Efficient Sequential ATPG Based on Partitioned Finite-State-Machine Traversal *

Qingwei Wu and Michael S. Hsiao {*qiwu3, hsiao*}*@vt.edu*
Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA

Abstract

We present a new automatic test pattern generation algorithm for sequential circuits by traversing the partitioned state spaces. The new features include: (1) non-disjoint state groups are obtained such that two different state groups may have common flip-flops, (2) partial state transition graphs (STGs) are constructed at run time for each state group, (3) spectral information for state variables are extracted and the spectral information of the state variables helps to identify the behavior of the flip-flops in the frequency domain. This information will help us to intelligently partition the state space. We focus only on the STGs for the flip-flops that are grouped together instead of building the STG for the entire circuit, and the ATPG tries to traverse all states and transitions within each partial STG. By exercising states visited and arcs traversed, the vectors generated often lead to the detection of hard faults. Since we limit a maximum size any state group can be, construction of partitioned STGs is feasible even for very large sequential circuits. Only logic simulation is needed in our ATPG; as a result, the execution time is greatly reduced while achieving high fault coverages compared with other test generators. For some large sequential circuits, highest fault coverages have been achieved.

1 Introduction

ATPG (Automatic test pattern generation) is playing an increasingly important role both in testing and verification of digital systems in recent years. Deterministic ATPGs [1–5] target a certain fault at a time and generate a test sequence for that fault. However, for hard-to-detect faults, aborts are possible due to extensive backtracks in the search space. This can result in unsatisfactorily low fault coverage. A sequence of random patterns can be used as a prefix to first remove some faults to alleviate the computation cost of the ATPG. However, it may only detect the easy-to-detect faults, thus the reduction in computation may not be significant.

Simulation-based test generation [6–12] is an alternative method of ATPG. It can be broadly divided into fault-simulation based and logic-simulation based categories. In the former, the test generator repeats fault simulation to gather information on targeted faults to guide the test generator search. Since fault simulation can be costly, its application for large sequential circuit may be limited. On the other hand, instead of using fault coverage metrics (which need calling of fault simulation), logic-simulation-based test generators use characteristics embedded in the fault-free circuit to guide the test generator with only logic simulation. If successful, large numbers of faults can be detected at a significantly reduced cost compared with deterministic and fault-simulation based ATPG. In the work by Saab et al., [8], the characteristic targeted by the ATPG is the number of events activated during logic simulation in the fault-free circuit. The test generator tries to maximize this circuit activity, with hopes that increased activity will lead to detection of more faults. In LOCSTEP [9], it was observed that the test sequence generated by deterministic test generators generally traverses through a sequence of states, and the new states reached inside the sequence usually helped in improving the fault coverage. Based on this observation, LOCSTEP tries to maximize the number of states reached. Since the number of states in a large sequential circuit can be huge, the test set sizes obtained can be large. In [7], attempt is made to differentiate between new states reached. To achieve this, state partitioning is used. The ATPG search tries to favor those states that are deemed more valuable, and hence more likely to detect the hard faults.

Despite the advances made in logic-simulation-based ATPGs, one disadvantage of logic-simulation based test generators remains: the fault coverage for large sequential circuits can still be lower when compared with that of fault-simulation-based counter-part, such as those reported in [6, 11, 12].

Our goal in this work is to develop an efficient ATPG that relies only on exploring embedded characteristics within the sequential circuit, in order to achieve very high fault coverages. Furthermore, these characteristics should be obtained with *only* logic-simulation to reduce

*This work is supported in part by NSF Grants CCR-0196470, CCR-0098304, CCR-0305881, and a grant from NJ Commission on Science & Technology.

0-7803-8106-8/03 $17.00 Copyright 2003 IEEE

the computation costs. To do this, we address the following issues: (1) state partitioning by extracting the spectral information of the states that the current test sequence has traversed, and (2) state and transition coverage within each partitioned state set. State partitioning and grouping are an inherent part of this work. It is simply dividing the flip-flops in a circuit into smaller groups, which is illustrated in Figure 1. Essentially, state partitioning breaks the global state into groups of flip-flops. However, the way the flip-flops should be divided is a difficult problem [7].

