D. Moundanos, J. A. Abraham and Y. V. Hoskote,
"A Unified Framework for Design Validation and Manufacturing Test,"
Proceedings, IEEE International Test Conference, Washington, D.C., October 1996.

A Unified Framework for Design Validation and Manufacturing Test *

Dinos Moundanos Jacob A. Abraham
Computer Engineering Research Center
University of Texas at Austin
Austin, TX 78712

Yatin V. Hoskote
Intel Development Labs
Intel Corporation
Hillsboro, OR 97124

Abstract

New approaches to address the difficult problems in test are necessary if its current status as a major bottleneck in the production of quality integrated circuits is to be changed. In this paper we propose a new direction for solving the test problem using powerful methods already employed for the formal verification of large circuits. More specifically, we will discuss how abstraction techniques can assist conventional ATPG tools when attacking hard to detect faults. The same abstractions can also be used in design verification to increase the level of confidence in a design following simulation, by providing a meaningful measure of the coverage achieved by the verification vectors. In this sense, our approach is geared toward providing a unified framework for design validation and manufacturing test.

1 Introduction

The rate at which design and fabrication technologies are advancing is causing test tools to fall behind rapidly. Testing is now, more than ever, becoming a major bottleneck in the production of quality integrated circuits. Ideally, one would like to deal with test early in the design cycle, in a way that would make it easier to address the difficulties commonly encountered later when testing large and complex circuits. For example, the designers could identify certain areas in the design that would benefit from adding testability features and use synthesis techniques to add those features at the behavioral level of description.

In this paper we suggest another way in which the structure-based testing process can benefit from an early examination of high-level information. Designs are specified in a language like VHDL, Verilog or even C, and are then synthesized automatically from these high level specifications by CAD tools. In our approach we combine abstraction techniques with well known formal verification techniques at the behavioral level of description with the objective of assisting the testing process in two ways. First, we identify redundancies which the lower level ATPG tools need not address, and second, we provide test vectors (generated in a very fast and efficient fashion) that will detect hard faults on which conventional test generators spend a lot of resources, or even abort.

Our approach is based on the Extracted Control Flow Machine (ECFM) model introduced in [10]. The extraction of the control flow machine can be done at any level of description and is based on the underlying mathematical model of Finite State Machines(FSMs). Most importantly, the extraction is independent of the description style that the designers use to specify the circuit. The ECFM has a much smaller state space but exhibits the same control flow as the original circuit. Most contemporary designs have a datapath of substantial size which contributes to a huge state space. However, efficient techniques are available for testing datapath circuitry at lower levels of abstraction. In our experience, attempting to cover the control part of the state space is enough to exercise most of the significant design errors.

We also demonstrate how our abstraction model can be used to provide valuable information in a design verification environment. Design verification is the process of checking the conformity of a design to its functional specification very early in the design cycle. While significant progress has been made in automating the verification process and using formal verification tools at the lower levels of the design hierarchy, simulation is still the primary means of checking the correctness of a high-level description. Functional tests attempting to exercise the behavior of the design are generated (usually manually) long before the low level implementation is available. We provide quanti-

*This research was supported in part by the Semiconductor Research Corporation under Contract 95-DP-388 and in part by the Texas Advanced Technology Development and Transfer Program under Project 003658-469 at the University of Texas at Austin.

D. Moundanos, J. A. Abraham and Y. V. Hoskote,
"A Unified Framework for Design Validation and Manufacturing Test,"
Proceedings, IEEE International Test Conference, Washington, D.C., October 1996.

tative metrics that give a pragmatic and meaningful measure of the quality of these tests. There is, of-course, a difference between design verification which is contacted on a software model and physical testing which is performed on the actual chip; all signals and latches are observable during simulation-based verification.

Our design validation and test framework consists of three phases. The first step involves extracting the ECFM model from the VHDL description. We assume that we are provided with a functional test suite for a given design. Based on the ECFM we calculate coverage metrics to quantify the quality of this test suite and identify instances of behavior not covered by the verification vectors. The second step involves the automatic generation of vectors to exercise the untested behavior. The same vectors can be used for the detection of hard stuck-at faults. The third step of our framework targets stuck-at faults exclusively. For a set of hard to detect faults in the control part of a circuit we use our abstraction techniques combined with formal verification techniques to either classify them as redundant or generate tests to detect them.

