TEST GENERATION FOR SEQUENTIAL FINITE STATE MACHINES

by

Hi-Keung Tony Ma, Srinivas Devadas,
A. Richard Newton, and
Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M87/36

29 May 1987

COVER PAGE

TEST GENERATION FOR SEQUENTIAL FINITE STATE MACHINES

by

Hi-Keung Tony Ma, Srinivas Devadas, A. Richard Newton,
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M87/36

29 May 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TEST GENERATION FOR SEQUENTIAL FINITE STATE MACHINES

by

Hi-Keung Tony Ma, Srinivas Devadas, A. Richard Newton,
and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M87/36

29 May 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Test Generation for Sequential Finite State Machines

Hi-Keung Tony Ma, Srinivas Devadas
A. Richard Newton and Alberto Sangiovanni-Vincentelli
Department of Electrical Engineering and Computer Sciences
Cory Hall
University of California, Berkeley, CA. 94720

Abstract

We present a novel approach to test pattern generation for synchronous sequential finite state machines. Our approach involves first extracting part of the State Transition Graph (STG) of the finite state machine, a Moore or a Mealy machine, using purely structural information, i.e. the gate-level description of the sequential circuit. The construction of the partial STG is based on an efficient state-enumeration algorithm that aims at finding paths from the reset state to different valid states (states reachable from the reset state) in the STG. For circuits with relatively few states, a partial STG including all the valid states is built. For circuits with a large number of states, only a subset of states is included in the partial STG. We show how test sequences for line stuck-at faults can be efficiently generated using the partial STG in conjunction with fault excitation-and-propagation and state justification algorithms based on the concept of state enumeration. We have successfully generated tests for finite state machines with a large number of states using reasonable amounts of CPU time and obtained close to maximum possible fault coverages.

Acknowledgements

This work is supported in part by the Semiconductor Research Corporation under grant 442427-52055, the Defense Advanced Research Projects Agency under contract N00039-86-R-0365 and by a grant from AT&T Bell Laboratories. Their support is gratefully acknowledged.

1. INTRODUCTION

Test generation for sequential circuits has long been recognized as a difficult task[Hen64, Bou71, Bre76, Mic83]. Unstructured random sequential designs are very difficult to test. One common approach to improve the testability of a sequential circuit is to add test points to the circuitry so that tests can be applied more readily and fault effects can be observed better. But this method is not systematic and relies on designer ingenuity. Asynchronous sequential designs containing races that may operate improperly due to hazards further complicate the test generation process. The identification of all feedback lines to construct a model (iterative array) for test generation for asynchronous designs is not a simple task. Tests generated using the model may be incorrect and require validation through simulations.

A popular approach to solving the problem of test generation for sequential circuits is to make all the memory elements controllable and observable, i.e. complete scan design[Eic77, Agr84]. scan design approaches have been successfully used to reduce the complexity of the problem of test generation for sequential circuits by transforming it into a combinational one which is considerably less difficult. The design rules of Scan designs also constrain the sequential circuits to synchronous ones so that the normal operation of the sequential circuit is free of races and hazards. However, there are situations where the cost in terms of area and performance of complete scan design is unaffordable. In addition, even though the general sequential testing problem is very difficult, there may be cases where test generation can be effective. Simply making all the memory elements scannable in a sequential circuit without even first investigating how difficult is the problem of generating tests for it could unduly incur unnecessary area cost.

The difficulty in generating a test usually lies with: 1) setting the states of the memory elements into a certain combination so that the fault under test is excited; 2) propagating the fault effect to the primary outputs. An input sequence is usually required in both cases (if such a sequence exists). In general, the longer the length of the shortest input sequence needed to perform steps 1) and 2), the more difficult it is to find an input sequence to test the circuit. Both

approaches mentioned above attempt to shorten the length of the input sequence. In the scan design approach, the length of the input sequence is reduced to one when all memory elements are made scannable.

