Search-Space Pruning Heuristics for Path Sensitization in Test Pattern Generation

by

João P. Marques Silva and Karem A. Sakallah

CSE-TR-178-93

THE UNIVERSITY OF MICHIGAN

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
Ann Arbor, Michigan 48109-2122
USA

Search-Space Pruning Heuristics for Path Sensitization in Test Pattern Generation

João P. Marques Silva and Karem A. Sakallah

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan 48109-2122

October 8, 1993

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. While the conditions for path sensitization are different for different applications, the search mechanism need not be. In this paper we propose and demonstrate the effectiveness of several new deterministic techniques for search-space pruning for test pattern generation. These techniques are based on a dynamic analysis of the search process and can be viewed as extensions of methods that were introduced in FAN and SOCRATES. 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 redundant faults and in generating tests for difficult faults.

1 Introduction

Path sensitization is common to test pattern generation, delay-fault testing and timing analysis, and can be posed as the problem of finding a valid truth assignment of the nodes in the circuit that sensitizes a path with a given property. The conditions to sensitize a path depend on the application. In recent years, extensive work has been done in developing techniques to prune the search space associated with path sensitization problems, particularly in test pattern generation [1, 5-9, 13-16, 19]. These techniques can be categorized as deterministic (e.g. unique sensitization points, head lines, static/dynamic learning, search space equivalence relations) and non-deterministic (e.g. simple and multiple backtracing).

In this paper we propose new deterministic heuristics to further improve search-space pruning in path sensitization. These techniques are based on a dynamic analysis of the search process. We start by illustrating how *unique sensitization points* [5] can be efficiently determined dynamically. Our algorithm has linear time complexity, in contrast to the algorithm suggested in SOCRATES [16] which has worst-case quadratic time complexity.

We, then, show that the notion of *head lin*es can be naturally extended to dynamic situations, and thus the size of the set of head lines can be reduced as the search process evolves. We provide a linear time algorithm that determines, at each node in the decision tree, a reduced set of head lines.

These dynamic concepts are associated with search-space pruning to guide the search. In case of inconsistencies, we provide a method to determine nodes which must have assume certain values to avoid 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], backtracking is always performed to the previous node in the decision tree, i.e. *chronologic backtracking*. In some situations backtracking can *provably* be performed to some other node in the decision tree, thus saving a large number of backtracks. Our dependency-backtracking algorithm is provably *complete*, in the sense that a solution is found if a solution exists, and has linear time complexity. Furthermore, if no backtracking is required the algorithm introduces no overhead in the search process. Dependency-directed backtracking schemes were originally proposed in [17] in an application of artificial intelligence techniques to circuit analysis.

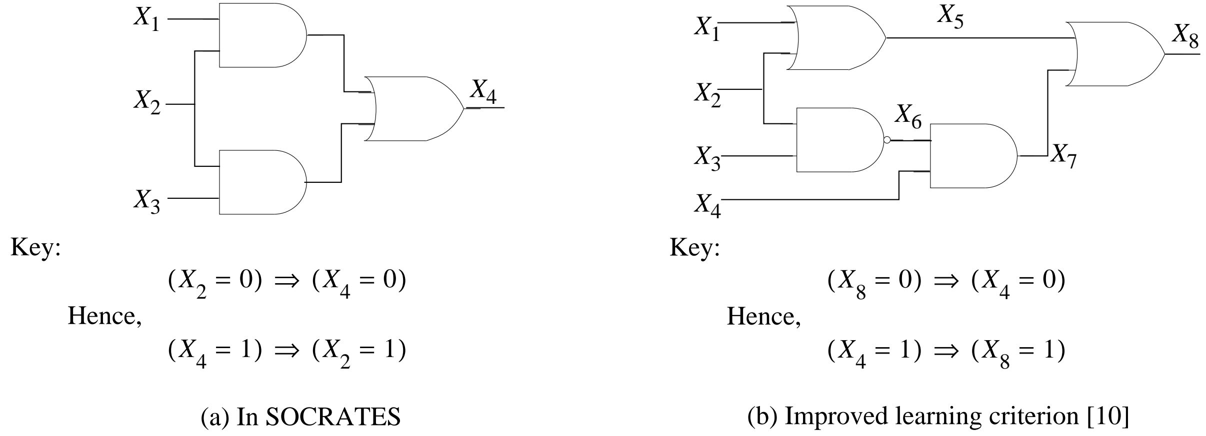
The new techniques for search-space pruning have been incorporated in a path sensitization algorithm LEAP (LEvel-dependent Analysis in Path sensitization). The basic path sensitization algorithm has been used to implement a test-pattern generation system, TG-LEAP, which can also run customized versions of PODEM, FAN, and SOCRATES.

In the next section we review concepts common to decision procedures used in path sensitization. We also introduce the basic concepts required to implement *failure-driven assertions* and *dependency-directed backtracking*. In Section 3 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 proving redundancy and in detecting difficult faults. In Section 5 directions for future research are described.

2 Definitions

The underlying algorithm for path sensitization is assumed to be a PODEM-based decision procedure [7], where decisions are made with respect to the primary inputs (or to the head lines). Throughout the paper we use the concepts of D-frontier, J-frontier, X-path, backward/forward implications, head line, unique sensitization points, and other concepts used in path sensitization for test-pattern generation, and which are described in detail in [1]. Furthermore, LEAP can perform static learning as proposed in [15], but the extended learning criterion of [10] is used, which includes static learning due to backward implications. Examples of static learning are shown in Fig. 1. Other concepts used in path sensitization are analyzed in the next sections.

In order to implement some of the techniques proposed in LEAP, the following additional definitions are required. A circuit is envisioned as a directed graph $G = (V, E)$, where primary inputs and gate outputs are represented as vertices in the graph. A directed edge (u, v) represents an input u of a gate with output v. Any algorithm defined on G with



(a) In SOCRATES

(b) Improved learning criterion [10]

Figure 1: Static learning

complexity $\Theta(|V| + |E|)$ is said to have linear time complexity, because the number of edges is assumed to be linearly related to the number of vertices (i.e., gates with bounded fanin).