The rest of the paper is organized as follows: In section 2 we briefly discuss previous work in this area and clearly distinguish it from our work. In section 3 we review the ECFM model. In section 4 we describe our validation framework and in section 5 we present experimental results on a microprocessor. We conclude and give directions for future work in section 6.

```
architecture BEHAVIOR of EXAMPLE is
 type STATE is (S1, S2, S3);
 signal CS : STATE;
 signal r, m, regA, regB : bit-vector(15 downto 0);
begin
process
begin
 wait until CLK = '1' and not(CLK'stable);
 case CS is
   when S1 =>
     flag-bit <= '0';
     if (START = '1') then
       CS <= S2; r <= regA; m <= regB;
     else
       CS <= S1;
     end if;
   when S2 =>
     CS <= S3; if (r < m) then flag-bit <= '1'; end if;
   when S3 =>
     CS <= S1;
     if (flag-bit = '1') then
       regA <= m; regB <= r;
     end if;
     DONE <= '1';
 end case;
end process;
end BEHAVIOR;
```

Figure 1: VHDL description of simple example

2 Related Work

Some researchers have utilized test suites generated from conventional ATPG tools used for structural testing to do design verification. For example, the approach followed in [1, 11] 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 it is effective at the gate or lower levels in catching common designer mistakes, it is primarily structural and does not give an indication of the extent to which the behavior of the design has been exercised. The coverage measure used in our approach depends only on the functional behavior and does not require the availability of a final synthesized net list. Other researchers have earlier developed a fault model called "single-state-transition (SST) fault" [4] in the state transition level which causes the destination state of a transition to be faulty while retaining its correct I/O behavior. Test generation for SST faults is a three-step process using explicit state enumeration techniques. The tests generated for SST faults are then used to evaluate stuck-at fault coverage. However, this approach is only effective for small circuits.

Information obtained from the high-level description has been used to produce stuck-at fault test suites or to assist a lower level test generation tool. In [5], an extended finite state machine (EFSM) model is extracted from the behavioral description and is exhaustively traversed to generate tests that are later used for stuck-at fault testing. The EFSM is a state machine with far fewer states than the machine representing the original circuit and exhaustive traversal of this machine ensures that all statements in the original behavioral program are executed. However, this machine is not small enough for some complex circuits. Our approach differs from this in two respects. First, the EFSM extraction depends heavily on the description style of the HDL code, unlike our extraction procedure. Secondly, the coverage measure used is quite different. We believe that checking execution of every statement in the HDL program is not an accurate measure of functional (behavioral) coverage.

D. Moundanos, J. A. Abraham and Y. V. Hoskote,
"A Unified Framework for Design Validation and Manufacturing Test,"
Proceedings, IEEE International Test Conference, Washington, D.C., October 1996.

For example, consider the portion of behavioral VHDL code in Figure 1. In this simple VHDL program the values of the two 32-bit registers regA, regB get swapped if the value of regA is less than the value of regB or remain unchanged otherwise. Setting START low in the first clock period and then using a value of 10, for example, in regA and 11 in regB with START high in the second clock period will exercise all the statements in the VHDL program and should imply 100% functional coverage. However, as is obvious, we have not tested the case where regA is not less than regB and no swapping takes place. Thus, even in such a simple machine, HDL coverage is seen to be an incorrect measure because of its inherent dependence on the program's syntactic style. The work in [9] bears similarities to the abstraction techniques employed in our framework in attempting to use an FSM model of microprocessor control logic to generate test programs used in simulation. However, the authors address the problem of mapping back to the original circuit by artificially injecting the appropriate values during simulation, while we propose the use of conventional ATPG techniques to expand the test sequence generated on the abstracted model. Additionally we employ the abstraction techniques to address the problems of redundancy identification and manufacturing test.

Formal verification techniques have been used before in an ATPG context mainly for redundancy identification [12, 6]. Our method of identifying redundancies is quite similar to the approaches described in these papers, but we focus on the high-level behavior. Thus, by employing abstraction techniques we are able to handle much bigger circuits. In the next section we will review the extracted control flow machine.

3 The Extracted Control Flow Machine Model

Most practical circuits have a large data state space which is intractable by present techniques (even those employing implicit state enumeration). In most applications, including design verification and test generation, the flow of control is of prime interest. However, extracting the control machine from a circuit description is not an easy task at any level of detail, including the behavioral level, because the control circuitry cannot be distinguished easily from the data path.

The ECFM of a sequential circuit is extracted by looking at the finite state machine representing the complete circuit. The difficulty in identifying the control circuitry often lies in defining the interface of the control unit with the rest of the circuit, and not in differentiating the control registers from registers holding pure data. In the ECFM methodology, the designer determines which registers are to be considered as contributing to the control state space and which make up the data. The key issue here is that we are not interested in the data part of the state space. Consequently, we drop the data registers from the circuit and determine the dependency set of the primary outputs and control state variables to find the new extended set of circuit inputs.

