SEARCH STATE EQUIVALENCE FOR REDUNDANCY IDENTIFICATION AND TEST GENERATION

John Giraldi
IBM Corporation
PO Box 950
Poughkeepsie, N.Y. 12602-0950

Michael L. Bushnell
ECE Dept., Rutgers University
P.O. Box 909
Piscataway, N.J. 08855-0909

ABSTRACT

We present new extensions to the **EST**[1] *algorithm, which accelerates combinational circuit* Redundancy Identification *and* Automatic Test Pattern Generation (*ATPG*) *algorithms, in particular* **SOCRATES**. **EST** *detects equivalent* search states, *which are saved for all faults during ATPG. The* search space *is reduced by using learned* Search State *equivalences to detect previously-encountered search states (possibly from prior faults) and to make internal node assignments. We present two extensions to* **EST**. *The first ensures that each portion of the ATPG search space is explored only once. The second applies* headline objectives *in parallel, rather than serially. For the 1985 ISCAS combinational benchmarks,* **EST** *accelerates* **SOCRATES** *by* 6.53 *times, when all faults are targeted, and by* 5.81 *times, when used with* random pattern generation, fault simulation *and* fault dropping. *This acceleration was achieved with minimal memory overhead.*

INTRODUCTION

We are entering the era of *Ultra Large Scale Integrated* circuits (*ULSI*), which are expected to have as many as 10 million transistors on a single chip by the end of the century. Future mainframe computers may have as many as one million logic gates. Because of these dramatic increases in circuit size and the decreasing controllability and observability of internal circuit nodes, testing costs now exceed 50 per cent of the total chip cost. Design for testability techniques, scan design[4] and *built-in self-testing* (*BIST*)[10] techniques have been developed to deal with these problems. Scan design reduces the general sequential circuit test generation problem to a combinational circuit test generation problem. BIST techniques replace external *automatic test equipment* and test patterns with internal circuitry that generates a pseudo-random sequence of test patterns, exercises the circuit and compresses the output response to see whether the circuit is faulty. Regrettably, BIST requires that the circuit-under-test have no redundant logic or faults, which, in turn, requires the existence of an extraordinarily fast combinational circuit *redundancy identification* (*RI*) algorithm.

Automatic Test Pattern Generation (*ATPG*) for combinational circuits has been extensively studied. Roth[12] first formulated the path-sensitization techniques for testing, Goel[8] introduced backtracing, Fujiwara[5] introduced the multiple backtrace method, Kirkland and Mercer[9] applied dominator theory to fault effect propagation and Schulz[13] pioneered the use of learning techniques to reduce the number of required implications. Here, we are reporting extensions to EST[7][1], a learning algorithm for combinational circuit redundancy identification and test generation. The EST project made these contributions:

- We characterize the search state of the ATPG algorithm using the *E-frontier*, a cut-set of the circuit induced by a set of *primary input* (*PI*) assignments. The frontier is a partition between the circuit part labeled with 0, 1, D and $\overline{D}$ signals and the part labeled with X signals.
- We showed that matching of E-frontiers could be used to cause early backup from infeasible searches or early search termination with a test pattern.
- EST was the first test pattern generator to use knowledge about the search space for a prior fault to accelerate search for a test of the current fault.
- EST was the first ATPG program to use its knowledge of opportunistically-discovered redundant faults in the circuit to further reduce the search space for later faults.

EST accelerated the **TOPS**[9] ATPG algorithm 328 times and enabled it to handle all faults in the ISCAS '85 benchmarks[2]. **EST** accelerates the **SOCRATES**[13] algorithm 5.81 times on the same benchmarks.

Results are presented for **EST** running with **SOCRATES**. We also introduce two new test pat-

[1] Acronym for *E*quivalent *ST*ate Hashing algorithm.

INTERNATIONAL TEST CONFERENCE 1991

tern generation heuristics and search space management techniques that simultaneously accelerate **EST** and further reduce its memory requirements by dramatically reducing the search space.

EST characterizes and learns its search space, opportunistically discovers equivalent search states and uses this knowledge to backup earlier than other algorithms can to avoid previously-searched areas of the space. The implication stack contents define the search space traversed by the ATPG algorithm. A partial simulation[15] of the implication stack contents defines the *ATPG Search State*[7]. When fault simulation fails to identify the current fault as tested, **EST** uses E-frontiers to find test patterns for the current fault using tests for previous faults. When we match an E-frontier for the current fault against a learned E-frontier, which represents an infeasible search space, we replace the implication stack contents with the E-frontier and force a backup. We will show that this operation is equivalent to the Boolean *contra-positive* and causes **EST** to avoid ever reentering large infeasible search areas that were previously explored.

The second contribution is an improved multiple backtrace strategy. The multiple backtrace of **FAN**[5] and **SOCRATES** identifies many *fanout* and *headline* objectives that are applied to the implication stack serially. Instead, our heuristic identifies headline objectives as soon as possible and immediately applies them in parallel to the implication stack. We can identify test patterns earlier than **FAN** or **SOCRATES**, which waste time with serial implications.

We first present background material to define the *ATPG Search State*, the E-frontier and the early search termination procedure. We then present new E-frontier heuristics for maintaining the implication stack, a new use of multiple backtrace for immediate application of headline objectives and finally we present the results of these two new features for the combined algorithms **EST** and **SOCRATES**.

BACKGROUND

We first define our terminology, show how we characterize the redundancy identification search space and illustrate opportunistic recognition of equivalent search states with an example.