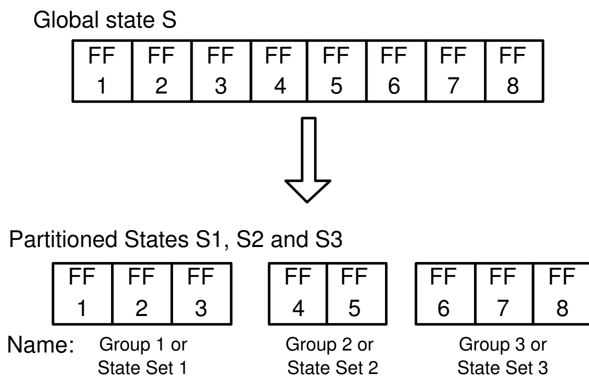

Figure 1: Illustration of State Partitioning

Since the quality of the partitioned space is important, methods in computing the partitions and groups are critical. The spectral information embedded in the states traversed from a given test sequence can help us partition the state variables. In particular, the flip-flops whose state-transition behavior contain dominant frequencies are grouped together. In the frequency domain, these flip-flops are observed to have correlation with one another. Furthermore, disjoint partitioning may not be the optimal way to divide the flip-flops. In other words, we allow state groups to share some common flip-flops. These overlapped state grouping can help the ATPG by observing how some specific flip-flops can contribute to different partitioned state spaces. Next, the STG (State Transition Graph) for each partial state space is dynamically constructed during the ATPG search. By traversing the states and arcs inside each partial STG (similar to the method in finite-state-machine (FSM) testing), the vectors generated help to exercise the circuit's state subspaces.

In terms of FSM testing, the idea is to map out the finite state machine embedded in the design and exercise the states and transitions within it. Recent work on FSM testing has yielded promising preliminary results. In the research by Goren and Ferguson [13], a fault coverage metric for a test suite for an FSM specification was derived and then extended for fault coverage estimation of interconnected FSMs. In the work by Wedler et al. [14], a technique for logic simulation with logic constraints given as equivalence relationships and constant value assignments is presented. And this simulation technique is a crucial ingredient in a practical implementation of FSM traversal. However, FSM testing remains to be a difficult problem when dealing with large sequential circuits with many flip-flops. In our work, since the number of flip-flops in each partitioned state set is limited to k, the largest number of states in any set is thus 2^k, we avoid the state explosion if we keep the size of k to be less than 16. In addition, if we can identify and group the flip-flops that belong to the controller of the circuit, the extracted STGs in our work can give us a view from the control-unit perspective, thereby providing additional guidance as we generate vectors. Finally, since only logic simulation is needed, the proposed technique can also contribute to generating effective simulation vectors for design validation. Our experimental results show that higher fault coverages can be achieved with very low computational needs, when compared to previously reported ATPGs that are based on logic simulation.

The remainder of the paper is organized as follows. Section 2 gives an overview and motivation for the new sequential ATPG approach. Section 3 explains the algorithm for extracting spectral information and its application to overlapped state grouping. Section 4 discusses the STG construction. Section 5 reports the experimental results, and Section 6 concludes the paper.

2 Overview and Motivation

Previously, state partitioning has shown promise in logic-simulation based test generation [7]. However, the state partitioning is mostly based on the structural information of the circuit; in other words, it is difficult to self-adapt to the change of test set on the run. Updating the state partition from time to time is important since re-partitioning of the state space may help in generating vectors that previous partition may have difficulty capturing to detect hard faults.

Before going further, we will first explain why spectral information can be helpful to partition the states. The spectral information reflects the behavior of the flip-flops in the frequency domain. If a flip-flop has a dominant frequency, it will exhibit a specific behavior in the frequency domain. Here is a simple example. Let us consider the circuit in Figure 2. The circuit switches between functional mode 1 and mode 2 at every clock cycle. A 4-bit counter and a 2-channel multiplexer are used to implement this specification. In the fault-free circuit, FF_1 and FF_5 will exhibit the same sequence (0101010101...). Obviously, both of these two flip-flops

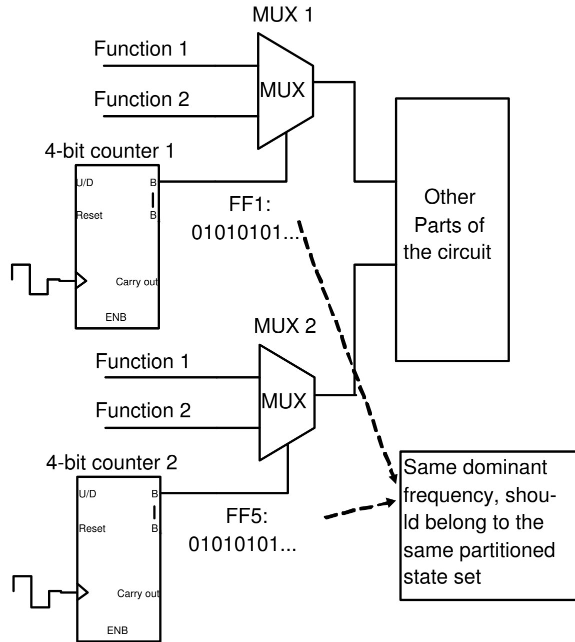

Figure 2: State partitioning by dominant frequency

have a single frequency, which is also their dominant frequency. Thus, it is reasonable to group these two flip-flops together. In general, any two flip-flops having the same dominant frequency will imply that they change their values together (note that they could change in opposite directions). This also suggests that there exists potential correlation between them. Note that in this simple example, the two flip-flops selected are both on the control path.