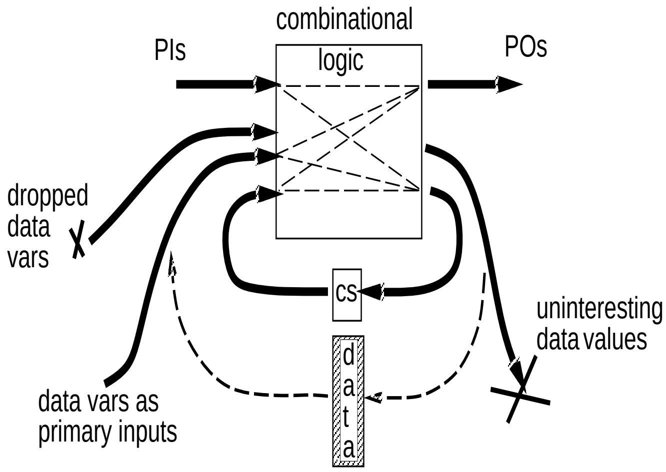

Figure 2: Extracting the ECFM from the FSM representing a sequential circuit

To better understand this process, consider the finite state machine in Figure 2. The shaded registers correspond to data registers. They are dropped from the circuit since we are not interested in the data part of the state space. We are also not interested in the actual data values, so we drop the inputs to these data registers from consideration (marked by the big cross on the right in Figure 2). We now determine the dependency set (shown by dotted lines) of the primary outputs and control state in terms of the primary inputs, control state variables and data variables. This forms the new set of primary inputs and may include some data variables. The rest of the data variables are dropped (marked by a small cross on the left in Figure 2). This dependency set is determined simply by looking at the support set of the output and next-state functions for the modified machine. Finally, **equivalent transitions** in this modified machine are grouped together. Two transitions are equivalent if they have the same effect on the control flow, i.e., they have the same current state, the same next state and they modify the output behavior of the machine in the same manner.

D. Moundanos, J. A. Abraham and Y. V. Hoskote,
"A Unified Framework for Design Validation and Manufacturing Test,"
Proceedings, IEEE International Test Conference, Washington, D.C., October 1996.

The actual state space of the ECFM is significantly smaller than that of the original circuit while still exhibiting the same control flow. Furthermore, introducing a wider datapath in the actual circuit does not result in any increase in the ECFM state space, which demonstrates the applicability of the model to even larger circuits. Note that the transitions now depend on an extended input set which includes data variables. While it may seem that the state space has decreased at the expense of the input space, the increase in input space is minimized by the grouping of equivalent transitions. Also, large input spaces are more easily handled by our implicit enumeration procedures than large state spaces. The ECFM retains the primary outputs of the actual circuit. Every state transition in the ECFM involves only assignment to primary outputs and no update of data.

As discussed earlier, it is not an easy task to extract a correct or satisfactorily approximate control machine from an HDL description since the extraction process depends on heuristics and the syntactic style. On the other hand, the ECFM extracted from the underlying state machine is independent of the description style of the HDL program and encodes the complete control flow in the original circuit. Note that synthesis of the complete underlying state machine from a VHDL program is a trivial task, given the state-of-the-art in automatic synthesis.

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 variables of the original circuit are primary inputs 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. For a more detailed discussion on the ECFM methodology the interested reader can refer to [10].

4 Description of a Unified Framework for Multi–level Design Validation and Test Generation

In this section we will describe the manner in which our abstraction techniques can assist or supplement existing methods of design validation. Our framework is applicable to many levels in the design hierarchy: from the behavioral level where early design verification is done, down to the gate level where structure-based testing is performed. We will first demonstrate our contribution toward improving any given simulation based design validation approach and then describe our approach to test generation for large circuits.

4.1 Computing Functional Coverage

We now describe how the ECFM can be used to derive a pragmatic estimate of the functional coverage provided by a suite of simulation test sequences. At this point we are assuming that the functional test suite (either generated manually or by means of an automatic process) is provided to us and we present a method for estimating the control flow coverage provided by that suite and ways to improve this coverage. This capability is critical, since it is obvious that design errors cannot be detected if the simulation never exercises the erroneous behavior.

4.1.1 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) [10]. Given an initial state or a set of possible initial states, we first compute the part of the design space that is actually reachable from those state(s). This is an essential pruning step to avoid looking at parts of the design that cannot be reached from the start state(s). The next step is to look at what part of the reachable state space is covered by the test vectors.

The two metrics are then given as,

$$SCM = \frac{Number\ of\ states\ visited}{Total\ number\ of\ reachable\ states} \quad (1)$$

$$TCM = \frac{Number\ of\ transitions\ traversed}{Total\ number\ of\ reachable\ transitions} \quad (2)$$

Obviously, TCM is the most comprehensive metric of the two. However, in certain cases SCM can give the designer useful hints as to how the quality of the tests can be improved. 100% SCM can be the first target. Covering all states first makes covering the task of covering the remaining transitions easier. Providing pointers to the part of the state space that is not covered by the given tests is also of paramount importance. This can greatly facilitate additions to the test suite to achieve better coverage. One should note that the ideal metric for calculating the functional coverage achieved by a test suite would be the the number of execution paths exercised by that suite, since in

D. Moundanos, J. A. Abraham and Y. V. Hoskote,
"A Unified Framework for Design Validation and Manufacturing Test,"
Proceedings, IEEE International Test Conference, Washington, D.C., October 1996.

