Dynamic Search-Space Pruning Techniques in Path Sensitization

João P. Marques Silva and Karem A. Sakallah
Department of Electrical Engineering and Computer Science
University of Michigan

Abstract— A powerful combinational path sensitization engine is required for the efficient implementation of tools for test pattern generation, timing analysis, and delay fault testing. Path sensitization can be posed as a search, in the n-dimensional Boolean space, for a consistent assignment of logic values to the circuit nodes which also satisfies a given condition. In this paper we propose and demonstrate the effectiveness of several new techniques for search-space pruning for test pattern generation. In particular, we present linear-time algorithms for dynamically identifying unique sensitization points and for dynamically maintaining reduced head line sets. In addition, we present two powerful mechanisms that drastically reduce the number of backtracks: failure-driven assertions and dependency-directed backtracking. Both mechanisms can be viewed as a form of learning while searching and have analogs in other application domains. These search pruning methods have been implemented in a generic path sensitization engine called LEAP. A test pattern generator, TG-LEAP, that uses this engine was also developed. We present experimental results that compare the effectiveness of our proposed search pruning strategies to those of PODEM, FAN, and SOCRATES. In particular, we show that LEAP is very efficient in identifying undetectable faults and in generating tests for difficult faults.

I. Introduction

Path sensitization is commonly used in test pattern generation, delay fault testing and timing analysis, and can be posed as the problem of finding a valid logic assignment to the circuit nodes that sensitizes one or more paths satisfying a particular application-specific property. Such assignments are typically found by a decision procedure that performs a directed search in the *n*-dimensional Boolean space. In recent years, most work on path sensitization has been concerned with the development of techniques for pruning the search space, particularly for test pattern generation [1, 5-9, 13-16, 21]. Some of these techniques, such as simple and multiple backtracing and the various controllability and observability measures, are *heuristic*; they may or may not lead to a reduction of the search space. On the other hand, techniques such as unique sensitization points, head lines, static/dynamic learning and search state equivalence are *non-heuristic*; if they apply, they are guaranteed to reduce the search space.

In this paper we introduce several new non-heuristic search-space pruning techniques, based on a dynamic analysis of the search process, and present experimental data that demonstrate their power for test pattern generation. We start by illustrating how *dynamic unique sensitization points* [5] can be identified in linear time; in contrast the algorithm suggested in SOCRATES [16] has worst-case quadratic time complexity. Next, we show that the notion of *head lin*es can be naturally extended to dynamic situations, allowing a reduction in the size of the set of head lines as the search process evolves. Dynamic head lines are identified, at each node in the decision tree, by a linear time algorithm.

When inconsistencies occur during the search, we provide a method for identifying nodes that must assume certain values to eliminate these inconsistencies. This is referred to as *failure-driven assertions* and can be viewed as a form of learning while searching [3]. We also introduce an algorithm to perform *dependency-directed backtracking*. In most algorithms for path sensitization such as the D-algorithm [14], PODEM [7], FAN [5], TOPS [8], SOCRATES [15] and EST [6], the search process always backtracks to the previous node in the decision tree, i.e. it performs *chronological* backtracking. In some situations, however, the search process can *provably* backtrack to some other node in the decision tree, yielding a significant reduction in the required number of backtracks while guaranteeing that a solution will be found if one exists. Our dependency-backtracking algorithm has linear time complexity and introduces negligible overhead if no backtracking is required. Dependency-directed backtracking schemes were originally proposed in [19] in an application of artificial intelligence techniques to circuit analysis.

The above search-space pruning techniques have been incorporated in a path sensitization engine called LEAP (LEvel-dependent Analysis in Path sensitization). LEAP has been used to implement a combinational timing analyzer, TA-LEAP [18] as well as a test pattern generation system, TG-LEAP which can also run customized versions of PODEM, FAN, and SOCRATES.

In the next section we introduce the basic concepts required to implement failure-driven assertions and dependency-directed backtracking. We also review concepts common to decision procedures used in path sensitization. In Section III we describe each of the new techniques, and detail the corresponding algorithmic implementation. Afterwards, we present a comprehensive set of results that illustrate the effectiveness of LEAP in identifying undetectable faults and in detecting difficult faults. In Section V directions for future research are described.

II. Definitions

The essential definitions required to describe LEAP are summarized in Fig. 1. We model a gate-level combinational circuit as a directed acyclic graph $C = (V_C, E_C)$ whose vertex set V_C corresponds to the circuit nodes and whose edge set E_C represents the input-to-output connections within the circuit gates. For typical circuits with bounded fanin, the number of edges is linearly related to the number of vertices. For such

This work was supported in part by the NSF under grant MIP-9014058 and in part by the Portuguese JNICT under project "Ciência".

