Test Generation for Sequential Circuits

HI-KEUNG TONY MA, SRINIVAS DEVADAS, A. RICHARD NEWTON, SENIOR MEMBER, IEEE, AND ALBERTO SANGIOVANNI-VINCENTELLI, FELLOW, IEEE

Abstract—**We present a novel approach to test pattern generation for synchronous sequential circuits. We have developed an efficient deterministic sequential test generation algorithm, based on extensions to the PODEM justification algorithm, which is effective for midsized sequential circuits. This algorithm can be used in conjunction with an incomplete scan design approach to generate tests for very large sequential circuits.**

Our approach to test generation involves first extracting part of the state transition graph (STG) of the Moore or Mealy finite state machine to be tested, 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 algorithms for fault excitation and propagation and state justification which are based on the *concept of state space 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 the maximum possible fault coverages.

For very large sequential circuits, an *incomplete scan design* approach to test generation has been developed. In this approach, the test generation algorithm is first used to generate tests for a large subset of faults in the circuit. A *small subset of memory elements* is then found, which if made observable and controllable will result in easy detection of all remaining irredundant but difficult-to-detect faults. The deterministic test generation algorithm is again used to generate tests for these faults in the modified circuit (the circuit with the identified memory elements made scannable). We can guarantee detection of all irredundant faults as in the complete scan design case, but at significantly less area and performance cost. The length of the test sequences for the faults can be bounded by a prescribed value—in general, a tradeoff exists between the number of memory elements required to make scannable and the maximum allowed length of the test sequence.

I. INTRODUCTION

TEST GENERATION for sequential circuits has long been recognized as a difficult task [1]–[4]. 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.

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 [5], [6]. Scan design approaches have been successfully used to reduce the complexity of the problem of test generation for sequential circuits by transforming the problem into that of combinational test generation, which is considerably less difficult. The design rules of scan design also constrain the sequential circuits to be synchronous so that the normal operation of the sequential circuit is free of critical races. However, there are situations where the cost in terms of area and/or performance and/or testing time of complete scan design is unaffordable. In addition, even though the general sequential testing problem is very difficult, there are cases where test generation can be effective. Simply making all the memory elements scannable in a sequential circuit without first ascertaining the difficulty of 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 and 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 [7]–[12] have been taken in the past to solve the problem of test generation for sequential circuits. They are either extensions of the classical D-algorithm or are based on random techniques [8], [11]. 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 too can be very time consuming for difficult faults that have

Manuscript received February 8, 1988; revised May 28, 1988. This work was supported in part by the Semiconductor Research Corporation under Grant 442427-52055, by the Defense Advanced Research Projects Agency under Contract N00039-86-R-0365, and by a grant from AT&T Bell Laboratories. The review of this paper was arranged by Associate Editor V. K. Agarwal.

This is an expanded version of the work originally presented at the ICCAD-87.

The authors are with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720.

IEEE Log Number 8822863.

0278-0070/88/1000-1081$01.00 © 1988 IEEE

only a few long test sequences. Our approach to test pattern generation for sequential finite state machines represents a significant departure from these methods.

We have developed an efficient deterministic sequential test generation algorithm based on extensions to the PODEM [13] justification algorithm. 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 new algorithms based on the concept of **state space enumeration**.

PODEM has been shown to be more efficient than the D-algorithm for combinational test generation. FAN [14] and TOP [15] are combinational test generation algorithms based on the implicit enumeration algorithm of PODEM, but incorporating additional acceleration techniques. Some of these techniques have been used in our sequential test generation algorithm. Others are currently being evaluated.

We assume the sequential circuit under test is synchronous and free of races 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 graphs (SGT) 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 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 stages 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 fault-excitation-and-propagation and state-justification algorithms to generate 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 the maximum possible fault coverages.

For very large sequential circuits, an incomplete scan design methodology has been developed. First, using the sequential testing algorithm, test sequences are generated for a large number of possible faults in the given sequential circuit. A **small subset of memory elements** is then found, which if made observable and controllable will result in easy detection of all remaining irredundant but difficult-to-detect faults. The deterministic test generation algorithm is again used to generate tests for these faults in the modified circuit (the circuit with the identified memory elements made scannable). We can guarantee detection of all irredundant faults as in the complete scan design case, but at less area and performance cost. The length of the test sequences for the faults can be bounded by a prescribed value—in general, a trade-off exists between the number of memory elements required to be made scannable and the maximum allowed length of the test sequence.

Preliminaries and basic definitions are given in Section II. The deterministic test generation algorithm is described in Section III. This algorithm has been implemented in the program, STALLION. Results obtained on several circuits using STALLION are presented in Section IV. In Section V, the incomplete scan design methodology to sequential test generation is introduced. Algorithms to identify the critical memory elements are presented. Results using STALLION in conjunction with the incomplete scan design approach are given in Section VI.

II. Preliminaries

A. Introduction

A general sequential circuit is shown in Fig. 1. It is realized by combinational logic and feedback registers. The general sequential test generation problem involves finding primary input sequences which can excite the faults in the circuit and propagate their effects to the primary outputs. No access to the inputs and outputs of the memory elements (the next state and present state lines, respectively) is given.

The initial state of the machine (e.g., when powered on) may be unspecified or specified. The fault model used in test generation also varies. We make certain assumptions regarding the sequential circuit to be tested.

1) The machine is assumed to have a **reset state,** R. All input test sequences begin from this reset state.
2) The fault model is assumed to be **single stuck-at.**
3) The memory elements are assumed to be represented as distinct logic primitives. Tests are generated for all single stuck-at fault in the combinational logic part of the sequential machine including faults on the present state lines, primary inputs, next state lines, and primary outputs. Faults within the flip-flops are not considered.

B. Difficulties in Sequential Test Generation

In combinational test generation, a single test vector suffices to excite a fault in the circuit and propagate its effect to the primary outputs. A sequential machine, however, has to be first placed in a state which can excite the fault and only then can the effect of the fault be propagated to the primary outputs. Placing the machine in the required state and propagating the effect of the fault to the primary outputs may each require a sequence of input vectors.