ATPG Search State. Figure 1, the *ATPG Decision Tree* (*ATPG-DT*), represents the test pattern search space for fault i of a conventional combinational branch-and-bound **ATPG** algorithm. Tree nodes correspond to the *ATPG Search States*[7] resulting from *partial simulation* of the *implication stack* contents, which consist of primary input and internal circuit node assignments. A path from the root to any internal node represents an implication stack assignment that generates the node (state). Inconsistent implication stack assignments resulting in backups are represented by rectangular tree leaves. Unproductive decisions by the algorithm correspond to edges only on paths to backup leaves. Algorithm performance is dramatically improved by minimizing the number of backups and unproductive decisions between backups. Circular leaves represent valid test patterns. Redundant faults have an ATPG-DT with no test pattern leaves.

EST uses ATPG search states to discover the same information that appears in a *Binary Decision Diagram* (*BDD*), while avoiding the effort of BDD computation. The *ATPG Binary Decision Diagram* (*ATPG-BDD*) for fault i of Figure 1 is represented by the ATPG-DT and the cross edge from node C' to C for fault i that is added after search states C' and C in the ATPG-DT are recognized as equivalent. We avoid exploring the sub-graph below node C', because the cross edge indicates that all leaves in this sub-graph represent backups. Therefore, we can backup and update the implication stack earlier than other algorithms.

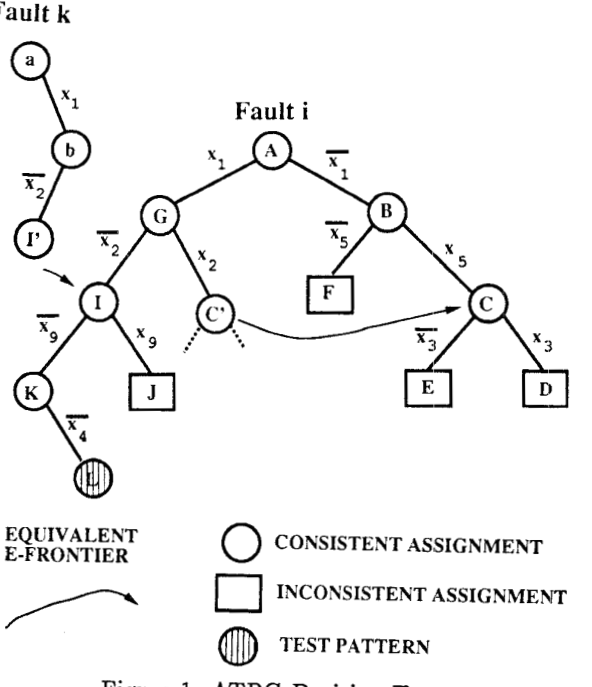

Figure 1: ATPG Decision Tree

Search states are opportunistically discovered and stored in an open hash table as *E-frontiers*[7], which uniquely identify an ATPG search state. An E-frontier is a complete, consistently-labeled circuit cut-set that is induced by the signal labelings in the implication

stack, which may contain either primary input or internal node values. Wei[15] first used partial simulation of the implication stack for logic verification. In a five-valued algebra, the frontier partitions the circuit into two parts: one with known signal values (0, 1, D, $\overline{D}$) and one with X values. In a nine-valued algebra[11], the second part may also contain the labelings $G0$ $(0/X)$, $G1$ $(1/X)$, $F0$ $(X/0)$ and $F1$ $(X/1)$. *Objectives* are generated during ATPG search, and represent required values, as yet unjustified, for signals in order to support previously-selected gate sensitization and fault path propagation decisions. Frontiers are generated during *X-path-check*, a depth-first search procedure that traverses the circuit graph from *primary outputs* ($PO's$) and internal node objectives to identify the current D-frontier and viable fault propagation paths from the fault site to the PO's. We perform *X-path-check* after we determine all implications from the implication stack contents. It is entirely possible for a frontier to contain an unjustifiable objective, which will be discovered when the ATPG algorithm later chooses another PI value that makes the inconsistency caused by the unjustifiable objective evident. E-frontiers in Figure 2 represent cut-sets using a five-valued algebra. Note that the frontiers of Figures 2b and c are equivalent but have different primary input assignments. E-frontiers for sensitized faults include D-frontiers as subsets. We use a *gate-oriented* representation of the E-frontier, which contains gate numbers and values, to save memory. One extra field is reserved in each E-frontier to represent the case where a fanout branch value on the frontier differs from its associated fanout stem value.

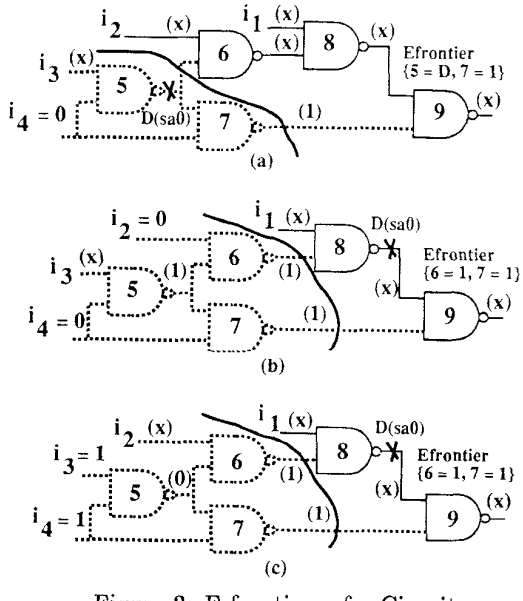

Figure 2: E-frontiers of a Circuit

EST never constructs an ATPG-BDD; instead, EST dynamically discovers BDD cross links through opportunistic matching of E-frontiers during ATPG search. An E-frontier is formed at each decision point of the ATPG algorithm, and is checked against the hash table for matches. When a match occurs, we have discovered a cross link in a BDD between the current search state and a previously-explored search state. Henceforth, we use ATPG-BDD's to illustrate the algorithm's behavior, but the EST algorithm never computes a BDD. This approach avoids all exponential-complexity algorithms to compute BDD's. The computing effort to form an E-frontier is proportional to some fraction of the number of gates in the circuit-under-test and hashing is a constant time algorithm. We do not even represent a BDD in the EST program data structures. Instead, we represent the E-frontier cut sets as arrays of (gate number, gate labeling) pairs and we hash the data in these arrays into a hash key, which determines the bucket of the hash table that stores the E-frontier. Other methods[3, 6] require additional overhead to precompute both good and failing machine BDD's, and force a predetermined expansion order for PI's during ATPG. Instead, we make use of the better PI ordering dynamically determined by a highly-effective multiple backtrace procedure[5, 13]. Since the complexity of BDD computation is worse than that of ATPG[1, 6, 3], our method is very attractive.

