Open access • Proceedings Article • DOI:10.1109/ICVD.2001.902656

A graph traversal based framework for sequential logic implication with an application to C-cycle redundancy identification — Source link

Jian-Kun Zhao, J.A. Newquist, Janak H. Patel

Institutions: University of Illinois at Urbana–Champaign

Published on: 03 Jan 2001 - International Conference on VLSI Design

Topics: Graph traversal, Sequential logic, Graph (abstract data type), Directed graph and Graph reduction

May 2000

UILU-ENG-00-2205
CRHC-98-14

University of Illinois at Urbana-Champaign

Do

A Graph Traversal Based Framework for Sequential Logic Implication with an Application to C-Cycle Redundancy Identification

Not

Jian-Kun Zhao, Jeffrey A. Newquist, and Janak H. Patel

Coordinated Science Laboratory
1308 West Main Street, Urbana, IL 61801

Take

SF 298 MASTER COPY KEEP THIS COPY FOR REPRODUCTION PURPOSES

REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1998	

4. TITLE AND SUBTITLE

A Graph Traversal Based Framework for Sequential Logic Application to C-cycle Redundancy Identification

5. FUNDING NUMBERS

SRC 97-DS-482
DABT63-95-C-0069

6. AUTHOR(S)

Jian-Kun Zhao, Jeffrey A. Newquist, and Janak H. Patel

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)

Coordinated Science Laboratory
University of Illinois
1308 W. Main St.
Urbana, IL 61801

8. PERFORMING ORGANIZATION REPORT NUMBER

(CRHC-98-14)
UILU-ENG-00-2005

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Semiconductor Research Corp.
P.O. Box 12053
Research Triangle Park, NC 27709-2053

DARPA
P.O. Box 12748
Ft. Huachuca, AZ
85670-2748

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This paper presents a new graph traversal based framework for sequential logic implication called GRAPH_SIMP. Due to the prohibitive time and space cost, few previous work target the discovery of sequential indirect implications that span multiple time frames. By using an efficient graph data structure and incorporating a graph reduction step into the implication generation process, our approach provides an efficient full support for sequential implication. Sequential logic implication has many useful applications, one of which is sequentially redundant fault identification. We show that sequential implications found by GRAPH_SIMP allow us to find more sequential redundancies than previously reported. Results of testing our implication algorithm against ISCAS89 circuits show that high implication coverage is essential to identifying redundant faults.

14. SUBJECT TERMS

ATPG, fault modeling

15. NUMBER IF PAGES

24

16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-102

A Graph Traversal Based Framework for Sequential Logic Implication with an Application to C-cycle Redundancy Identification

Jian-Kun Zhao (Contact Author)

Center for Reliable & High-Performance Computing
University of Illinois
1308 W. Main Street
Urbana, IL 61801
Phone: (217)-244-7174 FAX: (217)-244-5685
Email: j-zhao1@crhc.uiuc.edu

Jeffrey A. Newquist

Silicon Graphics, Inc.
2011 N. Shoreline Boulevard
Mountain View, California 94043-1389
Phone: (650)933-6255 FAX: (650)932-0925
Email: newquist@mti.sgi.com

Janak H. Patel

Center for Reliable & High-Performance Computing
University of Illinois
1308 W. Main Street
Urbana, IL 61801
Phone: (217)-333-6201 FAX: (217)-244-5685
Email: patel@crhc.uiuc.edu

Topics: ATPG and Fault Modeling

35 word abstract: The paper presents a new graph-based sequential implication framework which can discover large number of sequential indirect implications that span multiple time frames. Applying our implication results in sequential redundancy identification achieved better results than previously reported.

This research was supported in part by the Semiconductor Research Corporation under contract SRC 96-DP-109, in part by DARPA under contract DABT63-95-C-0069, and by Hewlett-Packard under an equipment grant.

A Graph Traversal Based Framework for Sequential Logic Implication with an Application to C-cycle Redundancy Identification

Abstract

This paper presents a new graph traversal based framework for sequential logic implication called GRAPH_SIMP. Due to the prohibitive time and space cost, few previous work target the discovery of sequential indirect implications that span multiple time frames. By using an efficient graph data structure and incorporating a graph reduction step into the implication generation process, our approach provides an efficient full support for sequential implication. Sequential logic implication has many useful applications, one of which is sequentially redundant fault identification. We show that sequential implications found by GRAPH_SIMP allow us to find more sequential redundancies than previously reported. Results of testing our implication algorithm against ISCAS89 circuits show that high implication coverage is essential to identifying redundant faults.

I Introduction

A number of different digital circuit analysis problems need to know the effects of asserting various logic values throughout a circuit: automatic test pattern generation (ATPG) [1], [2], [3], [4], untestable fault identification [5], [6], circuit optimization [7]- [10], and design verification [11]. Various solutions exist, and can be grouped into two major classes: *static learning* [1] and *dynamic learning.* In the context of logic circuits, learning refers to capturing the functional behavior of the circuit to more easily solve a given problem. Static learning algorithms are applied as a preprocessing step; in contrast, dynamic learning algorithms are performed as part of the circuit analysis procedure (e.g., during ATPG). In either case, logic implications are discovered and used to solve the various analysis problems.

A number of papers have dealt with implication algorithms[1] -[4], [10]-[15]. These algorithms are either structural based or Boolean satisfiability (SAT) based models. Kunz and Pradhan proposed a complete implication algorithm called recursive learning[15], which gurantees to find all necessary assignments under a partial set of node values. However, in practical implementation, the depth of recursion must be restricted to keep the time and space expense with reasonable bounds. As a result, some implications may not be found. In [14], Stoffel et al. proposed an implication engine which models recursive learning by AND-OR reasoning graphs. The working principle of AND-OR graph is the same as that of recursive learning in that both of them derive indirect implications by set intersection operation. Another graph-based implication engine proposed by Tafertshofter[16] inherits the characteristics of both structural based model and SAT based model. Their implication engine derives indirect implications through set operation and law of contraposition, which are considered as two major current techniques to discover indirect implications.