It has been shown [3] that faults in a general sequential circuit, like the one shown in Fig. 1, may require a test sequence of up to 4^n input test vectors, where n is the number of memory elements in the machine. The search space in sequential test generation is thus very large. It was shown in [3] that the algorithmic complexity of the extended D-algorithm is 4^n. Some faults in the circuit may be **redundant;** i.e., they cannot be detected by *any* test

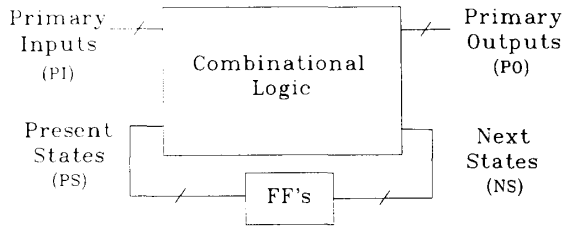

Fig. 1. A sequential circuit.

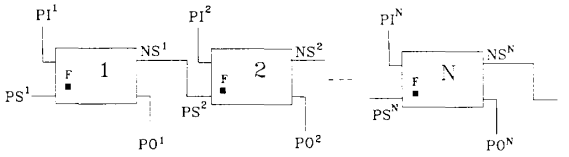

Fig. 2. Iterative array model for sequential circuit.

sequence. Large amounts of effort may be wasted in trying to generate tests for redundant faults, which are, in general, quite difficult to identify.

C. Basic Definitions

The conventional **iterative array model** [3] used in sequential test generation is shown in Fig. 2. The combinational logic block of the original sequential machine (Fig. 1) with a fault, F, to be detected in it has been duplicated in each time frame. Beginning with the present state lines in time frame 1, PS^1, set to the reset state values, we wish to produce an input sequence, PI^1, PI^2, $\cdots$, PI^n, for some n, which when applied to time frames 1, 2, $\cdots$, n propagates the effect of the fault F to the primary output lines of the nth time frame, PO^n. This input sequence is called a **test sequence** for the fault.

A state is considered as a bit vector of length equal to the number of memory elements (latches or flip-flops) in the sequential circuit. In general, a state is a **cube;** i.e., the values of the different bit positions (state lines) may be 0, 1, or x (don't care). A state with only 0's or 1's as bit values is called a **minterm** state.

A state is said to *cover* another state if the value of each bit position in the first state is either an x or is equal to the value of the corresponding bit position in the second state.

The process of finding an input sequence which places the machine, initially in its reset state, R, into a given state, S, is called **state justification.** The input sequence in question is called a **justification path.** State justification may be **forward** state justification or **backward** state justification, depending on whether the search is conducted from R forward or from S backward.

The search space in sequential test generation is deemed to be the product of two spaces, namely the **input space** and the **state space.** The **dimension** of each space is equal to the number of Boolean variables in the space. For example, the dimension of the input space is equal to the number of primary inputs. In our notation, these spaces correspond to the universal input cube and the universal state cube, respectively (the universal cube is a cube with all x entries of length equal to the dimension of the space).

A space can be **enumerated** by exhaustively searching a set of cubes which add up to the universal cube corresponding to that space. **Minterm enumeration** implies that each cube searched is a minterm. Minterm enumeration on an n-dimensional space implies that 2^n combinations have been searched. **Implicit enumeration** (or **implicit cube enumeration** or **cube enumeration**) involves exhaustively searching an n-dimensional space via cubes such that the number of cubes searched is significantly less than 2^n.

In a sequential circuit, a fault may be **redundant,** i.e., untestable. We differentiate between two kinds of redundancies in a sequential circuit. The first kind is deemed **combinationally redundant**—the effect of the fault cannot be propagated to the *primary outputs or the next state lines*, beginning from any state, with any input vector. A **sequentially redundant** fault is a fault which cannot be propagated to the *primary outputs* in 4^n time frames beginning from the reset state of the machine. Note that a combinationally redundant fault is also sequentially redundant.

D. Two Different Approaches to Test Generation

Given a reset state for the sequential machine, two different strategies can be employed to detect faults in the machine:

1) forward propagation from the reset state;
2) a two-stage process of forward propagation and state justification.

In the iterative array model, 1) corresponds to fixing the present state lines in the first time frame, PS^1, to the reset state values and attempting to find n and PI^1, $\cdots$, PI^n so as to propagate the fault(s) to PO^n.

In 2), the test generation process is decomposed into more tractable subproblems. First, an input sequence, $T1 = (PI^i, PI^{i+1}, \cdots, PI^n)$, and an initial state, $S0$, that excite and propagate the effect of the fault to the PO^n are found. This step is called **fault excitation and propagation** and $T1$ is called the **fault propagation** or simply the **propagation** sequence. Next, **state justification** on $S0$ is performed; i.e., a path from R to $S0$ is found consisting of another sequence of input vectors, $T0 = (PI^1, PI^2, \cdots, PI^{i-1})$. $T0$ is called the **setup sequence** or the **justification sequence.** The test sequence is the concatenation of the setup and propagation sequences, $T0$ and $T1$.

It should be noted that both fault propagation and state justification, in general, need a sequence of input vectors. An irredundant fault may be such that its effect can be propagated to the *next state lines alone* in the ith time frame. Time frames $i + 1$ through n are required to propagate the effect of the fault to the primary outputs. Similarly, given an initial state, $S0$, even a minimum-length