In the absence of inconsistencies, the search procedure proceeds as a sequence of decisions each of which is followed by the corresponding implications performed in a *breadth-first* manner. A decision in this context refers to the *elective* assignment of a specific value (0 or 1) 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 elective assignment of other vertices. The sequence of decisions is represented by a directed graph T, referred to as the *decision tree*. Each node, θ, in the decision tree is referred to as a *decision node*, 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. The implications associated with a given decision are represented by a directed *implication graph* I, which describes how the implication sequence evolves. Each node, ϕ, in the implication graph is characterized by two integer parameters:

1. $d(\phi)$, which is the decision level of the decision node responsible for the implication sequence;
2. $p(\phi)$, defined as the *implication* level.

In addition, $S(\phi)$ will be used to denote the predecessors of ϕ in the implication graph, which represent the vertices causing the implication of ϕ. The elements of $S(\phi)$ will also be referred to as the *implication parents* of ϕ.

The decision and implication levels of an implied node ϕ are calculated according to:

$$d(\phi) = max\{d(\zeta) | \zeta \in S(\phi)\} \tag{1}$$

and,

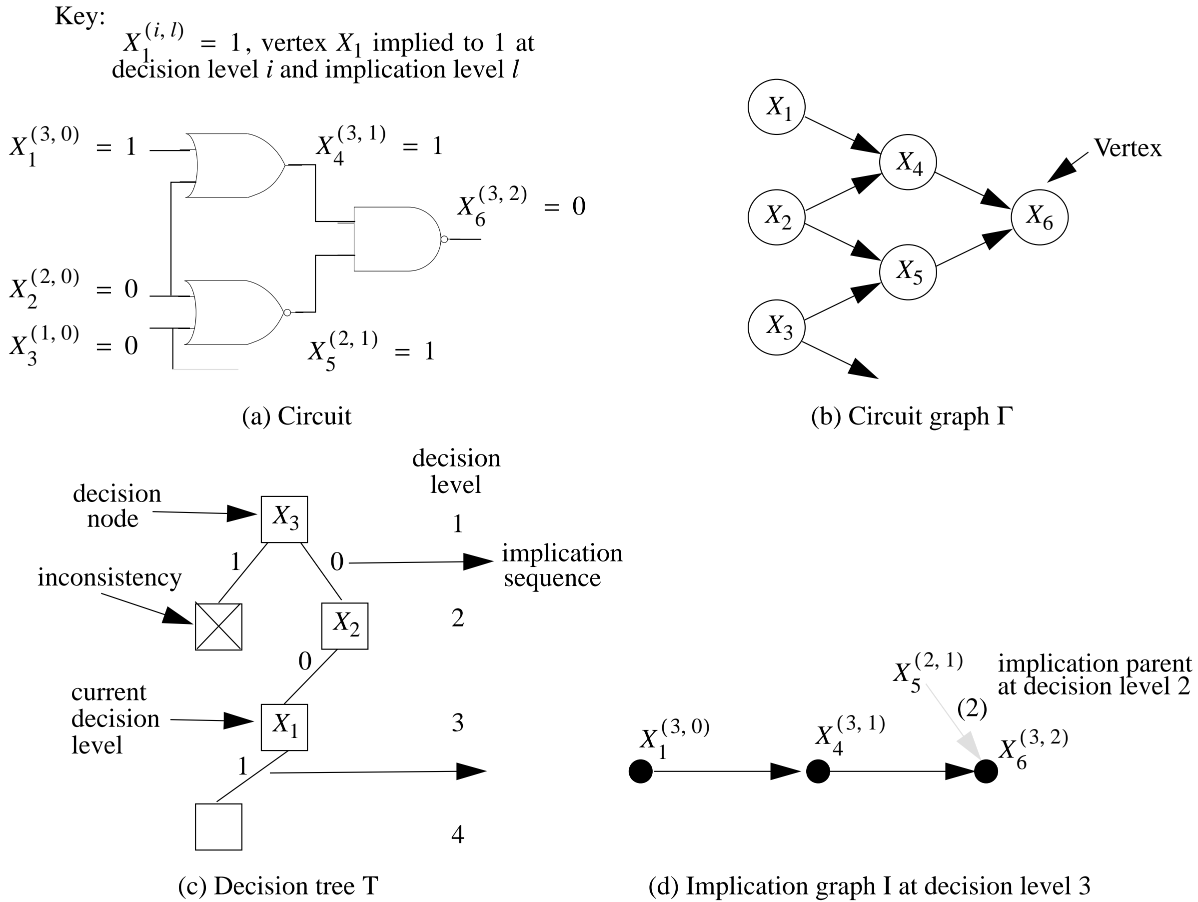
$$p(\phi) = 1 + max\{p(\zeta) | \zeta \in S(\phi) \wedge d(\zeta) = d(\phi)\} \tag{2}$$

The implication level of a decision node is 0 by definition.

Throughout 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.

In Fig. 2, an example circuit is shown, with the corresponding circuit graph, decision tree, and implication graph for a specific decision. In Fig. 2-a, and after assigning X_1 to 1 at decision level 3, the implication parent of X_4 is X_1, and the implication parents of X_6 are X_4 and X_5.

While the search process evolves inconsistencies may occur. These inconsistencies are categorized as *vertex* or *path*. A vertex inconsistency occurs when the logic values of the inputs and output of a gate are not consistent. A path inconsistency occurs when the D-frontier becomes empty after some decision. The two types of inconsistency are illustrated in Fig. 3. In Fig. 3-b the D-frontier is assumed to be composed only of X_3 and X_4.



(a) Circuit (b) Circuit graph Γ

(c) Decision tree T (d) Implication graph I at decision level 3

Figure 2: Structures associated with the search procedure

3 Level-Dependent Analysis

3.1 Dynamic Evaluation of Unique Sensitization Points

A *unique sensitization point* is a gate in a circuit which must propagate an error for a given fault to be detected [5]. The inputs to this gate, which cannot propagate the error signal, must assume non-controlling values and are referred to as *unique sensitization implications*. In Fig. 4-a, an example of a unique sensitization point and corresponding unique sensitization implication are shown. Both vertices X_6 and X_7 can propagate the error signal currently on X_2. Assuming the D-frontier to be composed of only X_6 and X_7, then the error signal propagates from X_2 to X_9 only if X_8 assumes a non-controlling value, i.e. 0.