Several approaches[Bre71, Sch75, Mar78, Mal85, Nit85, Sht85] have been taken in the past to solve the problem of test generation for sequential circuits. They are either extensions to the classical D-Algorithm or based on random techniques[Sch75, Nit85]. When the number of states of the circuit is large and the tests demand long input sequences, they can be quite ineffective for test generation. This is because no *a priori* knowledge of the length of the test sequence is available. In the extended D-Algorithm methods, a large amount of effort may be wasted in trying to find short sequence tests for faults that require long ones. Random testing techniques are based on continuous simulations and grading of test vectors according to simulation results. They can be very time consuming for difficult faults that have only a few long test sequences. Our approach to test pattern generation for sequential finite state machines represents a significant departure from these methods and is largely based on the concept of state enumeration. The problem of generating tests for faults that require a lengthy input sequence is handled efficiently by the intelligent use of information contained in a partial State Transition Graph (STG) and the integration of a few new algorithms based on the concept of state enumeration.

We assume the sequential circuit under test is synchronous and free of races and hazards under simple design rules. We also assume there is a reset state for the synchronous sequential machine and memory elements such as D flip-flops are identified and represented as logical primitives to facilitate loop cutting in transforming the synchronous sequential circuit into an iterative array. We first extract a part of the State Transition Graph (STG) of the finite state machine using purely structural information, i.e. the gate-level description of a sequential circuit. The construction of the partial STG is based on an efficient state-enumeration algorithm that finds paths from the reset state to different valid states (states reachable from the reset state) in the STG. For circuits with relatively few states, a partial STG including all valid states is built. For circuits with a

large number of states, only a subset of valid states is included in the partial STG. The partial STG is then used in conjunction with efficient enumeration-based fault excitation-and-propagation and state justification algorithms for generating tests for line stuck-at faults. We have successfully generated tests for finite state machines with a large number of states using reasonable amounts of CPU time and obtained close to maximum possible fault coverages.

The following section outlines the test generation process. Extraction of the fully or partially connected state transition graph from the logic level finite state machine is described in Section 3. The enumeration-based fault excitation-and-propagation and state justification algorithms are described in Section 4 and 5 respectively. Section 6 describes the detection of a special class of redundant faults. Results for a number of finite state machines are presented in Section 7.

2. THE TEST GENERATION PROCESS

Assuming the complete State Transition Graph (STG) of a sequential circuit is available, test generation for a fault under test can be done by first finding an input sequence $T1$ and an initial state $S0$ that excite and propagate the effect of the fault to the primary outputs within (4^n-2^n) time frames, where n is the number of latches in the sequential circuit (see Theorem 1). We assume a reset state for the machine from which all test sequences begin. Then every path from the reset state to any state $S1$ that covers $S0$, a potential setup sequence, in the complete STG is fault simulated. If a path $T0$ (setup sequence) to a state $S1$ that covers $S0$ can be found under fault conditions, a test sequence $T2$ is generated by concatenating the path $T0$ with $T1$. Even though a setup sequence $T0$ may not be found, the fault may still be detected by one of the potential setup sequences through fault simulation. If this is the case, that particular potential setup sequence itself can serve as a test sequence $T2$. If no test sequence can be found, a new input sequence $T1$ and a new initial state $S0$ which is disjoint from all previously generated ones is searched and the procedure is repeated.

The algorithm is complete, i.e. if a fault is testable, a test will be found given sufficient time. The main drawbacks of this method are: (1) the memory storage for the complete STG may be

unreasonably large and the generation of the complete STG may demand astronomical CPU time; (2) fault simulation of all potential setup sequences is extremely time consuming. A remedy to (1) is to generate the potential setup sequences on-the-fly using a backward justification algorithm that searches for paths from the reset state to the *S*0's under fault-free conditions. No information of the STG is required/used.

A test generation algorithm following the ideas presented above is as follows.

Algorithm Structure 1

(1) Find an (new) input sequence *T*1 and an (new) initial state *S*0 that will excite and propagate the effect of the fault under test to the primary outputs within (4^n-2^n) time frames using the state-enumeration-based test generation algorithm (described in Section 4). If no solution exists, exit without a test.

(2) Find a (new) path *T*0 (potential setup sequence) from the reset state to the initial state *S*0 using a backward justification algorithm. If no solution exists, go to (1).

(3) Fault simulate the potential setup sequence *T*0. If it detects the fault, generate the test sequence *T*2 from *T*0 and go to (5). Else if it is a valid setup sequence, go to (4). Else if *T*0 neither detects the fault nor is a setup sequence go to (2).

(4) Concatenate the input sequence *T*0 that represents the path from the reset state to the initial state *S*0 with *T*1 to form *T*2 which is the test sequence for the fault under test.

(5) Exit with a test sequence.

