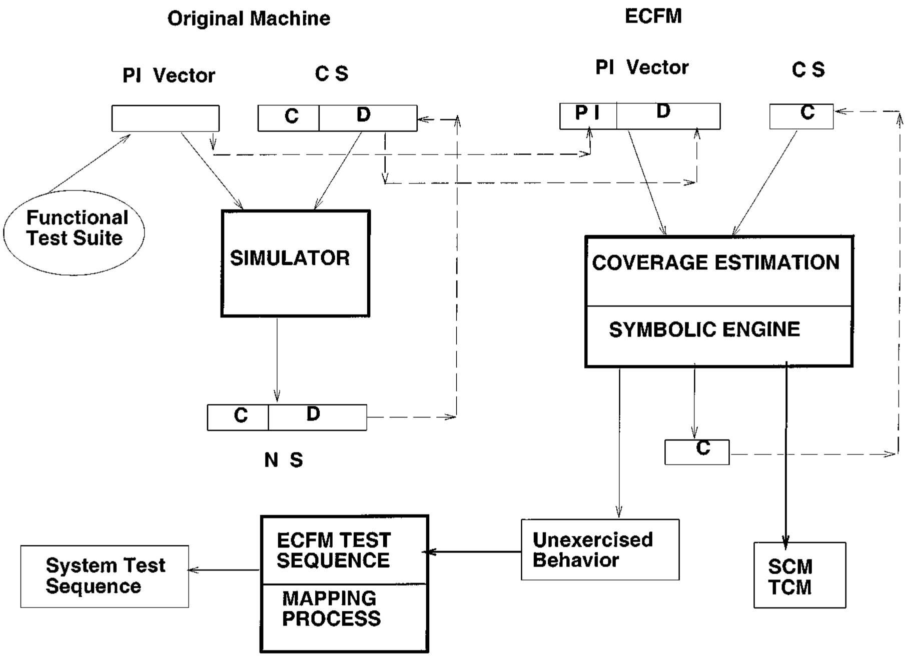

Fig. 5. Coverage-Directed Test Generation System.

lead us all the way back to the reset state, in which case a new transition sequence is started. The input vectors to the ECFM may consist of both Primary Inputs (PIs) and data registers. This means that the transition sequences generated in the way described above cannot be directly used for simulation of the original unit under test. The data values need to be justified, and if that is not possible the part of the transition sequence after the "unjustifiable" vector is dropped. There are several ways in which the justification can be achieved. For example one can use high level functional information to put the right values into the data registers at the right time. A more detailed discussion on this issue can be found in Section 5.3.

5.2. Test Generation for Event Sequences

At this point we need some definitions:

Definition 2. An input sequence that can be applied to a partially specified circuit C in state s such that the destination states for all induced transitions are specified is said to be **applicable** to circuit C in state s.

Definition 3. State s_1 in Finite State Machine M_1 is compatible with state s_2 in Finite State Machine M_2 iff every input sequence that is applicable to M_1 in state s_1 is also applicable to M_2 in state s_2 and causes the two machines to produce the same output sequence when applied to M_1 in initial state s_1 and M_2 in initial state s_2.

In the case of ESFSMs some states are characterized by non deterministic behavior in the sense that the next state is not unique. Therefore, the notion of compatibility has to be modified to account for this non determinism [15].

Definition 4. State s_1 in ESFSM M_1 is loosely k-compatible with state s_2 in Finite State Machine M_2 if s_1 is k-compatible with s_2 for all input sequences of length lesser than or equal to k that contain one and only one applicable input vector for each non deterministic state and all applicable input vectors for each deterministic state.

Definition 5. State s_1 in ESFSM M_1 is loosely compatible with state s_2 in Finite State Machine M_2 if s_1 is compatible with s_2 for all input sequences that contain one and only one applicable input vector for each non deterministic state and all applicable input vectors for each deterministic state.

The problem of proving the loose compatibility of the ESFSM with the design ECFM is formulated as a fixed point problem of discovering the set of states in the design ECFM that are loosely compatible with the start state of the EFSM.

We first compute the set of pairs of 1-compatible states in the ESFSM and the ECFM of the design. The predicate γ given below encodes this set of transition pairs,

$$\gamma(\vec{x}, \vec{q}_a, \vec{q}_b) = \bigwedge_{i=1}^{l} \left(\lambda_{a_i}(\vec{x}, \vec{q}_a) \odot \lambda_{b_i}(\vec{x}, \vec{q}_b)\right) \tag{5}$$

where subscript a is used for the ESFSM and subscript b is used for the ECFM of the design.