Early Search Path Termination. E-frontiers allow for early identification of search path termination conditions. Consider the ATPG-BDD of Figure 1 representing the search space for fault i. Node letters represent search states (E-frontiers) at each decision point in the ATPG algorithm. Search starts at the root node A and proceeds in depth-first search order up to node G with implications resulting in inconsistent assignments at nodes D, E and F. At node G, $x2$ is set to 1 with implication resulting in search state C'. We hash the E-frontier at state C', determine that it is equivalent to the E-frontier of inconsistent state C and backup earlier than other algorithms, which waste time exploring implications from C'.

Figure 1 shows another early search termination. Test pattern search for fault k uses the ATPG-BDD of the search for the previous fault, i. When $x2$ is set to 0 in state b for fault k, we create search state I', hash its E-frontier and find it to be equivalent to the E-frontier of node I for fault i. For sensitized faults, a D-frontier is a subset of the E-frontier and therefore all subsequent PI assignments for fault k will be identical to those previously made for fault i. Since node I for fault i is also sensitized, the unassigned PI's for fault k are set to those corresponding PI assignments for fault i and search terminates immediately with a

test pattern for fault k. Circuit decompositions represented by equivalent E-frontiers are identical, so ATPG search produces identical results for the decompositions and we need not repeat the search after an E-frontier match. Prior work[7] shows that this procedure generates tests for faults that cannot be dropped by fault simulation.

IMPLICATIONS FROM FRONTIERS

E-frontiers are most useful for characterizing a known infeasible search space K and setting the ATPG algorithm's implication stack so that no sequence of decision variables will ever be generated to lead the algorithm into region K again. This technique requires a base ATPG algorithm such as **FAN**[5], which reduces the ATPG search space by including internal circuit node assignments on the implication stack. Many different PI assignments may imply a single set of internal node assignments g_1, as in $PI_1 \rightarrow g_1$, $PI_i \rightarrow g_1$ and $PI_n \rightarrow g_1$. When the stack value g_1 is changed to $\overline{g_1}$ for a backup, we use the *contra-positive* to show that $\overline{g_1} \rightarrow \overline{PI_1}$, $\overline{g_1} \rightarrow \overline{PI_i}$ and $\overline{g_1} \rightarrow \overline{PI_n}$. PI assignments implying g_1 in the search space are avoided as long as $\overline{g_1}$ remains on the implication stack, so each infeasible region is searched only once.

To implement our method, we extend the use of internal circuit node assignments on the implication stack to include the implications from equivalent E-frontiers. E-frontier E_a defines a non-solution area of the ATPG search space represented by the area K below node E_a in the ATPG-BDD of Figure 3. Different sets of PI assignments imply the same E-frontier E_a, as in $P_a \rightarrow E_a$ and $P_b \rightarrow E_a$. In general, there are many different PI assignments P_k that can lead the ATPG algorithm back into region K. Each node N_k in K is generated by the partial simulation of two assignments: (1) the internal circuit node values of E_a and (2) those remaining PI values represented by edges from node E_a to N_k. Each node N_k in K has an E-frontier E_k. Let P_a and P_b represent previous and current PI assignments, respectively, that each produce E-frontier E_a. We can avoid generating any subsequent PI assignment P_l, which also produces E_a, and any P_k, which produces a node N_k in K, by replacing the implication stack contents with the internal circuit node assignments of E_a, when E_a produced by P_b is determined to be equivalent to E_a previously produced by P_a. From implication stack assignments of P_l and P_k we see that

$$P_l \rightarrow E_a, \quad P_k \rightarrow E_k \rightarrow E_a. \tag{1}$$

To prevent later reentry into K, **EST** replaces its implication stack contents with E_a when it encounters E_a for the second time during search. After substitution, **EST** immediately generates E_a again and backs up, because frontier hashing indicates that E_a represents an infeasible search region. The backup procedure generates the assignments $\overline{E_a}$ on the stack. From the Boolean contra-positive, we see that:

$$\overline{E_a} \rightarrow \overline{P_l} \text{ and } \overline{E_a} \rightarrow \overline{E_k} \rightarrow \overline{P_k}, \tag{2}$$

for all P_k that cause entry into infeasible search space K. We have shown that every node in K, including E_a, has the internal circuit node assignments of E_a. Since $\overline{E_a}$ is on the implication stack, the non-solution area identified by K will be avoided. In addition, a valid test pattern will have internal circuit node assignments from $\overline{E_a}$, since all those PI assignments that cause the internal circuit node assignments of E_a are in a known non-solution area of the search space. Therefore, this algorithm searches each infeasible search area in the ATPG-BDD only once whereas other algorithms may search infeasible areas repeatedly.