Even though this algorithm is potentially effective, backward justification in general is difficult when the setup sequence is long. In addition, some states may need to be justified more than once. Therefore, an important enhancement is to generate *a partial STG containing as many valid states* (and paths from the reset states to them) as possible provided that the partial STG extraction process (through forward enumeration as described in Section 3) is carried out efficiently. Note that the partial STG may contain all the valid states in the complete STG but con-

tains much fewer edges. States and edges may be *added* to the partial STG via backward justification during test generation.

The second drawback mentioned above, i.e. that fault simulation of all potential setup sequences is very time consuming, does not actually pose a problem. From the observations in our experiments, if $T0$ is an invalid setup sequence, it is very likely to be a test sequence. Therefore, there is rarely the need for fault simulation of more than one potential setup sequence for a fault. Finally, an efficient test generation algorithm combining the advantages of forward enumeration and backward justification by using the partial STG is as follows.

Algorithm Structure 2

(1) Find an (new) input sequence $T1$ and an (new) initial state $S0$ that will excite and propagate the effect of the fault under test to the primary outputs within a prescribed number of time frames using the state-enumeration-based test generation algorithm (described in Section 4). If no solution exists, exit without a test.

(2) Search for a path (potential setup sequence) $T0$ from the reset state to $S0$ in the partial STG. If it is found, go to (5).

(3) If the partial STG includes all valid states, go to (1).

(4) Find a path $T0$ from the reset state to the initial state $S0$ using the state justification algorithm (described in Section 5). If no solution exists, go to (1).

(5) Fault simulate the potential setup sequence $T0$. If it detects the fault, generate the test sequence $T2$ from $T0$ and go to (7). Else if it is a valid setup sequence, continue. Else go to (1).

(6) Concatenate the input sequence $T0$ that represents the path from the reset state to the initial state $S0$ with $T1$ to form $T2$ which is the test sequence for the fault under test.

(7) Exit with a test sequence.

The initial state *S0* can be a cube containing don't care bits or a min-term with every state bit specified. In the case of a cube, a path from the reset state to a minterm covered by *S0* can serve the purpose of a setup sequence.

3. STATE TRANSITION GRAPH EXTRACTION

The inputs to the logic level extraction program is the combinational logic block CLB of the finite state machine and information about latch inputs and outputs, i.e. present and next state lines. The output is a partial State Transition Graph (STG) of the finite state machine. A node in the STG represents a distinct state and an edge between two nodes represents an input combination (cube) that drives the finite state machine from one specific state to another.

The STG extraction first sequentially cube-enumerates all fanout edges from the given reset state. Whenever a new edge is found, it is added to the current STG if the next state it fans into does not exist in the STG. Each next state is then picked as a new starting state. The procedure is repeated until no more distinct valid state can be found. All the edges in the complete STG will be implicitly, but exhaustively enumerated. The partial STG constructed is a tree, i.e. there is only a single path from the reset state to any other state. This is to restrict the storage space for the partial STG so that synchronous sequential machines with very large number of states can be handled.

The algorithm used to enumerate the fanout edges from a state is an extension to the implicit enumeration algorithm of PODEM[Goe81]. Initially, values of all primary inputs and next states of the logic level finite state machine are set to unknown. The logic level circuit is simulated with the present state lines fixed at their specified values. An unknown next state line is then picked and a path is backtraced from it to an unknown primary input with the objective to set the value of the chosen next state line to a known one. A 1 or 0 is assigned to that primary input. The circuit is then simulated again. The setting of primary inputs and simulation of the circuit is continued until all next state lines are set to known values - a fanout edge is enumerated. Whenever an edge is found, we backtrack to where a primary input is first set to a known value and assigned it an opposite value. We then repeat the simulation and primary input setting. When no more backtracking

can be done, all the edges from a state are implicitly, but exhaustively enumerated.