To find the set of loosely $k+1$-compatible state pairs LC_{k+1} from the set of loosely k-compatible pairs LC_k, we find the **inverse image** and **pre-image** of the set LC_k, i.e., the pairs of states which must end up in loosely k-compatible states in one transition, and the pairs of states which could end up in loosely k-compatible states in one transition. We further ensure that the transitions to LC_k from its pre-image satisfy the predicate γ.

$$\begin{aligned} \mathrm{LC}_0 &= 1 \\ \mathrm{LC}_{k+1}(\vec{q}_a, \vec{q}_b) &= \mathrm{LC}_k(\vec{q}_a, \vec{q}_b) \bigwedge [(\neg \mathrm{ND}(\vec{q}_a) \\ &\quad \wedge \forall \vec{x} : \mathrm{LC}_k(\vec{Q}_a(\vec{x}, \vec{q}_a), \vec{Q}_b(\vec{x}, \vec{q}_b)) \\ &\quad \bigvee (\mathrm{ND}(\vec{q}_a) \wedge \exists \vec{x} : (\gamma(\vec{x}, \vec{q}_a, \vec{q}_b) \\ &\quad \wedge \mathrm{LC}_k(\vec{Q}_a(\vec{x}, \vec{q}_a), \vec{Q}_b(\vec{x}, \vec{q}_b))))] \end{aligned} \tag{6}$$

where the predicate ND represents the non-deterministic states in the ESFSM. In this equation current state variables $\vec{q}_a, \vec{q}_b$ are substituted by next state variables $\vec{Q}_a, \vec{Q}_b$ and the corresponding next state (transition) functions are composed into the graph.

The set of loosely compatible state pairs WC is obtained when the fixed-point is reached.

$$\mathrm{LC}(\vec{q}_a, \vec{q}_b) = \mathrm{LC}_k(\vec{q}_a, \vec{q}_b) \quad \text{if } \mathrm{LC}_{k+1} = \mathrm{LC}_k \tag{7}$$

The set of ECFM states compatible with the start state of the ESFSM $\vec{q}_{a_{\text{start}}}$ has characteristic function given by

$$\mathrm{LC}(\vec{q}_a, \vec{q}_b)_{\vec{q}_a = \vec{q}_{a_{\text{start}}}} \tag{8}$$

To continue with test generation for Event Sequences, in the case of good behavior not covered we check the compatibility of the start state of the Event Sequence FSM (ESFSM) with the ECFM of the design. If the set of loosely compatible states is non empty we form the product machine (note that in the above equations building the product machine was not required, we just utilized the next state functions and not the transition relation) and traverse the state space from initial states. The traversal is done by picking new states via compatible transitions, while backward traversal is not necessary. The procedure terminates either when a final state is picked as next state or when a previously picked transition is chosen again, since in this case we have entered a loop. The PickOne function just picks one cube from the ON-set of the function given as argument to it. An outline of the algorithm is given in Fig. 6.

In the case of bad behavior observed during simulation we want to provide a witness sequence that will help the designer identify the cause of the problem. We can either identify the vector sequence from the input test suite that caused the bad behavior to occur, or utilize the above described procedure to generate the witness. Our experience indicates that the above procedure usually generates shorter sequences.

During test generation mapping back to the original machine may be necessary according to the discussion in Section 5.3.

```
$s_0$ = Initial State of Product Machine;
$Picked = 0$;
Build Transition Relation($T$);
Build Compatible Transitions($\gamma$);
Build Loosely Compatible State Pairs(LC);
$\mathrm{LC}(\vec{Q}) = \mathrm{Compose}(\mathrm{LC}(\vec{q}), \vec{Q})$;
$i = 0$;
repeat
    $pred = s_i(\vec{q}) \wedge T(\vec{x}, \vec{q}, \vec{Q})$;
    if ($s_i$ contains a **non-deterministic** state)
        $pred = pred \wedge \gamma(\vec{x}, \vec{q})$;
        $pred = pred \wedge \mathrm{LC}(\vec{Q})$;
    endif;
    $pick = \mathrm{PickOne}(pred)$;
    if ($pick \subset Picked$)
        break;
    endif;
    $Picked = Picked \cup pick$;
    $(inp_i, s_{i+1}) = \exists \vec{q} : pick(\vec{x}, \vec{q}, \vec{Q})$;
    $i = i + 1$;
    $s_i(\vec{q}) = \mathrm{Compose}(s_i(\vec{Q}), \vec{q})$;
until ($s_i$ contains END state);
```

Fig. 6. Procedure for Event Sequence Test Generation.