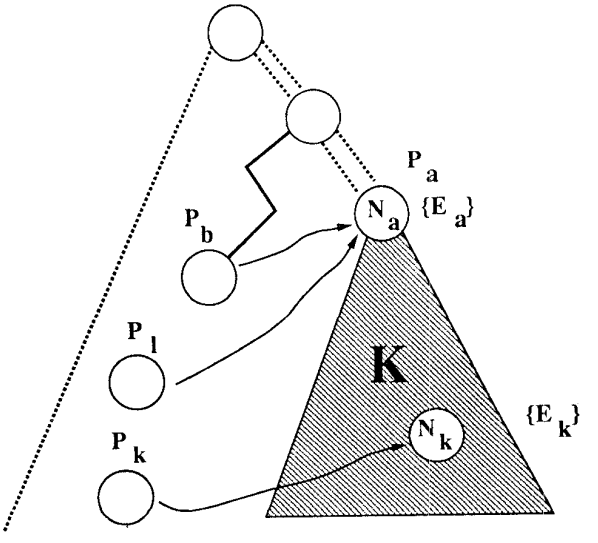

Figure 3: ATPG Search Binary Decision Diagram

It should be noted that we do not use the entire E-frontier when we replace the implication stack with the E-frontier. When replacement occurs, the internal node assignments of the E-frontier are much more important than the PI assignments. The flow chart of Figure 4 shows the algorithm for determining the new implication stack contents from the internal node assignments of an equivalent E-frontier, which is known to represent a non-solution search space for the current fault. Step 1 determines whether any internal node values are left in the E-frontier. If so, one is selected. We reject any node that is a PI, since Fujiwara and Shimono[5] have shown that internal node values imply many PI assignments. We reject internal nodes on a path from the fault site to

a PO (Steps 3 and 4) to avoid blocking any fault propagation path. We accept internal nodes whose output is due to an input with a *controlling* value (Step 5) or to a *learned implication* (Step 6). If the matched E-frontier list is empty and the new implication stack is also empty (Step 8), then we back up the implication stack (Step 9). Otherwise, we sort the new internal circuit nodes by gate level on the implication stack as if they were chosen by the *fanout objective* method of FAN[5]. Stack entries that are least likely to change are assigned to circuit nodes closest to the PO's. Nodes closest to the PI's are put at the top of the stack and are most likely to be changed.

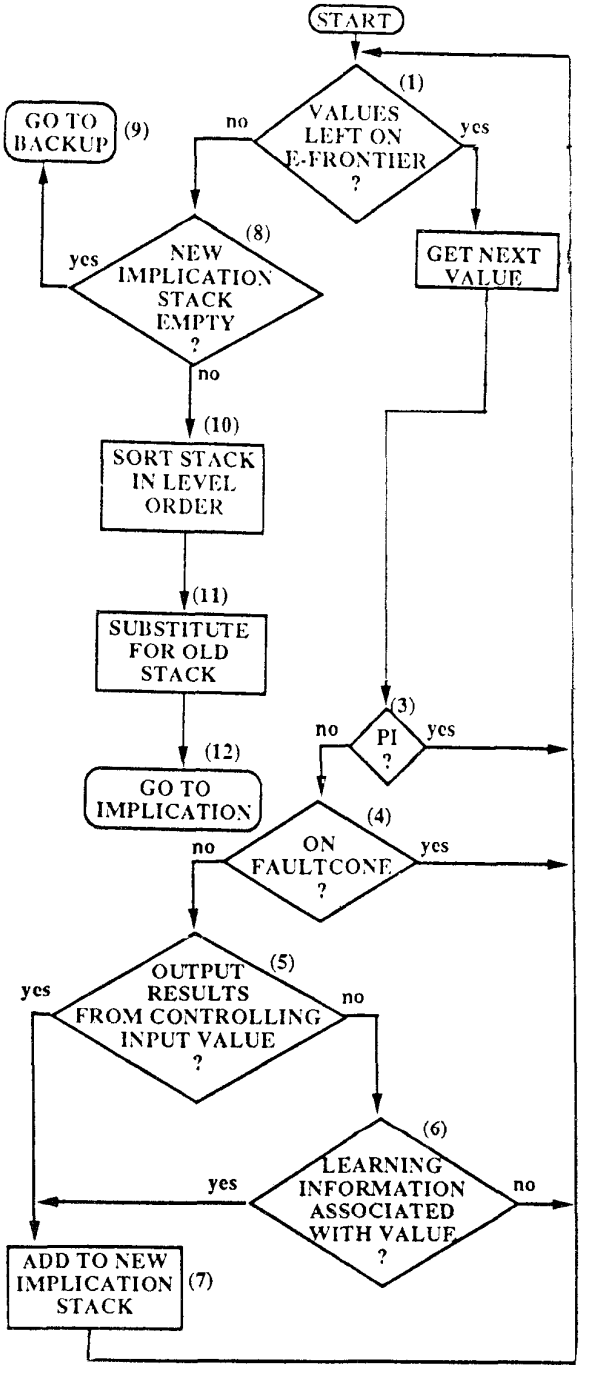

Figure 4: E-frontier Substitution Algorithm

Our decision to include circuit nodes with controlling values on an input (Step 5) in the implication stack is similar to the choice made by **DIJUST**[14], which implicitly partitions the solution areas. **DIJUST** is part of **PROTEUS**[15] and uses disjoint cubes of an unjustified line during implications to justify an unjustified line. Unlike **DIJUST**, which chooses the possible solution cubes based on a static heuristic, we can dynamically identify the *disjoint* condition (the E-frontier) defining a large non-solution search space area. We also choose internal nodes from the E-frontier that identify *learned implications*, since Schulz has shown that learning occurs many times for a circuit node and that each learned implication is useful as long as the circuit node and its value remain on the implication stack.