The extraction process can proceed in either a depth-first or a breadth-first fashion. In the breadth-first fashion, the path from the reset state to any state in the partial STG is the shortest one. The test sequences generated are shorter but the total number of test sequences is greater than using a depth-first algorithm. There are hard limits, $L1$ and $L2$, for the total number of states to be included in the final STG and the number of states at each level from the given initial state. $L1$ is used to restrict the memory usage and $L2$ restricts the maximum length of the test sequence. The pseudo-code below illustrates the partial STG extraction process in a depth-first fashion. Extract() is initially called with the reset state of the sequential circuit and the level equal to 0.

```
Extract(State, level)
{
  PresentState = State;
  PrimaryInput = unknown;
  simulate the circuit;

  while (not all edges have been enumerated) {

    if ((TotalNumStates ≥ L1) ||
      (NumStates[level] ≥ L2)) break;

    if (not all NextState lines are set) {
      find_new_pi_assignment();
      simulate with current set of pi assignments;
    }
    else {
      if (NextState is not in the partial STG) {
        add NextState to partial STG;
        TotalNumStates = TotalNumStates + 1;
        NumStates[level] = NumStates + 1;

        Extract(NextState, level + 1);
      }
      else {
        backtrack to the last set primary input
        and assign an alternative value to it;

        simulate with current set of pi assignments;

      }
    }
```

```
  }
}
```

An alternative to the backtracing/backtracking approach to STG enumeration described above is forward simulation on the input space given a starting present state. The forward simulation process begins with all the input lines set to unknown values. Inputs are set randomly to 0 or 1 in a pre-specified order till all the next state lines are all set to known values. Backtracking on primary input values is done after setting all next state lines. However, this approach is less efficient than the approach described earlier because a primary input value may be unnecessarily set in order to set the next state lines. This can lead to a great amount of redundant simulations. On the contrary, in the backtracing/bracktracking approach, the backtracing process makes sure that the next primary input to be set and the simulation following the value-setting always contribute to the setting of the next state lines.

4. THE FAULT EXCITATION-AND-PROPAGATION ALGORITHM

The Fault Excitation-and-Propagation algorithm (FEP) is based on the decision tree concept of the test pattern generation algorithm PODEM. FEP uses the conventional iterative array model for generating an input sequence $T1$ and an initial state $S0$ to excite and propagate the effect of the fault under test to the primary outputs within a prescribed number of time frames. The iterative array is considered wholly as a combinational circuit with primary inputs of different time frames time-indexed and the present state lines of the first time frame treated as pseudo inputs. The initial state $S0$ is specified by the pseudo inputs values. FEP first tries to propagate the fault effect to the primary outputs of the first time frame. If it fails, it will use the primary outputs of the second time frame for fault propagation and so on until the prescribed number of time frames is reached.

The number of time frames required by FEP to ensure a complete test generation algorithm can be proved to be a maximum of (4^n-2^n) where n is the number of latches in the sequential circuit. Forward search techniques based on the D-Algorithm starting from the reset state require 4^n

time frames to ensure completeness [Bre76]. In our complete algorithm, FEP requires (4^n–2^n) time frames and fault-free backward justification requires 2^n time frames.

Theorem 1: If $T1$ exists for any $S0$, its length is bounded by (4^n–2^n) in a n-latch sequential circuit.

Proof: Each state vector $y(i)$ in $T1$ will have one of five values, namely (0, 1, x, D, $\overline{D}$). The values D and $\overline{D}$ correspond to composite values for faulty and fault-free circuits - if a value of a line is 0 (1) in a fault-free circuit and 1 (0) in the faulty circuit the line is deemed to have a value, D ($\overline{D}$). If a test exists, the x's can be replaced by 0's and 1's and hence only four values need to be considered. Thus, 4^n unique states exist. $S0$ can be any of 2^n possible values since FEP begins from a fault-free state. It is clear that in testing a circuit, it is never necessary to enter the same state twice - all the $y(i)$ can be restricted to be unique. If $T1$, starting from a particular fault-free state, $S1$, enters another fault-free state, $S2$, we have a shorter test sequence starting from $S2$. So the upper bound on the length of $T1$, if it exists, occurs when $T1$ enters all possible faulty states starting from a fault-free state and this bound is (4^n–2^n).

FEP uses *two decision trees*, one for the primary inputs of different time frames and the other for the initial state $S0$, as opposed to only one in PODEM. The two decision trees are built in a similar way through the backtracing and backtracking processes as used in PODEM. The present state lines of the first time frame are treated similarly as the primary inputs during the fault excitation-and-propagation process. Values of the present state lines and primary inputs of different time frames are continuously set one at a time through the backtracing process and the iterative array is simulated whenever a primary input or a pseudo iput is set to a known value. The value-setting-and-simulation process continues until the effect of the fault under test is excited and propagated to the primary outputs of at least one of the time frames or when the backtracking limit is reached. Backtracking takes place whenever it has established that under the current set of primary input and pseudo input assignments, the effect of the fault under test cannot be excited and/or observed at the primary outputs of the specified time frame with further input assignments. Backtracking during the search for $T1$ and $S0$ is done on both decision trees.