5.3. Justification and Mapping Back

As was mentioned earlier when performing coverage-directed test generation we need to either come up with a sequence which will bring the original machine from the initial state to a specific state or a sequence that will take the machine from a given state to another given state. When performing test generation for event sequences, the test sequence generated on the ECFM is not directly applicable to the original circuit. This sequence has to be expanded and mapped back to the original circuit. First, for inputs that have been abstracted away in the ECFM model, random values have to be provided. This will expand the sequence horizontally. Second, all data conflicts in the generated sequence must be resolved by justifying the values of those PI's which are data registers in the original circuit. This will expand the sequence vertically (see Figs. 7 and 8).

5.3.1. ATPG Based Mapping Back. We propose the use of ATPG techniques for performing both justification and mapping back. We used HITEC [16] to demonstrate the validity of our approach. HITEC can produce a justification sequence for a given state back to a "don't care" initial state.

In general, we have a pair of partially or completely specified states and need to generate a sequence that

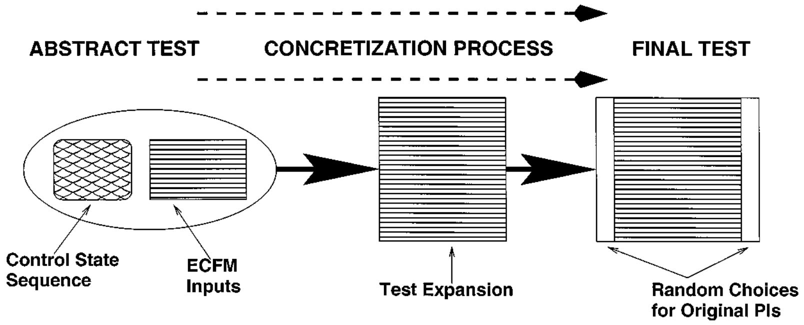

Fig. 7. The mapping back problem.

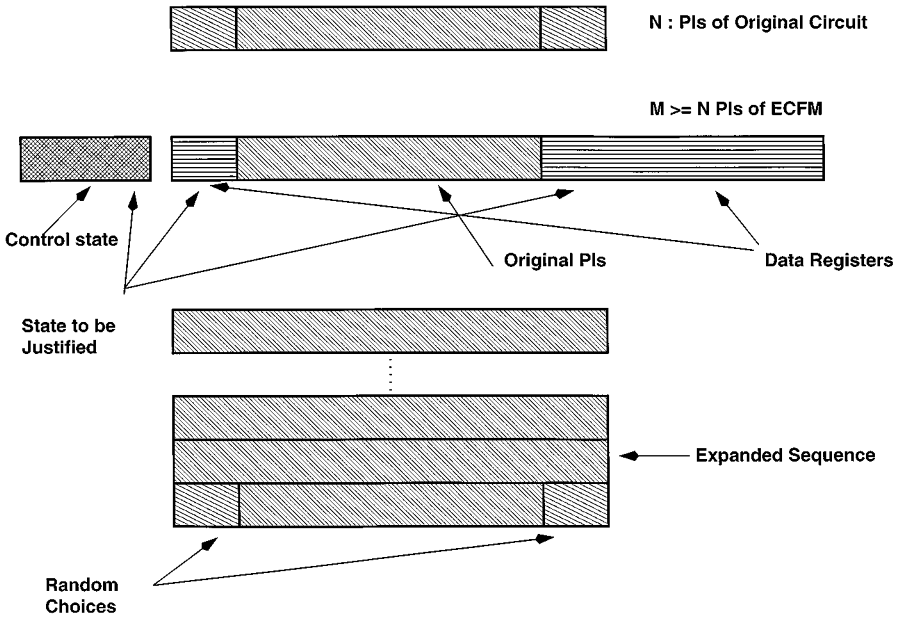

Fig. 8. The mapping back problem-detailed view.

will bring the original machine from the first to the second state. Each of these states consists of a control part retained in the ECFM and a data part which is captured in the transitions of the ECFM. In this case we take the justification sequence generated by HITEC for the second state and look for a subsequence that involves the first state. This subsequence is then incorporated in our test. If the first state is not involved then we try to justify the second state from the reset state. However, the first alternative is preferable since it generally leads to shorter test suites. If HITEC cannot produce a justification sequence then the corresponding vectors are dropped from our test. We force HITEC to generate the justification sequence that we need for a given state by introducing an AND gate "implementing" that state. The output of that gate becomes PO and we introduce a stuck-at-zero fault on it. Additionally we "cache" justification sequences so that if the same state needs to be justified again we do not have to go through the sequence expansion process again.

We believe that code specifically written to provide the functionality required by our technique will improve the efficiency of our approach, both in terms of speed and length of the generated justification sequences. Results shown for verification of sequential circuits using modified ATPG techniques such as "partial justification" [17] provide encouraging evidence for this statement.