The evaluation of unique sensitization points was proposed in FAN [5], TOPS [8], and SOCRATES [15] as a pre-

(a) Vertex inconsistency (b) Path inconsistency

Figure 3: Inconsistency types

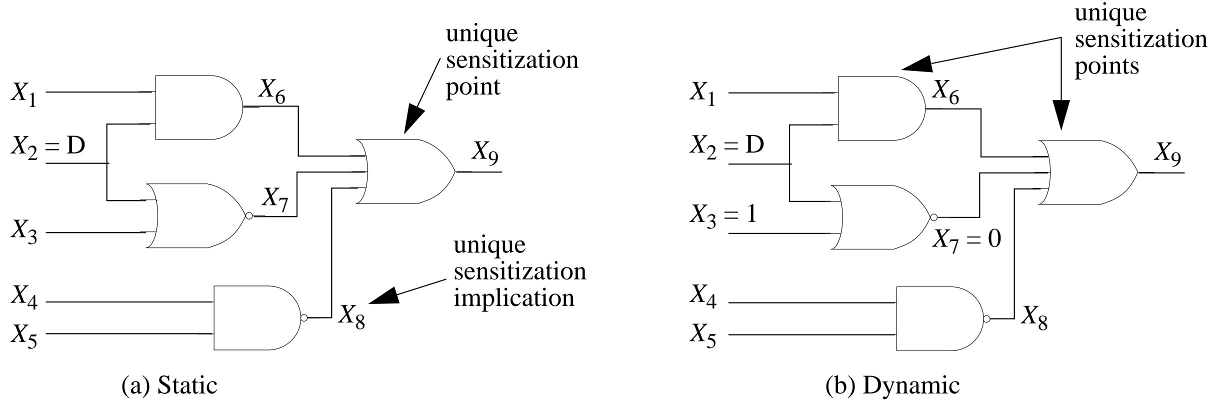


Figure 4: Unique sensitization points and implications

processing step and is based on the concept of dominators [18]. However, as the search process evolves, other unique sensitization points may exist, which cannot be determined with pre-processing techniques. An example of a dynamic unique sensitization point is shown in Fig. 4-b. Because X_3 has been implied to 1, X_7 is implied to 0. Hence, for the error signal to propagate from X_2 to X_9, X_1 must assume the non-controlling value of X_6. X_6 is not a static dominator of X_2, but due to the implication of X_3, X_6 becomes a dynamic dominator of X_2.

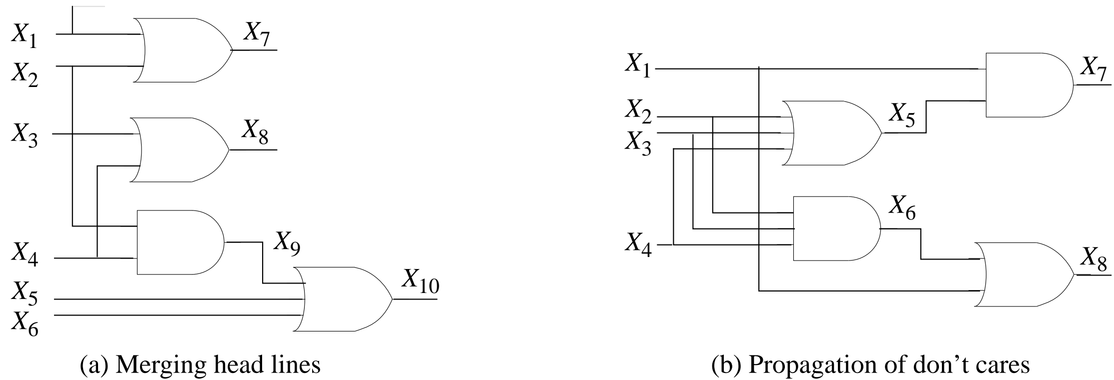
The evaluation of dynamic unique sensitization points may be required to prove the redundancy of some faults [16]. In a second version of SOCRATES [16], an approach to dynamically compute unique sensitization points based on the intersection of the dynamic dominators of each vertex in the D-frontier is given. This algorithm has quadratic time complexity because it requires the intersection of lists of dominator vertices. Furthermore, computing the lists of dynamic dominators of each vertex in the D-frontier also has worst-case quadratic time complexity on the size of the circuit graph. In SOCRATES, the dynamic evaluation of unique sensitization points is only applied to difficult faults, that are otherwise aborted [16].

3.1.1 Levelized Breadth-First Traversal

If we envision a circuit as a directed graph and identify the vertices in the D-frontier, then a levelized breadth-first traversal from the vertices in the D-frontier and until a primary output is reached, identifies the existence of an X-path and also identifies the dynamic unique sensitization points. A levelized breadth-first traversal basically ensures that a vertex at topological level k is not processed before any vertex with topological level less than k, which must also be processed. Hence, a unique sensitization point u is processed only after all vertices in its transitive fanin have been processed, and before any of the vertices in its transitive fanout are scheduled to be processed. If the *width* of the breadth-first traversal is defined as the number of vertices scheduled to be processed, then whenever the width is one, a dynamic unique sensitization point has been reached. If the width of the breadth first traversal ever reaches zero, then there is no X-path from a vertex in the D-frontier to a primary output. Finally, we note that the levelized breadth-first traversal has linear time complexity, and can be implemented with the same overhead as any procedure to identify X-paths.

3.1.2 Implication Parents due to Unique Sensitization Points

Whenever a vertex u is a dynamic unique sensitization implication, the implication parents of u are defined as the set of vertices constraining the D-frontier at the current decision level. A vertex v is said to constrain the D-frontier at decision level k if at decision level k-1 vertex v is in the D-frontier, and at decision level k v is either implied to 0 or 1. The vertices constraining the D-frontier implicitly cause the unique sensitization implications, and hence can be understood as the implication parents of derived unique sensitization implications.