31st ACM/IEEE Design Automation Conference ®

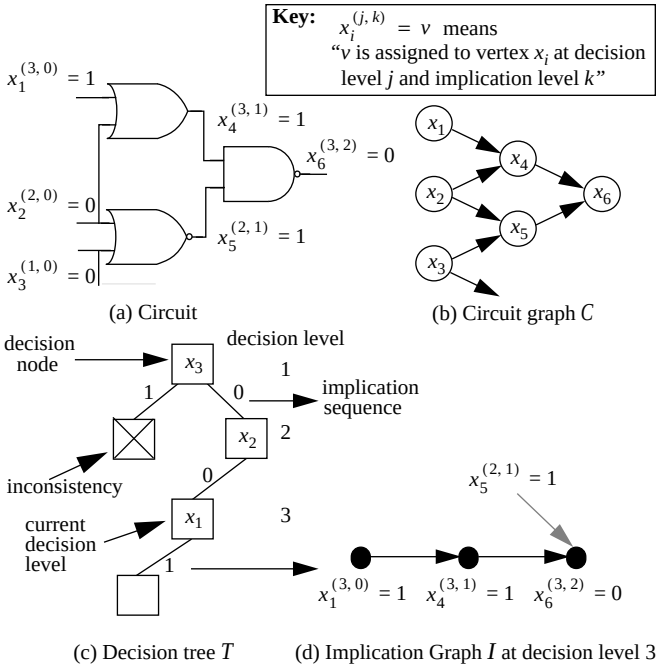

Fig. 1. Structures associated with a circuit during the search process

circuits, an algorithm whose run time is $\Theta(|V_C| + |E_C|)$ is said to have linear time complexity. Circuit vertices are denoted by labels such as x_1, x_2, etc.

In the absence of inconsistencies, the search process can be viewed as sequence of decisions each of which is, possibly, followed by an appropriate sequence of implications. A decision in this context refers to the *elective* assignment of a binary value to a given vertex in the circuit graph. An implication, on the other hand, refers to the *forced* assignment of a value to a vertex due to the current assignments of other vertices. Implications are triggered by decisions and are performed *breadth-first* through the circuit graph. The state of the search process is implicitly maintained using two dynamic data structures: a *decision tree* $T = (V_T, E_T)$ that records the decision sequence, and an *implication graph* $I = (V_I, E_I)$ that captures the cause-and-effect chains of forced assignments. A node θ in the decision tree (a *decision* node) corresponds to a circuit vertex x_i and is characterized by its *decision* level (depth) $d(\theta)$ in the tree. The root decision node is defined to be at decision level 1. Directed edges emanating from θ represent the possible binary assignments to x_i. A node ϕ in the implication graph is a predicate that denotes the forced assignment of a logic value v to a circuit vertex x_i. The predecessors of ϕ in the implication graph are referred to as its *antecedents* $Ant(\phi) \equiv \{\zeta \in V_I | (\zeta, \phi) \in E_I\}$. Assuming that the assignments $x_1 = v_1, \ldots, x_m = v_m$ are associated with antecedents $\zeta_1, \ldots, \zeta_m$, the edges directed from $Ant(\phi)$ to ϕ correspond to the implication $(x_1 = v_1) \wedge \ldots \wedge (x_m = v_m) \Rightarrow (x_i = v_i)$. Node ϕ is also characterized by two integer parameters: $d(\phi)$, the decision level of the decision node that triggered the implication sequence for ϕ, and $p(\phi)$, the *implication* level of ϕ, that are calculated according to:

$$d(\phi) = \max \{ d(\zeta) | \zeta \in Ant(\phi) \} \quad (1)$$

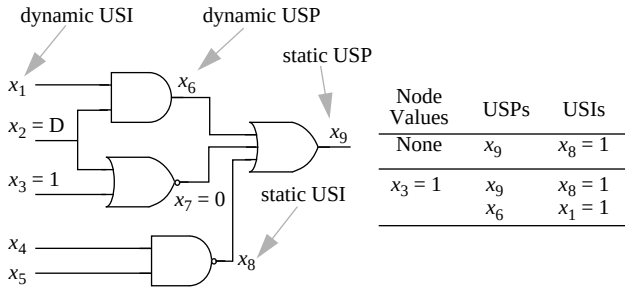

Fig. 2. USPs and USIs

$$p(\phi) = 1 + \max \{ p(\zeta) | \zeta \in Ant(\phi) \wedge d(\zeta) = d(\phi) \} \quad (2)$$

By definition, the implication level of a decision node is 0. In this paper, the notation $x_i^{(j,k)} = v$ should be interpreted to mean that circuit vertex x_i is assigned the value v at decision level j and implication level k.

Inconsistencies that arise during the search process are of two types. A *vertex inconsistency* occurs when the logic values of the inputs and output of a gate are not consistent. A *path inconsistency* occurs when there exists no *X*-path to propagate an error signal.

In the sequel we also refer to other commonly-used concepts in test pattern generation including head lines, D-frontier, J-frontier, backward/forward implications, unique sensitization points, non-local implications, etc. (e.g. see [1]).