5.3.2. Using High Level Information for Mapping Back. Our technique is modeled after the behavioral Test Generation approach. In behavioral level Test Generation a fault model is usually assumed. This can be something like stuck at or stuck open faults for control lines or some way of modeling faults in functional operation blocks (for example a microoperation fault when a arithmetic or relational operator is turned into another). Then a four step process is employed:

- Fault Sensitization
- Justification
- Fault Effect Propagation
- Justification

In our technique we only need to justify required values at the register outputs. So we do not assume any fault model and we do not follow the four steps described above. We support a synthesizable subset of the Verilog HDL. The following statements are supported: IF, CASE, simple FOR loops, blocking assignments, continuous assignments, event control statements. We also support multiple processes as long as the same variable is not being assigned in more than one.

The first step in this process is register identification for which we have two pieces of information. One is the RTL description itself and the other is the information on control latches that is provided by the designer during the ECFM extraction process. We use the following three criteria to identify registers in Verilog.

1. Process has an edge. Everything on the Left Hand Side (LHS) of assignments is considered a latch.
2. A Variable not assigned in every control path is considered a latch.
3. If the Sensistivity List (which is defined as whatever appears in the always process's control event expression) is not a superset of the Input list (which is defined as the collection of variables appearing on RHS of assignments, and contitions of conditional statements) everything on the LHS of assignments is considered a latch.

The circuit description in Verilog HDL is translated into a Control-Data Flow Graph. The Control Flow Graph contains nodes corresponding to conditions and transitions among them, and thus captures sequencing and execution paths. The DataFlow Graph contains nodes corresponding to data registers and operation nodes (data transformation and test). The connection between the two is done by associating control transitions and operation nodes. This will become more clear by means of an example. In Fig. 9 we show the description of a simple counter with reset and clock signals in

```
module counter(clk,rst,ld,in,out);
  input clk,rst,ld,in;
  output out;
  reg [31:0] in,out,creg;
  always begin
    if(rst==1) creg=0;
    else begin
      creg=creg;
      @(posedge clk)
      if(clk==0) creg=creg;
      else if(ld==1) creg=in;
      else if(creg==64) creg=256;
      else if(creg ≤ 512) creg=creg+16;
      else creg=creg-511;
    end
    out=creg;
  end
endmodule
```

Fig. 9. Verilog description of small example.

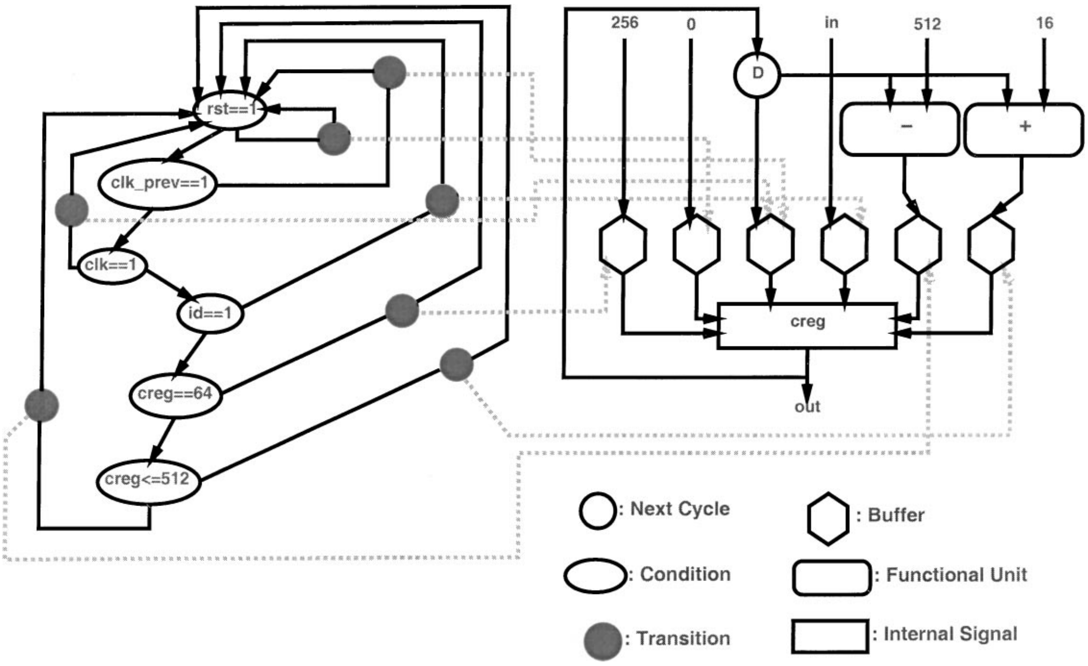

Fig. 10. Control-DataFlow Graph for simple example.