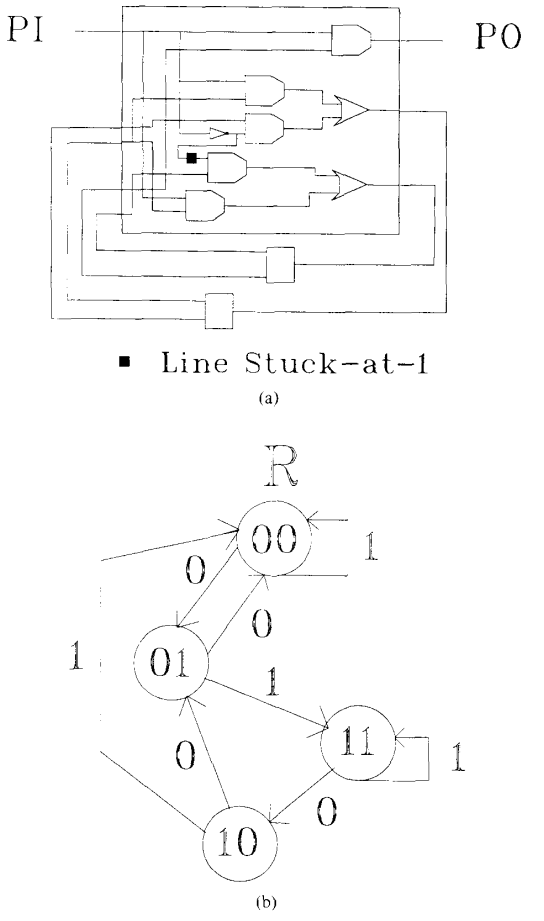

Fig. 3. (a) Sequential machine. (b) State transition graph.

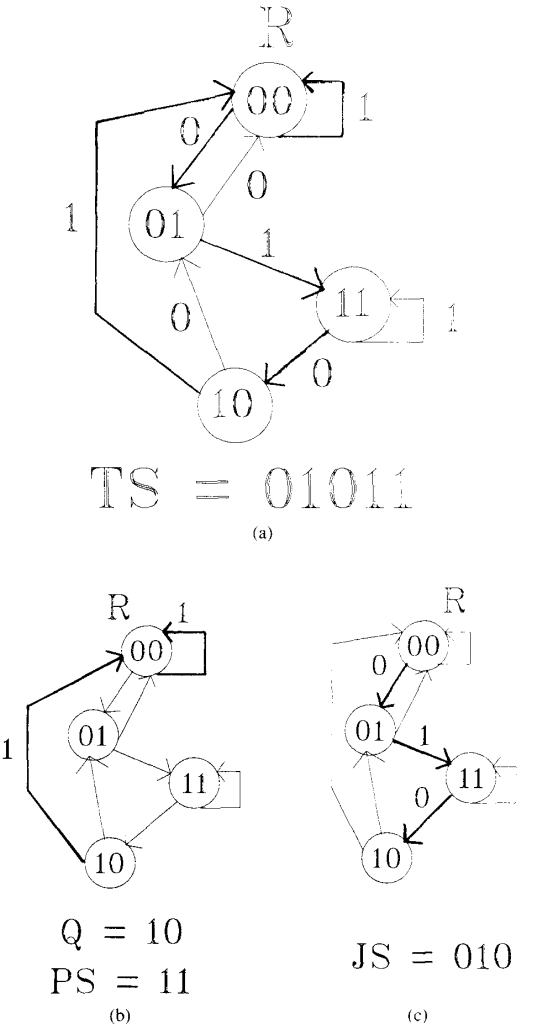

Fig. 4. (a) Test sequence using Approach 1. (b) Propagation sequence. (c) Justification sequence.

justification path for $S0$ may require more than one input vector.

In Figs. 3 and 4, these two approaches are illustrated. We have a sequential circuit with two latches in Fig. 3(a). A stuck-at fault on a line in the combinational logic is to be detected. The state transition graph of the circuit, with the reset state marked R, is shown in Fig. 3(b). In Fig. 4(a), the test sequence, TS, produced using **Approach 1** is shown (highlighted in the state transition graph). In Fig. 4(b) the propagation sequence, PS, and initial exciting state, Q, using **Approach 2** are shown. The state justification sequence, JS, is shown in Fig. 4(c).

III. A Deterministic Sequential Test Generation Algorithm

In our approach to sequential test generation, two different steps of forward fault propagation and state justification are performed. A deterministic sequential test generation algorithm, incorporating new techniques for fault propagation and state justification based on extensions to the PODEM justification algorithm, is presented in this section. Information contained in a partial state transition graph of the sequential circuit is exploited to facilitate detection of faults requiring long test sequences.

This section is organized as follows. In subsection A, the overall strategy used in generating test sequences for single stuck-at faults is described. Extraction of the fully or partially connected state transition graph from the logic-level sequential circuit is described in subsection B. The fault-excitation-and-propagation and state-justification algorithms are described in subsections C and D, respectively. The detection of a special class of redundant faults is described in subsection E.

A. The Overall Strategy

We first outline a test generation procedure which assumes the existence of a complete state transition graph (STG) description of the *fault-free* sequential circuit.

Assuming that the complete STG of the fault-free sequential circuit is available, test generation for a fault un-

der 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 in the primary outputs within 4^n time frames, where n is the number of latches in the sequential circuit. Then, every path from the reset state, R, to any state $S1$ that is covered by $S0$, a potential setup sequence, in the complete STG is fault simulated. Since the STG corresponds to the fault-free machine, it is not guaranteed that the path from R to $S1$ exists under faulty conditions. 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 $T2$. Multiple paths in the STG can be tried as potential setup sequences. If no test sequence can be found, a new fault propagation sequence $T1$ and a new initial state $S0$ which is *disjoint* from all previously generated ones are 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 that 1) the memory storage for the complete STG may be unreasonably large and the generation of the complete STG may demand astronomical CPU time and 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 state justification algorithm that searches for paths from the reset state to the $S0$'s under fault-free conditions. Another alternative to 1) is to perform state justification under faulty conditions so as to ensure that the justification path found is a setup sequence. In either case, 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 a (new) fault propagation sequence $T1$ and a (new) initial state $S0$ that will excite and propagate the effect of the fault under test to the primary outputs within 4^n time frames using the fault-excitation-and-propagation algorithm (described below in subsection C). If no solution exists, exit without a test.
2) Find a (new) path $T0$ (potential setup sequence) from the reset state to the initial state $S0$ using a state justification algorithm in the fault-free machine. If no solution exists, go to 1).
3) Fault simulate the potential setup sequence $T0$. If it detects the fault, generate the test sequence $T2$ directly from $T0$ and go to 5). Else if it is a valid setup sequence, go to 4). Else if $T0$ neither detects the fault nor is a setup sequence go to 2).
4) Concatenate the setup sequence $T0$ that represents the justification 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.
5) Exit with a test sequence.