(a) Merging head lines (b) Propagation of don't cares

Figure 5: Dynamic head lines

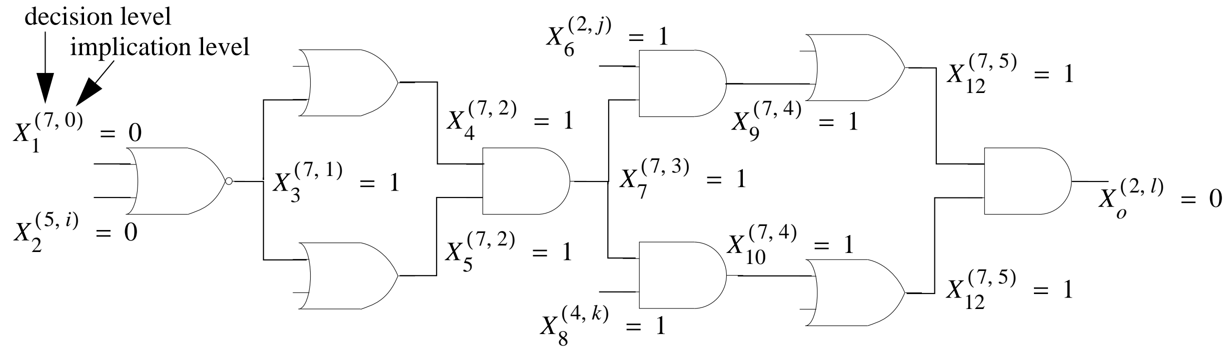
3.2 Dynamic Evaluation of Head Lines and Don't Cares

In FAN [5] and SOCRATES [15], head lines are defined as the outputs of fanout-free sub-circuits. Head lines 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. In Fig. 5-a, an example of such a situation is given. 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, with 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.

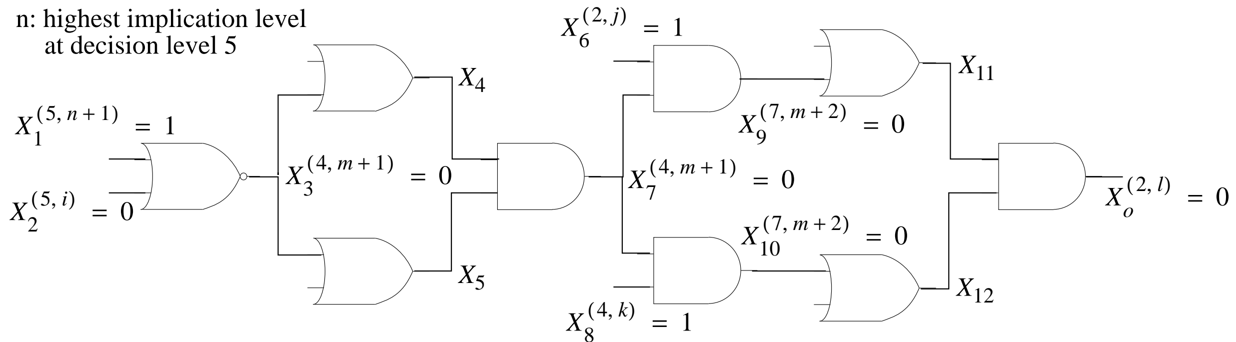
In [9] the concept of *don't care* vertices was introduced; it basically denotes a vertex that cannot affect fault propagation after some decision level, and has been used to speed up the search process by reducing the number of implications performed. We can also use the concept of don't care vertices to further extend the dynamic evaluation of head lines. In Fig. 5-b, an example of such a situation is given. Let us assume that the vertex corresponding to the first decision is X_1. If X_1 is assigned to 1, then X_8 is implied to 1. The effective fanout of X_6 becomes 0, and X_6 is said to be a *don't care* since its value cannot propagate forward, and cannot affect the path sensitization problem. Because X_6 is a don't care, the effective fanout of vertices X_2, X_3 and X_4 can be decreased by 1, and these three head line vertices become dynamically fanout-free. Consequently, X_5 becomes a new head line, and because X_7 only depends on X_5 after decision level 1, X_7 becomes a new head line. Vertices X_2, X_3, X_4 and X_5 are covered by the new head line X_7, and the search space after decision level 1 is reduced from $\{X_2, X_3, X_4\}$ to $\{X_7\}$. On the other hand, if at the first decision level X_1 is assigned to 0, then a similar reasoning applies, and the final search space is $\{X_8\}$.

As both examples show, merging head lines at a given decision level depends strongly on the decisions made at the current and past decision levels.

In LEAP, at each decision level and after all implications of the current decision have been performed, an initial set of new don't cares is determined. This set corresponds to primary outputs that can no longer propagate the error signal, and to the non-implied vertices that have all fanout vertices implied, i.e. vertices with effective fanout of 0. Start-



(a) Implication sequence



(b) With derived assertions

Figure 6: Failure-driven assertions

ing from the initial set of don't cares, a levelized backward traversal on the circuit structure is performed to update the effective number of fanout vertices of each vertex. If the effective number of fanout vertices of a vertex v reaches zero, then v becomes a don't care. 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 further note that whenever an unjustified vertex v becomes a new head line, v also becomes justified.

3.3 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 again during the search process. Let us consider the example circuit with the dynamic situation shown in Fig. 6-a. 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. We further note the evolution of the implication levels within decision level 7.

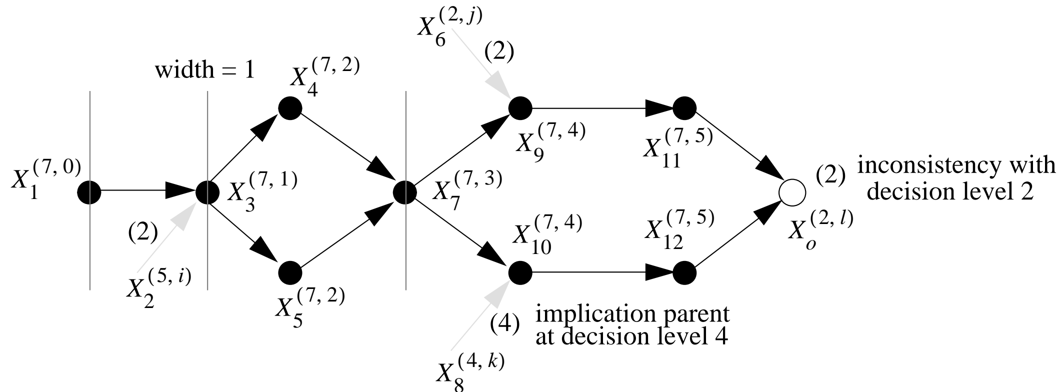


Figure 7: Implication graph associated with decision X_1 = 0

Vertices X_1, X_3 and X_7, with asserted values due to inconsistencies, are referred to as *failure-driven assertions*. A similar reasoning could be applied in case of a path inconsistency.