IMPLICATION EXAMPLE

Figure 5a represents an example circuit for which E-frontier substitution is performed. The circuit has a sa1 PI fault with $\overline{x_5}$ as a unique sensitization[5]. The algorithm selects variables x_3, x_4 and x_1 in search order, and backs up at ATPG-BDD node K_1 in Figure 5b because gate g_{12}, the only observable PO, becomes a constant 1. The algorithm backs up by changing the assignment x_1 to $\overline{x_1}$, which also leads to a backup at node K_2 for the same reason. The algorithm rejects both $\overline{x_1}$ and x_4 and adds first $\overline{x_4}$ and then $\overline{x_2}$ to the implication stack. The E-frontiers generated after each implication are associated with corresponding ATPG-BDD nodes as shown in Figure 5d. Partial simulation results in an E-frontier equivalent to that at node K_3. The algorithm replaces all current implication stack values with gate values g_8 and $\overline{g_9}$ from the E-frontier for node K_3. The new ATPG-BDD for this implication stack is shown in Figure 5c. As expected, partial simulation of the new implication stack results in a backup at node K_6 due to an equivalent E-frontier K_3 (see Figure 5d). The algorithm rejects gate value $\overline{g_9}$ and replaces it with g_9 to prevent reentry into the infeasible region defined by K_3. (We assume that there are additional undiscovered ways to reenter this infeasible region.) The fault is now propagated to a PO. The algorithm justifies g_9 by choosing new implication stack values of x_4 and $\overline{x_3}$, respectively, resulting in a test pattern for the fault, as shown in Figure 5e. We need not justify g_8 because its effects are cut off at gate g_{11}. By definition, the test pattern leaf for the search space lies outside of the in-

feasible search area defined by the E-frontier at node K_3 in Figure 5b.

MULTIPLE BACKTRACE AND IMPLICATION

We now present an extension to multiple backtrace that accelerates ATPG search. For many faults, it is possible to generate the target fault test pattern from the first multiple backtrace rather than by using many multiple backtraces[5, 13]. In the following discussion, a *headline* refers to a circuit node that separates the circuit-under-test into two parts controlled by disjoint sets of primary inputs. *Fanout* (*headline*) *objectives* are objectives for fanout stems (headlines). In our enhanced procedure, the first set of *initial objectives* is backtraced