Even though this algorithm is potentially effective, state justification in general is difficult when the setup sequence is long. In addition, some states may need to be justified more than once. Enhancements to this basic algorithm are discussed in the next section.

Enhancing the Basic Strategy: As mentioned in Section II, state justification can be performed in two different ways, via forward searching or backward searching. These two different search techniques are illustrated in Fig. 5(a) and (b). In forward state justification, if a single input vector which places the machine from R to the state to be justified, Q, cannot be found, a state Q', reachable from R via a single input vector, is found. Then a path from Q' to Q is searched for. In backward state justification, if a single input vector which places the machine from R to the state to be justified, Q, cannot be found, a state Q', which reaches Q via a single input vector, is found. Then, a path from R to Q' is searched for.

Both forward and backward state justification, in faulty or fault-free machines, can be extremely time consuming as the search space is very large. It is true that, given the justification algorithms used, some states are easier to justify backward than forward and vice versa.

Thus, an important enhancement to the test generation strategy, which *combines the advantages of forward and backward state justification*, is to generate *a partial STG containing as many valid states* (and paths from the reset state to them) as possible through forward searching/enumeration (as described in subsection III-B) and to use a backward justification algorithm on the fly during test generation if the initial state, $S0$, required for a fault does not exist in the partial STG. Note that the partial STG may contain all the valid states in the complete STG but contains many fewer edges. States and edges may be *added* to the partial STG via backward justification during test generation.

The second drawback of **Algorithm Structure 1,** that the 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 *by itself.* There is no need to concatenate $T0$ with $T1$ to produce a test sequence, $T2$. Therefore, there is rarely the need for fault simulation of more than one potential setup sequence for a fault. If $T0$ is not a setup sequence or a test sequence it means that the effects of the fault have been propagated to the next state lines (but not the outputs). $T0$ can thus serve as a starting point for a test sequence.

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 a (new) fault propagation sequence $T1$ and a (new) initial state $S0$ that will excite and propagate

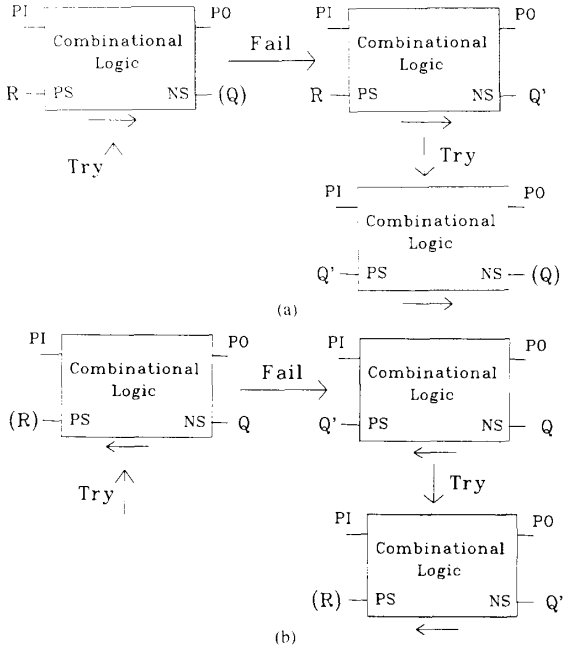

Fig. 5. (a) Forward state justification. (b) Backward state justification.

the effect of the fault under test to the primary outputs within a prescribed number of time frames using the fault-excitation-and-propagation algorithm (described in subsection III-C). 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 backward state justification algorithm (described in subsection III-D). If no solution exists, go to 1).
5) Fault simulate the potential setup sequence $T0$. If it detects the fault, generate the test sequence $T2$ directly from $T0$ and go to 7). Else if it is a valid setup sequence, continue. Else attempt to use $T0$ as a starting point for a test sequence. Else go to 1).
6) Concatenate the setup 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 minterm 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.

B. State Transition Graph Extraction

The input to the logic-level extraction program is the combinational logic specification of the finite state machine as well as information on 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 minterm state, and an edge between two nodes represents an input combination (cube) that drives the finite state machine for one specific state to another.

The STG extraction algorithm first cube-enumerates all fan-out 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 states can be found. Since each state reachable from the reset state is picked as a new starting state and the fan-out of each starting state is enumerated, 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 a very large number of states can be handled.