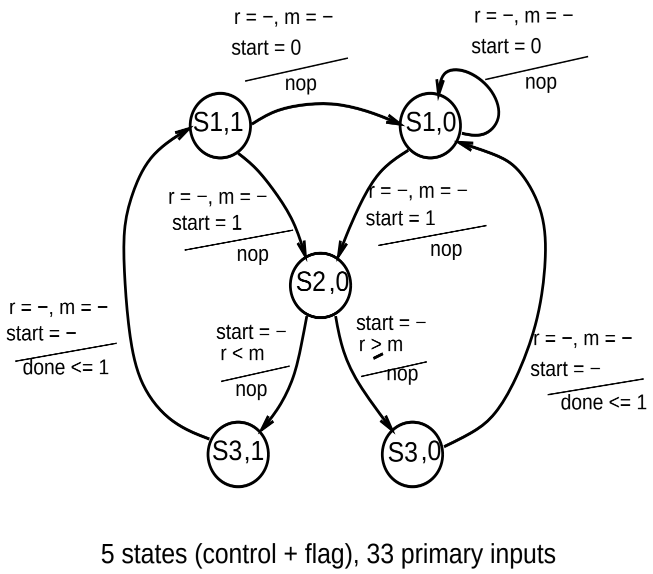

Figure 3: ECFM for circuit described in Figure 1

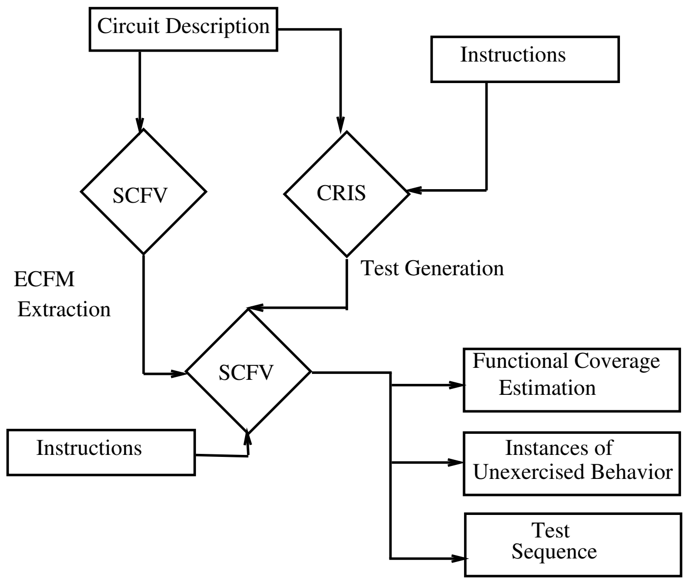

Figure 4: System Outline

many cases the order in which certain transitions are traversed is important. However this has exponential complexity, and so the next best alternative is the metrics defined above.

The biggest stumbling block to this straightforward approach has been the size of circuits. Computing the set of reachable states for large circuits is not currently feasible even when implicit enumeration methods [15] are being used. However, it is important to realize that it is not the designer's goal to exercise all transitions of the state machine as this will be prohibitively expensive for circuits with a data path of even moderate size. What designers are interested in, is whether the control behavior of the circuit has been fully tested or not. Given this fact, the ECFM model can be used for this purpose. The coverage metrics are now computed on the ECFM model and reflect the amount of control behavior exercised during the simulation. Thus, the set of simulation sequences and other data values collected during the simulation runs are fed to the ECFM after simulation of the original circuit is carried out and coverage of the simulation suite is determined.

Consider the ECFM for the circuit described in Figure 1, which is shown in Figure 3. The dashes denote don't cares and grouping of equivalent transitions is not shown. Grouping would result in just 8 groups of transitions. In this ECFM the flag-bit is considered part of the control state. Registers regA and regB are pure data and are dropped from the circuit. However, **r** and **m** are considered primary inputs because the control flow depends on them. Ensuring that every transition in the ECFM, modulo transition equivalence, is exercised during verification ensures that both cases of swapping registers **regA** and **regB** and leaving them untouched are covered. The values to be fed as data variables **r** and **m**, which are primary inputs to the ECFM, can easily be determined during the simulation runs of the actual circuit performed earlier during design verification. These values are fed to the ECFM along with the input vectors while computing the coverage.

Our method of extracting the ECFM, grouping transitions and calculating the two metrics described above utilizes ideas that have been extensively used in formal design verification over the last few years. It uses Ordered Binary Decision Diagrams (OBDDs) [2] and has been described in detail in [10]. An outline of our system is shown in Figure 4.

We use SCFV, the tool that we have developed, for the extraction of the ECFM from a circuit described at any level of abstraction. Next, we feed SCFV with the ECFM description, a test sequence and information about the legal instructions applicable to the original circuit. SCFV computes the coverage metrics described earlier, and reports instances of unexercised behavior in terms of states not visited and transitions not covered in the control part of the design by the provided test suite. This is valuable information for the designer since it can be used to enhance the test set so that the whole control state space is covered. SCFV can also automatically provide a test sequence targeting transitions not covered by the original test

D. Moundanos, J. A. Abraham and Y. V. Hoskote,
"A Unified Framework for Design Validation and Manufacturing Test,"
Proceedings, IEEE International Test Conference, Washington, D.C., October 1996.

sequence.