The proposed approach dynamically analyzes the spectral information of the FFs from the states traversed and uses this information to periodically re-partition the states if necessary. The benefit of this approach over the static state partitioning is illustrated in Figure 3. Suppose current partition results in two state sets: $\{FF_1$, $FF_2\}$ and $\{FF_3$, $FF_4\}$ and the current test set has successfully reached the complete state space within each partitioned state space as shown in the figure. In static state partitioning, it is difficult to evaluate the contribution of a vector without looking at the global state if the complete state space for each partition has been reached. As a result, differentiation of vectors is not possible. For example, suppose there is a new vector V that drives the circuit into a new global state (0010). However, because both of the partitioned states (00 and 10) have been reached *separately* before, it is difficult to compare the value of this vector with another that also results in a new, but different, global state.

By noting that FF_1 and FF_3 have the same dominant frequency, (sequence 0101 and 1010 have the same frequency, only different phase), within our approach, they will be re-partitioned based on the observed dominant frequencies embedded. Likewise, FF_2 and FF_4 also share a dominant frequency, and they will be grouped together in the new partitioning. Now for vector V, since it reaches two new partitioned states, i.e., 01 for set $\{FF_1, FF_3\}$ and 00 for $\{FF_2, FF_4\}$, vector V will be viewed as one that reaches some new partitioned states.

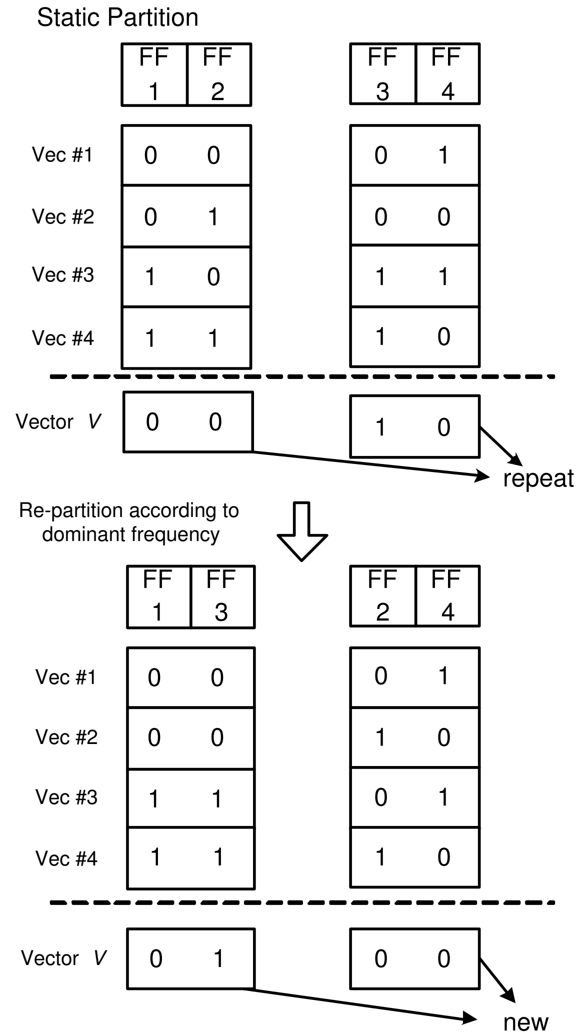

Figure 3: Illustration of the power of dynamical partition

Since we favor visiting new and useful states, we would like a mechanism that helps us to achieve this goal. In [11], it was observed that when one randomly picks one vector from the compaction test set and repeats it for several time frames, improvement in fault coverage can be achieved. From the state traversal point of view, we found that holding a vector (as opposed to cascading of different random vectors) indeed is also helpful to drive the circuit into more new states.

Based on this observation, we incorporated holding into our framework. However, instead of simply picking and holding one vector randomly, we do it more intelligently by preferring the vector sequence (from holding the same vector for several time frames) that will bring the circuit into more new partitioned states. It's important to note that repartition of state variables is not performed indefinitely. We reserve the repartition step only when the present state partition cannot help to identify additional useful vectors; in other words, the search cannot expand the current partitioned state space any more.

3 Overlapped State Grouping

Because what we care about most is the information embedded in the states the current test set has traversed, we want to employ signal processing techniques to extract this information. Frequency decomposition is the most commonly used technique in signal processing. A signal can be projected onto a set of independent waveforms, where each waveform has a distinct frequency. In choosing the projection matrix, we want one that can work well with non-sinusoidal waveforms, since the pattern in the FFs generally are non-sinusoidal. Hadamard transform is a well-known non-sinusoidal orthogonal transform in signal processing.