The algorithm used to cube-enumerate the fan-out edges from a state is an extension to the implicit enumeration algorithm of PODEM [13]. Initially, all the primary inputs and next states of the logic-level finite state machine are set to unknown values. 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 of setting the value of the chosen next state line to a known value. 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 are continued until all next state lines are set to known values—a fan-out edge is enumerated. Whenever an edge is found, but rejected because it leads to a state already in the partial STG, backtracking is performed on the input cube to where a primary input was first set to a known value. The opposite value is assigned to it. 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 gathered 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 pseudocode 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;
         }
      }
   }
}
```

In Fig. 6, the STG extraction process for the sequential machine of Fig. 3 is illustrated. Beginning with the reset state, the extraction algorithm proceeds in depth-first fashion, building up the partial STG.

C. 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. It uses nine-valued simulation, as opposed to the conventional five-valued simulation used in PODEM, to handle the multiple-fault effect in sequential test generation (a fault is repeated in each time frame). FEP uses the conventional iterative array model, shown in Fig. 2, for generating an input propagation 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 initial state $S0$ produced by FEP is a fault-free state (line values 0, 1, or x). The iterative array is considered wholly as a combinational circuit with primary inputs of different time frames time-indexed (PI^1, PI^2, etc.) and the present state lines of the first time frame treated as pseudoinputs (PS^1, PS^2, etc.). The initial state $S0$ is specified by the pseudoinput 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.

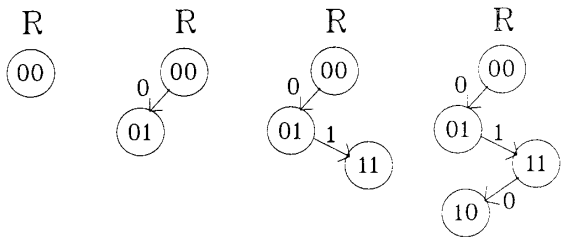

Fig. 6. STG extraction.

If a fault is combinationally redundant, FEP will not be able to propagate the effect of the fault to the primary outputs or the next state lines of the first time frame. FEP continues to the second and succeeding time frames only if the effect of the fault has been propagated to the next state lines of the previous time frame.

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 through backtracing and backtracking processes similar to those used in PODEM. The present state lines of the first time frame are treated similarly to 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 pseudoinput 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 the backtracking limit is reached. Backtracking takes place whenever it can be established that under the current set of primary input and pseudoinput 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. This check is performed similar to PODEM. Backtracking during the search for $T1$ and $S0$ is done on both decision trees.

FEP employs *the concept of disjoint state space enumeration* to make sure that all the tests it generates for a specific fault will have disjoint initial states $S0$; this is necessary because of the loop in the test generation process described in subsection III-A. Whenever the search for a new test is begun, the primary input decision tree ($D1$) for the previous test is scratched completely but the present state decision tree ($D2$) of the initial state $S0$ is retained. Immediately, backtracking is done on $D2$. 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 $S0$'s is related to how FEP is used in the test generation process described in subsection III-A. For a specific fault, a new test

is requested only if we cannot find a path from the reset state to the $S0$ in the previous test, either in the extracted STG or through the state justification algorithm described in subsection III-A. Therefore, all test generated for a specific fault should have disjoint $S0$'s.

A single decision tree could have been used instead of two separated ones as described above. 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 pseudoinput (present state line) is first set to a known value and assign it the opposite value. This means that some primary inputs may be preset to 0 or 1 values when beginning the search for a new initial state $S0$ (which is disjoint from the previous one, since one state bit value has been changed to an opposite value). 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 primary inputs as possible. Therefore the double decision tree method is used.

D. The State Justification Algorithm

Given a goal state $S0$, the state justification algorithmn (SJ) attempts to find a path (setup sequence) from the reset state to it. $S0$ 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 $S0$.

SJ performs backward justification from $S0$ to R, given a prescribed limit on the number of backtracks, to bound CPU time usage. SJ is used only if $S0$ does not exist in partial STG. First, SJ sets the next state lines to $S0$ and enumerates all the fan-in edges to $S0$. SJ then checks to see whether any of the states the edges fan out from cover the reset state or a state in the partial STG. If such a state exists, a path is found. Otherwise, SJ picks each fan-in state as a new goal state and carries out fan-in 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 fan-in edge enumeration algorithm is an extension to the PODEM enumeration algorithm. Here, we have multiple line (the next state line) values to be justified simultaneously rather than a single output line as in PODEM. The concept of state space enumeration is also employed in SJ. There are two decision trees to be maintained as in subsection III-C, 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 backtracking 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 fan-in edge is begun, $D1$ is completely scratched but $D2$ is retained. Immediately, backtracking is done on $D2$. Then, the enumeration procedure is repeated again with a new fan-out state. All edges (fanning out of disjoint states) fanning into a state have been implicitly enumerated when no more backtracking is possible. The pseudocode below illustrates the state justification algorithm proceeding in depth-first fashion. Breadth-first search is an alternative.

```
Justify_State(State)
{
   PresentStateLines (ps) = unknown;
   PrimaryInputs (pi) = unknown;
   simulate the circuit;

   while (not all fan-in 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 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 fan-in state is found) {
         if (fan-in state covers reset state in partial STG)
         {
            a path is found;
            return;

