SEQUENTIAL TEST PATTERN GENERATION: USING IMPLICIT STG TRAVERSAL TECHNIQUES TO GENERATE TESTS AND IDENTIFY REDUNDANCIES IN SEQUENTIAL CIRCUITS

by

Carol Wawrukiewicz, Alexander Saldanha, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M94/4

1 February 1994

SEQUENTIAL TEST PATTERN GENERATION: USING IMPLICIT STG TRAVERSAL TECHNIQUES TO GENERATE TESTS AND IDENTIFY REDUNDANCIES IN SEQUENTIAL CIRCUITS

by

Carol Wawrukiewicz, Alexander Saldanha, Robert K. Brayton,
and Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M94/4

1 February 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Sequential Test Pattern Generation: Using Implicit STG Traversal Techniques to Generate Tests and Identify Redundancies in Sequential Circuits

Carol Wawrukiewicz, Alexander Saldanha, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley

Abstract

The implicit STG traversal techniques developed recently in the verification community can also be used to generate tests and identify redundancies in sequential circuits. A sequential ATPG system which uses these techniques has been implemented in SIS. This system uses two different ATPG algorithms. The first is Ghosh's 3-step test generation procedure, modified to use implicit STG traversal techniques. This 3-step algorithm is fast, but is not guaranteed to produce a test for a fault. The second ATPG algorithm builds the product of the good and faulty circuit, and then implicitly traverses this product machine, just as in implicit circuit verification. If this traversal proves the good and faulty circuits equivalent, then the fault is redundant; otherwise the differentiating sequence is a test for a fault. This second algorithm is guaranteed to produce a test for a fault or else prove the fault redundant, but is more expensive than the first algorithm. Results of applying the SIS ATPG algorithms to the ISCAS '89 benchmark circuits are presented, and these results are compared to those of STEED and VERITAS. Algorithms for generating small test sets and for performing redundancy removal--both based upon the same algorithms used in general ATPG--are also presented, along with the results of applying them to the ISCAS circuits. Finally, suggestions for handling larger circuits--circuits which cannot be tested by any existing ATPG algorithm--are presented.

1 Introduction

Testing is an important step in the manufacture of integrated circuits. Although a circuit's design may have met all specifications, defects may have been introduced into the circuit during fabrication. Thus, after fabrication, it is important to test each circuit to determine whether or not it functions according to specification. The goal of test generation is simply to generate a set of tests that can be used to detect a high percentage of faulty circuits. Of course, as integrated circuits grow in size and complexity, it becomes more difficult to generate good test sets.

In recent years, very effective automatic test pattern generators for combinational circuits have been developed [9], [19], [20]. However, the more difficult task of automatic test pattern generation (ATPG) for sequential circuits remains an open problem. This is the primary problem addressed in this research.

ATPG for sequential circuits is difficult because, in general, a sequence of test vectors is required to test for a fault. The first part of the sequence is used to put the circuit into a state such that the fault can be excited, and the remainder of the sequence is used to propagate the effect of the fault to the primary outputs, where it can be observed.

The obvious solution to the problem of sequential ATPG, then, would seem to be to design circuits using only scan latches, latches that are directly controllable and observable. This transforms the problem of sequential test pattern generation into the simpler problem of combinational test pattern generation. Unfortunately, there are area and performance penalties associated with using scan latches, and testing time is usually longer when scan latches are used. It is because of these disadvantages that a solution to the problem of sequential ATPG is important.

As previously mentioned, the primary task of a test generator is to generate tests that detect a high percentage of faulty circuits. However, because the cost of testing a fabricated circuit depends on the size of the test set, small test sets are also desirable, especially if many thousands of circuits are to be tested. Like the general ATPG problem, the generation of small test sets for combinational circuits has been addressed extensively, but the same problem for sequential circuits has received little attention, again because of the difficulty of the problem. Section 3 of this paper addresses this problem of generating test sets which both detect a high percentage of faults and are relatively small.

The final topic addressed in this paper is that of redundancy removal, a circuit optimization technique which uses test generation as a subroutine. Redundancy removal takes advantage of redundant faults, faults for which no test sequence differentiates between the good and faulty circuit. If a stuck-at fault is redundant, then the faulty line can be replaced by a constant value without affecting the input-output behavior of the circuit. This process is called redundancy removal, and it reduces the area of the circuit. Test generation techniques can be used to identify, redundancies; thus, test generation is a subroutine used during redundancy removal. Redundancy removal is the topic of Section 4.

2 Sequential ATPG

The following subsections address the general problem of sequential test pattern generation. First, in Section 2.1, a few assumptions and definitions are presented. Then previous work in the field of sequential ATPG is discussed in Section 2.2. In Section 2.3, the algorithms that have been implemented in SIS are presented, followed by experimental results in Section 2.4. Finally, Section 2.5 contains conclusions about the effectiveness of the SIS ATPG algorithms.

2.1 Preliminaries

Before test generation can be performed, two models must be chosen, the circuit model and the fault model. The circuit model describes the behavior of a fault-free circuit; the fault model models the defects which can be introduced during fabrication. Given a particular circuit and a fault model, it is possible to generate a list of all faults which can occur according to these models; the task of the test generator is to generate a set of tests which can detect a high percentage of these faults. Many different circuit behavior and fault models can be used, but in this paper, only one type of circuit model and one type of fault model are considered; circuit behavior is described at the logic level, and faults are modeled as single stuck-at faults. This choice of models is a compromise between high-level behavioral models and low-level technology-dependent models. All further discussion of test generation in this paper assumes this particular model.

We shall also assume throughout this paper that all circuits have a reset state. This is not an unreasonable assumption, since a circuit which cannot be reset to some known state is not of much use. For circuits for which no reset state is given, Pixley, et. al. have shown a method to calculate a reset state, along with the reset sequence required to put the circuit into the reset state [16].

2.2 Previous Work

Because combinational ATPG is a well-solved problem, many sequential test pattern generators use the technique of time-frame expansion to convert a sequential ATPG problem into a combinational ATPG problem [1]. In this approach, the behavior of a sequential circuit over *n* time frames is simulated by the operation of a "time-frame expanded" combinational circuit. This time-frame expanded circuit is composed of *n* copies of the combinational logic of the sequential circuit, with the next state lines in time frame *i* connected to the present state lines in time frame *i*+1. A combinational test generator can then be used to generate tests for the time-frame expanded circuit, and these combinational tests can be decomposed to form sequential tests of length *n* for faults in the original sequential circuit. The problem with this method is two-fold. First, before test generation, it is impossible to know how long the test for a particular fault will be, i.e. how large to make *n*. If *n* is chosen too small, then no test will be found, and the process must be repeated with a larger *n*; if *n* is too large, then the combinational test generator does unneeded work. Secondly, the size of the time-frame expanded circuit required to find a test for a fault is directly proportional to the number of test vectors required to test the fault in the sequential circuit. Thus, the longer the test, the larger the combinational ATPG problem that must be solved.

In 1988, Ma, et. al. suggested dividing the generation of a sequential test sequence into two phases, justification and propagation [13]. Justification is the process of finding an input sequence that takes the circuit from the reset state to some specified state S; propagation is the process of finding an input sequence that, when applied from state S, excites the fault and then propagates the effect of the fault to a primary output. This two-phase approach was implemented in the program STALLION. Although STALLION solved the problems of justification and propagation using time-frame expansion techniques, it used information about the state transition graph (STG) of the circuit to aid in the process of combinational test generation on the time-frame expanded circuit. Thus, STALLION was the first sequential test generation program which used the sequential behavior of the circuit to aid in test generation.

Ghosh took the ideas of STALLION a few steps further in his program STEED [8]. In STEED, the problem of sequential test generation is divided into three phases. In the first phase, a combinational test for the fault is generated, assuming that present state lines are directly controllable and next state lines are directly observable. This combinational test can be divided into two parts, the setting of the input lines, called the excitation vector, and the setting of the present state lines, called the excitation state. The excitation state is a setting of the present state lines which allows the fault to be excited, and the excitation vector is a setting of primary inputs that excites the fault, when applied from the excitation state. The next step in Ghosh's algorithm is to justify the excitation state. If the justification sequence, followed by the excitation vector, propagates the effect of the fault to the primary outputs, then the test generation is accomplished. If, however, the excitation vector propagates the effect of the fault only to the next state lines, then a propagation sequence must be found. This sequence propagates the effect of the fault to one of the primary outputs, where the effect can be observed. In STEED, justification and propagation are performed by traversing the STG of the circuit, rather than through time-frame expansion. The STG traversal in STEED is semi-implicit; it is accomplished by repeatedly intersecting the cubes which represent the output functions of the circuit. STG traversals in STEED are first attempted using the STG of the fault-free machine; this allows many justification and propagation sequences to be wholly or partially reused, i.e. used for more than one fault. This greatly increases the efficiency of STEED. Although the sequences obtained from the fault-free machine are not guaranteed to work in the faulty machine, Ghosh showed empirically that they usually do.

While Ghosh was developing STEED, a new implicit STG traversal technique was being developed in the verification community [6], [20]. In this technique, the transition relation of a circuit is represented by the binary decision diagram (BDD) of its characteristic function, and sets of states are also represented by BDD's [3]. It is then simple to calculate the set of states S' reachable in one step from another set of states S; this is just an image computation, involving a single set intersection operation followed by an existential quantification and variable substitution. All of these operations can be performed efficiently on BDD's. Image computations can thus be used to perform an implicit breadth-first-search traversal of a circuit's state transition graph, starting from the circuit reset state. In turn, this STG traversal technique can be used to verify the equivalence of circuits. Given two circuits, the product machine of the two circuits is built, and the STG of the product machine is then traversed. If, from every reachable state of the product machine, the two circuits have exactly the same output, the circuits are equivalent; otherwise there exists a sequence of input vectors which leads to one output in one machine and a different output in the other machine, and thus the two machines are not equivalent.

Although this implicit traversal technique was developed originally for use in sequential circuit verification, it can also be used to improve upon Ghosh's test generation algorithm. This was done by researchers at the University of Colorado at Boulder, in their VERITAS ATPG system [5].

VERITAS uses the same basic structure as does STEED, but the implementation varies significantly. First, in VERITAS, STG traversal of the good machine is performed implicitly, using the implicit STG *traversal* technique described above. Use of this technique speeds the identification of redundant faults and the derivation of justification sequences. In addition, in cases where a fault is not tested or proven redundant during the 3-step algorithm, the product of the good and faulty machine is built and traversed using implicit STG traversal. This good/faulty product machine traversal, if completed, either proves the fault redundant, or generates a test sequence. Thus, on any circuit for which product machine traversal can be performed, VERITAS obtains 100% test fault coverage.