Verilog HDL. In Fig. 10 we show the corresponding Control-DataFlow Graph.

During the justification process, if the initial values are consistent with the required ones then we are done. Otherwise justification has to span multiple time frames. Problems might arise due to counters, timers and structures with large sequential depth. For example counters may present problems because of the unrolling that needs to be performed. The problem will be with something like the reset logic of a counter that may have to be explicitly unrolled to set up initial conditions.

A Data Register **r** can be the target of several assignments within the same process. Assume we need to justify value **v** on register **r**. We pick an assignment **A** with LHS **r** and RHS **e**. The justification is done by assigning **v** to **e**. If **e** is a primary input and the assignment is not controled by conditionals we are done, otherwise several objectives are added to the List of Justification Objectives. Justification may have to proceed through Arithmetic $(+, -)$ and Logic Operators. In the latter case we use standard structural level ATPG rules. For Arithmetic Operations the number of choices needs to be controlled. Justifying through an adder for example presents a lot of choices which we cannot allow the tool to explore exhaustively. If conflicts arise during this process we have to backtrack. There are two categories of backtrack points: one has to do with which assignment was picked for the register and the other is related to the values selected for justification through logic or arithmetic operators. Finally each assignment and justification constraint is given a time tag to signify when it is supposed to take place.

To illustrate this process consider the example described in Fig. 9 again. Assume that we want to justify 256 on the output. One way would be to get it from the input, but let us assume for illustration purposes that we want to achieve this utilizing an internal assignment. That means that we need to put the value 256 to the creg. Table 1 shows the constraints and values for inputs to do that. However, it is clear that we need to justify the value 64 on creg before we can get to 256. So we go back in time to do that (two more time frames as Table 2 shows).

The collection of all constraints at a given time point is kept as a BBD. The BDD can become incosistent as a result of a justification operation at which point backtracking is initiated. We use the following Boolean encoding: Assuming that N is the number of control states, a state $v \in N$ is encoded as an assignment $\psi_N(v)$ to a vector $\vec{v}$ of n Boolean variables with $n \geq \lceil \log(N) \rceil$ For a given m constraints at a given time frame we introduce a vector $\vec{k}$ of m Boolean variables $k_1, \ldots, k_m$

Table 1. Justification constraints.

Time frame	PI vectors	Statement
1	(CLK==0,T)	OUT=256
2	(CLK==1,T)(RST==0,T)(LD==0,T)(OUT==64,T)	OUT=256

Table 2. Justification constraints contd.

Time frame	PI vectors	Statement
1	(CLK==0,T)	OUT=256
2	(CLK==1,T)(RST==0,T)(LD==1,T)(IN==64,T)	OUT=256
3	(CLK==0,T)(RST==0,T)	OUT=256
4	(CLK==1,T)(RST==0,T)(LD==0,T)	OUT=256

where k_i is set to 1 when the corresponding constraint occurs positively. Constraints which are complement of each other are encoded with complemented variables. It is a unification process over a Boolean algebra.

For optimization purposes we use memorization (caching) so that we do not repeat previously done work. All generated justification sequences along with the initial objective are cached for future reuse. Other less obvious optimizations are based on "semantic" principles. If what needs to be justified is a set of bits belonging to a status register it is not important how that is accomplished as long as it is. Also in some cases the relative values of two data registers are of importance and not their absolute values. In both of these situations previous justification sequences can be reused.

6. Experimental Results

We applied our methodology to two microprocessors. The circuit statistics are given in Table 3. The first four columns provide information about the original circuit (number of primary inputs and outputs and DFFs in the circuit). The remaining four columns provide statistics for it's ECFM. Column 8 gives the extraction time of the ECFM model. The number of DFFs in the ECFM is significantly smaller than the number of DFFs in the original circuit while there is an increase in the number of primary inputs. The number of POs in the ECFM is either the same as the number of POs in the original circuit or smaller depending on whether the circuit has data outputs. Data outputs are discarded in the ECFM model.

Table 3. Circuit statistics.