Test sequence generation is performed on the ECFM. This sequence may not directly correspond to a sequence in the original machine because of **data conflicts**. This is the case because some of the original registers have now become PIs and can thus assume any value at any time. It is then possible for two consecutive vectors in the sequence to have inconsistent values for the same data register. In this case we can use conventional ATPG techniques to supplement the initial sequence with a subsequence that would justify the state encoded in the second of those vectors. This justification process is facilitated by ensuring that as many PI's as possible have don't care values. The technique of expanding the test sequence generated on the ECFM so that it becomes applicable to the original circuit will be described in more detail in the next section.

An uncovered transition is reported as the control state from which it originates and the assignments to the inputs in the ECFM necessary for that transition to be taken. Some of the PIs in the extended set of primary inputs of the ECFM are registers in the original machine. Test generation involves exercising the transition by first generating a sequence of vectors that will bring the machine to the state of interest (initialization) and then by appending one more vector corresponding to this transition (excitation).

There are two points that need to be made about this approach. First, one can substitute CRIS [14] with any other tool that could give an initial test sequence. A functional test sequence manually generated by designers could also be used. We employed CRIS because of lack of a functional test sequence for the microprocessor on which we applied our methodology. Second, there might be some data registers that are dropped completely from the ECFM if they do not appear in any dependency set. We can take the destination index register (DIR) as an example. If it is considered as part of the datapath, it is deleted from the state space. Since we are not interested in the register file, no output or next state function depends on the DIR and this register does not appear in the ECFM at all. But if the designers want to make sure that all registers are accessed, i.e., the DIR takes all valid values during simulation, then they can go back and retain the DIR as part of the control space and re-extract the ECFM.

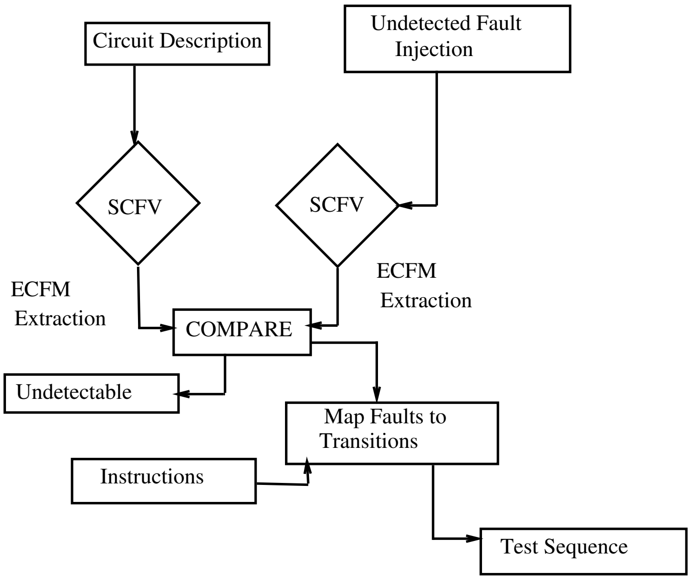

Figure 5: Method Outline

4.2 Using ECFM in structure–based test generation

We employ the extracted control flow machine model of a circuit described at the behavioral level to assist or supplement the structure-based test generation process for that circuit. We can provide the lower level tools with information about redundant faults in the control part of the design, faults that can then be removed from the faultlist when doing test generation at the gate level. Additionally, for faults that are hard to detect by ATPG we can very efficiently generate a test sequence that would detect them. An outline of our algorithm is shown in Figure 5.

4.2.1 Description of our Method

Our algorithm consists of the following steps:

- Given the gate level description of the circuit generate the complete fault list.
- Use a conventional low-level test generation tool like CRIS or HITEC [13] to detect as many faults as possible in a reasonable amount of time (in the order of a few hours).
- Extract the ECFM of the fault-free circuit.
- For each one of the aborted faults
  1. Inject the fault in the circuit.
  2. Extract the ECFM of the faulty circuit.

D. Moundanos, J. A. Abraham and Y. V. Hoskote,
"A Unified Framework for Design Validation and Manufacturing Test,"
Proceedings, IEEE International Test Conference, Washington, D.C., October 1996.

- If the fault belongs in the control part, perform an equivalence check between the ECFMs of the faulty and fault-free circuits.
  1. If the two ECFMs are equivalent then the fault is untestable.
  2. If they are not then generate a test sequence to detect that fault.