Another difference between VERITAS and STEED is that VERITAS makes more extensive use of random test generation. During the preliminary random test generation phase, VERITAS records not only tested faults, but also faults that have been excited. Next, it uses reachability information and random combinational test generation to generate random reachable combinational tests for unexcited faults. Finally, it uses random propagation sequence generation to generate propagation sequences for the randomly excited faults. In addition, during the 3-step test generation algorithm, VERITAS uses only random propagation, rather than the deterministic fault-free propagation used in STEED.

Because of its implicit STG traversal techniques and sophisticated random test generation, VERITAS runs more than 15 times faster than STEED on the ISCAS '89 benchmark circuits, and also obtains better fault coverage.

Although VERITAS outperforms STEED, it cannot handle circuits any larger than those that can be handled by STEED, which is a serious limitation. Neither program can generate tests for the larger circuits--the circuits with more than 50 latches--in the ISCAS '89 benchmark set. This limitation is shared by all sequential test generators that exist today.

2.3 Test Generation Algorithm

The sequential test generation algorithm described in this section is much like the VERITAS system. Like VERITAS, it is based on the three-step algorithm of STEED, it uses implicit BDD image computation techniques to perform STG traversal, and it performs good/faulty product machine traversal as a last resort. However, unlike VERITAS, our algorithm can also use BDD image computation techniques to deterministically generate propagation sequences. Further, combinational test generation is not performed by a topology-based algorithm like PODEM [9], as it is in both STEED and VERITAS, but rather by the algebraically-based program TEGUS [19]. Our entire test generation algorithm is outlined if Figure 2.1; this algorithm will be described in more detail in the remainder of this section.

Figure 2.1: Test Generation Algorithm

```
Sequential-ATPG
{
    build-bdds;
    traverse-STG;
    random-test-generation;
    for each fault {
        Fault = UNTESTED;
        get-reachable-combinational-test;
        if (no reachable combinational test exists) {
            Fault = REDUNDANT
            continue;
        }
        fault-free-justify-excitation-state;
        fault-simulate-to-get-min-just-sequence;
        if (MinJustSequence is a test) {
            TestSequence = MinJustSequence;
            Fault = TESTED;
            continue;
        }
        random-propagate;
        if (propagated) {
            TestSequence = MinJustSequence + PropagationSequence;
            Fault = TESTED;
            continue;
        }
        deterministic-fault-free-propagate;
        if (propagated) {
            TestSequence = MinJustSequence + PropagationSequence;
            fault-simulate(TestSequence);
            if (TestSequence is a test) {
                Fault = TESTED;
            }
        }
        if (number of unsimulated test sequences = NumberToFaultSimulateAtOneTime) {
            fault simulate to find untested faults that are tested by test sequences;
        }
    }
    for each untested fault {
        good-faulty-product-machine-traversal;
        if (good and faulty machines are equivalent) {
            fault = REDUNDANT
        } else {
            TestSequence = DifferentiatingSequence;
            fault = TESTED;
        }
        if (number of unsimulated test sequences = NumberToFaultSimulateAtOneTime) {
            fault simulate to find other faults that are tested by test sequences;
        }
    }
}
```

Figure 2.2: STG Traversal Algorithm

```
traverse-STG
{
    TotalSet = $\emptyset$;
    for (i = 0 to $\infty$) {
        if (i = 0) ReachedStates[i] = ResetState;
        else ReachedStates[i] = implicit-compute-next-states(ReachedStates[i-1]);
        if (ReachedStates[i] $\subseteq$ TotalSet) then return;
        TotalSet = TotalSet $\cup$ ReachedStates[i];
    }
}
```

Figure 2.3: Fault-Free Justification Algorithm

```
fault-free-justify-excitation-state;
{
    if (ExcitationState has already been justified) {
        reuse-justification-sequence;
        return;
    }
    for (i = 0 to STGdepth-1)
        ReachedExcitationState = ExcitationState $\cap$ ReachedStates[i];
        if (ReachedExcitationState $\neq \emptyset$) {
            break;
        }
    }
    CurrentState = ReachedExcitationState;
    for (j = i to 1) {
        Preimage = implicit-compute-reverse-image(CurrentState);
        ReachablePreimage = Preimage $\cap$ ReachedStates[i-1];
        (InputVector, CurrentState) = get-one-minterm(ReachablePreimage);
    }
    return;
}
```

Preprocessing and Random Test Generation

Before entering the 3-step deterministic test generation loop, several preliminary operations are performed. First, the BDD's of the external output and next state functions are constructed, using the Long BDD package from Carnegie Mellon University [12]. These functions are built for two circuits: the fault-free circuit and the product of the fault-free circuit with itself. The BDD's for the fault-free circuit are used during STG traversal; the BDD's of the product machine are used during fault-free propagation.

Next, the STG of the circuit is traversed to generate the set of reachable states of the system, as outlined in Figure 2.2. This traversal is performed implicitly, in breadth-first-search order, using the implicit STG traversal algorithm described in Section 2.2. The states reached during each step of the traversal are stored as BDD's in an array ReachedStates for use during justification; in addition, the BDD representing the total reachable set is computed for use during combinational test generation.

Finally, if desired, random test generation is performed. The random sequences generated in this step have a default length equal to the STG depth of the circuit. The random test generation procedure uses a single fault/parallel pattern fault simulator to determine which faults are tested by the random sequences.

Three-Step Deterministic Test Generation

After these preliminary steps, the main loop of the program is entered. For each fault not tested during RTG, the first step is to obtain an excitation state and excitation vector using a combinational test generator. Because unreachable excitation states cannot be justified, the combinational test generator has been modified so that it returns only "reachable" combinational tests, i.e. combinational tests that define excitation states that are reachable in the fault-free machine. Hence, this combinational test generation step can also be used to identify some sequential redundancies. In particular, if there is no "reachable" combinational test for a fault, then the fault is sequentially redundant [13]. Using the notation of Cho, et. al. [5], we shall refer to such faults as sequentially non-excitable faults (SNE faults). Empirical results show that most redundant faults in the ISCAS circuits are SNE faults (see Table 2.2 of Section 2.4).

The combinational test generator used in our test generator [19] is based on Larrabee's system of test generation using boolean satisfiability [11]. In this system, the circuit and fault are represented as a boolean formula such that a satisfying assignment for the formula defines a test for the fault. The advantage to using this system is that it is very easy to place restrictions on the combinational tests that are returned; the restrictions are applied by adding additional clauses to the boolean formula. Thus, to add the condition that all combinational tests must be reachable, we simply add clauses which enforce this condition. Thus, only "reachable" tests will satisfy the boolean formula. If the combinational test pattern generator reports that no reachable combinational test exists, then the fault is redundant, and the algorithm goes on to the next fault. If, on the other hand, a reachable combinational test does exist, then one such test is returned by the combinational test generator, and the algorithm goes on to the second step.

The second step is to justify--in the fault-free circuit--the excitation state defined by the combinational test. A fault-free justification sequence is guaranteed to exist, since only reachable excitation states are generated by the combinational test generator. The fault-free justification process uses the results of the circuit STG traversal, as outlined in Figure 2.3.

The fault-free justification procedure first checks whether or not the current excitation state has been previously justified. If it has been, then the fault-free justification sequence is simply looked up in a table of previously-generated results. If the excitation state has not been previously justified, then the justification process begins. This process uses the ReachedStates array calculated during the circuit STG traversal. The first step is to find the smallest i such that ReachedStates[i] contains the excitation state. This ensures that the procedure calculates the shortest possible fault-free justification sequence for this excitation state. After the excitation state is located in ReachedStates, reverse image computation techniques are used to compute the actual sequence of input vectors which lead to the excitation state.

After the justification sequence has been computed, it is fault simulated to assure that the fault-free justification sequence actually leads to the excitation state in the faulty machine. If it does not, then some corrupted transition must have been encountered during application of the fault-free justification sequence. The state from which the first corrupted transition is encountered is taken as the new excitation state, and the part of the justification sequence which leads to this state is the new justification sequence (called MinJustSequence in Figure 2.1).

If the justification sequence, followed by the excitation vector, propagates the effect of the fault to a primary output, then the fault has been tested. Otherwise, the next step is to find a fault-free propagation sequence that does propagate the effect to the primary outputs. Note that if the excitation vector does not propagate the fault effect directly to the primary outputs, then it must propagate it the next state lines, i.e. the excitation vector produces one state in the good machine, and a different state in the faulty machine. Hence, the propagation sequence is simply a sequence which differentiates between these two states.

Random propagation is tried first, since it is much less expensive than deterministic propagation. This procedure utilizes the same single fault/parallel pattern fault simulator that is used during random test generation.

If random propagation does not find a propagation sequence, then deterministic propagation is performed, as outlined in Figure 2.4. Like justification, deterministic propagation is performed using the fault-free machine. The propagation procedure first checks whether or not the current good and faulty states have been previously differentiated. It they have, then the fault-free propagation sequence is looked up in a table of previously-generated results. If they have not been previously differentiated, then the deterministic propagation process begins. Like justification, this process is based on STG traversal, this time in the fault-free product machine. The initial state in the traversal is the product state composed of the good and faulty states. Implicit STG traversal is used to explore every product state reachable from this initial state, as outlined in Figure 2.5. After each image computation, the output of the product machine is checked. If the output of the good and faulty machines differ for any reached state, the good and faulty states have been differentiated, and reverse image computations are performed to obtain the distinguishing sequence. If, at some point, the states reached after an image computation are contained in the total set of reached states, i.e. if a fixed point is reached, then the entire product machine has been traversed. If the good and faulty machine outputs have been equal during the entire traversal, then the good and faulty states cannot be differentiated in the fault-free machine.