The state grouping algorithm using dominant frequency information is outlined below. We consider a Hadamard matrix with order m.

```
// step 1: frequency decomposition for each FF
for each test segment (2^m vectors) {
    for (each FF in the circuit) {
        coefficient vector c_i = H × s_i;
        for (each element c_ij)
            FF_frequency_component[i][j] += c_ij ;
    }
}

// step 2: judge if dominant frequency exists
for (each FF i in the circuit) {
    get the maximum coefficient FF_MaxFre[i];
    get the 2^nd max coeff FF_2ndMaxFre[i];
    if (maximum coefficient > threshold {
        dominant freq = FF_MaxFre[i];
        2^nd dominant freq = FF_2ndMaxFre[i];
    }
    else
        this flip-flop has no dominant frequency;
}

// step 3: grouping the states
initial set Φ = all FFs with dominant freq;
while (Φ not empty) {
    pick one flip-flop FF_i;
    Φ_i = all FFs in Φ with the same dominant
        frequency as FF_i;
    place all flops in Φ_i into the same set;
    if (non-overlapped state group)
        remove all FFs in Φ_i from Φ;
}
```

The above algorithm works as follows. H_m is the Hadamard transform matrix of order m and S_i is the state bit sequence of flip-flop $\#i$. A flip-flop will have 2^m possible frequency components, corresponding to the 2^m vertical vectors of Hadamard matrix H_m. Noting the fact that $H_m = H_m^{-1}$, we can obtain the following equation (using c_i to substitute $H_m \times S_i$):

$$S_i = H_m^{-1} \times H_m \times S_i = H_m^{-1} \times (H_m \times S_i) = H_m \times c_i$$

$$\Longrightarrow S_i = \sum_{j=0}^{2^m-1} c_{ij} \cdot h_j$$

where c_{ij} is the jth element of c_i, and h_j is the jth vertical vector of H.

Based on the above equation, S_i can be represented as the sum of frequency components embedded in the signal multiplied by the corresponding coefficient. And it is clear that the coefficient vector can be obtained by $c_i = H_m \times S_i$.

For every segment of the states traversed, the coefficient vector will be calculated for every flip-flop. Then, these vectors will add up together to reflect the final coefficient of each frequency component. The two highest coefficients will be identified and stored. If the highest coefficient is larger than the predetermined threshold (we set threshold to be half of the sum of the coefficients), it indicates that a dominant frequency for the corresponding flip-flop exists. Once a flip-flop has a dominant frequency, we also record the second highest coefficient for it using a similar computation. For example, suppose the order of Hadamard matrix is 3 and the final coefficient vector for $FF1$ is $(7\ 0\ 18\ 0\ 5\ 0\ 1\ 2)^{-1}$, then the highest coefficient is 18, whose position in the coefficient vector is 3, and the sum is 33. Since 18 is greater than half of 33, $FF1$ has dominant frequency 3, corresponding to the third vector of H_3:$(1\ 1\ \text{-}1\ \text{-}1\ 1\ 1\ \text{-}1\ \text{-}1)^{-1}$. The second highest coefficient is 7, whose position in the coefficient vector is 1, so the second dominant frequency of $FF1$ is 1, corresponding to the first vector of H_3: $(1\ 1\ 1\ 1\ 1\ 1\ 1\ 1)^{-1}$.

Once the dominant and second dominant frequencies have been computed for each flip-flop, the flip-flops with same dominant frequency are placed into the same group. Because we set a maximum size of each state

group as k, if the number of flip-flops with same dominant frequency is greater than k, we partition them further into smaller groups according to their second dominant frequencies. If there still remains groups with sizes greater than k, we simply partition them randomly into smaller groups. This process is illustrated in Figure 4. For the rest of the flip-flops that have no dominant frequency, we partition them randomly.