In Fig. 6-b, the implied vertices after asserting vertices are shown. Assuming no inconsistency is detected (because the decision node is complemented), the next decision level is 8 and no vertex is effectively implied at decision level 7.

During the search process, and whenever an inconsistency is found, the causes of the inconsistency are examined. This basically entails determining the vertices and decision levels which 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 level to assert vertices.

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

1. Identifying which decision levels constrain D-propagation by removing vertices from the D-frontier.
2. Tracing the implication parents 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 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 *parent tracing* on the implication sequence leading to the inconsistency.

3.3.1 Parent Tracing

Parent 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 implication parents which are also implied at the current decision level. The decision levels of implication parents other than the current decision level are recorded. By definition, the implication level of any implication parent of a vertex v, implied at the same decision level, is lower than the implication level of v. 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 the D-frontier, any decision level that directly contributes to the inconsistency is recorded. Hence if a vertex v 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. 6, and illustrate how assertions are derived. In Fig. 7 the implication graph describing the information provided by the implication levels and by the implication parents is shown. Starting from the inconsistent vertices X_o, X_{11} and X_{12}, the graph is traversed in reverse levelized breadth-first manner. 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 for X_1. In this example we assume that no decision levels are recorded due to constraining the D-frontier.

Although the example consists of forward implications, parent tracing can be used with backward implications or non-local implications because it is based on implication levels and on implication parents, and not on topological levels.

Finally, we note that by performing parent tracing every time an inconsistency is found, and by recording the decision levels constraining the D-frontier, it is possible to always have an accurate definition of which decision levels were relevant for the inconsistencies detected. This information is what allows performing dependency-directed backtracking.

Our approach to derive failure-driven assertions eventually determines the same information that can be determined with dynamic learning as proposed in SOCRATES [16], but there are some relevant differences. Dynamic learning has quadratic time complexity and has to be performed after each decision. For this reason dynamic learning is only used for extremely difficult faults [16]. On the other hand, computing failure-driven assertions has linear time complexity and is only performed after an inconsistency is found. However, in some cases failure-driven assertions may take several decisions to derive the same information that can be derived with dynamic learning after one decision. Therefore, we could hypothetically construct examples where dynamic learning, with a smaller decision tree, would perform better than computing failure-driven assertions. On average, however, we believe that computing failure-driven assertions is more efficient than dynamic learning while providing equivalent information.

3.4 Dependency-Directed Backtracking

To illustrate how dependency-directed backtracking can improve the search process over chronologic backtracking, we study the example circuit in Fig. 8-a. 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 *X4*. 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 the D-frontier. 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. 8-b. This information can be obtained by considering recorded decision levels which constrain the D-frontier, and by performing parent tracing from the vertices constraining the D-frontier at the current decision level, i.e. Z_1 and Z_2. We note that decision levels 2 and 3 do not constrain the D-frontier and do not contribute to an 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 Fig. 8-c). Again a path inconsistency is detected. Furthermore, we note that only decision levels 1 and 4 contribute to the path inconsistency. This information can be obtained again by considering recorded decision levels which constrain the D-frontier, and by performing parent tracing from the vertices constraining the D-frontier at the current decision level.

Assigning X_4 to both logic values causes inconsistencies, hence it is necessary to backtrack. In chronologic backtracking schemes, the last non-complemented decision is tried, which corresponds to X_3 in the example. However, the