Figure 2.4: Deterministic Fault-Free Propagation Algorithm

```
deterministic-fault-free-propagate
{
    if (GoodState and FaultyState have already been differentiated) {
        reuse-propagation-sequence;
        return;
    }
    InitialState = build-product-machine-init-state(GoodState, FaultyState);
    product-machine-traversal(InitialState);
    return;
}
```

Figure 2.5: Product Machine Traversal Algorithm

```
product-machine-traversal(StartState)
{
    TotalSet = $\emptyset$;
    ProductReachedStates[0] = StartState;
    for (i = 0 to $\infty$) {
        if (good-and-faulty-outputs-not-equal(ProductReachedStates)) {
            ReachablePropagationCube = get-cube-with-different-outputs;
            break;
        } else {
            if (ProductReachedStates[i] $\subseteq$ TotalSet) {
                /* no differentiation sequence exists */
                return;
            }
            TotalSet = TotalSet $\cup$ ProductReachedSets[i];
            ProductReachedSets[i+1] = implicit-compute-next-states(ProductReachedSets[i]);
        }
    }
    for (j = i to 1) {
        (InputVector, CurrentState) = get-one-minterm(ReachablePropagationCube);
        PropagationCube = implicit-compute-reverse-image(CurrentState);
        ReachablePropagationCube = PropagationCube $\cap$ ProductReachedStates[i-1];
    }
    (InputVector, CurrentState) = get-one-minterm(ReachablePropagationCube);
    return;
}
```

If a propagation sequence is found during deterministic propagation, it must be fault simulated to determine whether or not it works when the fault is present. If it does work in the presence of the fault, then the fault has been tested. Otherwise, the 3-step algorithm has failed to find a test for the fault, and yet has not proved the fault redundant, since propagation was not attempted using the faulty machine. Therefore, the fault is abandoned for the present time; it will be tested or proven redundant later during good/faulty product machine traversal.

Good-Faulty Product Machine Traversal

If the fault has not been tested or proven redundant during the 3-step algorithm, the algorithm resorts to good-faulty product machine traversal, as outlined in Figure 2.6. The first step is to insert the fault into the product machine, thus creating the good-faulty product machine out of the good/good product machine. Next, this good-faulty product machine is traversed. If a differentiating sequence is found during this traversal, then this sequence is a test for the fault. Otherwise, the good and faulty circuit are equivalent, and thus the fault is redundant. We shall call faults that are proven redundant during good/ faulty product machine traversal non-distinguishable faults (ND faults) [5]. Such faults can be excited (if they could not be excited, they would have been proven redundant during combinational test generation), but the effect of this excitation cannot be propagated to the primary outputs.

Fault Simulation

Throughout the test generation algorithm, the deterministically-generated test sequences are fault simulated on all untested faults in the hopes of testing some of the untested faults. Rather than fault simulating each time a new test sequence is generated, the algorithm waits until several test sequences have been generated, so that these sequences can be simulated in parallel, using a single fault/parallel pattern simulator. The number of sequences simulated at one time is user-defined. For circuits in which simulation is very fast compared to deterministic test generation, this number should be chosen small; for circuits in which simulation is relatively slow compared to deterministic test generation, this parameter should be large.

Figure 2.6: Good/Faulty Product Machine Traversal Algorithm

```
good-faulty-product-machine-traversal;
{
    insert-fault-into-product-machine;
    recompute-bdds;
    product-machine-traversal(ProductResetState);
    remove-fault-from-product-machine;
}
```

2.4 Results

The test generation algorithm described in the previous section has been implemented in SIS, the UC Berkeley sequential synthesis tool [18], using the Long BDD package [12]. This section presents the results of this algorithm to generate tests for circuits in the ISCAS '89 sequential benchmark set [2].

Table 2.1 gives statistics about each circuit tested, including the number of primary inputs and outputs, the number of latches and gates, the number of faults (after fault collapsing), the STG depth, and the number of reachable states. Because the ISCAS circuits do not have specified reset states, a reset state of all zeros was assumed for each circuit.

Table 2.1: Circuit Statistics

Circuit	Inputs	Outputs	Latches	Gates	Faults	Depth	States
s208	11	2	8	96	177	17	17
s298	3	6	14	119	275	19	218
s344	9	11	15	160	267	7	2625
s349	9	11	15	161	272	7	2625
s382	3	6	21	158	346	151	8865
s386	7	7	6	159	321	8	17
s400	3	6	21	162	367	151	8865
s420	19	2	16	196	356	17	17
s444	3	6	21	181	407	151	8865
s510	19	7	6	211	466	47	47
s526	3	6	21	193	492	151	8868
s526n	3	6	21	194	491	151	8868
s641	35	24	19	379	406	7	1544
s713	35	23	19	393	508	7	1544
s820	18	19	5	289	773	11	25
s832	18	19	5	287	789	11	25
s838	35	2	32	390	711	17	17
s953	16	23	29	395	893	11	504
s1196	14	14	18	529	1044	3	2616
s1238	14	14	18	508	1118	3	2616
s1488	8	19	6	653	1202	22	48
s1494	8	19	6	647	1218	22	48

Table 2.2 shows the results of test generation. The first three columns list the total number of faults, the number of faults for which a test was generated (Tested), and the number of faults that were proven *redundant* (Redund). In the next two columns, the redundant faults are broken up into the two types of redundant faults, sequentially non-excitable faults (SNE Red) and non-distinguishable faults (ND Red). The next column shows the number of good/faulty product machine traversals performed (PMT) during test generation. Test fault coverage (TFC) and test generation time are shown in the last two columns. All times in this paper were obtained on a DEC 5000/125. Because of the equivalence verification phase of the algorithm, fault coverage is 100% in all circuits for which tests could be generated.

Table 2.2: SIS ATPG Results

Circuit	Faults	Tested	Redund	SNE Red	ND Red	PMT	TFC (%)	Time (s)
s208	177	116	61	56	5	6	100	7
s298	275	240	35	35	0	0	100	13
s344	267	261	6	6	0	0	100	13
s349	272	264	8	8	0	0	100	14
s382	346	326	20	20	0	0	100	519
s386	321	253	68	68	0	0	100	7
s400	367	340	27	27	0	0	100	544
s420	356	144	212	207	5	6	100	20
s444	407	372	35	35	0	17	100	540
s510	466	466	0	0	0	0	100	6
s526	492	402	90	90	0	0	100	688
s526n	491	403	88	88	0	0	100	681
s641	406	346	60	60	0	0	100	30
s713	508	407	101	101	0	0	100	38
s820	773	739	34	34	0	0	100	38
s832	789	739	50	50	0	0	100	40
s838	711	195	516	511	5	6	100	73
s953	893	883	10	10	0	0	100	42
s1196	1044	1041	3	0	3	18	100	151
s1238	1118	1049	69	66	3	23	100	246
s1488	1202	1162	40	40	0	0	100	112
s1494	1218	1167	51	51	0	0	100	130

Table 2.3 compares the speed of SIS test generation with that of STEED [8] and VERITAS [5]. Runtimes that appear in brackets signify that the test algorithm did not achieve 100% test fault coverage on the circuit; in this case, the test fault coverage follows the runtime. For instance, on circuit s208, STEED runtime was 4 seconds, and 97.0% of the faults were covered by this set or proven redundant. If the runtime is unbracketed, then the algorithm achieved 100% test fault coverage.

Table 2.3: Comparison with STEED and VERITAS

Circuit	STEED* (s)	VERITAS* (s)	SIS (s)	$\frac{\text{SIS}}{\text{VERITAS}}$
s208	{ 4 / 97.0% }	6	7	1.2
s298	{ 6 / 99.0% }	5	13	2.6
s344	5	5	13	2.6
s349	6	5	14	2.8
s382	{ 1320 / 95.2% }	244	519	2.1
s386	4	4	7	1.8
s400	{ 1200 / 95.8% }	244	544	2.2
s420	{ 440 / 91.2% }	56	20	0.4
s444	{ 1992 / 95.6% }	190	540	2.8
s510	{ 7 / 99.8% }	9	6	0.7
s526	{ 1060 / 91.0% }	338	688	2.0
s526n	{ 1040 / 91.0% }	428	681	1.6
s641	{ 10200 / 93.1% }	19	30	1.6
s713	{ 10440 / 93.1% }	26	38	1.5
s820	120	50	38	0.8
s832	{ 120 / 99.7% }	61	40	0.7
s838	{1500 / 80.5% }	425	73	0.2
s953	29	50	42	0.8
s1196	{ 4080 / 98.7% }	51	151	3.0
s1238	{ 3600 / 99.0% }	65	246	3.8
s1488	133	105	112	1.1
s1494	147	129	130	1.0
TOTAL	624 min.	42 min.	66 min.	1.6

* STEED times were divided by 3, and VERITAS times were multiplied by 1.25, to account for computer speed differences. (VERITAS test generation was performed on a DEC5000/200; STEED was run on a VAX-11/8800.)

Because both SIS and VERITAS can perform good/faulty product machine traversal, both algorithms obtain 100% fault coverage on all circuits for which they can generate tests. In contrast, STEED, which cannot perform product machine traversal, does not obtain 100% fault coverage on all circuits. Further, because SIS and VERITAS use implicit traversal techniques, they are much faster than STEED. Overall, SIS is 1.6 times slower than VERITAS, but SIS does outperform VERITAS on some circuits. The remainder of this section discusses why SIS and VERITAS perform differently on different circuits and makes suggestions for improving the running time of SIS. The following table, which shows the relative amounts of time spent in each part of the SIS test generation algorithm, will be useful during this discussion.

Table 2.4: SIS Runtime Profile

Build BDDs & Setup	RTG	STG Traversal	SAT Formula Setup	SAT Solve	Just.	Random Prop.	Det. Prop.	Good/ Faulty PMT	Fault Sim.
11%	14%	9%	12%	4%	15%	1%	13%	10%	11%

The circuits s208, s420, and s838 shed some light on one of the advantages of the SIS system. These three circuits are very similar; each is a digital fractional multiplier, and they each have the same STG depth and number of reachable states. Further, in each circuit, only 5 of the redundant faults are ND faults, that is, almost all the redundant faults can be detected by the combinational test generator. The major difference between the circuits from a testing perspective is in the number of SNE faults; s208 has 56 SNE faults, s420 has 207 SNE faults, and s838 has 511 SNE faults. As shown in Table 2.3, SIS and VERITAS take about the same amount of time to test s208, but SIS runs 3 times faster on s420, and is 6 times faster on s838. This leads us to conclude that the satisfiability-based combinational test generator in SIS more efficiently identifies SNE redundancies than the topology-based generator of VERITAS.

This conclusion also explains the success of SIS in testing s820 and s832. These circuits have 34 and 50 SNE faults, respectively, and by our hypothesis, SIS identifies them much more quickly than does VERITAS.