In this algorithm we concentrate on the faults that belong to the control part of the circuit. Basically out of all the aborted faults we choose those that cause the BDDs for the next state functions and/or the outputs of the faulty and fault-free ECFMs to be different. We choose to use a conventional ATPG tool at the beginning to detect the easy faults. Conceptually one could use our method for all the faults in the faultlist that belong to the control part of the machine. However, depending on the size of the ECFM for the given machine, this might be computationally expensive. So initially we use an ATPG tool and we impose a reasonable timeout limit on it. Additionally, we make the assumption that the circuits on which we apply our methodology have a reset state and ignore the faults affecting the resetting function. Since we have a reset state, an untestable fault, identified by the procedure above, is a redundant fault. There can be two types of redundant faults in the ECFM. First, we can have faults that do not affect the reachable part of the ECFM of the circuit. Secondly, there may be a fault that changes the reachable part of the control flow machine but does not affect the output behavior. This kind of fault will also be classified as redundant by the above method. The equivalence check that we employ in the above process tests the equivalence of the reset state (and all the states reachable from reset) in the faulty and fault-free circuit in terms of their I/O behavior. If no discrepancy in the I/O behavior is found the fault is redundant, otherwise a test can be generated.

At this point we need to consider what it means for the original circuit if a fault is characterized as redundant in the ECFM of that circuit. We have the following theorem:

Theorem: If a fault in the control part of a circuit is classified as redundant in the ECFM of that circuit, then that fault is also redundant in the original machine.

Proof: A fault is redundant in the original circuit and in the ECFM if there exists no input (distinguishing) sequence that would cause the values at the Primary Outputs to differ. This means that there are no values which, when loaded in the data registers, would cause the effects of the fault to be propagated to the POs. In the ECFM the data registers have been freed, some of them becoming PIs. On the other hand in the original circuit the data registers can only take a subset of the values that they can take in the ECFM. So if a fault is redundant in the ECFM it will also be redundant in the original circuit.

4.2.2 Test generation

In this section we describe how implicit state enumeration techniques can be used to generate differentiating sequences between two finite state machines [15, 6]. We first need some terminology.

A Mealy machine is defined by the 5-tuple $(I, O, s, S, \Delta, \Lambda)$ where $I = \{0,1\}^m$ is the input space, $O = \{0,1\}^l$ is the output space, s is the initial state, $S = \{0,1\}^n$ is the state space, $\Delta : I \times S \to S$ is the next-state functional vector, $\Delta = [\delta_1 ... \delta_n]$, and $\Lambda : I \times S \to O$ is the output functional vector, $\Lambda = [\lambda_1 ... \lambda_l]$. Let $\vec{q}$ represent the vector of present-state variables, $\vec{Q}$ the vector of next-state variables, $\vec{x}$ the input vector and $\vec{y}$ the output vector. Thus, we have $Q_i = \delta_i(\vec{x}, \vec{q}),\ i \in \{1..n\}$ and $y_j = \lambda_j(\vec{x}, \vec{q}),\ j \in \{1..l\}$.

The following concepts and operators are in wide use in the area of formal verification using implicit state enumeration procedures [3, 7]:

Definition: A **transition relation** is a relation T on the inputs I and the set of states S, such that, $T(\vec{x}, s_i, s_j) = 1$ if for input $\vec{x}$ in current state s_i, the next state is s_j.

Given an FSM with next state functional vector $\Delta(\vec{x}, \vec{q})$, the transition relation of the machine can be written as:

$$T(\vec{x}, \vec{q}, \vec{Q}) = \prod_{i=1}^{n} Q_i \odot \delta_i(\vec{x}, \vec{q}) \qquad (3)$$

where $\odot$ signifies the XNOR operation.

A set of states in an FSM is represented by its characteristic function, which is a Boolean predicate on the state variables.

Definition: The **characteristic function** of a set of states $s \subseteq S$ in an FSM is a Boolean function C on the state variables $\vec{q}$ such that $C(\vec{q})$ is true whenever the state represented by that valuation of the state variables belongs to s.

Manipulation of sets of states and traversing the state space of a state machine implicitly is accomplished through three operators, the Image, PreImage operators.

Definition: Given the characteristic function $C(\vec{q})$ of a set of states s in an FSM with transition relation

D. Moundanos, J. A. Abraham and Y. V. Hoskote,
"A Unified Framework for Design Validation and Manufacturing Test,"
Proceedings, IEEE International Test Conference, Washington, D.C., October 1996.