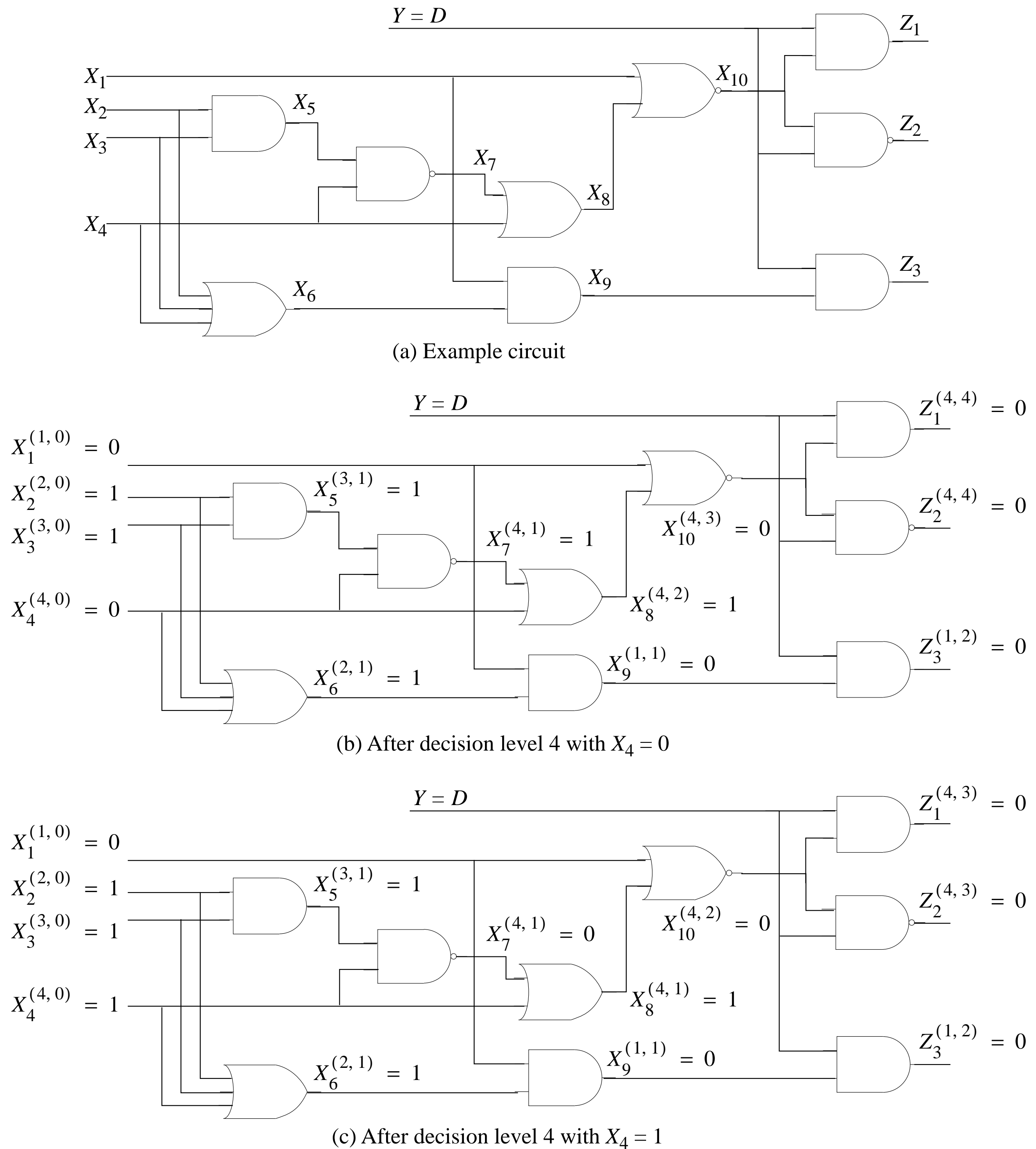


Figure 8: Analyzing inconsistencies for backtracking purposes

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 chronologic backtracking schemes is illustrated in Fig. 9.

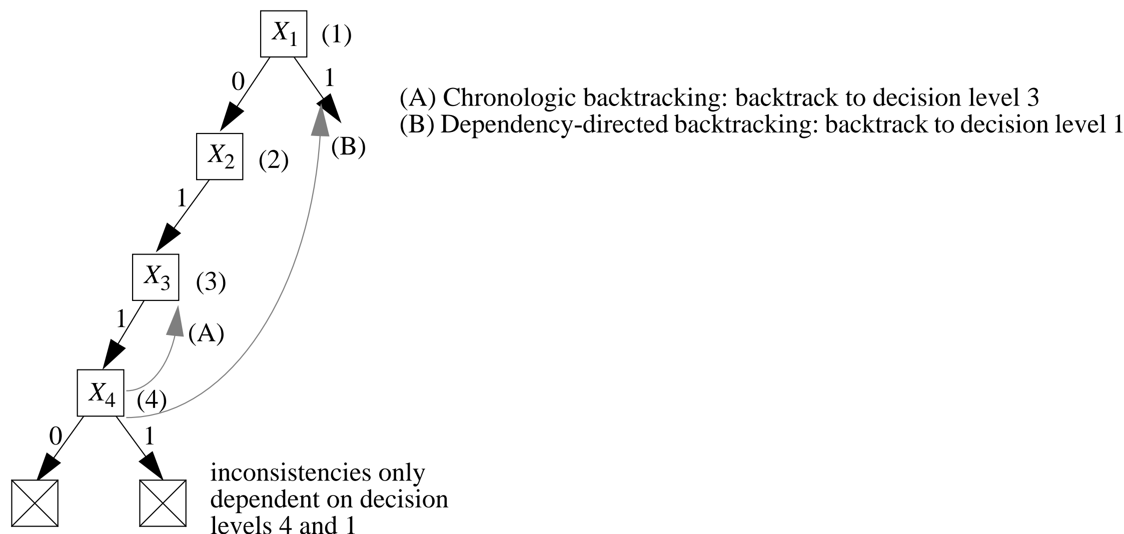


Figure 9: Dependency-directed versus chronologic backtracking

As suggested in the previous section, the information required to implement the proposed dependency-directed backtracking scheme is obtained by recording the decision levels constraining the D-frontier and by performing parent tracing after each inconsistency is detected. 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 v 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. Parent 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.

In [11] and [12] some different forms of dependency-directed backtracking were proposed for test-pattern generation in sequential circuits. However, the backtracking scheme proposed in [11] is only sketched and no experimental results are given. The backtracking scheme proposed in [12] uses a concept equivalent to decision levels to decide the backtracking point in the decision tree, but as described in [12], the analysis is performed local to an inconsistency and hence it may not to be complete. Consequently, some detectable faults may not be detected.

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. When finding the solution to some problem, this improved backtracking scheme never requires more backtracks than the common chronologic backtracking, provided a fixed ordering of the decision vertices is assumed. In the worst-case scenario, both schemes result in the same number of backtracks; in such a situation dependency-directed backtracking introduces a constant overhead to the total running time.

4 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 deterministic heuristics 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 3.1. 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 3.1. 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 deterministic heuristics of each algorithm, only structural controllability/observability measures are used [1]. Furthermore, only the following *backtracing* schemes were tested:

1. Simple backtracing, starting by trying to satisfy the most difficult controllability problems and afterwards trying to satisfy the most simple observability problems.
2. Multiple backtracing, as proposed in [5], but using structural controllability/observability measures.

In TG-LEAP backtracing is *always* performed to a head line in opposition to the backtracing schemes in FAN and SOCRATES, where backtracing can stop at fanout points [5], [15]. This option is intended to allow using the path sensitization algorithm in other applications, mainly timing analysis and delay-fault testing.

Furthermore, no redundancy removal techniques are used [1], [17]. In the tests performed, each path sensitization problem is intended to be analyzed individually, and updating redundant information on 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 simplistically reduced fault set for each circuit. For comparison purposes, all faults of the collapsed fault set of each benchmark circuit 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 Table 1, some characteristics as well as the pre-processing time for static learning of each benchmark circuit are shown. Static learning is used only in SOCRATES* and LEAP.