III. Level-Dependent Analysis

A. Dynamic Unique Sensitization Points

A *unique sensitization point* (USP) is a gate that must propagate the error for a given fault to be detected [5]. This, in turn, implies that an input of this gate to which the error cannot be propagated must assume a noncontrolling value; such an implication is referred to as a *unique sensitization implication* (USI). The identification of *static* USPs was proposed in FAN, TOPS, and SOCRATES as a pre-processing step and was based on the concept of dominators [20]. It is possible, however, for *dynamic* USPs to emerge as the search process evolves; such UPSs cannot be identified with pre-processing techniques. A second version of SOCRATES [16] finds dynamic USPs by intersecting the *dynamic dominators* of each vertex on the D-frontier. This algorithm has quadratic time complexity since it requires the intersection of lists of dominator vertices. In addition, generating these lists for each vertex on the D-frontier has worst-case quadratic time complexity. Because of this high computational cost, SOCRATES invokes dynamic USP evaluation only for difficult faults that would otherwise be aborted. Examples of static and dynamic USPs and USIs are shown in Fig. 2.

Fortunately, dynamic USPs can be found much more simply by a linear time *levelized* breadth-first traversal of the circuit graph with overhead comparable to those of X-path check procedures. Starting from the vertices on the D-frontier, successive vertices that are on X-paths are visited in *level order*

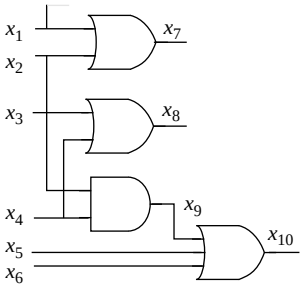

Fig. 3. Dynamic head lines

until the primary outputs are reached. A dynamic USP is identified whenever the width of this traversal, defined as the number of vertices scheduled to be processed next, drops to *one*; a traversal width of *zero* indicates that there are no X-paths from the D-frontier to the primary outputs.

B. Dynamic Evaluation of Head Lines

In FAN and SOCRATES, head lines are defined as the outputs of fanout-free sub-circuits and, thus, can be satisfied to any logic value in linear time. By using head lines instead of primary inputs, the search space can be effectively reduced. Head lines have been determined statically, as a pre-processing phase prior to computing the test pattern for each fault. However, as the search process evolves, it may be possible to define new head lines as a function of other head lines. An example of such a situation is shown in Fig. 3. Initially the set of head lines corresponds to the primary inputs $\{x_1, x_2, x_3, x_4, x_5, x_6\}$. Let us assume that the first decision, at decision level 1, corresponds to $x_2 = 1$, which implies x_7 to 1. We note that at this decision level the value of x_9 is uniquely determined by the value of x_4, because the value of x_2 is 1. Let us further assume that the next decision corresponds to $x_3 = 1$, which implies x_8 to 1. Because x_4 cannot affect vertices other than x_9, the *effective* fanout of x_4 is one after decision level 2. Since x_4 is a head line and it is fanout-free, then x_9 is a new head line at decision level 2. However, now x_{10} is driven by three fanout-free head lines, x_9, x_5 and x_6, and thus x_{10} also becomes a new head line. Each time a new head line is defined, the fanin head lines become fanout-free vertices covered by the new head lines. Consequently, after decision level 2, the set of dynamic head lines becomes $\{x_1, x_{10}\}$ instead of the static set $\{x_1, x_4, x_5, x_6\}$, and the dimension of the search space is reduced to half.

In LEAP, at each decision level and after all implications of the current decision have been performed, a levelized backward traversal of the circuit graph is performed to update the effective number of fanout vertices of each vertex. Afterwards, the current set of head lines is examined to determine whether a subset of head lines can be merged into a new head line. A vertex driven by head lines all of which become dynamically fanout-free is a new head line. The process of merging head lines into new head lines is repeated until no more new head lines can be derived. We note that whenever an unjustified vertex x_i becomes a new head line, x_i also becomes justified. In [17], dynamic head lines are further extended by using topological *don't cares* [9].

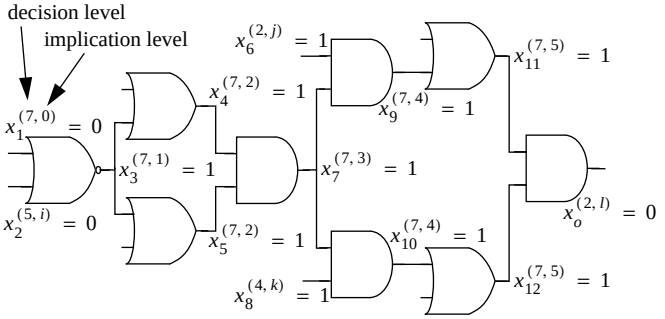

Fig. 4. Failure-driven assertions

C. Failure-Driven Assertions

In LEAP, whenever an inconsistency is found, its causes are analyzed and an attempt is made to avoid repeating implications that would lead to the same inconsistency later in the search process. Let us consider the example circuit with the dynamic situation shown in Fig. 4. The current decision level is assumed to be 7, and x_1 is assigned to 0. This causes vertices x_3, x_4, x_5, x_7, x_9, x_{10}, x_{11} and x_{12} to be implied to 1, and leads to a vertex inconsistency at x_o, which is required to be 0 at decision level 2. An analysis of the dynamic situation in the circuit shows that only decision levels 2, 4 and 5 contribute to the inconsistency. Furthermore, x_7 *cannot* assume value 1 above decision level 4; with the values of x_6, x_8 and x_o, implied at decision levels less or equal to 4, a vertex inconsistency occurs if x_7 assumes value 1 above decision level 4. Similarly, x_3 cannot assume value 1 above decision level 4. Consequently, both vertices must be asserted to value 0 at decision level 4. On the other hand, x_1 cannot assume value 0 above decision level 5, because otherwise the same implication sequence would take place, and an inconsistency would occur. Vertices x_1, x_3 and x_7, with asserted values due to inconsistencies, are referred to as *failure-driven assertions*. These assertions also cause the implication of x_9 and x_{10} to 0. We note that failure-driven assertions can also be determined in case of path inconsistencies.