However, contrary to our hypothesis, there are many circuits with many SNE faults for which VERITAS matches or outperforms SIS. In each of these cases, there is at least one other part of the test generation process which we believe overwhelms the SNE redundancy identification time.

First, in s1488 and s1494, fault simulation accounts for almost 50% of test generation time. VERITAS has a more sophisticated fault simulator than does SIS; it uses compiled simulation to simulate the good circuit, and single fault/parallel pattern simulation to simulate faulty circuits. This explains the fact that SIS and VERITAS perform about equally on these circuits; although SIS more quickly identifies the SNE faults, VERITAS fault simulates more quickly. Presumably, then, if the SIS fault simulator were improved to match that of VERITAS, SIS would outperform VERITAS on these two circuits.

Secondly, there is another large group of circuits with SNE faults--s382, s400, s444, s526, s526n, s1196, and s1238--for which product machine traversals take 45-70% of test generation time. This is another area in which VERITAS has a more sophisticated implementation. VERITAS uses constrained product machine traversal, which takes advantage of the fact that the good and faulty machines are usually very similar. Rather than building the entire product machine, it builds a constrained product machine, which does not include the parts of the faulty machine that cannot be affected by the fault. This speeds the product machine traversal in many cases.

Another reason that VERITAS spends less time than SIS on product machine traversal is that VERITAS has a more sophisticated RTG procedure than SIS. (The VERITAS RTG procedure is described in Section 2.2.) This allows VERITAS to randomly test many more faults than does SIS; VERITAS tests 84% of the ISCAS circuit faults using RTG, while SIS tests just 55% of faults using RTG. This means that VERITAS performs fewer deterministic test generations than does SIS, i.e. it does not need to perform as many combinational test generations, justifications, propagations, and good/faulty product machine traversals as does SIS.

One more possible SIS modification should be mentioned. The SAT formula setup time is very large right now--12% of test generation time on the ISCAS benchmarks. This time includes the time to build the part of the boolean formula that restricts the combinational solutions to be reachable. In the current implementation, this part of the formula is rebuilt for each fault. However, since the set of reachable states is constant throughout the algorithm, this part of the formula should be built just once, and then reused for each fault. This would reduce the SAT formula setup time.

2.5 Conclusions

The SIS sequential test generator has been described and shown to be competitive with the current state-of-the-art sequential test generator VERITAS. Although SIS is 1.6 times slower than VERITAS overall on the ISCAS '89 benchmark circuits, it does outperform VERITAS on many circuits. By combining the best of both algorithms, it should be possible to create a test generator that slightly outperforms both SIS and VERITAS. This hybrid test generator would include the random test generator, fault simulator, and constrained product machine traversal of VERITAS, and the combinational satisfiability-based test generator of SIS. Based on the percentage of its runtime that VERITAS spends on combinational test generation, we estimate that a VERITAS/SIS hybrid test generator would run 15-25% faster than VERITAS.

The real problem, though, is that this hybrid algorithm would only outperform both algorithms on the problems that both can already solve; the hybrid would be no more able to solve larger problems than either SIS or VERITAS or STEED. Thus, rather than concentrating time and effort on the problem of incrementally improving SIS or VERITAS, future work should examine the problem of how to generate tests for larger circuits, circuits on which SIS and VERITAS cannot perform test generation at all. This topic is addressed further in Section 5.

3 Generating Small Test Sets

In this section, the problem of generating small test sets for sequential circuits is addressed. Section 3.1 is an introduction and discussion of previous work in this area. Section 3.2 discusses the exact formulation of the problem, and Section 3.3 describes the heuristics which have been implemented. In Section 3.4, the results of implementing these heuristics are presented, and finally, in Section 3.5, conclusions are drawn.

3.1 Introduction and Previous Work

The problem of generating small test sets for combinational circuits has been well-addressed. However, the more difficult task of reducing test set size for sequential circuits has not been examined as extensively. This is a difficult problem to formulate exactly, and nothing has been published on the exact formulation or solution of this problem. Instead, existing research on the test set size problem has concentrated on the use of heuristics to reduce test set size as much as possible. These heuristics, described in the following paragraphs, essentially attempt to maximize the overlap between test sequences so that many tests can be applied simultaneously.

Fault simulation is one obvious, easily implemented, and frequently used method of reducing both test set size and test generation time. After each test is generated, it is simulated on all the untested faults, in the hope that it will also test other faults. If the test sequence does test other faults, then not only does the test generator not have to spend time generating a test for these other faults, but also the test set contains a single sequence that covers each of these faults, rather than one test sequence for each fault.

"Test sequence appending" is another way to increase the overlap of sequential tests. Most sequential ATPG systems generate every test from either the reset state or an unknown state. However, it is also possible to generate a test for a fault by appending test vectors onto an already generated test sequence. If the number of appended vectors is much smaller than the number of vectors required to generate a test from the reset state or an unknown state, the size of the test set may be reduced. Ono et. al. have shown that this technique can be used to reduce test set size, especially for circuits with no reset state [15].

The previous two techniques are used during test generation to reduce the size of the test set as it is being generated; there are also several heuristics that can be used after test generation to reduce the size of the test set.

Compaction is one such technique. Compaction methods take advantage of the fact that ATPG often specifies only a subset of the input values in a test sequence; the remaining unspecified inputs can be assigned arbitrarily. Given a set of incompletely specified test sequences, compaction algorithms attempt to set the unspecified input values such that many tests can be applied simultaneously, i.e. overlapped. If two or more tests can be overlapped, then the resulting test set is smaller than the original, in which each test sequence was applied separately. Compaction techniques are commonly used on combinational circuits; Niermann, et. al. [14] have extended these techniques for use on sequential test sets. Of course, the problem of sequential compaction is much more difficult than that of combinational compaction, since sequences can overlap in many different ways.

Reverse fault simulation is another post-processing test set reduction technique. In this technique, after all tests have been generated, the tests are simply fault simulated in the reverse order from that in which they were generated, in hopes of identifying some redundant tests. These redundant tests can then be eliminated from the test set. Cho, et. al have shown this to be very effective on the ISCAS '89 benchmark circuits [5].

3.2 The Exact Problem

The previous work described above demonstrates that heuristics are effective in reducing the size of sequential test sets. However, because the problem has not been solved exactly, it is impossible to know how close the resulting solutions are to the exact solution. We have examined the exact problem, and have formulated it as a modified shortest common superstring problem. However, the problem as formulated is NP-complete, and has no known efficient approximation algorithms, and thus we could not use this formulation to develop an efficient solution or approximation for the exact problem.

Our formulation is based upon the shortest common superstring problem [7], an NP-complete problem:

Shortest common superstring problem:

INSTANCE: A finite alphabet A and a finite set R of strings from A^*

QUESTION: What is the shortest string w in A^* such that each string x in R is a substring of w, i.e. $w = w_0 x w_1$ where each w_j is in A^*?

The following modified problem can be used to solve the smallest test set problem:

Modified shortest common superstring problem:

INSTANCE: A finite alphabet A and a finite set T of finite sets of strings from A^*.

QUESTION: What is the shortest string w in A^* such that at least one string from each set T_i in T is a substring of w?

The above problem can be used to solve the smallest test set problem for a circuit by defining A and T as follows: Let V be the set of all possible input vectors to the circuit and S be the set of all reachable states of the circuit. Define A to be the finite alphabet with elements $a_i = v_i s_i$. Let F be the set of all possible faults in the circuit. Let T_{fi} be the set of tests for the i^{th} fault f_i. Note that elements of T_{fi} are strings from A^*. Finally, define T to be the set of all T_{fi}, over all f_i in F.

3.3 Heuristics and Small Set Algorithm

Although we could not solve the problem of generating the minimum-sized test set for a circuit, we were still interested in evaluating the effectiveness of different combinations of heuristics. To do this, we added five test set size reducing heuristics to the SIS test generation algorithm, and evaluated the efficacy of various combinations of these heuristics. Three of the heuristics--fault simulation, test sequence appending, and reverse fault simulation--have already been described in Section 3.1. In addition to these, we considered two other heuristics.

The first is a "short individual test sequence" heuristic. The three aforementioned heuristics all attempt to overlap test sequences, but they cannot do anything about individual test sequences that are unnecessarily long. Hence, we thought that it might be effective to simply try to generate very short test sequences for individual faults, in the hope that the union of these short test sequences would be a small test set. This goal can be accomplished by using good/faulty product machine traversal to generate tests, since this method is guaranteed to generate minimum-length test sequences. To decrease the run time when this heuristic is used, SNE faults are identified during a preprocessing step, using the combinational test generator.

Finally, we examined the effect of using no random test generation and propagation. The reason for trying this approach is similar to the reason for trying the short individual test sequence heuristic. Random tests in general can be much longer than deterministically-generated tests, and thus it is possible that the use of random tests could cause test sets to be unnecessarily large.

Figure 3.1 shows the effect of using each of the five heuristics on a small example. Figure 3.1(a) lists the three faults of the example, along with the three test sequences that exist for each fault. Test sequences are represented as strings of vectors; for instance, test sequence V_4V_1 is a test of length two, consisting of input vector V_4 followed by input vector V_1. Each test sequence must be applied from the reset state of the machine. Assume that the first test sequence listed for each fault is the test obtained by the 3-step algorithm, the second test listed is another test for the fault that was not obtained by the ATPG algorithm, and the third test listed is the one obtained by the good/faulty product machine traversal algorithm. Assume that there are no other tests for these faults.

Figure 3.1(b) shows the two test sequences generated by the random test generator.

Figure 3.1(c) shows the test sets obtained with and without each test-set reducing heuristics, assuming that random test generation occurs before deterministic test generation, and that the faults are considered in the order Fault1, Fault2, Fault3. For example, when no heuristics are used, the test set contains the three sequences $V_1V_4V_5V_1V_2$, $V_4V_1V_2$, and $V_2V_3V_4V_1V_2$. The first test sequence, generated during random test generation, is a test for Fault1. The second and third sequences are generated for Fault2 and Fault3 respectively using the 3-step test generation algorithm. Also shown in Figure 3.1(c) is the best solution obtainable through using a combination of the five heuristics, and the exact solution.