Table 1: Statistics of the benchmark circuits

Circuit	Gates	PIs	POs	Faults	Redundant faults	Pre-processing time (in sec)	Non-local implications
C432	160	36	7	524	4	0.097	138
C499	202	41	32	758	8	0.208	40
C880	383	60	26	942	0	0.250	116
C1355	546	41	32	1574	8	1.047	208
C1908	880	33	25	1879	9	1.910	1310
C2670	1193	233	140	2747	117	2.718	1951
C3540	1669	50	22	3428	137	16.37	6906
C5315	2307	178	123	5350	59	4.723	3609
C6288	2406	32	32	7744	34	0.961	830
C7552	3512	207	108	7550	131	13.90	10139

The results of running each algorithm with simple backtracing are shown in Table 2. The number of detected, redundant and aborted faults is denoted by **#D**, **#R** and **#A**, respectively. The total number of aborted faults for each algorithm is also given.

Table 2: Results with simple backtracing (backtrack limit set to 500)

Circuit	Faults	PODEM*			FAN*			SOCRATES*			LEAP		
		#D	#R	#A	#D	#R	#A	#D	#R	#A	#D	#R	#A
C432	524	519	0	5	519	1	4	519	2	3	520	4	0
C499	758	750	0	8	750	8	0	750	8	0	750	8	0
C880	942	942	0	0	942	0	0	942	0	0	942	0	0
C1355	1574	1566	0	8	1566	8	0	1566	8	0	1566	8	0
C1908	1879	1864	6	9	1868	9	2	1868	9	2	1868	9	2
C2670	2747	2626	62	59	2630	86	31	2630	93	24	2630	117	0
C3540	3428	3282	114	32	3287	132	9	3291	137	0	3291	137	0
C5315	5350	5291	55	4	5291	59	0	5391	59	0	5291	59	0
C6288	7744	7675	34	35	7700	34	10	7710	34	0	7710	34	0
C7552	7550	7375	62	113	7388	73	89	7390	77	83	7417	131	2
Total	32496			273			145			112			4

As the results show, with simple backtracing, LEAP performs better that any of the other algorithms, and only four

Table 3: Results with multiple backtracing (backtrack limit set to 500)

Circuit	Faults	PODEM*			FAN*			SOCRATES*			LEAP		
		#D	#R	#A	#D	#R	#A	#D	#R	#A	#D	#R	#A
C432	524	520	1	3	520	1	3	520	2	2	432	4	0
C499	758	750	0	8	750	8	0	750	8	0	750	8	0
C880	942	942	0	0	942	0	0	942	0	0	942	0	0
C1355	1574	1566	0	8	1566	8	0	1566	8	0	1566	8	0
C1908	1879	1861	6	12	1866	7	6	1870	9	0	1870	9	0
C2670	2747	2630	68	49	2628	86	33	2630	93	24	2630	117	0
C3540	3428	3281	114	33	3284	132	12	3291	137	0	3291	137	0
C5315	5350	5290	53	7	5291	59	0	5291	59	0	5291	59	0
C6288	7744	7703	34	7	7708	34	2	7695	34	15	7708	34	2
C7552	7550	7369	62	119	7349	77	124	7368	77	105	7419	131	0
Total	32496			246			180			146			2

detectable faults out of a total of 32496 are aborted. Of the three other algorithms, SOCRATES* performs better than FAN*, which performs better than PODEM*.

In Table 3, the results of running each algorithm using multiple backtracing are shown. With both backtracing schemes, LEAP is able to prove redundant *all* the redundant faults, and with multiple backtracing only aborts two detectable faults of circuit C6288. We note that with simple backtracing LEAP detects these two faults without backtracking. For SOCRATES* and FAN* the results improve for C432, C1908 and C6288. On the other hand, the results are worse for C7552, and this causes the total number of aborted faults of SOCRATES* and FAN* to increase with multiple backtracing.

Furthermore, with multiple backtracing, and 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.

In Table 4 the running times per fault for each of the algorithms is shown, with both simple and multiple backtracing. For circuits where several faults are aborted by the other algorithms, LEAP performs better. However, in circuits where SOCRATES* does not abort any fault (e.g. C499, C880, C1355, C1908, C3540 and C5315), the average running time per fault of LEAP is higher. The main reason for the higher running times is related to the dynamic evaluation of head lines as proposed in Section 3.2. At each decision level, dynamic head line evaluation introduces overhead linearly related to the size of the circuit graph. Hence, a small constant is introduced to the running times, as the results indicate.

Table 4: Run time per fault (in seconds) with simple and with multiple backtracing

Circuit	PODEM*		FAN*		SOCRATES*		LEAP	
	Simple	**Multiple**	**Simple**	**Multiple**	**Simple**	**Multiple**	**Simple**	**Multiple**
C432	0.046	0.081	0.033	0.071	0.033	0.059	0.024	0.040
C499	0.093	0.211	0.049	0.134	0.055	0.137	0.065	0.144
C880	0.027	0.036	0.030	0.040	0.033	0.041	0.040	0.044
C1355	0.154	0.307	0.137	0.280	0.127	0.277	0.143	0.290
C1908	0.161	0.217	0.135	0.147	0.112	0.126	0.126	0.136
C2670	0.223	0.349	0.129	0.260	0.147	0.264	0.130	0.210
C3540	0.215	0.282	0.108	0.166	0.103	0.152	0.118	0.154
C5315	0.090	0.147	0.113	0.159	0.126	0.164	0.149	0.180
C6288	0.423	0.579	0.253	0.555	0.292	0.782	0.320	0.634
C7552	0.283	0.519	0.266	0.435	0.274	0.436	0.292	0.384

For circuits where SOCRATES* aborts faults, LEAP has better run times. However, if the backtrack limit is reduced, SOCRATES* would eventually have better run times at the cost of some aborted faults.

As Table 4 indicates, the multiple backtracing scheme used introduces an important overhead when compared with the results for simple backtracing. In most of the circuits, the average run time per fault almost doubles with multiple backtracing. The main reason for this difference in run times is due to the difference between the number of vertices traversed by multiple backtracing and the number of vertices traversed by simple backtracing.

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 5; columns labeled **#B** denote the number of backtracks and the column labeled **#A** denotes the number of assertions determined by LEAP. For C432 the backtrack limit was set to 50000, and for the other circuits the backtrack limit was set to 10000. 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 which require backtracking, several assertions are determined by analyzing the causes of the inconsistencies.