	Original			ECFM			
Circuit	PIs	POs	DFFs	PIs	POs	DFFs	Extraction
Viper	33	53	251	75	1	5	8.8s
gl85	17	27	256	44	11	14	8.5s

The Viper microprocessor [8] is a 32-bit microprocessor. It has four general purpose registers, two ALU registers along with a memory address register and an instruction register. The netlist has approximately 6000 gates. Memory is not included in this description. The approximately 40 instructions include arithmetic, comparison, Boolean instructions along with instructions for reading and writing to the memory. The Viper halts operation if an exception is raised, which occurs if an illegal instruction is fetched or overflow occurs. The gl85 circuit is a model of the 8085 microprocessor. This model uses 8-bit input and output buses in place of the 8-bit bidirectional address-data bus. The instruction set includes data transfer, arithmetic, logic, branch, stack, IO, machine control instructions. Each instruction has one, two or three bytes. Operation of the gl85 proceeds under the control of two state machines. The gl85 is considerably more complex than the viper and traditional ATPG cannot deal effectively with it. To illustrate this we attempted ATPG directly on the gl85 model on a 200 MHz UltraSparc II with 1GB of memory. After 11 hours of CPU time the efficiency achieved was only 17%. This emphasizes the need to employ abstractions as proposed in this paper. This is also the reason why we did not use a test suite generated by an automated tool like CRIS or HITEC for gl85.

Table 4 presents our validation coverage analysis results on the above circuits. Column 2 gives the number

Table 4. Functional Coverage results.

Circuit	Vectors	s-a Cov (%)	SCM (%)	TCM (%)	Time (mins)
Viper	5959	91.46	100	91.46	8.84
gl85	24307	79.78	93.81	29.15	37.63

Table 5. Functional Test Generation results.

			#Just			SA Covg. (%)	
Tool	Circ.	#Vec.	Total	Succ.	Time (s)	All	Control
FTGEN	gl85	57307	2472	1417	200000	53.10	72.98
FTGEN	Viper	19813	1856	1373	9867	81.78	94.17
FTGEN*	Viper	26200	1856	1687	7100	82.82	95.21

of test vectors applied. Columns 4 and 5 give the SCM and TCM coverage metrics for the corresponding circuits and Column 6 gives the time to compute the coverage metrics in minutes. Take the Viper as an example. Since the state space for the Viper is in the order of 10^{75}, it is not possible to compute the reachable state space using current techniques. Most of the state space is due to the register file. So we evaluate the functional coverage on the ECFM of the Viper (for which formal verification techniques are applicable) which has 32 states out of which 17 are reachable from the initial state. Our system also identified instances of unexercised behavior. For example the circuit does not enter the halt state with the comparison flag bit set. This implies that an instruction causing the machine to enter the halt state was not tested after an instruction causing the flag to be set. For the gl85, 7296 states out of the possible 16384 states in the ECFM are reachable.

Table 4 also correlates the stuck-at fault coverage and the functional coverage for the test suites that we utilized in our experiments. The test sequence used for the viper was obtained using CRIS [19]. The test sequence for gl85 was taken from a manually generated functional test [20]. This table is included because the SCM and TCM coverage metrics are used to drive the test generation process as described earlier.

Although the relation between our functional coverage metrics and the stuck-at fault coverage is not clear, an interesting observation can be made. High stuck-at coverage does not guarantee high functional coverage. However, we cannot say anything conclusive about high TCM coverage providing high stuck-at coverage. Furthermore, our metrics are designed to gauge functional coverage on a software model of the design where every signal is observable, as opposed to stuck-at fault coverage which is associated with manufacturing tests designed to be applied on the actual chip.

Table 5 presents the results on Coverage-Directed Test Generation for the two microprocessors. Our objective is to get 100% state coverage, or terminate the process when a timeout limit of 200000 seconds is reached. The first column specifies the tool used to obtain the corresponding results. FTGEN is the tool that utilizes ATPG techniques for justification and mapping back, while FTGEN* is the tool that utilizes the approach described in Section 5.3.2. The 3rd column gives the number of vectors generated, the 4th column gives the number of times justification was necessary, the 5th column shows the number justification was succesfull, the 6th column gives the overall time required, and the last two columns give the stuck-at fault coverage that the generated sequences achieve, for the complete fault list or the control fault list (faults present in the ECFM). Although stuck-at fault coverage is not necessarily an accurate measure of the quality of a functional test sequence, we believe that it is a good indication of its effectiveness. For the viper the objective of 100% state coverage was achieved while for gl85 the process timed out. As can be seen for the viper FTGEN* produces a sequence that achieves better coverage faster. The reason why FTGEN* produces better results on the viper is the fact that we are able to succesfully justify more objectives than when using ATPG (in FTGEN) and that results in less number of vectors being dropped. Test sequences are quite lengthy, which is a common characteristic of functional test generation. However, this approach produces a test sequence with much better coverage compared to directly applying ATPG on the circuits. Table 5 gives some indication that doing the justification at the RT-Level is a good alternative to other proposed methods [5, 21, 6].