1. $new_0 = s_0 = Reached =$ initial state(s);
2. $i = 0$;
3. $\tau = \bigwedge_k \lambda_{p_k}$;
4. while $(new_i \neq 0)$
5.   if $(new_i \wedge \bar{\tau} \neq 0)$
6. $(inp_n,\ s_n) =$ PickOneCube$(new_i \wedge \bar{\tau})$;
7. $j = n - 1$;
8.   while $(j \geq 0)$
9. $pred = \mathbf{PreImage}(\Delta_p,\ s_{j+1})$;
10. $pred = pred \wedge new_j$;
11. $(inp_j,\ s_j) =$ PickOneCube$(pred)$;
12. $j = j - 1$;
13.   endwhile;
14.   return;
15.   endif;
16. $To = \mathbf{Image}(\Delta_p,\ new_i)$;
17. $i = i + 1$;
18. $new_i = To - Reached$;
19. $Reached = Reached \cup new_i$;
20. endwhile

Figure 6: Procedure to generate distinguishing sequence

T,

$$img(T, C) = \{v \mid \exists x[u \in C \wedge T(x, u, v)]\}$$
$$pre(T, C) = \{u \mid \exists x[v \in C \wedge T(x, u, v)]\}$$

The image of a set of states C is the set of states which can be **reached from** a state in C by one transition and the pre-image is the set of states **from which** a transition **can** go to a state in C.

The procedure implementing the algorithm for generating differentiating sequences is shown in Figure 6. Initially the product machine of the faulty and fault-free machines is formed. The characteristic function of the set of transitions with matching outputs is generated (step 3). Starting from a given initial state of the product machine (s_0), the algorithm performs a forward traversal, saving the set of newly visited states (new_i) at each step, until a discrepancy in the outputs for the component machines is encountered (step 5). A sequence of input vectors (inp_j) to this product machine state is then generated (steps 6-14) using the saved state sets (new_i) to do a backward traversal from the state showing incorrect outputs (s_n) back to the initial state (s_0). The PickOneCube routine just picks one cube from the ON-set of the function given as the argument.

4.2.3 Test Sequence Expansion

As was mentioned earlier, 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. Secondly, all data conflicts in the generated sequence must be resolved by justifying the values of those PI's which were data registers in the original circuit. We propose the use of ATPG techniques for performing this justifications. While in the process of implementing our own code, we used HITEC to demonstrate the validity of our approach. HITEC can produce a justification sequence for a given state assuming a don't care initial state.

There are two cases that need to be handled. We may have to generate a justification sequence for a given state (set of values for PI's that were data registers in the original circuit) to a don't care state. In this case the sequence generated by HITEC is directly applicable. In the other case we have a pair of partially or completely specified states and need to generate a sequence that will bring the original machine from the first to the second state. 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 HITEC cannot produce a justification sequence then the corresponding vectors are dropped from our test. The way we force HITEC to generate the justification sequence that we need for a given state, is 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.

Dropping vectors was not frequently necessary for the circuit to which this technique was applied. However the applicability of the method needs to be evaluated in larger examples. 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.

5 Experimental Results

We applied our methodology to the Viper microprocessor [8]. The Viper is a 32-bit microprocessor

D. Moundanos, J. A. Abraham and Y. V. Hoskote,
"A Unified Framework for Design Validation and Manufacturing Test,"
Proceedings, IEEE International Test Conference, Washington, D.C., October 1996.

designed at RSRE, England for safety-critical applications. It has four general purpose registers, two ALU registers along with a memory address register and an instruction register. The number of PIs is 33. The netlist has a total of 251 flip-flops and approximately 6000 gates. The total number of faults injected when doing structure-based test generation is 10591. 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 ocurs.

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. We identified 5 flip-flops as encoding the flow of control. These include the control, status and overflow flip-flops. So the ECFM has 5 flip-flops and 75 inputs. It has 17 reachable states. For lack of a functional test suite for the Viper we evaluated the functional coverage of a test sequence of 5962 vectors generated by the CRIS test generator. This sequence was generated in approximately 6 hours and detects 9687 stuck-at faults out of the 10591 faults in the complete fault list, giving a stuck-at fault coverage of 91.46%. We evaluated the functional coverage on the ECFM for these vectors. SCM was 100% and TCM was 91.13%. Evaluating the functional coverage required 10.75 minutes and extracting the ECFM required 0.51 seconds on a sparc10. Our system also identified instances of unexercised behavior in terms of uncovered transitions. The designer can then write tests to exercise these transitions or we can automatically generate a test sequence that would exercise all the uncovered transitions (modulo transition equivalence) in the ECFM. This sequence can be mapped back to the original machine in a manner similar to the one described in the previous section.

The next step is to apply the procedure described in Figure 5 to perform test generation for the remaining 904 faults not detected by the original sequence generated by CRIS. To evaluate the performance of our approach we used HITEC to perform test generation on this reduced faultlist. HITEC, after running for 3 days, identified 86 faults as redundant, was able to generate tests for 536 faults and aborted for the remaining 282 faults. We applied our approach to each fault in the reduced faultlist. That is, for each fault we inject that fault, extract the ECFM for the faulty circuit, perform an equivalence check against the fault-free ECFM and, if they are not equivalent, generate a test sequence. This process takes on the average 8.6 seconds for each fault (0.5 seconds for the ECFM extraction and 8.1 seconds for the equivalence check which includes the time for the sequence generation if the two ECFMs are not equivalent). Additionally we have an overhead associated with the generation of justification sequences using HITEC. Currently we set the backtrack limit to 100000 and the whole process takes 2-3 seconds. Furthermore the average justification sequence for the Viper is in the order of 10 vectors (i.e., 2 instructions).