FEP employs *the concept of disjoint state enumeration* to make sure that all the tests it generates for a specific fault will have disjoint initial states *S*0's; this is necessary because of the loop in the test generation process described in Section 2. Whenever the search for a new test is begun, the primary input decision tree (*D*1) for the previous test is scratched completely but the present state decision tree (*D*2) of the initial state *S*0 is retained. Immediately backtracking is done on *D*2. Then the value-setting-and-simulation process is carried out as described above. The reason that tests generated for a specific fault by FEP should all have disjoint *S*0's is related to how FEP is used in the test generation process as described in Section 2. For a specific fault, a new test is requested only if we cannot find the path from the reset state to the *S*0 in the previous test, neither in the extracted STG nor through the state justification algorithm described in Section 5. Therefore, all tests generated for a specific fault should have disjoint *S*0's.

A single decision tree could have been used instead of two separated ones as described above. And instead of completely resetting all primary input values to unknown, i.e. scratching the entire primary input decision tree, when a new search is started, one can simply backtrack on the single decision tree to where a pseudo input is first set to a known value and assigned it an opposite value. But due to the inherent characteristics of the enumeration approach of PODEM, it is more efficient to begin a search with as small a number of preset inputs as possible. Therefore the double decision tree method is used.

5. THE STATE JUSTIFICATION ALGORITHM

Given a goal state *S*0, the state justification algorithm (SJ) attempts to find a path (setup sequence) from the reset state to it. *S*0 can be a cube containing don't care state bits or a minterm with every state bit specified. In the case of a cube, SJ needs only to find a path to any minterm state that is covered by *S*0.

First, SJ sequentially enumerates all the fanin edges to *S*0. It then checks whether any state the edges fanout from covers the reset state. If such a state exists, a path is found. Otherwise, SJ picks each fanin state as a new goal state and carries out fanin edge enumeration again. The

procedure is repeated until a path is found or no path can be found. SJ actually proceeds in a depth-first fashion and there is a limit on the maximum length of the justification sequence.

The edge enumeration algorithm is an extension to the enumeration algorithm PLOVER in [Ma87]. The difference is that here we have multiple line (the next state lines) values to be justified simultaneously rather than a single output line as in PLOVER. The concept of state enumeration is also employed in SJ. There are two decision trees to be maintained as in Section 4, i.e. one ($D1$) for the primary inputs and the other ($D2$) for the present state lines. All the present state lines and primary inputs are set to unknown values initially. Through backtracing and backtracking processes, the primary inputs and present state lines are continuously set to some known values, 1 or 0, until all the next state lines are found to be set to their specified values through simulation. Whenever the search for a new fanin edge is begun, $D1$ is completely scratched but $D2$ is retained. Immediately backtracking is done on $D2$. Then the enumeration procedure is repeated again. All edges (with disjoint fanin states) fanning out of a state are enumerated when no more backtracking is possible. The pseudo code below illustrates the state justification algorithm

proceeding in depth-first fashion. Breadth-first search is an alternative.

```
Justify_State(State)
{
  PresentState (ps) = unknown;
  PrimaryInput (pi) = unknown;
  while (not all fanin states to State are enumerated) {
    while (not all the NextState lines are justified) {

      find_new_pi/ps_assignment();
      simulate circuit with current set of pi/ps assignments;

      if (there are conflicts on NextState line values) {

        backtrack to the last set pi in D1 or ps in D2
                  and assign an alternative value to it;
        simulate with current set of pi and ps assignments;
      }
    }
    if (a fanin state is found) {
      if (fanin state covers reset state) {
        a path is found;
        return;
      }
    }
    else Justify_State(fanin state);

    if (a path is not found) {

          /* scratch D1 */
      scratch all pi assignments;

      backtrack to the last set ps in D2 and
      assign an alternative value to it;
      simulate with current set of ps assignments;
    }
  }
}
```

6. DETECTION OF REDUNDANT FAULTS

The difficulty in test generation for sequential circuits does not just lie with finding tests for the difficult testable faults. The determination of redundant faults is equally formidable if not more difficult. Obtaining a low fault coverage does not necessary mean the test generator is inadequate if we can show that the fault coverage is close to the maximum achievable value. However, to determine whether faults, that no test has been generated for, are redundant or testable may demand astronomical CPU times. For the purpose of judging how close the fault coverage obtained by our test generator is to the maximum possible value, we find all the redundant faults based on Theorem 2 given below and treat other undetected faults as possibly testable faults. This gives a pessimistic estimate of the number of redundant faults in a given circuit.