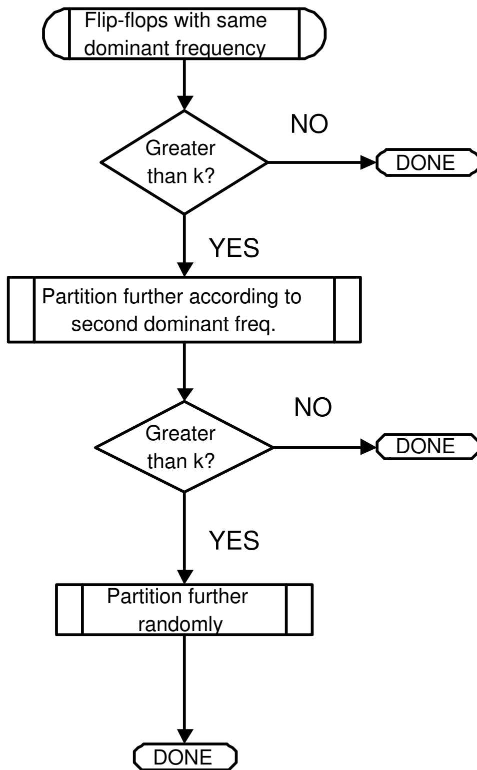

Figure 4: Algorithm for Partitioning States with dominant frequency

In [7], the state partitioning is disjoint, which means, each flip-flop belongs to *exactly* one state set. However, in doing so, this may prevent us from obtaining effective vectors. Consider the portion of a circuit shown in Figure 5. Suppose we have disjoint state partitions $\{FF1, FF2, FF3\}$ and $\{FF4, FF5\}$ and partitioned states (001) and (01) have been reached *separately* for these two partitions before. In order to propagate the fault-effect of the fault f in the circuit, it would require traversing through a *complete* state of $(FF1, FF2, FF3, FF4, FF5) = 00101$. Under the disjoint state partitioning, any vector that drives the circuit into this desired state will not be captured, since every partitioned states ('001' and '01') has been reached. However, in our overlapped state grouping, if $FF3$ belongs to both state sets, we result in these two state groups: $\{FF1, FF2, FF3\}$ and $\{FF3, FF4, FF5\}$. Under this overlapping state grouping, partitioned state (101) for state group $\{FF3, FF4, FF5\}$ is new and the vector driving the circuit into state (00101) will be captured. This is illustrated in Figure 6.

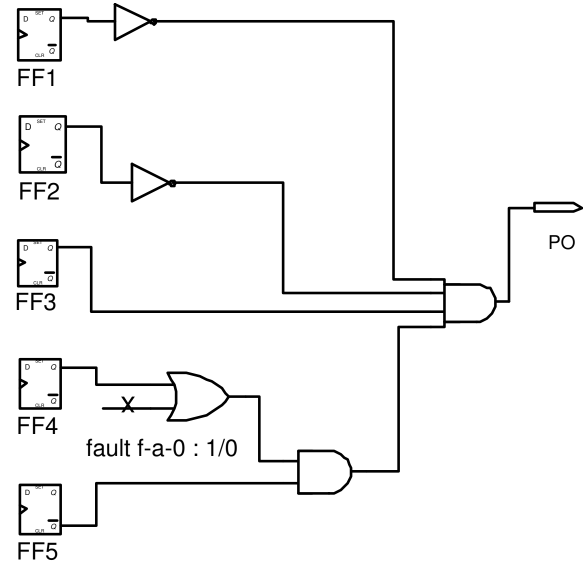

Figure 5: Example Circuit of overlapped State Grouping

By modifying step 3 of the algorithm discussed earlier, we can allow for overlapped state grouping. THe entire step 3 is now divided into three phases, as outlined below:

// step 3: overlapped state grouping
// phase 1: partitioning using dominant frequency
initial set Φ = all flip-flops with same dominant freq;
while (Φ not empty) {
    pick one flip-flop FF_i;
    Φ_i = all FFs in Φ with the same dominant
        freq as FF_i;
    place all FFs in Φ_i into the same set;
    remove all FFs in Φ_i from Φ;
}

// phase 2: partitioning using second dominant freq.;
initial set Φ = all flip-flops that have second
dominant frequency;
while (Φ not empty) {
    pick one flip-flop FF_i;
    Φ_i = all FFs in Φ with the same dominant
        freq as FF_i;
    place all FFs in Φ_i into the same set;
    remove all FFs in Φ_i from Φ;
}