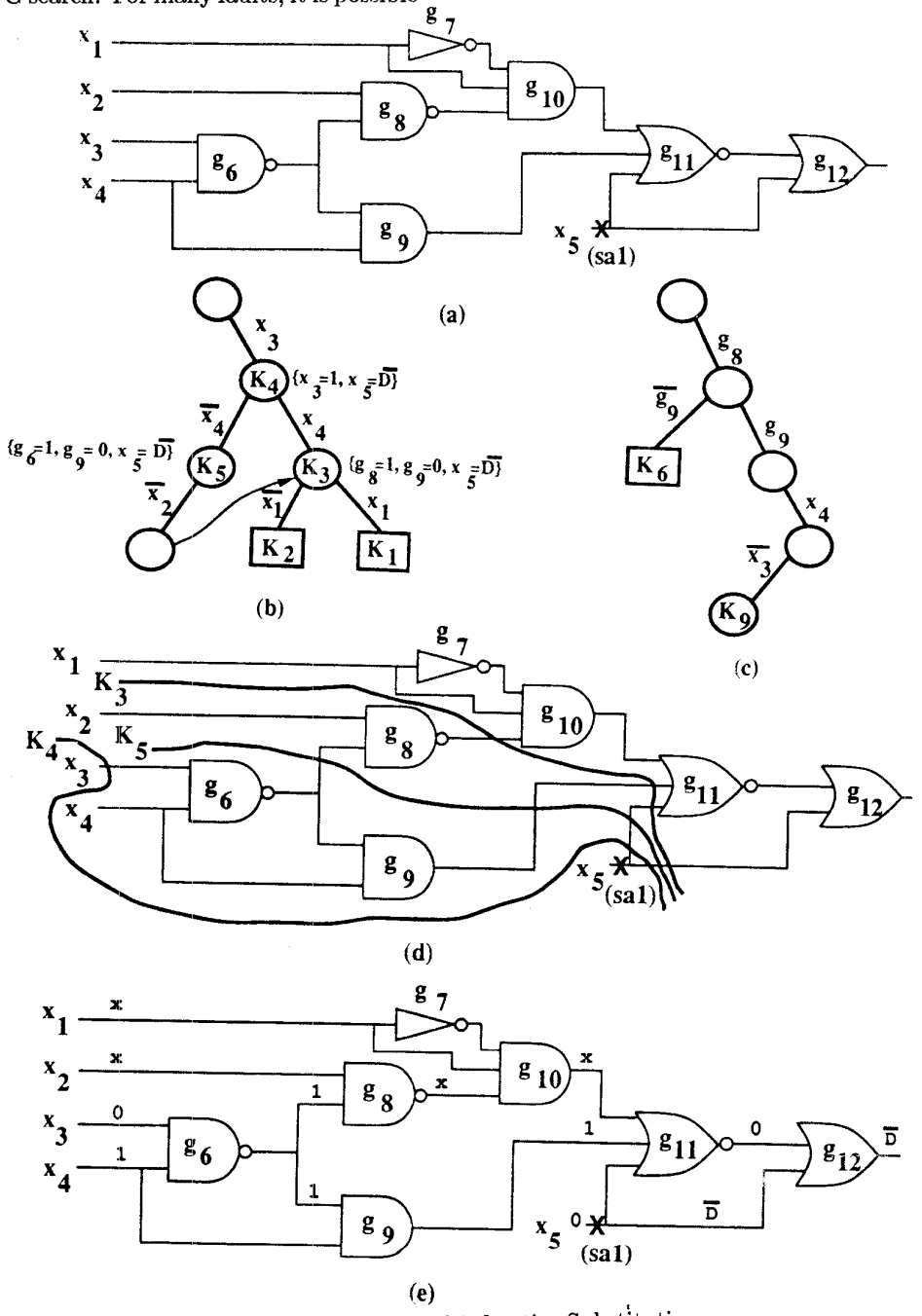

Figure 5: Example of E-frontier Substitution

to *headlines* without stopping at *fanout objectives*. Unlike **FAN** and **SOCRATES**, **EST** applies all resulting headline assignments simultaneously. We reject all assignments if they result in an inconsistency and alternatively proceed with test generation using the conventional multiple backtrace. When all objectives of the multiple backtrace are headlines, we simultaneously add all of them to the implication stack and imply, using Schulz's criteria[13] to order the headlines on the stack. If this results in an inconsistency, we remove headline objectives from the top of the stack without backing up until no inconsistency remains. Test generation then proceeds as before. These *complete multiple backtrace* and *improved implication* procedures generate test patterns earlier, on average, than **SOCRATES** does, since those headlines not resulting in backups are applied simultaneously during the earliest multiple backtrace in which they are generated. Mistakes made by this procedure are not always unmade by the base ATPG algorithm, but instead the E-frontier mechanism enables the algorithm to "branch" out of non-solution areas with little penalty to overall computing time.

Table 1: Characteristics of the ISCAS '85 Benchmarks

#	# Faults	# PI's	# Gates
c432	528	36	160
c499	758	41	202
c880	1017	60	383
c1355	1646	41	546
c1908	2054	33	880
c2670	3006	233	1193
c3540	3935	50	1669
c5315	5878	178	2307
c6288	7760	32	2406
c7552	8197	207	3512

Sometimes headline and fanout objectives not contributing to fault justification or propagation are selected for the implication stack by Fujiwara's procedure[5]. This leads to additional nodes in the decision graph and additional unjustified lines to be explored during multiple backtrace. The **FAN** and **SOCRATES** algorithms use significant computing to reduce the search space by examining all objectives and eliminating non-contributing ones. However, **EST** does not eliminate these non-contributing objectives. Such additional nodes in the decision tree are instead quickly recognized as equivalent to other nodes by equivalent E-frontier recognition, a considerably less time-consuming operation than the **SOCRATES** method.

RESULTS

The **EST** ATPG algorithm[7], the *complete multiple backtrace* and the *improved implication* procedure were implemented with the **SOCRATES** algorithm[13] in the *C* programming language. We compared the resulting performance to that of **SOCRATES** alone. We present results with all faults targeted and no *fault dropping* as well as results incorporating *random test pattern generation* (*RPG*), *fault simulation* (*FSIM*) and *fault dropping*. The experiments comparing **EST** and **SOCRATES** ran on a IBM RS/6000 computer, a 29 MIPS machine with 32 Megabytes of memory, using test cases from the ISCAS '85 benchmarks[2]. The **SOCRATES** and **EST** implementations do not include Schulz's dynamic learning stage DYN_2[13], which benefited only two faults in benchmark *c432*. However, **EST** is able to process these two faults without difficulty. Therefore, we dropped *c432* from the overall speedup comparisons. Table 1 summarizes the ISCAS '85 benchmark characteristics. We note some

Table 2: **EST** Performance with All Faults Targeted

CKT		SOCRATES					EST									
#	Redundant	Implications	Backups	CPU Time (sec.)	Memory (meg.)	# Test Patterns	Implications	Backups	Early Backups	ATPG Faults Total	ATPG Faults Hash Test	CPU Time (sec.)	Memory (meg.)	# Test Patterns	Speed-up Ratio	Memory Increase Ratio
c432	4	33039	12707	221.8	0.71	201	2863	917	732	528	19	34.3	1.21	266		
c499	8	29745	658	295.3	1.13	446	12090	387	43	758	202	141.9	2.61	486	2.08	2.31
c880	0	10773	29	90.0	0.98	439	3612	18	0	1017	288	43.1	1.56	472	2.09	1.59
c1355	8	58008	382	2742.3	2.12	471	4910	43	0	1646	950	135.9	2.87	454	20.18	1.35
c1908	9	33438	98	1005.6	2.58	846	10625	142	10	2054	959	255.5	3.81	837	3.94	1.48
c2670	117	38560	952	1325.3	2.15	851	8420	395	74	3006	1281	230.7	3.29	872	5.74	1.53
c3540	137	42507	978	1539.1	3.23	1470	17229	867	87	3935	1247	643.8	6.30	1552	2.39	1.95
c5315	59	71061	429	2927.3	4.49	2059	18873	410	73	5878	2752	724.8	6.46	2117	4.04	1.44
c6288	34	242760	30924	29659.9	6.91	2753	40912	1401	157	7760	4413	4075.0	15.23	1995	7.28	2.20
c7552	131	195586	5085	16574.7	5.82	2984	35180	1525	227	8197	5061	1504.7	9.70	3285	11.02	1.67
Ave.															6.53	1.72

distortions of our CPU time measurements. **EST** uses a table-driven, interpretive implication procedure and a table-driven, interpretive *single-pattern-single-fault* fault simulator. Naturally, our CPU times could be significantly shortened if we instead used efficient implication and fault simulation algorithms, particularly a *parallel-pattern-single-fault* method. However, the identical distortions affect our implementation of **SOCRATES**, so the *Speedup Ratio* and *Memory Usage* results reported below are still valid. The slow table-driven implication and backtracing procedures, in particular, in this implementation of **SOCRATES** cause **SOCRATES** CPU times to be slower in this experiment, given the hardware used, than the times reported earlier by Schulz[13]. Nonetheless, the results remain valid because **EST** is an algorithm to speed up any test-pattern generator, and because the experiments were run on the combination of **EST** and **SOCRATES**. Any implementation improvements or optimizations that benefit the **SOCRATES** times here will similarly benefit the **EST** times.