Figure 3.1: Effect of Heuristics on an Example

Figure 3.1(a): Faults and corresponding test sequences

	Tests for Fault1	Tests for Fault2	Tests for *Fault3*
Test generated by 3-step algorithm	$V_1V_4V_5$	$V_4V_1V_2$	$V_2V_3V_4V_1V_2$
Another test for the fault	$V_4V_1V_3$	$V_1V_4V_5V_1V_2V_2$	V_5V_5
Test generated by good/faulty PMT	V_2	$V_2V_3V_3$	V_4V_1

Figure 3.1(b): Sequences generated by random test generator

Random Sequence 1	$V_1V_4V_5V_1V_2$
Random Sequence 2	$V_6V_3V_1V_3V_2$

Figure 3.1(c): Test Sets Generated

Heuristic(s) Used	Test Set Obtained	Number of vectors in Test Set
None	$V_1V_4V_5V_1V_2$ $V_4V_1V_2$ $V_2V_3V_4V_1V_2$	13
Fault simulation	$V_1V_4V_5V_1V_2$ $V_4V_1V_2$	8
Test-sequence appending	$V_1V_4V_5V_1V_2V_2$ $V_2V_3V_4V_1V_2$	11
Reverse fault simulation	$V_2V_3V_4V_1V_2$ $V_4V_1V_2$	8
Short individual test sequences	$V_1V_4V_5V_1V_2$ $V_2V_3V_3$ V_4V_1	10
No RTG	$V_1V_4V_5$ $V_4V_1V_2$ $V_2V_3V_4V_1V_2$	11
Reverse fault simulation, Short individual test sequences, No RTG	V_4V_1 $V_2V_3V_3$	5
EXACT SOLUTION	$V_4V_1V_2$ V_2	4

3.4 Results

The five heuristics described in Section 3.3 have been implemented in SIS. After implementation, experiments were run to determine which combinations of heuristics led to the smallest test sets for the ISCAS '89 circuits. These experiments were performed in two stages. In the first stage, every combination of the five heuristics was run on a small but representative set of the ISCAS circuits, with the goal of eliminating those combinations that gave very large test sets. The 8 combinations that gave the smallest test sets were then tested on the complete set of circuits.

Figure 3.2 shows the abbreviation used for each heuristic in tables throughout this section.

The results of the first stage of testing are shown in Table 3.1. In this stage, only 8 of the ISCAS '89 circuits were tested: s208, s298, s344, s386, s510, s641, s820, and s1488. The first five columns show the combination of heuristics used; an X in a heuristic column signifies that the heuristic was used. The next three columns show the total number of test sequences, total number of test vectors, and total test generation time for all 8 circuits. The final column (Best) shows the number of circuits for which the combination of heuristics produced the best solution, i.e. that with the smallest test set, among all combinations. (For two circuits, more than one heuristic combination gave the best solution, which is why this column totals to more than eight.) The rows of heuristic combinations are sorted according to the number of test vectors produced.

Figure 3.2: Abbreviations for Heuristics

Heuristic	Abbreviation
Fault simulation	FS
Test sequence appending	TSA
Reverse fault simulation	RFS
Short individual test sequences	SITS
No RTG	NRTG

Table 3.1: Results of applying combinations of heuristics to 8 circuits in the ISCAS '89 benchmark set

FS	TSA	RFS	SITS	NRTG	Overall number of sequences	Overall number of test vectors	Overall time (s)	Best
X	X	X		X	131	2,348	1,232	5
X	X			X	212	2,895	1,130	
X	X	X			169	3,009	724	1
X	X	X	X	X	225	3,028	1,957	2
X		X		X	365	3,559	902	1
X	X	X	X		196	3,601	1,398	
X		X			295	3,754	511	1
X	X		X	X	375	3,815	1,759	
X		X	X	X	495	4,045	3,252	
X		X	X		332	4,067	1,614	1
		X	X		405	4,813	4,116	
		X	X	X	571	4,824	8,308	
		X			390	4,826	822	
		X		X	524	4,853	1,695	
X	X				318	4,981	597	
X				X	575	5,388	733	
	X	X			278	5,561	1,506	
X	X		X		327	5,622	1,215	
	X	X	X		258	6,351	3,529	
	X	X		X	323	6,577	4,258	
X					514	6,674	331	
X			X	X	894	6,808	3,019	
X			X		546	6,851	1,353	
	X	X	X	X	306	6,910	5,180	
	X				731	11,197	954	
	X		X		771	13,888	2,988	
	X			X	1,289	14,121	3,211	
	X		X	X	1,766	16,602	4,556	
			X		1,456	17,246	3,567	
					1,456	18,254	356	
			X	X	3,583	25,116	7,718	
				X	3,583	29,863	1,267	

Table 3.1 shows that no one combination of heuristics works best on every circuit. However, the goal in this first stage of testing was simply to identify a subset of heuristic combinations to use in the second stage of testing. We chose the first 10 combinations from Table 3.1, since all combinations after these 10 give overall test sets more than twice as large as the smallest overall test set. In addition, the best heuristic combination for every circuit tested in this stage is included among these 10 combinations. After choosing these 10 combinations, we eliminated the 2 combinations which did not use reverse fault simulation, since reverse fault simulation can only reduce test set size. (Reverse fault simulation is a post-processing step which removes redundant test sequences from the test set, and thus a combination which includes reverse fault simulation cannot do worse than the same combination without reverse fault simulation.) The remaining eight combinations were then used to generate tests for a larger set of circuits. Table 3.2 shows the results of applying these 8 heuristic combinations to the complete ISCAS benchmark set. For each circuit, each column shows the result of using one of the 8 heuristic combinations. Each table entry contains first the test set size, followed by the test generation time. For each circuit, the best result, i.e. the smallest test set size obtained, is shown in bold type.

Table 3.2: Small Test Set Results (Test Set Size/ Test Generation Time) for Eight Best Combinations of Heuristics (Best result for each circuit is shown in bold type.)

	FS	FS	FS	FS	FS	FS	FS	FS
	TSA	TSA	TSA		TSA			
Circuit	RFS	RFS	RFS	RFS	RFS	RFS	RFS	RFS
			SITS		SITS		SITS	SITS
	NRTG		NRTG	NRTG			NRTG	
s208	187 / 15	203 / 13	**155 / 15**	208 / 10	160 / 13	213 / 9	168 / 16	194 / 14
s298	220 / 41	204 / 19	273 / 68	**167 / 22**	225 / 45	193 / 15	176 / 69	202 / 44
s344	98 / 290	**91 / 16**	93 / 271	104 / 295	97 / 70	**91 / 15**	127 / 792	**91 / 74**
s349	99 / 293	**91 / 17**	102 / 323	104 / 301	97 / 70	**91 / 16**	131 / 825	**91 / 77**
s382	**918 / 524**	1146 / 552	924 / 1879	1212 / 604	925 / 1430	1214 / 596	1247 / 1980	1247 / 1944
s386	**156 / 12**	167 / 10	161 / 17	198 / 10	193 / 14	219 / 9	213 / 21	233 / 17
s400	**923 / 556**	1041 / 549	973 / 1943	1211 / 641	939 / 1567	1213 / 583	1248 / 2034	1248 / 1931
s420	186 / 40	238 / 34	**173 / 52**	188 / 30	228 / 44	240 / 28	199 / 59	245 / 49
s444	887 / 392	875 / 406	910 / 1074	1505 / 687	**732 / 973**	1510 / 709	1205 / 1335	1205 / 1211
s510	**237 / 27**	522 / 20	**237 / 91**	682 / 30	527 / 53	682 / 20	900 / 482	758 / 163
s526	2125 / 792	1976 / 752	2131 / 1713	2003 / 739	1968 / 1550	1941 / 752	**1757 / 1675**	**1757 / 2114**
s526n	2125 / 753	1976 / 736	2131 / 1682	2003 / 688	1968 / 1503	1941 / 745	**1757 / 1640**	**1757 / 1623**
s641	**202 / 174**	209 / 64	260 / 344	269 / 78	259 / 178	231 / 40	363 / 491	274 / 180
s713	**174 / 171**	209 / 77	258 / 315	258 / 91	240 / 165	231 / 50	321 / 438	276 / 173
s820	**496 / 166**	653 / 110	539 / 289	743 / 96	683 / 318	797 / 90	788 / 477	858 / 428
s832	**498 / 172**	638 / 109	580 / 314	727 / 98	718 / 312	811 / 90	768 / 484	840 / 445
s838	**203 / 138**	214 / 126	204 / 218	223 / 118	228 / 163	268 / 109	223 / 246	249 / 184
s953	367 / 412	**339 / 276**	438 / 1732	714 / 393	415 / 1435	675 / 249	812 / 2561	745 / 2331
s1196	407 / 767	418 / 413	**355 / 960**	437 / 604	395 / 494	445 / 328	393 / 1049	434 / 516
s1238	422 / 930	443 / 537	**356 / 1060**	459 / 734	413 / 626	462 / 441	411 / 1095	456 / 610
s1488	**752 / 495**	960 / 474	1310 / 807	1188 / 354	1457 / 684	1328 / 320	1310 / 789	1457 / 665
s1494	**780 / 445**	1044 / 427	1279 / 762	1174 / 349	1426 / 638	1250 / 311	1279 / 740	1426 / 625
Total	12462 / 7605	13657 / 5737	13672 / 15859	15777 / 6972	14293 / 12345	16046 / 5525	15796 / 19298	16043 / 15418

The results of Table 3.2 again show that no one combination of heuristics is best for every circuit. Thus, given unlimited test generation time, it would be best to generate tests 8 different times, using each of the heuristic combinations shown in Table 3.2, and then to pick the smallest test set of the 8 produced. Given a time constraint, the first combination of heuristics--fault simulation, test set appending, reverse fault simulation, and no random test generation--should be used, since it gives the best solution for half of the circuits, and is within 10-30% of the best solution on all other circuits. In future discussion, we shall call the version of our ATPG algorithm which uses this first combination of heuristics "test set reducing SIS" (TSR-SIS).

Table 3.3 compares the general SIS ATPG algorithm of Section 2 (SIS) with TSR-SIS. For each circuit tested, the table shows the number of vectors produced by the general SIS ATPG algorithm (SIS Vectors), the number of vectors produced by TSR-SIS (TSR-SIS Vectors), and the ratio of these two values. In the final three columns, the table shows the time taken by the two algorithms, and the ratio of the two times.

Overall, TSR-SIS produces test sets that are 3.1 times smaller than those produced by SIS, and it takes 1.9 times as much time as SIS. This is not a large time penalty, considering the amount of chip testing time that will be saved because of the much-reduced test set size.