// phase 3: check for subset and superset

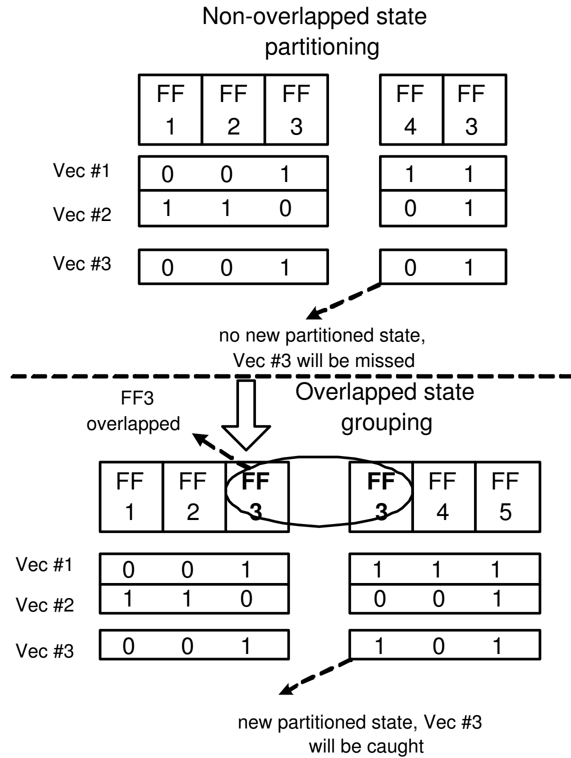

Figure 6: Illustration of the Benefit of Overlapped State Grouping

// between state sets
for (each state set P_i) {
  for (every other set P_j) {
    if ($P_i \subseteq P_j$)
      remove P_i from state sets;
    else if ($P_i \supseteq P_j$)
      remove P_j from state sets;
  }
}

The above algorithm works as follows. In the first phase, we group the flip-flops according to their dominant frequencies, while in the second phase, grouping is performed according to their second dominant frequencies. As discussed above, the second dominant frequency can also be significant enough to describe the behavior of the FFs in the frequency domain. So it can also be used as a criteria to partition the states. After these two phases, some state sets may share common flip-flops. So, if one state set is a proper subset of another state set, we remove the smaller set since it will not contribute for selecting vectors with the presence of the bigger set. This step is performed in phase-3 of the algorithm.

4 Dynamic STG for partitioned states

When we explain the ATPG process, in which the partitioned state spaces are used to guide the search for effective vectors. In traversing the FSM for each state group, not only is the number of unique states visited within each state group important, but also the specific paths through which the states are visited. This urges us to construct the STG (State Transition Graph) for each state group. Each node in the STG represents a state in the state space for the corresponding FSM, and the edges indicate the transitions between states. Initially, the STG consists of only nodes and no edges. The edges are added as transitions are traversed by the vectors generated.

Consider a STG for a state group, where nodes A and B have been visited via a path with the associated edge $A \rightarrow B$. Now suppose a new vector sequence drives the circuit to node A from B, finding a new edge: $B \rightarrow A$. In state partitions without STG such as in [7], this vector sequence will not be added to the test set since no new state in the partitioned state space is reached. However, our approach considers this vector pair useful because it traverses through a new transition in the STG.

In general, it makes sense to assign more weight to the nodes in the STG than to the edges, since reaching more states is of higher value. However, transitions within the STG can also play an important role, as different paths through the STG are now taken into account. In our method, we set the weight for the nodes to be 0.7, while the weight for the edges to be 0.3 (we tried other different weights, such as 0.6 vs. 0.4, 0.8 vs. 0.2. We found that the weights of 0.7 vs. 0.3 yielded best results). However, these weights can easily be changed to suit the ATPG needs.

Recall that we set a limit of k (with $k < 16$) to be the maximum size of each state group. So, even for very large circuits, it is still computationally feasible to construct the STGs for all the state sets. In our approach, the goal is trying to best construct the STG for each state group dynamically, with hopes that it will help drive the circuit to reach interesting and new states, thus detecting hard faults.

Although we favor new states, sometimes repeating of a visited state is necessary for detecting a fault. For example, consider the circuit in Figure 7. Suppose flip-flops $\{FF1, FF2, FF3\}$ are placed in the same set. To detect the fault f, we need to excite the fault and propagate the fault effect. Thus, $\{FF1, FF2, FF3\}$ are required to be (101) for at least two consecutive time frames. If repeating of a state is disfavored, we may miss the necessary sequence to detect this fault. Repeating of states can be obtained by taking advantage of the self-loops inside the STG. This is illustrated in Figure

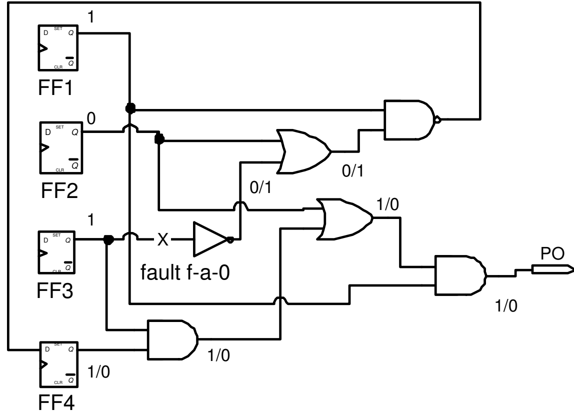

Figure 7: Example Circuit to illustrate self-loop in STG

8. Nevertheless, traversing through self-loops cannot be

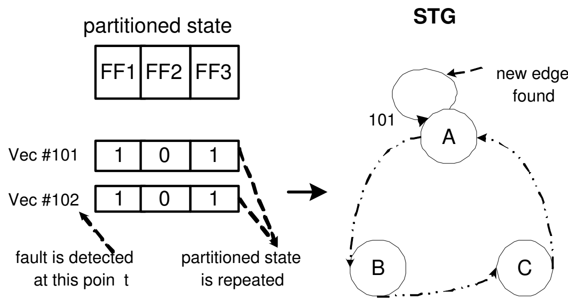

Figure 8: Illustration of self-loop in STG

repeated indefinitely, and this is taken into account in our ATPG.