After an inconsistency is detected, it is necessary to determine the vertices and decision levels that contributed to the inconsistency. Given a vertex or path inconsistency, we want to determine all the vertices that directly contributed to the inconsistency at the current decision level and at past decision levels. We also want to compute which decision levels besides the current decision level affected the inconsistency, to decide at which decision levels to assert vertices.

The process of identifying vertices and decision levels affecting an inconsistency is divided into two phases:

1. Determining decision levels that constrain D-propagation. Propagation of an error signal is constrained whenever some possible propagation paths to the primary outputs are eliminated.
2. Tracing the antecedents from the inconsistent vertex or from the set of vertices defining a path inconsistency until the decision vertex.

The first phase is used only to identify decision levels which

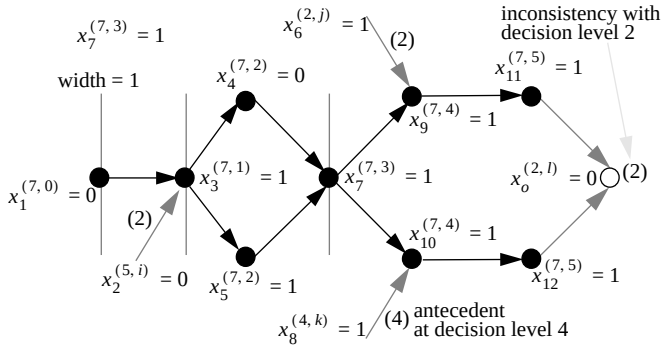

Fig. 5. Implication graph for decision $x_1 = 0$

effectively contribute to the inconsistency and which will not be identified by the second phase. The second phase determines the remaining decision levels which contribute to the inconsistency and determines which vertices can be asserted to a fixed value at some decision level. To obtain this information we perform *antecedent tracing* on the implication sequence leading to the inconsistency.

Antecedent tracing at decision level *k* corresponds to a reverse levelized breadth-first traversal on the implication level of each vertex implied at decision level *k*, from the inconsistency point until the vertex associated with the current decision. The inconsistency point denotes the set of vertices responsible for a vertex or path inconsistency. Each vertex after being processed schedules for future processing its antecedents that are also implied at the current decision level. The decision levels of antecedents other than those at the current decision level are recorded. By definition, the implication level of any antecedent of a vertex x_i, implied at the same decision level, is lower than the implication level of x_i. The partial order thus defined assures that whenever the width of the reverse levelized breadth-first traversal reaches *one*, the vertex to be processed next in the breadth-first traversal can be asserted to the complement of its current value at the highest decision level recorded so far, since its current value alone generates an implication sequence leading to an inconsistency. We note that since phase 1 records the decision levels constraining D-propagation, any decision level that directly contributes to the inconsistency is recorded. Hence if a vertex x_i is asserted to value *v* at some decision level *j*, it cannot provably assume a different value after decision level *j*.

We refer now to the example of Fig. 4, and illustrate how assertions are derived. In Fig. 5 the implication graph describing the information provided by the implication levels and by the antecedents is shown. Given the inconsistent vertices x_o, x_{11} and x_{12}, vertices x_{11} and x_{12} are scheduled for starting the reverse levelized breadth-first traversal and the decision level of x_o is recorded. During the traversal, the breadth-first width reaches one on vertices x_7, x_3 and x_1. Thus, each of these vertices can be asserted to the complement of its current logic value. The decision levels at which the vertices are asserted are defined by the decision levels other than 7 which have been recorded from the inconsistency point until the vertex being asserted is processed. In the example shown the decision levels recorded are 2 and 4 for x_7 and x_3, and 2, 4 and 5

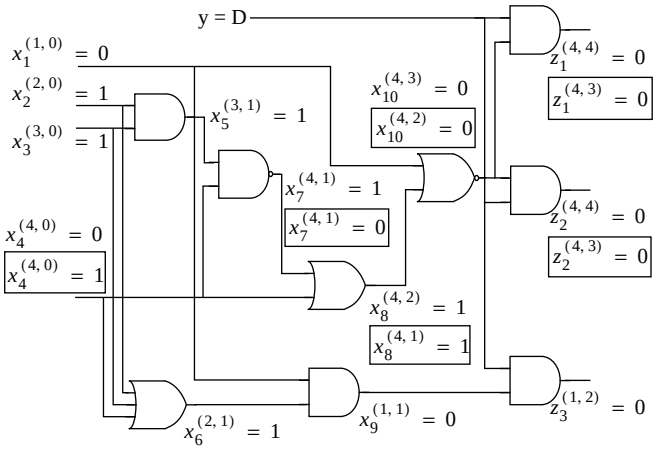

Fig. 6. Dependency-directed backtracking

for x_1. For this example we assume that no decision levels are recorded due to constraining D-propagation.