Table 5: Handling difficult faults with multiple backtracing (time in seconds)

Circuit R: redundant D: detectable	PODEM*		FAN*		SOCRATES*		LEAP		
	#B	Time	#B	Time	#B	Time	#B	#A	Time
C432 (R) 259gat s-a-1	> 50000	389	5618	56.63	793	4.93	33	66	0.36
C432 (R) 347gat s-a-1	> 50000	288.5	5740	43.94	921	6.65	11	20	0.11
C1908 (R) 565 s-a-1	> 10000	147.2	> 10000	161.2	0	0.031	0	0	0.052
C2670 (R) 2282 s-a-1	> 10000	127.3	> 10000	150.5	> 10000	203.8	9	16	0.24
C2670 (R) 2417 s-a-1	> 10000	126.8	1872	57.41	> 10000	227	4	6	0.16
C7552 (D) 3695 s-a-1	> 10000	119.3	> 10000	92.94	>10000	95.05	110	48	2.56

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* caused by static learning, which in some situations may cause the multiple backtracing scheme used to make several wrong assignments, which result in SOCRATES* not being able to detect the fault or to prove the fault redundant.

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*.

Some of the relevant statistics of running LEAP with simple and multiple backtracing on each of the benchmark circuits are shown in Table 6; columns labeled **S** denote simple backtracing whereas columns labeled **M** indicate multiple backtracing. For LEAP, the number of decisions is usually small when compared with the number of primary inputs of each circuit. The average number of decisions depends on the backtracing scheme chosen, and none of the backtracing schemes implemented seems to be definitely better. On average, the number of backtracks per fault is negligible; the only exception being C7552. The average number of failure-driven assertions illustrates the applicability of this deterministic heuristic. Since the average number of backtracks is small, the number of assertions is also

Table 6: Statistics for LEAP (average numbers per fault) with a backtrack limit of 500

Circuit	Decisions		Backtracks		Assertions		Unique sensitization implications		Head lines	
	S	**M**	**S**	**M**	**S**	**M**	**S**	**M**	**S**	**M**
C432	12.44	10.94	0.235	0.141	1.20	0.48	3.98	3.96	2.02	1.41
C499	33.74	35.12	0.169	0.098	2.09	2.46	0.61	0.93	0.51	0.07
C880	10.49	7.31	0.008	0.000	0.03	0.00	2.60	2.10	3.60	2.04
C1355	32.88	32.67	0.000	0.000	1.91	0.20	0.71	0.59	0.02	0.01
C1908	17.83	12.57	0.856	0.039	1.21	0.14	1.87	2.10	0.71	0.44
C2670	16.92	17.47	0.076	0.111	0.69	0.33	2.83	3.02	12.71	12.71
C3540	11.42	9.64	0.009	0.037	0.42	0.30	2.80	2.86	0.43	0.38
C5315	9.76	10.36	0.022	0.023	0.22	0.09	2.20	2.23	0.88	0.83
C6288	26.93	27.84	0.033	0.347	0.35	0.70	0.31	0.32	0.01	0.01
C7552	33.68	21.97	1.190	1.272	1.00	1.07	2.99	3.09	1.03	1.01

necessarily small. Except for specific circuits (C499, C1355 and C6288), a reasonable number of unique sensitization implications is determined for each fault. Furthermore, for most circuits, several static as well as dynamic head lines are found and reduced. C432, C880, and C2670 appear to be specially suitable for determining head lines. For circuits C499, C1355 and C6288, on the contrary, the number of head lines is small.

The results of Table 2 and of Table 3 suggest that PODEM*, with simple backtracing, can be used to detect most of the faults with very little overhead. On the other hand, using LEAP is preferable to detect or prove redundant the more difficult faults. Hence, we ran PODEM*, with simple backtracing and a backtrack limit of 5, on all the benchmark circuits. Afterwards, we ran LEAP, with multiple backtracing and a backtrack limit of 500, on the set of faults aborted by PODEM*. The results obtained are shown in Table 7. 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. For C499, C880 and C1355 some of the algorithms discussed can perform better alone without aborting faults. For the remaining benchmark circuits, using the combination of PODEM* followed by LEAP achieves a much 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.

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 [13], [15] and [19]. We further note that 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.

Table 7: Results using PODEM* followed by LEAP

Circuit	PODEM* (backtrack limit of 5)				LEAP (backtrack limit of 500)				Time/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	

5 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 [7], FAN [5] and SOCRATES [15].

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 4 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 space equivalence [6] and dominance [4] relations to further prune the search space. Actually, search space equivalence relations provide a complementary scheme with respect to dependency-directed backtracking; search space 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. It is our goal to evaluate possible applications of LEAP to these areas, where the path sensitization problems are usually more difficult than in test pattern generation.

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 FOR-

TRAN”. In Proc. Int. Symp. Circuits and Systems, 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”. Proc. 22nd Fault Tolerant Comput. Symp., 1992.

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

[6] J. Giraldi, and M. L. Bushnell, “EST: The New Frontier in Automatic Test-Pattern Generation”. In Proc. 27th Design Automation Conf., 1990.

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

[8] T. Kirkland, and M. Ray Mercer, “A Topological Search Algorithm for ATPG”. In Proc. 24th Design Automation Conf., 1987.

[9] A. Lioy, “Adaptive Backtrace and Dynamic Partitioning Enhance APTG”. In Proc. Int. Conf. Computer Design, 1988.

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

[11] S. Mallela, and S. Wu, “A Sequential Circuit Test Generation System”. In Proc. Int. Test Conf., 1985.

[12] R. Marlett, “An Effective Test Generation System for Sequential Circuits”. In Proc. 23th Design Automation Conf., 1986.

[13] J. Rajski, and H. Cox, “A Method to Calculate Necessary Assignments in Algorithmic Test Pattern Generation”. In Proc. Int. Test Conf., 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 et. al., “SOCRATES: A Highly Efficient Automatic Test Pattern Generation System”. IEEE Trans. on Computer-Aided Design, vol. 7, no. 1, January 1988, pp. 126-137.

[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, July 1989, pp. 811-816.

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

[18] R. E. Tarjan, “Finding Dominators in Directed Graphs”. SIAM J. Comput., vol. 3, pp. 62-89, 1974.

[19] J. A. Waicukauski, et. al., “ATPG for Ultra-Large Structured Designs”. In Proc. Int. Test Conf., 1990.