Definition 1: An edge in the State Transition Graph is said to be *corrupted* by a stuck-at fault if the effect of the fault can be excited and propagated to the primary outputs and/or next state lines by the input vector corresponding to the edge with the present state lines values set to the fanin state of the edge.

Theorem 2: In order for a stuck-at fault to be detected, the fault should at least corrupt one fanout edge from a valid state that is reachable from the reset state in the state transition graph.

Proof: In order to detect a fault we need a test sequence starting from the reset state and ending with a corrupted edge in the STG. If a fault does not corrupt any fanout edge from a valid state in the STG, no test sequence can detect the fault since no corrupted edge can be reached from the reset state.

Determining this special class of redundant faults requires the extraction of a partial STG containing all valid states reachable from the reset states. The procedure to find these redundant faults is based on the FEP algorithm described in Section 4. A single time frame is used and all next state lines are treated as primary outputs. We generate all tests, for a potential redundant fault, with disjoint initial states. If none of the initial states exists in the partial STG, the fault

under test is redundant.

7. RESULTS

Results and time profiles for six finite state machines are given in Table 1 and 2 respectively. In the tables *m* and *s* stand for minutes and seconds respectively. For each example, the number of inputs (#inp), number of outputs (#out), number of gates (#gate), number of latches (#lat), number of equivalent faults (#eqv. faults), number of test sequences (#test seq.), total number of test vectors (#vect), maximum test sequence length (max. seq. len.), fault coverage, percentage of provably redundant faults (using Theorem 2), total fault coverage including detected and provably redundant faults (tfc), and CPU time on a VAX 11/8800 are indicated in Table 1. CPU times for extracting the partial state transition graph, test sequence generation, fault simulation, miscellaneous setup and for the entire test generation process are indicated in Table 2.

CKT	#inp	#out	#gate	#lat	#eqv. faults	#test seq.	#vec	max. seq. len.	fault cov. (%)	red.* fault (%)	tfc§ (%)	CPU† time
cse	7	7	192	4	680	96	472	8	99.71	0.29	100.0	53.2s
sse	7	7	130	6	486	46	284	10	84.57	15.23	99.8	69.9s
planet	7	19	606	6	2028	80	1191	26	97.39	2.56	99.95	12.6m
sand	9	6	555	6	1932	165	1077	24	94.36	5.18	99.54	22.4m
scf	27	54	959	8	3338	136	2238	21	94.37	3.86	98.23	83.0m
sbc	40	56	1011	28	3008	168	1063	24	95.68	2.66	98.34	62.1m

* percentage of provably redundant faults
§ total fault coverage including detected and provably redundant faults
† All times are obtained on a VAX 11/8800

Table 1: Results for 6 example circuits

CKT	STG Extraction	Test Generation	Fault Simulation	Miscell.	Total
cse	0.9s	8.3s	43.8s	0.2s	53.2s
sse	0.4s	52.2s	17.1s	0.2s	69.9s
planet	3.2s	1.2m	11.4m	0.7s	12.6m
sand	4.6s	10.7m	11.6m	0.6s	22.4m
scf	13.9s	11.5m	71.2m	1.2s	83.0m
sbc	12.4m	28.3m	21.4m	1.3s	62.1m

Table 2: Time profiles for example circuits

As can be seen our test generation technique obtains close to the maximum possible fault coverage in all the examples. The extraction of the STG consumes a relatively small amount of CPU time with respect to the total TPG time in all cases. Fault simulation constitutes a large percentage of total TPG time in most cases except in Example 2, as can be seen in Table 2. Our fault simulator uses the parallel-fault event-driven technique and a more sophisticated one using concurrent techniques will significantly speed up the test generation process. The reason that test generation time is the dominant constituent in the total CPU time in Example 2 is because a great amount of time is consumed in trying to find tests for the large number of reduntant faults.

The first five examples are finite state machines obtained from various industrial sources. The largest example SBC is the snooping bus controller[Hil86] in the SPUR chip set. It was syn-

thesized using the multiple level logic optimization system MIS[Bra86].