         }
         else Justify_State(fan-in 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;
      }
   }
}
```

E. Detection of Redundant Faults

The difficulty in test generation for sequential circuits does not just lie with finding tests for the difficult-to-detect but testable faults. The determination of redundant faults is equally formidable if not more difficult. Obtaining a low fault coverage does not necessarily mean that 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, which no test has

been generated for, are redundant or testable may demand an astronomical amount of CPU time.

As defined in Section II, two classes of redundant faults exist in a sequential circuit. Combinationally redundant faults are detected using the fault-excitation-and-propagation algorithm, FEP. In general, they are easier to find than sequentially redundant faults.

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 sequentially redundant faults based on Theorem 1, given below, and treat other undetected faults as possibly testable faults. This gives a lower bound on 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 line values set to the fan-in state of the edge.

Theorem 1: In order for a stuck-at fault to be detected, the fault should at least corrupt one fan-out 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 from the reset state and ending with a corrupted edge in the STG. If a fault does not corrupt any fan-out 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 state. The procedure to find these redundant faults is based on the FEP algorithm, described in subsection III-C. A single time frame is used and *all next state lines are treated as primary outputs.* (During test generation, the fault effect has to be propagated to the primary outputs alone, and therefore FEP may require multiple time frames.) It should be noted that FEP attempts to find a fault-free state and input vector which detects the given fault. We generate all tests, for a potential redundant fault, with disjoint initial states. If none of the initial states covers any of the states in the partial STG, the fault under test is redundant.

IV. Test Generation Results Using STALLION

The test generation algorithm described in the previous section have been implemented in the program STALLION. STALLION consists of about 10 000 lines of C code and runs in a VAX-UNIX environment.

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

As can be seen our test generation technique obtains close to the maximum possible fault coverage in all the

TABLE I
STATISTICS FOR EIGHT EXAMPLE CIRCUITS

CKT	#inp	#out	#gate	#lat	#eqv. faults
cse	7	7	192	4	680
sse	7	7	130	6	486
planet	7	19	606	6	2028
sand	9	6	555	6	1932
scf	27	54	959	8	3338
mult4	9	9	170	15	506
sbc	40	56	1011	28	3008
stage	131	64	2700	64	9155

TABLE II
TEST GENERATION RESULTS FOR CIRCUITS

CKT	#test seq.	#vec	max. seq. len.	fault cov. (%)	red.* fault (%)	tfc§ (%)	CPU† time
cse	96	472	8	99.71	0.29	100.0	53.2s
sse	46	284	10	84.57	15.23	99.8	69.9s
planet	80	1191	26	97.39	2.56	99.95	12.6m
sand	165	1077	24	94.36	5.18	99.54	22.4m
scf	136	2238	21	94.37	3.86	98.23	83.0m
mult4	17	120	24	96.25	2.96	99.21	40.6s
sbc	168	1063	24	95.68	2.66	98.34	62.1m
stage	139	425	26	93.97	6.03	100.0	154m

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

TABLE III
TIME PROFILES FOR 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
mult4	27.5s	9.7s	6.5s	0.2s	43.9s
sbc	12.4m	28.3m	21.4m	1.3s	62.1m
stage	10.1m	50.8m	94.1m	1.0m	154m

examples. The extraction of the partial 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 **sse**, as can be seen in Table III. Our fault simulator uses a 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 **sse** is that a great amount of time is consumed in trying to find tests for the large number of redundant faults.

The first five examples are finite state machines obtained from various industrial sources. The example **sbc** is the snooping bus controller [16] in the SPUR chip set. It was synthesized using the multiple level logic optimization system MIS [17].

So far, we have not been able to perform direct comparisons with other deterministic sequential test generation systems. However, performance figures have been quoted for the D-algorithm-based test generation system of [10] for some publicly available benchmarks. We reproduce the figures here along with STALLION's results for the same examples. The CPU times are in seconds on a VAX 11/8650. In all aspects and for all the examples STALLION's results are vastly superior to the TPG system of [10].

The test generator of [10] does not have an interactively running fault simulator. Thus the fault list is not updated and the test generator produces tests for each fault separately. Also, for each test, a new initialization is attempted. These two factors increase the run time for the test generator of [10].

The example **planet** has faults that require long test sequences (Table II). Because of this the differences between the performances of the test generator of [10] and STALLION are more marked in **planet** than in the other two examples.

V. An Incomplete Scan Design Approach

A. Introduction

In the previous sections, we presented a sequential testing algorithm effective for midsized sequential finite state machines. In this section, we present a new incomplete scan design approach to test generation for large sequential circuits. First, using the efficient sequential testing algorithm, STALLION, test sequences are generated for a large number of possible faults in the given circuit. A **small subset of memory elements** is then found, which if made observable and controllable will result in all remaining irredundant but previously difficult-to-detect faults being easily detected in the modified circuit by STALLION. We can guarantee detection of all irredundant faults as in the complete scan design case, but at significantly less area and performance cost. The length of the test sequences for the faults can be bounded by a prescribed value—in general, a trade-off exists between the number of memory elements required to be made scannable and the maximum allowed length of the test sequence.

Related work in this area has involved using functional vector sets to initially detect faults in the sequential circuit [18]. After functional vector fault simulation, a set of memory elements is made scannable so as to ensure that test sequences, each consisting of a *single* test vector, can detect the remaining undetected faults. Our approach is not restricted to using single vector test sequences to detect faults. A minimal set of critical memory elements is found which, when made scannable, allows the sequential test generation algorithm to detect the faults via *multiple* vector test sequences.