Finally, with the state groups, a side benefit is that we can obtain distinguishing sequences for each group, as well as unique input-output sequences. Thus, we can perform additional FSM testing such as state identification and state verification, if needed.

5 Experimental Results

We implemented the proposed overlapped state grouping and STG construction algorithm in C++. The experiments were conducted on a number of ISCAS89 [15] and ITC99 [16] benchmark circuits over a 2.4 GHz Pentium 4 with 512M RAM, running the Linux system. Because overlapped sate grouping is the core of the algorithm, we targeted mostly *large* sequential circuits having more than **50 flip-flops**. However, we also report results for some small circuits to show the effectiveness on smaller circuits.

In Table 1, we compared the results of our test generator with HITEC [1], a deterministic test generator, LOCSTEP [9], a logic-simulation based test generator that targets maximizing global states visited (no state partitioning) and the results from [7], where static state partitioning was considered. For each circuit tested, the number of faults detected, test set size, and ATPG time are reported for each of the three ATPGs, except for LOCSTEP, where the execution times are not available.

We allow for a maximum number of 80,000 vectors to be generated for each circuit, and we list the number of detected faults and test generation time as the point where the maximum fault coverage was first reached.

From Table 1, we can observe that our approach is very efficient in obtaining high fault coverages together with small test sets. By favoring traversal of specific paths in the partitioned STGs (whose transitions have not been exercised), we confine the circuit along the paths in FSM that will achieve high fault coverage quickly. Our execution times are slightly higher than [7] because we pay for the STG construction. However, the execution times are much shorter than HITEC. Note that for most of the circuits, very high coverages have already been achieved by [7], but the detection of a few extra faults is note-worthy, since these are very *hard-to-detect* faults, which most ATPGs spend most of their time on. For example, in the circuit s5378, HITEC detected 3238 faults, LOCSTEP detected 3059 faults, 3636 faults were detected in [7], and we were able to detect 3642 faults. This coverage is only 1 fault lower than the highest reported coverage of 3643 faults [12]. Finally, for circuits b12, b15, and b22 (all very hard-to-test circuits), our approach achieved significantly higher coverage than the previously reported approaches. In b12 and b15, highest coverages have been obtained by our method.

Table 2 compared our results with STRATEGATE [6], a state-of-the-art fault-simulation-based ATPG. Our new ATPG achieves higher coverages for s1423, s5378, b12 and b15, the same for s35932, while just slightly lower for b22. And the execution times are *several orders of magnitude* lower than STRATEGATE. For example, in circuit b12, STRATEGATE detected 1488 faults in 4 days of execution, while our method detected 1665 faults, which is highest reported so far, in only 124 seconds. Likewise, for b15, we obtained a higher fault coverage in 1.5 hours, compared to 7 days of execution with STRATEGATE.

6 Conclusion

In this paper, we have presented a new method in which spectral information extracted from traversed states are used to partition the state variables, and we presented a novel overlapped state grouping to benefit the ATPG search. We constructed STG (State Transition Graph) for each state group dynamically and this helps both test generation and Finite-State-Machine traversal within

Table 1: Experimental Results

Circuit	# of FFs	Deterministic ATPG			Logic-Simulation-Based ATPG							
		HITEC			LOCSTEP [9]		[7]			Our Approach		
		# Det	# Vec	Exe Time	# Det	# Vec	# Det	# Vec	Exe Time	# Det	# Vec	Exe Time
s382	21	301	1,463	90.0	**364**	5,354	**364**	757	6.6	**364**	568	7.2
s400	21	382	4,309	72.0	374	5,354	382	721	8.9	**383**	610	9.5
s1238	18	**1,283**	475	0.1	1,268	9,409	1,274	302	285	**1,283**	260	28.6
s1423	74	776	177	834.0	1,274	9,616	1,415	1,334	147.0	**1,416**	1,334	180.0
s5378	179	3,238	941	1104.0	3,059	7,545	3,636	1,123	212.0	**3,642**	1,479	240.0
s35932	1,728	34,902	240	-	34,812	2,408	**35,100**	208	24.0	**35,100**	215	30.0
b01	5	-	-	-	-	-	132	95	0.01	**133**	78	0.7
b12	121	-	-	-	-	-	1,541	3,645	301.75	**1,665**	6,758	124.0
b15	447	-	-	-	-	-	18,276	17,251	4,555	**18,302**	21,024	5,101
b22	709	-	-	-	-	-	30,684	31,065	16,914	**30,701**	38,312	19,105