We then utilized the sequence of 5959 vectors on the viper to monitor for specific event sequences. The whole process takes 18 mins. We are basically monitoring for good behavior and we are using 32 event sequences; 27 monitoring instruction decoding, 3 monitoring overflow conditions, 1 monitoring illegal instructions. The results are given in Table 6. The objective of this experiment is to measure how many user provided event sequences an independently generated test sequence covers, and augment that sequence with additional tests that exercise interesting uncovered behaviors.

Table 6. Control Event Sequence Coverage results.

31 Event Sequences monitoring	Type	Instances
	Comparison	335
	Boolean	478
Instruction	Arithmetic	93
Decoding(#27)	Call	45
	Memory	54
	I/O	87
Illegal instructions(#1)		310
	Addition	12
Overflow conditions(#3)	Subtraction	11
	Left Shifting	2

The Event Sequences used are given in the following:

- Illegal Instructions

```
INCLUDE TE {!CF && ((IR23&&IR22&&IR21&&IR20)||(IR23&&IR22&&IR21&&
            !IR20)  ||  (IR23&&IR22&& !IR21 &&IR20))}
        CA {STOP}{[1]}
```

- Overflow due to Addition

```
INCLUDE TE {!CF && !IR23 &&IR22 && ! IR21 && IR20)}
        CA {OVF}{[5]}
```

- Overflow due to Subtraction

```
INCLUDE TE {!CF && !IR23 &&IR22 && IR21 && IR20)}
        CA {OVF}{[5]}
```

- Overflow due to Left Shifting

```
INCLUDE TE {!CF && IR23 &&IR22 && !IR21 && !IR20 &&MF1 && !MF0)}
        CA {OVF}{[5]}
```

As was mentioned earlier the test sequence utilized does not test some instances of behavior. We generated an event sequence which has the setting of register B as the triggering event and the raising of the STOP flag within 2 to 6 cycles as the consequent action. Raising of the STOP flag signifies entering of the HALT state. We utilized the algorithms described in Section 5.2 for event sequence test generation and generated a sequence of two instructions that will cause the STOP flag to be set and the machine to enter the HALT state immediately after that. The first is a comparison instruction and the second is an illegal instruction. This process took 17.1 seconds. We then expanded this sequence by justifying the values that needed to be loaded to registers R and M, the registers used by the comparison instruction. This was done in 11 seconds.

We are currently working on applying these techniques to the gl85 microprocessor, as well as the DLX, a pipelined example.

7. Conclusion

In this paper we have addressed the problem of design validation by considering two very important issues associated with it. The first is the issue of functional coverage. We extended the previously introduced state and transition coverage metrics with additional metrics targeting control event sequences. Secondly we provided two techniques in automatic test generation. The first aims at generating critical transition tours in the control state space of the design (ECFM) and the second aims at exercising uninstantiated event sequences. Our future work involves the application of these techniques to larger more complicated examples(with emphasis on pipelined designs). Additionally we would like to be able to develop an iterative framework in our test generation approach where after some ECFM sequences have proven to be unrealizable in the context of the complete machine, other test sequences would be generated. Ideally one would like to be able to recognize unrealizable ECFM sequences on the fly. Finally we would also like to investigate the possibility of automating the process of discovering the event control sequences that need to be exercised. Currently we rely on the designer to provide these sequences.