Although the example consists of forward implications, antecedent tracing can be used with backward implications or nonlocal implications because it is based on implication levels and on antecedents, and not on topological levels. We further note that antecedent tracing is the key procedure for implementing dependency-directed backtracking.

D. Dependency-Directed Backtracking

To illustrate how dependency-directed backtracking can improve the search process over chronological backtracking, we study the example circuit in Fig. 6. Without loss of generality, we assume a simple backtracing scheme which chooses the input variables in the order x_1, x_2, x_3, and x_4, and that the order of choosing vertices in the D-frontier is z_1, z_2 and z_3. Furthermore, the simple backtracing scheme is assumed to choose x_1 over x_8, x_7 over x_4, and x_5 over x_4. *y* assumes value D, and z_1, z_2 and z_3 are assumed to be primary outputs. We further assume that none of the techniques introduced in the previous sections is applied. Our goal is to propagate the error signal in *y* to any of the primary outputs.

The first decision is $x_1 = 0$, which results from backtracing an initial objective of 1 on x_{10}. This decision causes the implication of x_9 to 0 and z_3 to 0. Since z_3 is removed from the D-frontier, decision level 1 is recorded as constraining D-propagation. The second decision is $x_2 = 1$, which also results from backtracing from x_{10}. This decision causes the implication of x_6 to 1. The third decision is $x_3 = 1$, which causes x_5 to be implied to 1. Finally, the fourth decision is $x_4 = 0$, which causes x_7 to be implied to 1, x_8 to be implied to 1, and x_{10}, z_1 and z_2 to be implied to 0. Hence a path inconsistency is detected, and the value of x_4 must be complemented. We note that only decision levels 1 and 4 contribute to the path inconsistency as shown in Fig. 6. This information can be obtained by considering recorded decision levels which constrain D-propagation, and by performing antecedent tracing from the vertices that constrain D-propagation at the current decision level, i.e. z_1 and z_2. We note that decision levels 2 and 3 do not

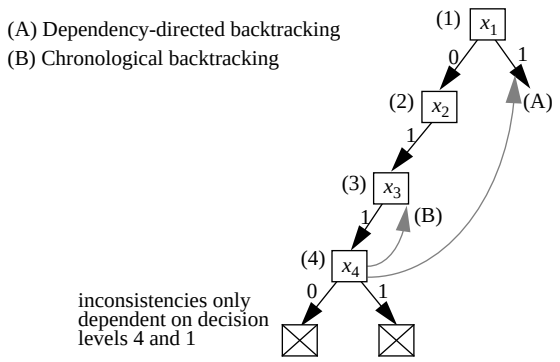

Fig. 7. Dependency-directed versus chronological backtracking

constrain D-propagation and do not contribute to any implication at decision level 4.

After complementing x_4, the new implications are x_7 implied to 0, x_8 implied to 1 and x_{10}, z_1 and z_2 implied to 0 (see highlighted assignments in Fig. 6). Again a path inconsistency is detected. Furthermore, we note that only decision levels 1 and 4 contribute to the path inconsistency.

Assigning x_4 to both logic values causes inconsistencies, hence it is necessary to backtrack. In chronological backtracking schemes, the last non-complemented decision is tried, which corresponds to x_3 in the example. However, the analysis of the decision levels that effectively contribute to both inconsistencies reveals that backtracking can be performed to decision level 1. Hence the value of x_1 is complemented and all decisions after decision level 1 are erased. By backtracking to decision level 1, it is proved that reconsidering the decisions at levels 2 or 3 could not allow path sensitization. The difference between dependency-directed backtracking and the chronological backtracking schemes is illustrated in Fig. 7.

As suggested in the previous section, the information required to implement the proposed dependency-directed backtracking scheme is obtained by recording the decision levels that constrain D-propagation and by performing antecedent tracing after detecting each inconsistency. Hence, all decision levels that contribute to inconsistencies are recorded, and each time both values of a decision node cause inconsistencies, the highest decision level that has been recorded in past inconsistencies is used as the backtracking decision level.

When backtracking to decision level k, it is necessary to identify lower decision levels that contribute to the implications at decision level k. The real cause for inconsistencies at higher decision levels can be related to these lower decision levels, which may not be recorded yet. However, by identifying all such decision levels, we could force backtracking to decision levels higher than the lowest decision level possible. For this reason, when processing inconsistencies, any vertex x_i implied at a lower decision level l is explicitly identified (i.e. marked as part of a set M_l). When backtracking to decision level l, only the vertices in M_l are processed. Antecedent tracing is performed for each of the vertices in M_l to identify lower decision levels that contribute to relevant implications at decision level l.