* '-' indicates that the data was not available; no execution times were available for LOCSTEP
* Time reported in seconds

Table 2: Comparison with STRATEGATE on Large Circuits

Ckt	STRATEGATE [6] Det	STRATEGATE [6] Vec	STRATEGATE [6] Time	Ours Det	Ours Vec	Ours Time
s1423	1,414	3,934	4,572 sec	**1,416**	1,334	180.0 sec
s5378	3,639	11,571	136,080 sec	**3,642**	1,479	240.0 sec
s35932	**35,100**	257	39,240 sec	**35,100**	215	30.0 sec
b12	1,488	33,113	4 days	**1,665**	6,758	124.0 sec
b15	17,921	83,465	7 days	**18,302**	21,024	5,101 sec
b22	**30,788**	22,059	3 days	30,701	38,312	19,105 sec

each of the state subspaces. These features require only logic simulation, thus making the proposed approach efficient. In addition, because we limit the sizes of the state groups, the STGs will remain to be small, allowing our method to be applicable to very large sequential circuits. Our experiments showed that very high fault coverages together with small test set can be achieved. In some circuits, highest fault coverages have been achieved.

References

[1] M. Niermann and J. H. Patel, "HITEC: A Test Generation Package for Sequential Circuits," *Proc. European Design Automation Conf.,* 1991, pp. 214-218.

[2] T. P. Kelsey, K. K. Saluja, and S. Y. Lee, "An Efficient Algorithm for Sequential Circuit Test Generation," *IEEE Trans. on Computers,* vol. 42, no. 11, pp. 1361-1371, Nov. 1993.

[3] I. Hamzaoglu and J. H. Patel, "New Techniques for Deterministic Test Pattern Generation," *Proc. of VTS,* 1998, pp. 446-452

[4] I. Hamzaoglu and J. H. Patel, "Deterministic test pattern generation techniques for sequential circuits," *Proc. Int'l Conf. Computer-Aided Design,* 2000, pp. 538 -543.

[5] M. Henftling, H. C. Wittmann and K. J. Antreich; "A single-path-oriented fault-effect propagation in digital circuits considering multiple-path sensitization", *Proc. Int'l Conf. Computer-Aided Design,* 2002, pp. 304 -309.

[6] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Sequential Circuit Test Generation Using Dynamic State Traversal," *Proc. European Design and Test Conf.,* 1997, pp. 22-28.

[7] S. Sheng and M. S. Hsiao, "Efficient Sequential Test Generation Based on Logic Simulation", *IEEE Design and Test of Computers,* vol. 19, no. 5, pp. 56-64, 2002

[8] D. G. Saab, Y. G. Saab, and J. A. Abraham, "CRIS: A Test Cultivation Program for Sequential VLSI Circuits," *Proc. Int'l Conf. Computer-Aided Design,* 1992, pp. 216-219.

[9] I. Pomeranz and S. M. Reddy, "LOCSTEP: A Logic-Simulation-Based Test Generation Proce-

dure," *IEEE Trans. CAD of Integrated Circuits and Systems,* vol. 16, no. 5, May 1997, pp. 544-554.

[10] E. M. Rudnick et al., "Application of Simple Genetic Algorithms to Sequential Circuit Test Generation," *Proc. European Design and Test Conf.,* 1994, pp. 40-45.

[11] R. Guo, I. Pomeranz, and S. M. Reddy, "A Fault Simulation Based Test Pattern Generator for Synchronous Sequential Circuits," *Proc. VLSI Test Symp.,* 1999, pp. 260-267.

[12] A. Giani et al., "Efficient Spectral Techniques for Sequential ATPG," *Proc. IEEE Design Automation and Test in Europe Conf.,* 2001, pp. 204-208.

[13] S. Goren and F. J. Ferguson, "Testing Finite State Machines Based on a Structural Coverage Metric", *Proc. Int'l Test Conf.,* 2002, pp. 773-780.

[14] M. Wedler et al., "Improving Structural FSM Traversal by Constraint-Satisfying Logic Simulation", *Proc. IEEE Computer Society Annual Symp. VLSI,* 2002.

[15] F. Brglez, D. Bryan, and K. Kozminski, Combinational profiles of sequential benchmark circuits, *Int. Symp. on Circuits and Systems,* 1989, pp. 1929-1934.

[16] S. Davidson and Panelists, ITC 99 benchmark circuits- preliminary results, *Proc. Int'l Test Conf.,* 1999, pp. 1125.