Table 2 shows the relative performance of **EST** and **SOCRATES** with all faults targeted and a backup limit of 200. Both algorithms identified all redundant faults, with no aborted faults, using the simplistically-reduced fault lists for the benchmarks. *Implications* gives the number of times the implication procedure ran after the implication stack changed; **EST** performed 79.0 % fewer implications than **SOCRATES**. **EST** performed 86.9 % fewer backups than **SOCRATES**. The *Speedup Ratio* shows that **EST** ran, on average, 6.53 times faster than **SOCRATES** and 20.18 times faster just on benchmark *c*1355. **EST** used at most 15.23 Megabytes of memory, 2.2 times as much as **SOCRATES**. *Early Backups* gives the number of times **EST** either substituted an E-frontier into the implication stack or backed up due to an E-frontier match. *Hash Test* is the number of faults (out of the total given under *ATPG Faults*) for which a test pattern was immediately formed after an equivalent E-frontier match into the ATPG-BDD for a previous fault. FSIM would not have found a test for these faults using the previous fault's test pattern. **EST** produced 2.0 % fewer test patterns than **SOCRATES**.

Table 3 compares the performance of **EST** and **SOCRATES** when using RPG, FSIM using a single-pattern single-fault fault simulator and *fault dropping* with a backup limit of 200. The RPG stage ends when 64 consecutive patterns do not produce a test pattern for the remaining faults. Both **EST** and **SOCRATES** were able to identify all redundant faults in the simplistically-reduced fault lists for the benchmarks. All terminology in the table is the same as before. *ATPG Faults* is the number of faults processed by the ATPG search algorithm. The discrepancies in the *ATPG Faults* columns between **EST** and **SOCRATES** arise because the two algorithms compute different test patterns for the same faults, and therefore fault dropping causes the **EST** search algorithm to process fewer faults than **SOCRATES**. Since this experiment produced exactly the same numbers of redundant faults as for the previous experiment (see Table 2), the redundant fault counts are not repeated. **EST** performed 54.5 % fewer total implications than **SOCRATES**, and backed up 64.5 % less often. No test pattern search for a fault was aborted by either algorithm. Table 4 gives the speedup and increased memory usage of **EST**, when compared to **SOCRATES**, with both algorithms using RPG, FSIM and fault dropping. **EST** ran, on average, 5.81 times faster than **SOCRATES** with the greatest speedup of 22.87 on benchmark *c*1355. **EST** used at most 5.64 Megabytes of memory, only 1.04 times as much

Table 3: **EST** Performance with Random Pattern Testing and Fault Simulation

CKT		SOCRATES							EST								
#	Static Learning Time (sec.)	Total ATPG Faults	Implications	Backups	CPU Time (sec.) RPG-FSIM	CPU Time (sec.) ATPG	Mem. (meg.)	# Test Patterns	ATPG Faults Total	ATPG Faults Hash Test	Implications	Backups	Early Backups	CPU Time (sec.) RPG-FSIM	CPU Time (sec.) ATPG	Mem. (meg.)	# Test Patterns
c432	.43	6	24650	12141	2.8	108.6	0.67	79	6	0	540	221	143	2.8	4.3	1.04	79
c499	1.1	11	107	0	4.7	0.8	1.04	67	11	0	23	0	0	4.7	0.2	1.27	67
c880	1.4	15	137	1	6.4	1.2	0.83	120	15	1	66	1	0	6.4	0.8	1.07	120
c1355	5.2	29	695	0	12.7	34.3	1.86	109	29	0	86	0	0	12.7	1.5	2.10	109
c1908	9.1	31	221	1	43.2	6.9	2.25	187	32	1	214	3	0	43.1	3.5	2.49	188
c2670	14.7	225	2781	8	44.8	71.6	1.66	211	212	36	360	5	3	44.0	7.7	1.93	198
c3540	26.7	169	275	2	68.0	6.3	2.51	259	168	0	208	2	0	67.2	4.9	2.78	258
c5315	42.0	80	559	42	80.8	23.2	3.37	223	80	10	309	28	7	80.8	6.0	3.60	223
c6288	52.6	34	34	0	73.3	1.0	5.42	63	34	0	17	0	0	73.1	0.5	5.64	63
c7552	88.9	285	5653	524	177.1	622.5	4.18	381	274	94	3472	164	14	176.0	112.6	4.56	370

as **SOCRATES**. **EST** produced 1.5 % fewer test patterns than **SOCRATES**. The absolute numbers of patterns are higher here than for Schulz's experiments[13], because we do not fault simulate the entire test pattern set in the reverse order of pattern generation after ATPG is completed. This operation would drop additional, unnecessary patterns.

Table 4: Increased Speed and Memory of **EST** Compared to **SOCRATES**

Circuit #	Speed-up Ratio	Memory Increase Ratio	Fault Coverage
c432			99.24
c499	4.0	1.22	98.94
c880	1.5	1.29	100.00
c1355	22.87	1.13	99.51
c1908	1.97	1.11	99.56
c2670	9.30	1.16	96.11
c3540	1.29	1.11	96.52
c5315	3.87	1.07	99.00
c6288	2.0	1.04	99.56
c7552	5.53	1.09	98.40
Average	5.81	1.14	

Table 5 compares the different phases used in Schulz's implementation of the **SOCRATES** algorithm with the phases used in our implementations of **EST** and **SOCRATES**. Dynamic *DYN_1* is Schulz's operation to detect dynamic dominators. Dynamic *DYN_2* is Schulz's full dynamic learning operation, which is not used in **EST** with **SOCRATES**. The probabilistic backtrace of **EST** is invoked when the number of 0's and 1's required at a fanout stem during the multiple backtrace are equal. In this case, **EST** generates a random number which it uses to determine the fanout stem assignment. This procedure is superior to that of **SOCRATES**, which always assigns a 0 when the votes for 0's and 1's are tied.