8. CONCLUSION

A novel approach to test generation for synchronous sequential finite state machine has been presented in this paper. The efficacy of our method stems from the integration of several new algorithms that are based on the concept of state enumeration. Our approach involves extracting a partial state transition graph and using it in conjunction with fault excitation-and-propagation and state justification algorithms in generating tests. The problem of generating tests for faults that require a lengthy input sequence is shown to be handled efficiently by our method through the intelligent use of the path information contained in the STG and the coordinated interaction of the various algorithms. We have successfully generated tests for finite state machines with a large number of states using reasonable amounts of CPU time and obtained close to maximum possible fault coverages. A new method of detecting a special class of redundant faults in determining how close the fault coverage obtained to the maximum possible value is also described.

9. REFERENCES

[Agr84] V. D. Agrawal, S. K. Jain and D. M. Singer, "Automation in Design For Testability", Proc. of Custom Integrated Circuits Conference, Rochester, NY, May 21-23, 1984.

[Bou71] W. G. Bouricius, E. P. Hsieh, G. R. Putzolu, J. P. Roth, P. R. Schneider and C. J. Tan, "Algorithms for Detection of Faults in Logic Circuits", IEEE Transcation on Computers, Vol. C-20, No. 11, Nov. 1971, pp. 1258-1264.

[Bra86] R. K. Brayton, E. Detjens, S. Krishna, T. Ma, P. McGeer, L. Pei, N. Phillips, R. Rudell, R. Segal, A. Wang, R. Yung and A. Sangiovanni-Vincentelli, "Multiple-Level Logic Optimizatin System", Proc. of IEEE Int. Conf. on CAD (ICCAD), Santa Clara, CA, Nov. 1986, pp. 356-359.

[Bre71] M. A. Breuer, "A Random and an Algorithmic Technique for Fault Detection Test Generation for Sequential Circuits", IEEE Transcation on Computers, Vol. C-20, No. 11, Nov. 1971, pp. 1366-1370.

[Bre76] M. A. Breuer and A. D. Friedman, "Diagnosis & Reliable Design Of Digital Systems", Computer Science Press, 1976.

[Eic77] E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability", Proc. of 14th Design Automation Conference, June 1977, pp. 462-468.

[Goe81] P. Goel, "An Implicit Enumeration Algorithm To Generate Tests for Combinational Logic Circuits", IEEE Transactions on Computers, Vol. C-30, Mar. 1981.

[Hen64] F. C. Hennie, "Fault detecting experiments for sequential circuits", Proc. of 5th Annual Symp. on Switching Circuit Theory and Logical Design, Princeton, New Jersey, Nov. 1964, pp. 95-110.

[Hil86] M. Hill et al., "Design Decisions in SPUR", IEEE COMPUTER, Vol. 19, No. 11, Nov. 1986, pp. 8-22.

[Ma87] H. K. T. Ma, R. S. Wei, S. Devadas and A. Sangiovanni-Vincentelli, "Logic Verification Algorithms and their Parallel Implementation", Proc. of 24th Design Automation Conference, Miami Beach, FLA, June 1987.

[Mal85] S. Mallela and S. Wu, "A Sequential Circuit Test Generation System", Proc. of 1985 International Test Conference, Philadelphia, PA, Oct. 1983, pp. 57-61.

[Mar78] R. A. Marlett, "EBT: A Comprehensive Test Generation Technique For Highly Sequential Circuits", Proc. of 15th Design Automation Conference, Las Vegas, NV, June 1978, pp. 332-339.

[Mic83] A. Miczo, "The Sequential ATPG: A Theoretical Limit", Proc. of 1983 International Test Conference, Philadelphia, PA, Oct. 1983, pp. 143-147.

[Nit85] S. Nitta, M. Kawamura and K. Hirabayashi, "Test Generation by Activation and Defect-drive (TEGAD)", INTEGRATION, the VLSI journal 3 (1985) 3-12.

[Sch75] H. D. Schnurmann, E. Lindbloom and R. G. Carpenter, "The Weighted Random Test-Pattern Generator", IEEE Transcation on Computers, Vol. C-24, No. 7, July 1975, pp. 695-700.

[Sht85] S. Shteingart, A. W. Nagle and J. Grason, "RTG: Automatic Register Level Test Generator", Proc. of 22th Design Automation Conference, Las Vegas, NV, June 1985, pp. 803-807.