Table 3.3: Comparison to general SIS ATPG algorithm

Circuit	SIS Vectors	TSR-SIS Vectors	SIS Vectors TSR-SIS Vectors	SIS Runtime	TSR-SIS Runtime	TSR-SIS Time SIS Time
s208	328	187	1.8	7	15	2.1
s298	398	220	1.8	13	41	3.2
s344	147	98	1.5	13	290	22.3
s349	140	99	1.4	14	293	20.9
s382	2072	918	2.3	519	524	1.0
s386	336	156	2.2	7	12	1.7
s400	2100	923	2.3	544	556	1.0
s420	235	186	1.3	20	40	2.0
s444	1961	887	2.2	540	392	0.7
s510	1190	237	5.0	6	27	4.5
s526	3607	2125	1.7	688	792	1.2
s526n	3607	2125	1.7	681	753	1.1
s641	827	202	4.1	30	174	5.8
s713	847	174	4.9	38	171	4.5
s820	1896	496	3.8	38	166	4.4
s832	1873	498	3.8	40	172	4.3
s838	271	203	1.3	73	138	1.9
s953	1832	367	5.0	42	412	9.8
s1196	5253	407	12.9	151	767	5.1
s1238	5179	422	12.3	246	930	3.8
s1488	2517	752	3.3	112	495	4.4
s1494	2619	780	3.4	130	445	3.4
Total/ Average	39235	12462	3.1	66 min	127 min.	1.9

Table 3.4 compares the best test set sizes obtained by TSR-SIS with the test set sizes obtained by STEED and VERITAS. The first three columns show the number of vectors produced by each algorithm for each circuit. Again, a value in brackets indicates that the algorithm did not achieve 100% fault coverage, and in such cases, the fault coverage is listed.

Table 3.4: Test Set Size Comparison with STEED and VERITAS

Circuit	STEED	VERITAS	TSR-SIS	STEED / TSR-SIS	VERITAS / TSR-SIS
s208	{ 195 / 97.0% }	192	187	1.0	1.0
s298	{ 280 / 99.0% }	119	220	1.3	0.5
s344	125	48	98	1.3	0.5
s349	120	56	99	1.2	0.6
s382	{ 1633 / 95.2% }	1028	918	1.8	1.1
s386	238	168	156	1.5	1.1
s400	{ 409 / 95.8% }	1091	923	0.4	1.2
s420	{ 808 / 91.2% }	187	186	4.3	1.0
s444	{ 994 / 95.6% }	1026	887	1.1	1.2
s510	{ 733 / 99.8% }	584	237	3.1	2.5
s526	{ 2037 / 91.0% }	1457	2125	1.0	0.7
s526n	{ 2287 / 91.0% }	1528	2125	1.1	0.7
s641	{ 327 / 93.1% }	134	202	1.6	0.7
s713	{ 315 / 93.1% }	139	174	1.8	0.8
s820	1304	785	496	2.6	1.6
s832	{ 1344 / 99.7% }	763	498	2.7	1.5
s838	{290 / 80.5% }	193	203	1.4	1.0
s953	1050	578	367	2.9	1.6
s1196	{ 545 / 98.7% }	376	407	1.3	0.9
s1238	{ 576 / 99.0% }	389	422	1.4	0.9
s1488	1310	1031	752	1.7	1.4
s1494	1374	1040	780	1.8	1.3
TOTAL	18294	12912	12462	1.5	1.0

Overall, TSR-SIS produces test sets that are 1.5 times smaller than those produced by STEED, despite the fact that STEED's test set does not obtain 100% fault coverage. This is because the only test set *reducing* heuristic used by STEED is fault simulation. On the other hand, TSR-SIS and VERITAS produce roughly the same overall test set size. This makes sense since VERITAS uses fault simulation, reverse fault simulation, and a heuristic that is similar to test set appending; this is almost the same combination of heuristics used by TSR-SIS. The test set appending in VERITAS takes place during random test generation. As described in Section 2.2, VERITAS has a sophisticated random test generator which is used to generate tests for 84% of the faults in the ISCAS circuits. This random test generator records not only faults covered by RTG, but also faults that were excited by the random test vectors. The random test generator then tried to extend the random tests in ways that test the excited faults. In other words, the random test generator attempts to build up a test for the excited faults from the end of already determined tests, which is similar to test set appending.

3.5 Conclusions

Five test-set size reducing heuristics have been added to the SIS test generation algorithm. On the basis of empirical evidence, we found 8 combinations of heuristics which seem to give the smallest test sizes, and identified one combination of heuristics which works best overall. This combination is the one which includes fault simulation, test set appending, reverse fault simulation, and no random test generation. Through use of these heuristics, the test set sizes of the ISCAS circuits were reduced by a factor of 3.1, with a factor of 1.9 increase in test generation time.

4 Sequential Redundancy Removal

The test generation algorithm described in Section 2 does more than generate tests for faults; it can also identify redundant faults. A redundant fault corresponds to a redundancy in the circuit itself; if there is no test for a line stuck-at 0, then that line can be set to a constant 0 without changing the input/output behavior of the circuit. The process of setting redundant lines to constant values is called redundancy removal. Often redundancy removal leads to gate reduction, and thus area reduction, as illustrated in Figure 4.1.

Figure 4.1(a) shows a part of a circuit containing a redundant line as shown. This redundant line can be replaced by a constant 1. Note that this causes the OR gate A to output a constant 1 as well, and thus this gate can be removed, its output replaced by a constant 1. After this replacement, one input of the AND gate B is a constant 1, and thus this gate always outputs the value of its second input. Therefore, gate B can be removed, its output replaced by the non-constant input line. Figure 4.1(b) shows the circuit after all of these replacements are performed.

In the remainder of this section, the redundancy removal algorithm is described in detail, and results of applying redundancy removal to the ISCAS benchmarks are shown.

Figure 4.1: Results of Redundancy Removal

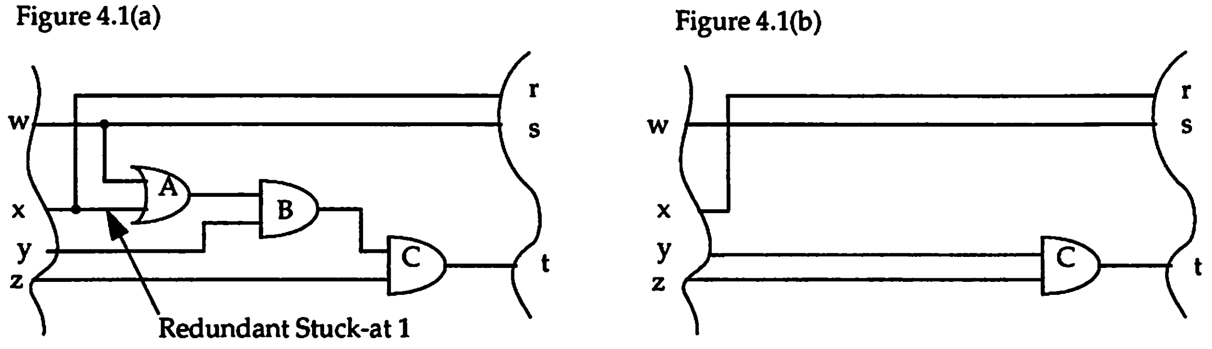
Figure 4.1(a)

Figure 4.1(b)

4.1 Redundancy Removal Algorithm

The redundancy removal algorithm, outlined in Figure 4.1 and 4.2, identifies and removes one redundant fault at a time. First, as in the ATPG algorithm, the output function BDD's are calculated, the circuit STG is traversed, and random test generation is performed. Next, the algorithm enters its main loop. Each time through the loop, one redundancy is identified and removed, until the circuit is irredundant.

The actual redundancy identification procedure, described in Figure 4.2, uses several methods to minimize the time spent making the circuit irredundant. First, each time the redundancy identification procedure is called, the algorithm looks first at unseen faults, since an unseen fault is more likely to be redundant than a fault that was testable during a previous loop. In addition, the redundancy identification procedure first identifies all SNE redundant faults, for two reasons. First, SNE faults can be identified more quickly than ND faults, since SNE faults can be identified using the combinational test generator, while ND faults can only be identified using product machine traversal. Secondly, removing an SNE redundant fault cannot change the output functions and reachable states of the circuit, and thus these do not have to be recalculated after an SNE fault is removed. Only after no more SNE faults exist does the algorithm use good/faulty product machine traversal to identify any remaining ND faults.

Going into a bit more detail, during the first stage of redundancy identification, the algorithm uses only the three-step test generation algorithm to find tests for previously unseen faults. Next, if all unseen faults are testable, the algorithm considers the remaining faults, those which have been tested during previous loops, but may now be redundant. First, all previously generated tests are fault simulated on these remaining faults. This step tests most of the faults. Next, the second stage of test-generation is entered. In this stage, the algorithm again uses the three-step test generation algorithm to test any untested faults and to identify SNE redundant faults. If no redundancies are found in this stage, then the final stage is entered. In this stage, good/faulty product machine traversal is used to test the remaining faults or to identify them as ND redundant faults.

Whenever an ND redundancy is removed, the circuit output functions and reachable states may change, and thus the output functions and reachable states are recalculated.

Figure 4.1: Redundancy Removal Algorithm

```
remove-redundancies
{
    build-bdds;
    traverse-STG;
    random-test-generation;
    do {
        RedundancyRemoved = detect-and-remove-redundancy;
        if (RedundancyRemoved and OutputFunctionsChanged) {
            build-bdds;
            traverse-STG;
        }
    } while (RedundancyRemoved)
}
```

Figure 4.2: Redundancy Detection Algorithm

```
detect-and-remove-redundancy
{
    for each never seen fault {
        three-step-atpg;
        if (Fault == REDUNDANT) {
            remove-redundancy;
            return TRUE;
        }
    }
    fault-simulate(AllTests, RemainingFaults);
    for each remaining untested fault {
        three-step-atpg;
        if (Fault == REDUNDANT) {
            remove-redundancy;
            return TRUE;
        }
    }
    for each remaining untested fault {
        good-faulty-product-machine-traversal;
        if (Fault == REDUNDANT) {
            remove-redundancy;
            OutputFunctionsChanged = TRUE;
            return TRUE;
        }
    }
}
```

4.2 Results

The redundancy removal algorithm described above has been implemented in SIS, using the UC Berkeley BDD package [18]. Tables 4.1 and 4.2 show the results of applying this algorithm to the ISCAS benchmarks. Table 4.1 shows the number of literals and gates in each circuit before and after redundancy removal and the time taken to make the circuit irredundant. Table 4.2 shows the number of latches before and after redundancy removal for those circuits in which the number of latches changed as a result of redundancy removal.