The dependency-directed backtracking scheme proposed here has negligible overhead in the absence of inconsistencies. When inconsistencies are detected, the time complexity of the algorithms for failure-driven assertions and dependency-directed backtracking is linear. Because all relevant dependencies are determined, the dependency-directed scheme proposed is *complete*, in contrast with other schemes proposed for sequential test pattern generation [11, 12].

IV. Results

The techniques described in the previous section have been incorporated in a path sensitization algorithm, LEAP, implemented in C++, which forms the core of a test pattern generation system, TG-LEAP. TG-LEAP can also run customized implementations of PODEM, FAN and SOCRATES, that employ the non-heuristic pruning techniques of each of these path sensitization algorithms. The implementation of PODEM [7], PODEM*, can perform both forward and backward implications, and thus must maintain a J-frontier. The implementation of FAN [5], FAN*, computes unique sensitization points dynamically whenever the size of the D-frontier is one using the algorithm described in Section III. The implementation of SOCRATES, SOCRATES*, implements the concepts described in [15] and also computes dynamic unique sensitization points, but using the algorithm proposed in Section A. Hence, SOCRATES* corresponds to a more efficient implementation of the deterministic heuristics in [15] and [16] until phase DYN_1 [16], but without the implementation of instruction 2 of the unique sensitization procedure [15]. The results given for SOCRATES also use the improved learning criterion of [10].

Because our main goal is to compare the non-heuristic pruning techniques of each algorithm, only structural controllability/observability measures are used [1]. In TG-LEAP backtracing *always* stops at a head line in opposition to the multiple backtracing schemes in FAN and SOCRATES, where backtracing can stop at fanout points. This option is intended to allow using LEAP in other applications, mainly timing analysis and delay fault testing. Furthermore, no redundancy removal techniques are used [1, 19]. In the tests performed, each path sensitization problem is intended to be analyzed individually, and updating the redundancy information of the circuit every time a fault is proved redundant, would eventually relate individual path sensitization problems.

In the following, several tests are performed on the ISCAS'85 [2] benchmark suite, using a collapsed fault set for each circuit. For comparison purposes, all faults in each collapsed fault set are targeted. This option is intended to allow a thorough evaluation of each of the path sensitization algorithms when applied to test pattern generation, especially in proving redundancy and finding tests for hard to detect faults. All the results shown were obtained on a DECstation 5000/240 with 32 Mbytes of RAM. In addition, all CPU times shown are in seconds.

The results of running each algorithm using the multiple backtracing of [5] are shown in Table I, where **#D**, **#R** and **#A**

TABLE I
RESULTS WITH MULTIPLE BACKTRACING

Circuit	Faults	PODEM*				FAN*				SOCRATES*				LEAP			
		#D	#R	#A	Time	#D	#R	#A	Time	#D	#R	#A	Time	#D	#R	#A	Time
C432	524	520	1	3	0.081	520	1	3	0.071	520	2	2	0.059	432	4	0	0.040
C499	758	750	0	8	0.211	750	8	0	0.134	750	8	0	0.137	750	8	0	0.144
C880	942	942	0	0	0.036	942	0	0	0.040	942	0	0	0.041	942	0	0	0.044
C1355	1574	1566	0	8	0.307	1566	8	0	0.280	1566	8	0	0.277	1566	8	0	0.290
C1908	1879	1861	6	12	0.217	1866	7	6	0.147	1870	9	0	0.126	1870	9	0	0.136
C2670	2747	2630	68	49	0.349	2628	86	33	0.260	2630	93	24	0.264	2630	117	0	0.210
C3540	3428	3281	114	33	0.282	3284	132	12	0.166	3291	137	0	0.152	3291	137	0	0.154
C5315	5350	5290	53	7	0.147	5291	59	0	0.159	5291	59	0	0.164	5291	59	0	0.180
C6288	7744	7703	34	7	0.579	7708	34	2	0.555	7695	34	15	0.782	7708	34	2	0.634
C7552	7550	7369	62	119	0.519	7349	77	124	0.435	7368	77	105	0.436	7419	131	0	0.384
Total	32496			246				180				146				2	

denote the number of detected, proved redundant and aborted faults, respectively. A backtrack limit of 500 was used. Columns labeled **Time** denote the average CPU time per fault for each algorithm.

LEAP is able to prove redundant *all* the redundant faults, and only aborts two detectable faults of circuit C6288, whereas the other path sensitization algorithms abort a larger number of faults. The run times for each algorithm, although similar, depend on the number of aborted faults. For circuits where SOCRATES* or FAN* abort no faults, the processing overhead of LEAP leads to slightly higher CPU times per fault. For circuits where FAN* and SOCRATES* abort a reasonable number of faults, LEAP performs better.

We further note that for C6288, FAN* performs better than SOCRATES*. We conjecture that since SOCRATES* uses non-local implications, for some faults this increases the original width of the J-frontier. This increased width may cause some wrong initial decisions, which are difficult to correct when the size of the decision tree becomes large. Although LEAP uses the same assignments as SOCRATES*, the initial wrong assignments are overcome by the dependency-directed backtracking scheme and by failure-driven assertions.