This section will focus on the algorithm used in the identification of the minimal subset of memory elements. The overall strategy used in the identification of the critical memory elements is described in subsection B. Heuristic selection algorithms used in the last stage of the identification process are described in subsection C.

B. The Global Strategy

The overall structure of the incomplete scan design algorithm is shown below. The algorithm incorporates the sequential test generation algorithm STALLION, described in Section II. It is given the sequential circuit, S, and a set of faults to be detected, F. It produces a set of test sequences, T, each beginning from the reset state of the machine, R, and identifies a set of memory elements, M, to be made scannable. The T are such that they detect the faults F in S^M, the sequential circuit S with the memory elements M made observable and controllable.

A fault in F may not have been detected by STALLION because of a failure to find a propagation sequence or a justification sequence or both. For each undetected fault due to a justification failure, the algorithm finds multiple sets of memory elements as well as a justification sequence corresponding to each set, such that making any one set of memory elements controllable results in detection of the fault. For each undetected fault due to a propagation failure, a set of memory elements and a propagation sequence are found, such that making any one of the elements in this set observable results in detection of the fault. Given different choices of sets of memory elements to detect all the faults in F, a selection algorithm selects a set corresponding to each fault so as to minimize the total number of memory elements that are to be made scannable.

Incomplete Scan Design Algorithm

1) Given the sequential circuit S, for each fault $f \in F$, attempt to find a fault-excitation-and-propagation sequence, $S0_f$ and $T1_f$, which will propagate the effect of f to the primary outputs if possible, else to the next state lines. The length of $T1_f$ is limited to *MAX_PROP_LEN*. If $T1_f$ does not propagate f to the primary outputs, a set of next state lines, NS_f, to

which the effect of f can be propagated to is found. The set of faults which can be propagated only to next state lines is called F^{NS}.

2) The distinct state vectors in $S0_f \; \forall f$ are found. Call this set of distinct state vectors K.
3) Generate *MAX_STATE* states, Q_i, at different levels, i, from R in the state transition graph (STG) of S. i varies from 1 to *MAX_LEVEL*. *MAX_STATE* is limited to 3200 to restrict memory usage.
4) For each $k \in K$, find the memory lines to be made scannable such that each state $q \in Q_i$ covers k. For each k generate *MAX_CHOICE* best (with the least number of lines) choices for line sets which, if made scannable, will result in the covering of k by $q \in Q_1$.
5) Given K and *MAX_CHOICE* choices of line sets for each $k \in K$, and a line set NS_f for each $f \in F^{NS}$, select a line set for each k and a single line from each NS_f line set so the number of distinct lines to be made scannable, M, is minimized.

In **step 1**, propagation sequences are found for faults in the sequential circuit, S, using the sequential test generation algorithm *STALLION*. For each fault STALLION produces a starting state, $S0_f$, and an input vector sequence, $T1_f$, which propagate the effect of the fault to either the primary outputs or the next state lines of S. While running STALLION, the present state lines of S are set last so the state vectors, $S0_f$, have as many don't care bits in them as possible. If the effect of the fault is propagated to the primary outputs, then it only remains to justify $S0_f$; otherwise the complete set of next state lines to which the effect of the fault can be propagated, NS_f, has to be found as well.

In **step 3** a large set of states in the state transition graph of S and paths from the reset state leading to each state are found. These states are extracted using the STG extraction algorithm described in subsection III-B. The number of states and the lengths of the justification paths are bounded by *MAX_STATE* and *MAX_LEVEL*, respectively.

The **distance** between two arbitrary bit vectors of length N, $A(i)$ and $B(i)$, where each bit can take the values of 0, 1, and x (don't care), is defined as the number of bits where $A(i)$ is 1 and $B(i)$ is 0 or *vice versa* over $i = 1, \cdots, N$. Given a state $q \in Q_i$ and a state $k \in K$, the distance between q and k then gives the number of state line values that q and k differ in. It is easy to see that if the state lines which are different q and k are made controllable, any justification sequence for q has to work for k. Don't care bits in k do not contribute to distance since they can take the values of 0 or 1. So to minimize distance between the states in K and Q_i, the number of don't care bits in K is maximized by setting the state lines last in **step 1.**

In **step 4,** for each pair of q and k, the set of lines which are to be made scannable for the justification sequence of q to be usable for k is found. Then, for each k, *MAX_CHOICE* such line sets with the least number of lines and, secondarily, the smallest justification sequence length are selected.

Given these *MAX_CHOICE* sets of lines for each k, a heuristic algorithm (**step 5**) is used to select one particular line set (corresponding to one particular $q \in Q_i$) for each k, so as to minimize the total number of distinct lines in all the line sets selected. Simultaneously, for each fault, f, which could not be propagated to the primary outputs, a line from NS_f (the set of next state lines to which f can be propagated) is selected. Two selection algorithms which have been employed are described in the next section.

After the heuristic selection process, a minimal number of state lines to be made scannable and justification sequences for K are identified. A set of test sequences, T, is formed by concatenating the justification sequences, J, with the propagation sequences, $T1$, generated using STALLION. T will detect all undetected faults in S^M, namely F.

C. Heuristic Algorithms for Selection

The subproblem to be solved is as follows. We have N elements (each corresponding to a fault), each with a set of line groups. The number of line groups may vary. We assume that for any given element none of the line groups is a superset or subset of any other line group of the same element. The goal is to identify a line group for each element such that the number of distinct lines in the selected line groups is minimum. Some of the elements may have a set of line groups, each containing a single line. For example, during test generation, faults that can only be propagated to next state lines will result in an element with the property mentioned above.

Two heuristic algorithms which produce minimal solutions are described below.

Algorithm 1