Table 4.1: Results of Redundancy Removal

Circuit	Literals Before	Literals After	After Before	Gates Before	Gates After	After Before	Time (s)
s208	166	79	0.48	96	41	0.43	9
s298	244	148	0.61	119	61	0.51	30
s344	269	205	0.76	160	108	0.68	103
s349	273	205	0.75	161	108	0.67	107
s382	306	225	0.74	158	97	0.61	1370
s386	347	196	0.56	159	86	0.54	38
s400	322	228	0.71	165	98	0.59	1557
s420	336	79	0.24	196	41	0.21	17
s444	352	244	0.69	181	106	0.59	2124
s510	424	392	0.92	211	179	0.85	28
s526	445	265	0.60	193	105	0.54	3461
s526n	445	265	0.60	194	105	0.54	3414
s641	539	229	0.42	379	111	0.29	79
s713	591	229	0.39	393	111	0.28	88
s820	757	669	0.88	289	241	0.83	70
s832	769	669	0.87	287	241	0.84	81
s838	670	76	0.11	390	38	0.10	52
s953	743	646	0.87	395	310	0.78	492
s1196	1009	864	0.86	529	387	0.73	747
s1238	1041	869	0.83	508	390	0.77	1187
s1488	1387	1218	0.88	653	528	0.81	189
s1494	1393	1218	0.87	647	528	0.82	233
Average			0.67			0.59	

Table 4.2: Results of Redundancy Removal

Circuit	Latches Before	Latches After
s208	8	5
s420	16	5
s641	19	15
s713	19	15
s838	32	5

Table 4.3 compares sequential redundancy removal with other area optimization techniques, including a modified sequential redundancy removal procedure, combinational redundancy removal, and sequential don't care minimization techniques [17]. The first column shows the number of literals and gates in the original circuit. The next four columns show the number of literals and gates in the circuits after four different optimization procedures, along with the time taken by each procedure. The first procedure is combinational redundancy removal (Comb. RR). The second procedure is the regular sequential redundancy removal procedure described in Section 4.1 (Seq. RR). The third procedure is "quick" sequential redundancy removal (Quick Seq. RR), a modified sequential redundancy removal in which only SNE redundancies are identified and removed. This quick procedure is generally faster than full sequential redundancy removal, since no justifications, propagations, or product machine traversals are performed, but it is not guaranteed to leave a circuit fully testable. The final optimization procedure is external don't care minimization (Exdc Min.). In this procedure, the unreachable states of the machine are used as external don't cares, and the circuit is then optimized using the full-simplify and script.rugged procedures of SIS [18].

The results of Table 4.3 show that combinational redundancy removal achieves the least reduction, but takes by far the least amount of time. Sequential redundancy removal reduces the literal count by almost twice as much as does combinational redundancy removal, but takes 10-1000 times more time. The quick sequential redundancy removal procedure achieves almost exactly the same reduction as does the full sequential redundancy removal procedure, which is to be expected, since most of the redundant faults in these circuits are SNE faults (see Table 2.2). Further, the quick procedure is much faster than the full sequential redundancy removal procedure; on average, the quick procedure uses ten times less time than the full procedure. The external don't care minimization procedure achieves twice as much reduction as does sequential redundancy removal, but on most circuits this procedure takes more time than does quick sequential redundancy removal. In particular, on the largest circuits, redundancy removal is much faster than don't care minimization.

Table 4.3: Comparison with Other Area Optimization Techniques

Circuit	Before (lit/gate)	Comb. RR (lit/gate/time)	Seq. RR (lit/gate/time)	Quick Seq. RR (lit/gate/time)	Exdc Min. (lit/gate/time)
s208	166 / 96	138 / 68 / 1	79 / 41 / 9	79 / 41 / 5	57 / 8 / 7
s298	244 / 119	200 / 75 / 1	148 / 61 / 30	148 / 61 / 9	108 / 24 / 20
s344	269 / 160	223 / 114 / 1	205 / 108 / 103	213 / 111 / 41	164 / 49 / 113
s349	273 / 161	223 / 114 / 1	205 / 108 / 107	217 / 111 / 38	164 / 49 / 109
s382	306 / 158	249 / 101 / 1	225 / 97 / 1370	225 / 97 / 152	147 / 32 / 77
s386	347 / 159	306 / 118 / 1	196 / 86 / 38	202 / 88 / 13	115 / 21 / 36
s400	322 / 165	252 / 102 / 2	228 / 98 / 1557	228 / 98 / 156	144 / 33 / 78
s420	336 / 196	275 / 135 / 2	79 / 41 / 17	79 / 41 / 12	57 / 8 / 12
s444	352 / 181	268 / 110 / 2	244 / 106 / 2124	244 / 106 / 54	141 / 34 / 66
s510	424 / 211	392 / 179 / 2	392 / 179 / 28	392 / 179 / 4	273 / 27 / 73
s526	445 / 193	391 / 140 / 2	265 / 105 / 3461	267 / 105 / 67	156 / 35 / 65
s526n	445 / 194	391 / 140 / 2	265 / 105 / 3414	267 / 105 / 70	155 / 33 / 66
s641	539 / 379	288 / 128 / 3	229 / 111 / 79	229 / 111 / 58	175 / 51 / 88
s713	591 / 393	288 / 128 / 7	229 / 111 / 88	229 / 111 / 65	175 / 51 / 93
s820	757 / 289	724 / 256 / 5	669 / 241 / 70	669 / 241 / 22	415 / 64 / 287
s832	769 / 287	724 / 256 / 9	669 / 241 / 81	669 / 241 / 28	406 / 62 / 244
s838	670 / 390	545 / 265 / 8	76 / 38 / 52	76 / 38 / 33	57 / 8 / 28
s953	743 / 395	667 / 319 / 6	646 / 310 / 492	650 / 312 / 380	440 / 88 / 284
s1196	1009 / 529	869 / 389 / 12	864 / 387 / 747	869 / 389 / 221	681 / 104 / 10899
s1238	1041 / 508	874 / 392 / 38	869 / 390 / 1187	876 / 393 / 377	735 / 111 / 1015
s1488	1387 / 653	1284 / 550 / 6	1218 / 528 / 189	1231 / 532 / 40	716 / 93 / 1462
s1494	1393 / 647	1284 / 550 / 12	1218 / 528 / 233	1233 / 533 / 48	722 / 94 / 1584
TOTAL (lit/gate)	12828 / 6463	10855 / 4364	9218 / 4020	9292 / 4044	6203 / 1079
Reduction (lit/gate)		15% / 32%	28% / 38%	28% / 37%	52% / 83%

We also examined the results of applying various combinations of quick sequential redundancy removal and don't care minimization. Table 4.4 shows the results achieved by don't care minimization alone, quick sequential redundancy removal followed by don't care minimization, and don't care minimization followed by quick sequential redundancy removal. As in Table 4.3, each column shows the number of literals and gates in the circuits after each procedure, along with the time taken by the procedure.

Table 4.4: Results of Applying Both Redundancy Removal and External Don't Care Minimization

Circuit	Before (lit/gate)	Exdc Min (lit/gate/time)	Quick Seq RR Exdc Min (lit/gate/time)	Exdc Min Quick Seq RR (lit/gate/time)
s208	166 / 96	57 / 8 / 7	56 / 8 / 10	57 / 8 / 7
s298	244 / 119	108 / 24 / 20	106 / 23 / 24	108 / 24 / 22
s344	269 / 160	164 / 49 / 113	164 / 49 / 140	164 / 49 / 144
s349	273 / 161	164 / 49 / 109	164 / 49 / 139	164 / 49 / 143
s382	306 / 158	147 / 32 / 77	140 / 32 / 224	147 / 32 / 142
s386	347 / 159	115 / 21 / 36	119 / 19 / 34	115 / 21 / 38
s400	322 / 165	144 / 33 / 78	136 / 32 / 227	144 / 33 / 136
s420	336 / 196	57 / 8 / 12	56 / 8 / 17	57 / 8 / 13
s444	352 / 181	141 / 34 / 66	141 / 31 / 103	141 / 34 / 136
s510	424 / 211	273 / 27 / 73	273 / 27 / 80	273 / 27 / 81
s526	445 / 193	156 / 35 / 65	135 / 31 / 137	154 / 34 / 117
s526n	445 / 194	155 / 33 / 66	135 / 31 / 142	155 / 33 / 101
s641	539 / 379	175 / 51 / 88	175 / 51 / 94	175 / 51 / 103
s713	591 / 393	175 / 51 / 93	175 / 51 / 101	175 / 51 / 116
s820	757 / 289	415 / 64 / 287	354 / 50 / 220	405 / 62 / 302
s832	769 / 287	406 / 62 / 244	371 / 55 / 231	390 / 60 / 265
s838	670 / 390	57 / 8 / 28	56 / 8 / 39	57 / 8 / 31
s953	743 / 395	440 / 88 / 284	447 / 89 / 686	440 / 88 / 3527
s1196	1009 / 529	681 / 104 / 10899	670 / 105 / 1442	681 / 104 / 11184
s1238	1041 / 508	735 / 111 / 1015	690 / 99 / 1090	732 / 110 / 1338
s1488	1387 / 653	716 / 93 / 1462	706 / 97 / 1638	716 / 93 / 1499
s1494	1393 / 647	722 / 94 / 1584	695 / 90 / 1685	717 / 92 / 1610
TOTAL (lit/gate)	12828 / 6463	6203 / 1079	5964 / 1035	6167 / 1071
Reduction (lit/gate)		52% / 83%	54% / 84%	52% / 83%

Table 4.4 shows that for almost every circuit, the combination of quick sequential redundancy removal followed by don't care minimization achieves only slightly more area reduction than don't care minimization alone, but takes about 40% more time, on average. Note, though, that for most of the largest circuits, this combination of procedures takes about the same or less time than don't care minimization alone. Further, for circuit s1196, this combination of procedures takes an order of magnitude less time than does don't care minimization. This suggests that sequential redundancy removal may aid in area optimization for large circuits--circuits that cannot be optimized efficiently using don't care minimization procedures. For such circuits, it is possible that sequential redundancy removal may reduce the size of the circuit enough so that external don't care minimization can be performed much more quickly.