The primary objective of LEAP is to be used with difficult faults, both redundant and detectable. To compare LEAP with the other algorithms, a small set of redundant and hard to detect faults was chosen from some of the benchmark circuits. The results obtained are shown in Table II; columns labeled **#B** denote the number of backtracks and the column labeled **#A** denotes the number of assertions determined by LEAP. A ** indicates that the fault was aborted after 50000 backtracks, and a * indicates that the fault was aborted after 10000 backtracks.

For all the redundant faults, LEAP proves redundancy with a reduced number of backtracks. On the other hand, the other algorithms cannot prove redundancy in most cases, even with a large backtrack limit. The difference of backtracks between SOCRATES* and LEAP illustrates the strength of the deterministic heuristics introduced in LEAP. For both algorithms, the decision tree created for each fault is the same until backtracking is required. Afterwards, while SOCRATES* usually requires a very large number of backtracks, LEAP manages to derive the information required to skip several decision tree nodes, thus proving redundancy with a very small number of backtracks. Furthermore, in each of the examples shown that require backtracking, several assertions are determined by analyzing the causes of the inconsistencies.

For fault 3695 s-a-1 in circuit C7552, although LEAP requires 110 backtracks to find a test pattern to detect the fault, none of the other algorithms is able to find a solution to the path sensitization problem in less than 10000 backtracks. This example further illustrates the applicability of the deterministic heuristics used in LEAP when compared to SOCRATES*. Finally, we note that for fault 2417 s-a-1 of C2670, FAN* manages to prove redundancy while SOCRATES* does not. From our experience, the reason seems to be the increased J-frontier in SOCRATES* due to static learning.

A possible solution to reduce the CPU time per fault is to run PODEM* followed by LEAP. Hence, we ran PODEM* on all the faults, with a backtrack limit of 5, and using a simple backtracing scheme [7, 17]. Afterwards, we ran LEAP, with a backtrack limit of 500, on the faults aborted by PODEM*. The results obtained are shown in Table III. The total number of faults analyzed by each algorithm is denoted by **#T**. The number of detected, redundant and aborted faults is denoted by #**D**, **#R** and **#A**, respectively. PODEM* detects a total 31645 detectable faults from a total of 32496 faults, proves redundant 287 faults, and aborts 564 faults. Afterwards, LEAP detects 344 faults from an initial total of 564, proves redundant 220 faults and aborts no faults. The combination of PODEM* followed by LEAP achieves better performance than any of the other algorithms alone. Furthermore, no fault is aborted. We note that the two faults aborted by LEAP with multiple backtracing for C6288, are detected without backtracks by LEAP or PODEM* using simple backtracing [17].

The results presented in this section are intended only to illustrate the effectiveness of LEAP for difficult faults, both redundant and detectable. In a complete test pattern generation system, fault simulation would be employed to reduce the test set size, and to randomly detect some difficult detectable faults, as proposed in [13], [15] and [21]. We further note that

TABLE II
HANDLING DIFFICULT FAULTS

Circuit R: redundant D: detectable	Podem*		Fan*		Socrates*		Leap		
	#B	Time	#B	Time	#B	Time	#B	#A	Time
C432 (R) 259gat s-a-1	**	389	5618	56.63	793	4.93	33	66	0.36
C432 (R) 347gat s-a-1	**	288.5	5740	43.94	921	6.65	11	20	0.11
C1908 (R) 565 s-a-1	*	147.2	*	161.2	0	0.031	0	0	0.052
C2670 (R) 2282 s-a-1	*	127.3	*	150.5	*	203.8	9	16	0.24
C2670 (R) 2417 s-a-1	*	126.8	1872	57.41	*	227	4	6	0.16
C7552 (D) 3695 s-a-1	*	119.3	*	92.94	*	95.05	110	48	2.56

our implementation of SOCRATES* has some relevant differences with respect to the original algorithm [15, 16]. SOCRATES uses an improved multiple backtracing procedure as well as improved controllability/observability measures to guide the decision procedure. Furthermore, SOCRATES* only implements one of the unique sensitization procedures of SOCRATES [16]. This justifies the differences in results observed between SOCRATES* and SOCRATES.

V. CONCLUSIONS

This paper introduces several new techniques to prune the search space in path sensitization problems. These techniques explore dynamic information provided by the search process, both before and after inconsistencies are detected. The techniques proposed have been incorporated in a path sensitization algorithm (LEAP), which experimental results show to be more suitable to prove redundancy and to find tests for hard to detect faults than customized implementations of PODEM, FAN and SOCRATES.

Despite the improvements introduced in LEAP, the search process is still extremely dependent on the ordering of assignments to the head lines as the results in Section IV show. Future work is mainly intended to overcome this problem and to improve the inconsistency processing schemes proposed in LEAP. A natural evolution consists in introducing search state equivalence [6] and dominance [4] relations to further prune the search space. Actually, search state equivalence relations provide a complementary scheme with respect to dependency-directed backtracking; search state equivalence relations avoid entering in regions of the search space equivalent to others searched before, while dependency-directed backtracking prunes the decision tree by avoiding reconsidering decisions that do not affect the inconsistencies found.