```
for(i = 1; i ≤ N; i++) {
   Element = e[i];
   for (j = 1; j ≤ Element.NumChoices; j++) {
      Solution = greedy(Element.Choice[j], Element)
   }
}
select best Solution;
greedy(lineGroup, element) {
   Lines = Lines ∪ lineGroup;
   for (i = 1; i ≤ N; i++) {
      if (e[i] ≠ element) {
         for (j = 1; j ≤ e[i].NumChoices; j++) {
            card = | Lines ∪ e[i].Choice[j[ |
         }
         pick k so card is minimum;
         Lines = Lines ∪ e[i].Choices[k[;
      }
   }
   return(Lines);
}
```

Algorithm 1 is a very fast greedy algorithm run with different starting points.

Algorithm 2

find all required lines, Lines;

while (choices to be made) {

pick N_c elements, $e_1, \cdots e_{N_c}$ and choices for the elements, $c_1, \cdots c_{N_c}$ minimizing $|\text{Lines} \cup e_i[c_i]| \quad i = 1, N_c$

mark choices made for e_i

}

Algorithm 2 first finds all lines which are definitely required, if any. For example, if for any element a line exists in all its choices, that line is definitely required. Then, N_c best elements and line groups in these elements are picked at a time. The complexity of this algorithm is $O(N^{N_c})$. The larger N_c is, the better the solution can potentially be ($N_c = N$ is exhaustive search) but more CPU time is required. We have found that $N_c = 3$ gives near-optimum results within acceptable run times.

TABLE IV
COMPARISON WITH TPG SYSTEM OF [10]

EXAMPLE	STALLION			TPG system of [10]		
	tfc§	#vect.	CPU time	tfc§	#vect.	CPU time
sse	99.80	284	69.9	99.50	676	1134
mult4	99.21	120	40.6	93.15	148	1490
planet	99.95	1191	754.0	61.00	132	19388

§ total fault coverage including detected and provably redundant faults

TABLE V
INCOMPLETE SCAN DESIGN RESULTS

EXAMPLE	#inp	#out	#gate	#lat	red. fault %	initial fault cov.	scan latches	final fault cov.	CPU time (secs)
sse	7	7	130	6	0.0	84.57	3	100.0	4.9
sand	9	6	555	6	0.21	94.31	2	99.79	5.3
scf	27	54	959	8	0.51	94.67	2	99.49	1.8
donfile	2	1	232	12	0.0	63.60	5	96.34	11.1
							9	100.0	14.3
sckt4	3	6	160	21	1.09	26.55	6	92.93	2.1
							9	98.91	2.4
sbc2	35	51	1011	33	2.83	81.25	5	95.68	9.1
							10	97.17	17.1
lex1	27	52	395	97	0.0	75.86	39	97.94	6.3

VI. Results Using Incomplete Scan Design

In this section, we present results obtained on several circuits using STALLION in conjunction with an incomplete scan design methodology. Results obtained on seven circuits are summarized in Table V. In the table, the number of inputs (#inp), outputs (#out), gates (#gate), and latches (#lat) in each circuit is indicated. The percentage of combinationally redundant faults (which cannot be detected even using complete scan design), initial fault coverage achieved by STALLION, the number of latches made scannable, the final fault coverage in the modified circuit, and the CPU times used for selection on a VAX 11/8650 are also given. The CPU times for sequential test generation varied between a minute for the smaller examples to an hour for the largest example, **sbc2,** on a VAX 11/8650. It should be noted that the CPU time used for sequential test generation can be bounded by limiting the number of backtracks allowed. However, if this is done, quite possibly fewer faults will be detected and more scan latches may be required.

As can be seen, making a small subset of latches scannable increases the fault coverage obtained to the maximum possible value or very close to the maximum possible value for all the circuits. For instance in example **sckt4,** making nine out of 21 latches scannable raised the fault coverage from 26.25 percent to the maximum possible value of 98.91 percent (1.09 percent of the faults are combinationally redundant). Depending on the fault coverage required, differing numbers of scan latches suffice. Trade-offs can be made for each example, as indicated in Table V.

The examples **donfile** and **sckt4** originally have a large number of sequentially redundant faults, which is why the fault coverage obtained by STALLION for these two examples is low. Making some memory elements scannable allows detection of these faults, increasing the fault coverage to the maximum possible value.

The results demonstrate the advantages of using a combined approach of sequential test generation and incomplete scan design. A large percentage of faults are detected using the efficient sequential test generation algorithm, and the remaining irredundant faults are detected by the same algorithm after making a minimal subset of flip-flops observable and controllable.

VII. Conclusions

A novel approach to test generation for synchronous sequential finite state machines has been presented in this paper. We have developed an efficient deterministic test generation algorithm for sequential circuits. The efficacy of our method stems from the integration of several new algorithms that are based on the concept of state space 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 have obtained close to the maximum possible fault coverages. A new method of detecting a special class of redundant faults in determining how close the fault coverage obtained is to the maximum possible value has also been described.

For very large sequential circuits, we have developed

an incomplete scan design approach to test generation. In this approach, the test generation algorithm is first used to generate tests for a large subset of faults in the circuit. A minimal set of critical memory elements is then found, which, if made observable and controllable, will result in easy detection of all remaining irredundant but difficult-to-detect faults. The deterministic test generation algorithm is again used to generate tests for these faults in the modified circuit (the circuit with the identified memory elements made scannable). We can guarantee detection of all irredundant faults as in the complete scan design case, but at significantly lower area and performance cost.

REFERENCES

[1] F. C. Hennie, "Fault detecting experiments for sequential circuits," in *Proc. 5th Ann. Symp. Switching Circuit Theory and Logical Design* (Princeton, NJ), Nov. 1974, pp. 95–110.

[2] W. G. Bouricius *et al.*, "Algorithms for detection of faults in logic circuits," *IEEE Trans. Comput.*, vol. C-20, Nov. 1971).

[3] M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*. Rockville, MD: Computer Science Press, 1976.

[4] A. Miczo, "The sequential ATPG: A theoretical limit," in *Proc. 1983 Int. Test Conf.* (Philadelphia, PA), Oct. 1983, pp. 143–147.

[5] E. B. Eichelberger and T. W. Williams, "A logic design structure for LSI testability," in *Proc. 14th Design Automat. Conf.*, June 1977, pp. 462–468.

[6] V. D. Agarwal, S. K. Jain and D. M. Singer, "Automation in design for testability," in *Proc. Custom Integrated Circuits Conf.* (Rochester, NY), May 21–23, 1984.

[7] M. A. Breuer, "A random and algorithmic technique for fault detection and test generation for sequential circuits," *IEEE Trans. Comput.*, vol. C-20, pp. 1366–1370, Nov. 1971.

[8] H. D. Schnurmann, E. Lindbloom, and R. G. Carpenter, "The weighted random test-pattern generator," *IEEE Trans. Comput.*, vol. C-24, pp. 695–700, July 1975.

[9] R. Marlett, "EBT: A comprehensive test generation technique for highly sequential circuits," in *Proc. 15th Design Automat. Conf.* (Las Vegas), June 1978, pp. 332–338.

[10] S. Mallela and S. Wu, "A sequential test generation system," in *Proc Int. Test Conf.* (Philadelphia, PA) Oct. 1985, pp. 57–61.

[11] S. Nitta, M. Kawamura, and K. Hirabayashi, "Test generation by activation and defect-drive (TEGAD)," *Integration, VLSI J.*, vol. 3, pp. 2–12, 1985.

[12] S. Shteingart, A. W. Nagle, and J. Grason, "RTG: Automatic register level test generator," in *Proc. 22nd Design Automat. Conf.* (Las Vegas), June 1985, pp. 803–807.

[13] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Trans. Comput.*, vol. C-30, pp. 215–222, Mar. 1981.

[14] H. Fujiwara, "FAN: A fanout-oriented test pattern generation algorithm," in *Proc. ISCAS 85*, June 1985, pp. 671–674.

[15] T. Kirkland and M. R. Mercer, "A topological search algorithm for ATPG," in *Proc. Design Automat. Conf.* (Miami Beach, FL), July 1987, pp. 502–508.

[16] M. Hill *et al.*, "Design decisions in SPUR," *IEEE Computer*, vol. 19, pp. 8–22, Nov. 1986.

[17] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: A multiple level logic optimization system," *IEEE Trans. Computer-Aided Design*, Nov. 1987.

[18] V. D. Agrawal *et al.*, "A complete solution to the partial scan problem," in *Proc. Int. Testing Conf.*, Sept. 1987.

*

Hi-Keung Tony Ma, for a photograph and a biography, please see page 712 of the June 1988 issue of this TRANSACTIONS.

*

Srinivas Devadas, for a photograph and a biography, please see page 712 of the June 1988 issue of this TRANSACTIONS.

*

A. Richard Newton (S'73–M'78–SM'86), for a photograph and a biography, please see page 722 of the June 1988 issue of this TRANSACTIONS.

*

Alberto Sangiovanni-Vincentelli (M'74–SM'81–F'83), for a photograph and a biography, please see page 519 of the April 1988 issue of this TRANSACTIONS.