Finally, Table 4.4 shows that the combination of external don't care minimization followed by quick sequential redundancy removal does not achieve significantly more reduction than does don't care minimization alone, showing that don't care minimization removes most or all of the redundancies in these circuits.

4.3 Conclusions

Sequential redundancy removal, a straightforward application of test generation procedures, has been shown to reduce the number of literals in the ISCAS sequential benchmark circuits by an average of 30% and to reduce the number of gates by an average of 40%. Further, a modified "quick" version of sequential redundancy removal, in which only SNE redundancies are removed, has been shown to achieve almost the same reduction, about ten times more quickly. Although the reduction achieved by sequential redundancy removal is greater than that achieved by combinational redundancy removal, it is much less than the reduction achieved by don't care minimization techniques when the unreachable states of the circuit are used as external don't cares. However, sequential redundancy removal can be useful in optimizing some circuits that are too large to be 0ptimized efficiently using don't care minimization. For these circuits, the quick sequential redundancy removal reduces the circuit enough that don't care minimization can proceed more quickly. In these cases, the combination of quick sequential redundancy removal followed by don't care minimization achieves more reduction, in less time, than either procedure alone.

5 Future Work

The results presented in this paper show that the sequential ATPG and redundancy removal algorithms that have been implemented in SIS are effective for small circuits--the ISCAS benchmark circuits with fewer than 50 latches; however, these algorithms fail on larger circuits, as does every existing test generation algorithm. For the next set of larger circuits, the limiting operation is STG traversal; the BDD's produced during the implicit STG traversal of the circuit become too large to be operated upon efficiently. Since the ATPG algorithms described in this paper are based upon reachability calculations, this is a serious problem, and new techniques will be needed if we are to test larger circuits.

There are several possible approaches to the problem of testing and identifying redundancies in larger circuits, some exact and some approximate. One approach is to break up a large circuit into a system of smaller, interacting finite state machines. Of course, the small finite state machines cannot be examined in isolation, since the behavior of each small machine is controlled by its environment, i.e. the other finite state machines. However, for each small finite state machine, the environment of the machine can be reduced with respect to the machine, i.e. behavior that does not affect the machine can be added to or subtracted from its environment. After the reduction of the environment, the product of the machine and the reduced environment can be constructed. This reduced product machine is not equivalent to the original large circuit; however, the behavior of the small machine in this product machine is exactly the same as its behavior in the original circuit. Thus, if the STG of the reduced product machine can be traversed using existing techniques, it will be possible to test faults and identify redundancies within the small machine. In order to test the entire original circuit, the environment reduction process can be repeated for each small machine.

This method will not work in cases where even the reduced machine cannot be traversed. In this case, a similar but approximate approach can be used. In this approach, the environment reduction is not exact, i.e. during the reduction, behavior which *does* affect the small machine is added to or subtracted from the environment. Because the reduction is approximate, the behavior of the small circuit within the reduced product machine is not exactly the same as its behavior within the original circuit. Thus, in general it is not possible to test faults or prove redundancies within the small machine. However, if the approximate reduction follows some rules, it is possible to test some faults and prove some redundancies. In particular, if the reduced environment is formed by only adding behavior to the environment, then any fault of the small machine which is redundant in the reduced product machine is also redundant in the original machine. If the reduced environment is formed by only subtracting behavior from the environment, then any sequence which tests a fault in the reduced machine will test that fault in the original machine.

Another similar approximate approach to the problem of testing large circuits is to simplify the BDD representation of either the circuit outputs, the circuit STG, or the reachable state set by adding and subtracting behavior. As with the approximate reduction of the environment described above, this approach is not exact, but if rules are applied to the approximation process, some testing and redundancy identification can be performed. Cho, et. al. at the University of Colorado at Boulder have used this approach to perform redundancy removal on the circuits in the ISCAS benchmark set that cannot be tested by existing methods [4]. They approximate the reachable set of the circuit as follows:

First, they partition the latches of the large circuit into disjoint subsets, and construct a submachine for each subset. Each submachine contains all the combinational logic of the original circuit, but only those latches that appear in its corresponding subset; the next and present state lines for all other latches become primary inputs and outputs. Each submachine is traversed, using the results of previous traversals to limit the inputs that can be applied during traversal. Because the results of each traversal can affect other traversals, the traversals are performed iteratively until a fixed point is reached--until the reachable states of the submachines do not change. At this point, an approximation to the reachable states of the entire circuit is formed by merging the results of the submachine traversals, i.e. by intersecting the BDD's of all the submachine reachable states.

Note that this approximation always contains the actual reachable state set, and thus it can be used to prove faults redundant--if a fault cannot be excited from any state in the overestimation of the reachable state set, then it certainly cannot be excited from any state in the reachable state set, and thus is redundant.

We have extended this method to obtain better approximations to the reachable state set [10]. In the Boulder approach, only disjoint partitions of latches are considered; in our extension, we consider subsets of latches which are not disjoint. In essence, we are generating several approximations to the reachable state set, and then intersecting them to obtain the final approximation. Although each individual approximation may be large, the hope is that they will be very different from each other, and thus their intersection will be small, i.e. close to the actual reachable state set. The results obtained by using this extension are shown in Table 5.1. For four different circuits, this table shows the exact number of reachable states, the number of states in the UC Boulder approximation, the number of states we obtained using the UC Boulder approximation method, and finally, the number of states we obtained by intersecting two approximations, i.e. by using non-disjoint latch subsets. Our extension gives reachable state approximations about 2 times smaller than those obtained by the Boulder approach for three of the four circuits.

Table 5.1: Approximate Reachability Results

	s526	s713	s953	s1238
Exact Number of States	8,868	1,544	504	2,616
UCBoulder Approximation-- disjoint latch subsets	15,200	5,300	--	18,447
Berkeley Approximation-- disjoint latch subsets	16,800	5,300	4,087	25,092
Berkeley Approximation-- non-disjoint latch subsets	15,500	2,362	2,040	13,801

We would like to extend our method even further by adding dynamic latch subset selection to our reachability approximation algorithm. At each step of the approximation process, the next submachine to be traversed would be chosen based upon the current approximation; we would like to choose a submachine whose traversal is likely to give an approximation very different from the current approximation, so that their intersection will be as small as possible. In this way, we hope to generate approximations that are even closer the actual reachable state set.

6 Conclusions

A sequential test generation and redundancy removal system, based on implicit STG traversal techniques, has been implemented in SIS. The test generator achieves 100% test fault coverage on the ISCAS '89 benchmark circuits that have fewer than 50 latches, but runs 1.6 times slower than the test generator VERITAS. The SIS test generator is slower than VERITAS because VERITAS uses more sophisticated fault simulation, random test generation, and product machine traversal techniques than SIS. However, the satisfiability-based combinational test generator of SIS identifies redundant faults much faster than the structurally-based combinational test generator of VERITAS. In addition, test set reducing heuristics have been added to the basic SIS ATPG algorithm, and the best combination of these heuristics has been identified. Use of these heuristics reduces test set size by a factor of 3.1 over tests sets produced by the basic SIS algorithm. The SIS redundancy removal algorithm reduces the number of gates in the ISCAS '89 benchmarks by 40%, but achieves less reduction than external don't care optimization procedures. However, for some large circuits, a combination of sequential redundancy removal and don't care optimization achieves more reduction, in less time, than either procedure alone. Finally, methods for testing and identifying redundancies in larger circuits--circuits on which existing sequential ATPG algorithms fail--have been suggested.

References

[1] M. Breuer and A. Friedman. *Diagnosis and Reliable Design of Digital Systems.* Computer Science Press, 1976.

[2] F. Brglez, D. Bryan, and K. Kozmiski. Combinational Profiles of Sequential Benchmark Circuits. In *Proceedings of the IEEE International Symposium on Circuits and Systems,* Portland, Oregon, May 1989.

[3] R. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers,* vol. C-35, August 1986, pp. 677-691.

[4] H. Cho, G. Hachtel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for Approximate FSM Traversal. In *Proceedings of the 30th Design Automation Conference,* June 1993, pp. 25-30.

[5] H. Cho, G. Hachtel, and F. Somenzi. Redundancy Identification/Removal and Test Generation for Sequential Circuits Using Implicit State Enumeration. In *IEEE Transactions on Computer-Aided-Design,* July 1993, pp. 935-945.

[6] O. Coudert and J. C. Madre. A Unified Framework for the Formal Verification of Sequential Circuits. In *Proceedings of the International Conference on Computer-Aided Design,* November 1990, pp. 126-129.

[7] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman and Company, 1979.

[8] A. Ghosh. *Techniques for Test Generation and Verification of VLSI Sequential Circuits.* Ph.D. thesis, University of California, Berkeley, September 1991.

[9] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. In *IEEE Transactions on Computers*, vol. C-30, March 1981, pp. 215-222.

[10] M. Khalaf. Approximating the Set of Reachable States for Finite State Machines. EE 290LS report, 1992.

[11] T. Larrabee. *Efficient Generation of Test Patterns Using Boolean Satisfiability.* Ph.D. thesis, Stanford University, 1990.

[12] D. Long. Personal communication, November 1993.

[13] H-K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli. Test Generation for Sequential Circuits. In *IEEE Transactions on Computer-Aided-Design*, October 1988, pp. 1081-1093.

[14] T. Niermann, R. K. Roy, J. H. Patel, and J. A. Abraham. Test Compaction for Sequential Circuits. In *IEEE Transactions on Computer-Aided-Design*, February 1992, pp. 260-267.

[15] T. Ono and M. Yoshida. A Test Generation Method for Sequential Circuits Based on Maximum Utilization of Internal States. In *Proceedings of the International Test Conference*, 1991.

[16] C. Pixley, S. Jeong, and G. Hachtel. Exact Calculation of Synchronization Sequences Based on Binary Decision Diagrams. In *Proceedings of the 29th Design Automation Conference*, 1992, pp. 620-623.

[17] H. Savoj, H. Touati, and R. Brayton. Extracting Local Don't Cares for Network Optimization. In *Proceedings of the International Conference on Computer-Aided Design*, November 1991.

[18] E. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proceedings of the International Conference on Computer Design*, October 1992.

[19] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. *Combinational Test Generation Using Satisfiability.* Memorandum No. UCB/ERL M92/112, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA, October 1992.

[20] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines Using BDDs. In *Proceedings of the International Conference on Computer-Aided Design*, November 1990, pp. 130-133.