Part of the motivation for developing LEAP is the construction of a highly efficient path sensitization algorithm with applications to other areas where path sensitization is required, mainly timing analysis and delay fault testing. Preliminary results of applying LEAP to timing analysis are given in [18]. Moreover, it is our goal to evaluate the application of LEAP to delay fault testing.

TABLE III
RESULTS USING PODEM* FOLLOWED BY LEAP

Circuit	PODEM*				LEAP				Time (sec/fault)
	#T	#D	#R	#A	#T	#D	#R	#A	
C432	524	519	0	5	5	1	4	0	0.031
C499	758	750	0	8	8	0	8	0	0.055
C880	942	940	0	2	2	2	0	0	0.027
C1355	1574	1566	0	8	8	0	8	0	0.142
C1908	1879	1818	6	55	55	52	3	0	0.079
C2670	2747	2624	49	74	74	6	68	0	0.070
C3540	3428	3262	100	66	66	29	37	0	0.093
C5315	5350	5268	46	36	36	23	13	0	0.075
C6288	7744	7534	34	176	176	176	0	0	0.213
C7552	7550	7364	52	134	134	55	79	0	0.189
Total	32496	31645	287	564	564	344	220	0	

REFERENCES

[1] M. Abramovici, M. A. Breuer and A. D. Friedman, *Digital Systems Testing and Testable Design*, Computer Science Press, 1990.

[2] F. Brglez, and H. Fujiwara, "A Neutral List of 10 Combinational Benchmark Circuits and a Target Translator in FORTRAN," in *Proc. Int. Symp. on Circ. and Sys.*, 1985.

[3] R. Dechter, "Learning While Searching in Constraint-Satisfaction Problems," Technical Report CSD-860049, University of California at Los Angeles, June 1986.

[4] T. Fujino and H. Fujiwara, "An Efficient Test Generation Algorithm Based on Search Space Dominance," in *Proc. 22nd Fault Tolerant Comput. Symp.*, pp. 246-253, 1992.

[5] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Trans. on Computers*, vol. 32, no. 12, pp. 1137-1144, December 1983.

[6] J. Giraldi and M. L. Bushnell, "EST: The New Frontier in Automatic Test pattern Generation," in *Proc. 27th Design Automation Conf.*, pp. 667-672, 1990.

[7] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. on Computers*, vol. 30, no. 3, pp. 215-222, March 1981.

[8] T. Kirkland and M. Ray Mercer, "A Topological Search Algorithm for ATPG," in *Proc. 24th Design Automation Conf.*, pp. 502-508, 1987.

[9] A. Lioy, "Adaptive Backtrace and Dynamic Partitioning Enhance APTG," in *Proc. Int. Conf. Computer Design*, pp. 62-65, 1988.

[10] W. Kunz and D. Pradhan, "Accelerated Dynamic Learning for Test Pattern Generation in Combinational Circuits," *IEEE Trans. on Computer-Aided Design*, vol. 12, no. 5, pp. 684-694, May 1993.

[11] S. Mallela and S. Wu, "A Sequential Circuit Test Generation System," in *Proc. Int. Test Conf.*, pp. 57-61, 1985.

[12] R. Marlett, "An Effective Test Generation System for Sequential Circuits," in *Proc. 23th Design Automation Conf.*, pp. 250-256, 1986.

[13] J. Rajski and H. Cox, "A Method to Calculate Necessary Assignments in Algorithmic Test Pattern Generation," in *Proc. Int. Test Conf.*, pp. 25-34, 1990.

[14] J. P. Roth, "Diagnosis of Automata Failures: a Calculus and a Method". *IBM J. Res. Develop.*, vol. 10, pp. 278-291, July 1966.

[15] M. H. Schulz, E. Trischler and T. M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Trans. on Computer-Aided Design*, vol. 7, no. 1, pp. 126-137, January 1988.

[16] M. H. Schulz, and E. Auth, "Improved Deterministic Test Pattern Generation with Applications to Redundancy Identification," *IEEE Trans. on Computer-Aided Design*, vol. 8, no. 7, pp. 811-816, July 1989.

[17] J. P. M. Silva and K. A. Sakallah, "Search-Space Pruning Heuristics for Path Sensitization in Test Pattern Generation," Technical Report CSE-TR-178-93, University of Michigan, October 1993.

[18] J. P. M. Silva and K. A. Sakallah, "Efficient and Robust Test Generation-Based Timing Analysis," in *Proc. Int. Symp. on Circ. and Sys.*, 1994, in press.

[19] R. M. Stallman and G. J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," in *Artificial Intelligence*, 9, pp. 135-196, 1977.

[20] R. E. Tarjan, "Finding Dominators in Directed Graphs," in *SIAM J. Comput.*, vol. 3, pp. 62-89, 1974.

[21] J. A. Waicukauski, P. A. Shupe, D. J. Giramma and A. Matin, "ATPG for Ultra-Large Structured Designs," in *Proc. Int. Test Conf.*, pp. 44-51, 1990.