CONCLUSIONS

We have presented new heuristics for a redundancy identification and test pattern generation algorithm, **EST**, which reduce the combinational ATPG search space further than **SOCRATES** can. **EST** uses equivalent search states identified during ATPG search to back up and form test patterns earlier than other algorithms. The first new heuristic uses E-frontiers of equivalent search states to modify the implication stack and avoid potentially-traversed as well as previously-traversed infeasible search spaces. This heuristic enables **EST** to search each region of the test pattern search space only once, if given sufficient memory. The second new heuristic applies headline objectives from the multiple backtrace procedure simultaneously and as soon as possible to the implication stack to identify test patterns earlier than conventional methods. For the 1985 ISCAS combinational benchmarks, **EST** accelerates the **SOCRATES** ATPG algorithm on average by 6.53 times, with minimal memory overhead, when all faults are targeted and 5.81 times, when used with random pattern generation, fault simulation and fault dropping. **EST** reduced the number of implications by 79.0 % and the number of backups by 86.9 %, when compared with **SOCRATES**.

Table 5: Phases of the **SOCRATES** and **EST** Algorithms

SOCRATES					**EST**				
Phase	Operation	Control-lability	Observa-bility	Back-ups	Phase	Operation	Control-lability	Observa-bility	Back-ups
1	RPG and Fault Dropping			0	1	RPG and Fault Dropping			0
2	Static Learning			0	2	Static Learning			0
3	Determ. ATPG with learned imp., unique sens.	COP	LEVEL	10	3	Determ. ATPG with learned imp., unique sens.	COP	LEVEL	10
4	Dynamic *DYN_1*	COP	LEVEL	10	4	Dynamic *DYN_1*	COP	LEVEL	10
5	Dynamic *DYN_2*	COP	LEVEL	10	5	Dynamic *DYN_1*	SCOPE	LEVEL	10
					6	Dynamic *DYN_1*, Prob. Backtrace	SCOPE	LEVEL	10
					7	Dynamic *DYN_1*, Prob. Backtrace	SCOPE	LEVEL	200

ACKNOWLEDGEMENTS: We thank the IBM Corporation for support from the Resident Study Program. The research reported here was partially supported by the Center for Computer Aids for Industrial Productivity (CAIP), an Advanced Technology Center of the New Jersey Commission on Science and Technology, at Rutgers University, N.J., and by its industrial affiliates. We thank the SIGDA of the ACM and the Micro-Electronics Center of North Carolina for providing the ISCAS '85 benchmark circuits.

References

[1] S. B. Akers. Functional Testing with Binary Decision Diagrams. In *Proc. of the 8th FTCS*, pages 82–92, June 1978.

[2] F. Brglez and H. Fujiwara. A Neutral Netlist of 10 Combinational Benchmark Circuits. In *Proc. of ISCAS; Special Session on ATPG and Fault Simulation*, pages 151–158, June 1985.

[3] K. Cho and R. E. Bryant. Test Pattern Generation for Sequential MOS Circuits by Symbolic Fault Simulation. In *Proc. of the 26th DAC*, pages 418–423, June 1989.

[4] E. B. Eichelberger and T. W. Williams. A Logic Design Structure for LSI Testability. *J. Des. Aut. and Fault. Tol. Comp.*, 2:165–178, May 1978.

[5] H. Fujiwara and T. Shimono. On the Acceleration of Test Generation Algorithms. In *Proc. of the 13th FTCS*, pages 98–105, June 1983.

[6] R. K. Gaede, M. R. Mercer, K. M. Butler, and D. E. Ross. CATAPULT: Concurrent Automatic Testing Allowing Parallelization and Using Limited Topology. In *Proc. of the 25th DAC*, pages 597–600, June 1988.

[7] J. Giraldi and M. L. Bushnell. The New Frontier in Automatic Test Pattern Generation. In *Proc. of the 27th DAC*, pages 667–672, June 1990.

[8] P. Goel. An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits. *IEEE Trans. on Computers*, C-30(3):215–222, March 1981.

[9] T. Kirkland and M. R. Mercer. A Topological Search Algorithm for ATPG. In *Proc. of the 24th DAC*, pages 502–508, June 1987.

[10] B. Konemann, J. Mucha, and G. Zweihoff. Built-In Test for Complex Digital Integrated Circuits. *IEEE J. Sold-State Circuits*, SC-15:315–319, June 1980.

[11] P. Muth. A Nine-Valued Circuit Model for Test Generation. *IEEE Trans. on Computers*, C-25:630–636, June 1976.

[12] J.P. Roth, W. G. Bouricius, and P. R. Schneider. Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits. *IEEE Trans. on Computers*, EC-16(10):567–580, Oct. 1967.

[13] M. Schulz and E. Auth. Improved Deterministic Test Pattern Generation with Applications to Redundancy Identification. *IEEE Trans. on CAD*, 8(7):811–816, July 1989.

[14] R. S. Wei and A. Sangiovanni-Vincentelli. New Front End Line Justification Algorithm. In *Proc. of the ITC*, pages 350–359, Oct. 1986.

[15] R. S. Wei and A. Sangiovanni-Vincentelli. PROTEUS: A Logic Verification System for Combinational Circuits. In *Proc. of the ITC*, pages 350–359, Oct. 1986.