Our method was able to identify 83 of the 86 faults as redundant. The remaining three faults do not belong to the control part of the circuit and so our approach is not conclusive for those. Furthermore our approach was able to detect 393 faults. Out of these 393 faults 176 had been detected by HITEC and 193 had been aborted. The test sequence generated consisted of 3739 vectors. When expanded the sequence consisted of 5440 tests. This sequence when fault simulated on the original machine gave a stuck-at coverage of 81.08%, detecting a lot more than the targeted faults. The overall coverage achieved when this sequence was appended to the original one was 94.98% while the efficiency was 95.77%. It should be made clear that our approach targets the faults that directly affect the control part of the machine which are identified by the procedure described above. In this way we have been able to very quickly generate tests for several hard to detect faults.

6 Conclusions

In this paper we have proposed a unified framework for multi-level design validation and test generation. We have demonstrated how abstraction techniques can be employed at the behavioral level of description to assist design verification by providing quantitative measures for the quality of the functional tests used during simulation. Additionally we have proposed a method of generating functional vectors that would exercise transitions not exercised by the original functional test sequence. The method is completely automatic and efficient, using powerful implicit enumeration techniques.

In addition we have proposed a technique for redundancy identification and test generation at the state transition level, again by using an abstraction that captures the control flow of the original circuit. Although only a small set of stuck-at faults is being targeted, the vector sequence generated is able to detect many faults, while the test generation time is in the

D. Moundanos, J. A. Abraham and Y. V. Hoskote,
"A Unified Framework for Design Validation and Manufacturing Test,"
Proceedings, IEEE International Test Conference, Washington, D.C., October 1996.

order of a few seconds for each fault. Conventional low-level test generation tools like HITEC abort for faults that were classified as redundant, or for which our method generated a test sequence in a matter of seconds. We are in the process of evaluating these techniques on bigger circuits.

Our approach demonstrates how high-level abstractions can help the design validation process at several levels of the design hierarchy. Moreover, abstractions make it possible to handle circuits that cannot be handled even when using implicit enumeration techniques. However, even our current approach based on a flattened circuit representation will fail for large real life circuits. Our future research involves developing techniques for the hierarchical extraction of the ECFM at the behavioral level making use of information about the symmetry of the system. This will allow us to deal with larger circuits.

References

[1] M. Abadir and H. Reghbati. Functional specification and testing of logic circuits. *Comp. and Math. with Appls.*, 11, No. 12:1143–1153, 1985.

[2] R. Bryant. Graph–based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35, No. 8:677–691, 1986.

[3] J. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential circuit verification using symbolic model checking. *Proc. 27th DAC*, pages 46–51, 1990.

[4] K. Cheng and J. Jou. A functional fault model for sequential machines. *IEEE Trans. on CAD*, 11, No. 9:1065–1073, 1992.

[5] K. Cheng and A. Krishnakumar. Automatic functional test generation using the extended finite state machine model. *Proc. 30th DAC*, pages 86–91, 1993.

[6] H. Cho, G. Hachtel, and F. Somenzi. Redundancy identification/removal and test generation for sequential circuits using implicit state enumeration. *IEEE Trans. on CAD*, 12, No. 7:935–945, 1993.

[7] O. Coudert, C. Berthet, and J. Madre. Verification of sequential machines using boolean functional vectors. *Formal VLSI Correctness Verification. Elsevier Sc.*, pages 179–196, 1990.

[8] W. Cullyer. *Implementing Safety–Critical Systems: The Viper Microprocessor*, pages 1–26. VLSI Specification, Verification and Synthesis. G. Birtwistle and P. Subrahmanyam, Eds. Kluwer Ac., 1988.

[9] R. Ho, C. Yang, M. Horowitz, and D. Dill. Architecture validation for processors. *Proc. 22nd International Symposium on Computer Architecture*, pages 404–413, 1995.

[10] Y. Hoskote, D. Moundanos, and J.A. Abraham. Automatic extraction of the control flow machine and application to evaluating coverage of verification vectors. *Proc. ICCD*, pages 532–537, 1995.

[11] S. Kang and S. Szygenda. Design validation: Comparing theoretical and empirical results of design error modeling. *IEEE Design and Test*, pages 18–26, 1994.

[12] John Moondanos and J. A. Abraham. Sequential redundancy identification using verification techniques. *Proc. ITC*, pages 197–205, 1992.

[13] T. Nierman and J. H. Patel. Hitec: A test generation package for sequential circuits. *Proc. EDAC*, pages 214–218, 1991.

[14] D. Saab, Y. Saab, and J. A. Abraham. Cris: A test cultivation program for sequential vlsi circuits. *Proc. ICCAD*, pages 216–219, 1992.

[15] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. *Proc. ICCAD*, pages 130–133, 1990.