References

1. Y.V. Hoskote, D. Moundanos, and J.A. Abraham, "Automatic Extraction of the Control Flow Machine and Application to Evaluating Coverage of Verification Vectors," *Proc. ICCD*, 1995, pp. 532–537.
2. R. Ho and M. Horowitz, "Validation Coverage Analysis for Complex Digital Designs," *Proc. ICCAD*, 1996, pp. 146–151.
3. J. Burch, E. Clarke, K. McMillan, and D. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," *Proc. 27th DAC*, 1990, pp. 46–51.
4. D. Moundanos, J.A. Abraham, and Y.V. Hoskote, "Abstraction Techniques for Validation Coverage Analysis and Test Generation," *IEEE Transactions on Computers*, Vol. 47, No. 1, pp. 2–14, 1998.
5. R. Ho, C. Yang, M. Horowitz, and D. Dill, "Architecture Validation for Processors," *Proc. 22nd International Symposium on Computer Architecture*, 1995, pp. 404–413.
6. D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal, "Coverage-Directed Test Generation Using Symbolic Techniques," *Proc. Formal Methods in CAD*, 1996.
7. M. Pierre, S. Yang, and D. Cassiday, "Functional VLSI Design Verification Methodology for the CM-5 Massively Parallel Supercomputer," *Proc. ICCD*, 1992, pp. 430–435.
8. S. Devadas, A. Ghosh, and K. Keutzer, "An Observability-Based Code Coverage Metric for Functional Simulation," *Proc. ICCAD*, 1996, pp. 418, 425.
9. K. Cheng and A. Krishnakumar, "Automatic Functional Test Generation Using the Extended Finite State Machine Model," *Proc. 30th DAC*, 1993, pp. 86–91.
10. A. Chandra, V. Ivengar, D. Jameson, R. Jawalekar, I. Nair, B. Rosen, M. Mullen, J. Yoon, R. Armoni, D. Geist, and Y. Wolfsthal, "AVPGEN-A Test Generator for Architecture Verification," *IEEE Transactions on VLSI Systems*, Vol. 3, No. 2, pp. 188–200, 1995.
11. K.D. Jones and J.P. Privitera, "The Automatic Generation of Functional Test Vectors for Rambus Designs," *Proc. DAC*, 1996, pp. 415–420.
12. M. Abadir and H. Reghbati, " Functional Specification and Testing of Logic Circuits," *Comp. and Math. with Appls.*, Vol. 11, No. 12, pp. 1143–1153, 1985.
13. P. Chung, Y. Wang, and I. Hajj, "Diagnosis and Correction of Logic Design Errors in Digital Circuits," *Proc. 30th DAC*, 1993, pp. 503–508.
14. S. Kang and S. Szygenda, "Design Validation: Comparing Theoretical and Empirical Results of Design Error Modeling," *IEEE Design and Test*, 1994, pp. 18–26.
15. Y.V. Hoskote, "Formal Techniques for Verification of Synchronous Sequential Circuits," Ph.D. Dissertation, ECE Dept., UT Austin, 1995.
16. T.M Niermann and J.H. Patel, "HITEC: A Test Generation Package for Sequential Circuits," *Proc. of EDAC*, 1991, pp. 214–218.
17. S.-Y. Huang, K.-T. Cheng, and K.-C. Chen, "AQUILA: An Equivalence Verifier for Large Sequential Circuits," *Proc. of Asia-Pacific DAC*, 1997.
18. W. Cullyer, "Implementing Safety-Critical Systems: The Viper Microprocessor," *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. Subrahmanyam (Eds.), Kluwer Ac., 1988, pp. 1–26.
19. D. Saab, Y. Saab, and J.A. Abraham, "Cris: A Test Cultivation Program for Sequential VLSI Circuits," *Proc. ICCAD*, 1992, pp. 216–219.
20. Jian Shen and J.A. Abraham, "Native Mode Functional Test Generation fot Microprocessors With Application to Self Test and Design Validation," *Proc. ITC*, 1998, pp. 990–999.
21. D. Moundanos, J.A. Abraham, and Y.V. Hoskote, "A Unified Framework for Design Validation and Manufacturing Test," *Proc. ITC*, 1996, pp. 875–884.

Dinos Moundanos received the B.S. degree in Computer Science from the National Technical University of Athens, Greece in 1992 and the MSEE and Ph.D. degrees from the Electrical and Computer Engineering Department at the University of Texas at Austin in 1994 and 1998 respectively. He is currently with the Advanced CAD Research Group at the Fujitsu Labs of America in Sunnyvale, California where he is involved in developing formal verification and validation technology for next generation designs. His research interests include fault tolerance, verification and validation of hardware systems, computer architecture and programming languages.

Jacob A. Abraham (S'71-M'74-SM'84-F'85) received the B.S. degree in electrical engineering from the University of Kerala, India in 1970. He received the M.S. degree in Electrical Engineering and the Ph.D. degree in Electrical Engineering and Computer Science from Stanford University, Stanford, California in 1971 and 1974 respectively. He is a professor in the Department of Electrical and Computer Engineering, director of the Computer Engineering Research Center, and holds a Cockrell Family Regents Chair in Engineering at the University of Texas at Austin. From 1975 to 1988 he was a member of the faculty of the University of Illinois at Urbana-Champaign. His research interests include VLSI design and Test, formal verification and fault-tolerant computing. He is the principal investigator for several contracts and grants in these areas and a consultant to industry and government on testing and fault-tolerant computing. He has written more than 200 publications and supervised more than 40 Ph.D. dissertations. Dr. Abraham is a member of the ACM and Sigma Xi and a fellow of the IEEE. He has served as an associate editor for the IEEE Transactions on VLSI Systems and as chair of the IEEE Computer Society Technical Committee on Fault-Tolerant Computing. He currently serves on the editorial board of the Journal of Electronic Testing: Theory and Applications.