In this paper we propose a new graph-based implication framework which is efficient in terms of both time and space. We focus on discussing the construction phase of this implication engine, which can be viewed as a static learning procedure. Compared with dynamic learning, static learning has several advantages. Dynamic learning is typically applied in the context of an ATPG, or other analysis algorithm, during branching steps. Implications found in dynamic learning are only valid under a specific situation of assignments, which limits the scope of discovered implications and causes common implications to be re-learned in another situation. In contrast, implications found through static learning are valid in all branching situations. By using statically learned implications, a branch-and-bound algorithm will spend considerably less time backtracking from incorrect decisions. Moreover, it is usually expensive to discover indirect implications during dynamic learning, whereas many indirect implications, especially those *unilateral indirect implications*[2], can be easily found in static learning . Since indirect implications play a critical role in many processes, it is of utmost importance to perform static learning as a preprocessing phase in many applications.

Our approach distinguishes from previous approaches in several aspects. First, few previous papers discuss sequential indirect implication that may involve multiple time frames. Even though some of the implication algorithms proposed before may be applied to sequential circuits, the implication engines used are mainly combinational and sequential indirect implications that span multiple time frames are not targeted. The reason for this may lie in the prohibitive time and space costs. The implication algorithm proposed here fully supports sequential indirect implication as well as combinational indirect implications. Experimental results show that the execution time spent by our algorithm is within reasonable bound. A second characteristic of our implication algorithm is the small memory space requirement, considering the huge number of indirect implications found. Usually, indirect implications are either put in external data structures or included into the implication engines. Neither of the two ways outperforms the other in saving storage space for indirect implications. Our experiments show that an extremely large number of sequential indirect implications can be derived in static learning, which causes storage space issue if no explicit measures are taken for space reduction. Our algorithm overcomes this issue by incorporating a graph reduction procedure into the construction process of the implication engine. This graph reduction approach significantly reduces the space consumption, making sequential implication a feasible and attractive tool to apply in many applications.

Indirect implications are very useful in many processes, such as logic optimization[10], logic verification[17], ATPG[2], and redundancy identification [5], [6], [7], [18]. In the later part of this paper, we present an application of our implication algorithm to sequential C-cycle redundancy identification using the FIRES algorithm proposed by Iyer et al.[6]. We also propose an efficient procedure called *STEM_ANALYSE*, to do the unobservability validation on stems, which is a critical step in FIRES. Applying the results of our implication algorithm, we achieved better results in sequential redundancy identification than the original FIRES did.

The rest of the paper is organized as follows. Section II discusses the basic concepts and data structures supporting the implication algorithm, Section III presents the implication algorithm, Section IV describes an application of the implicaton algorithm — C-cycle redundancy identification, Section V gives the experimental results, and Section VI concludes the paper.

II Basic Concepts and Data Structures

A Basic terms and concepts

We first define a few terms that will be used frequently throughout the algorithm description.

1. $[N, v, t]$: assign logic value v to node N in time frame t;

   (In combinational circuits, t is ignored. The default value for t is 0.)

2. $[M, w] \rightarrow [N, v, t]$: assignning value w to node M in the current time frame (time frame 0) implies another assignment: value v on node N in time frame t.

3. $impl[N, v, t]$: set of implications resulting from setting node N in time frame t to value v. In case t is not specified, $impl[N, v]$ represents the set of implications resulting from setting node N in the current time frame to value v.

Time frames are bounded by D flip-flops and the *current time frame* is always time frame 0. When implication is propagated across a D flip-flop, the time frame will be incremented or decremented correspondingly. For description convenience, for combinational circuits, the time frame part is omitted in assignment representation. For example, assignning value 0 to node A in a combinational circuit is represented as $[A, 0]$ instead of $[A, 0, 0]$.

For sequential circuits, static implication procedure is performed on all assignments in the *current time frame* (time frame 0).

The following laws are used in the implication generation process:

1. <u>Deriving implication set for an assignment in time frame t (non-current time frame)</u>

   $impl[N, v, t] = \{[M, w, t' + t] \mid [M, w, t'] \in impl[N, v]\}$;

2. <u>Forward implication</u>: If all the input values of a gate are known or one of the inputs is at the controlling value of the gate, then the output value of this gate can be uniquely determined from its input values. For example, for an AND gate, if one of the inputs is set to 0, then the output is 0; if all of the inputs are set to 1, then the output is 1.

3. <u>Backward implication</u>: Suppose we are generating implications of $[N, a]$. Let G be an unjustified gate in time frame t with m unspecified input nodes S_i and a specified output node Y.

   if G is an AND gate:

   if $[Y, 0] \in impl[N, a]$, $impl[N, a] = impl[N, a] \cup (\bigcap_{i=1}^{m} impl[S_i, 0, t])$

   if $[Y, 1] \in impl[N, a]$, $impl[N, a] = impl[N, a] \cup (\bigcup_{i=1}^{m} impl[S_i, 1, t])$

   If $Y = 1$, then all gate inputs are 1, and we can add the implications of setting these inputs to 1 to our list of implications. If $Y = 0$, we find implications resulting from setting each input to 0, and since at least one input must be 0, we add the common implications found. $impl[S_i, 0/1, t]$ can be derived using the first basic law described above.

if G is an OR gate:

if $[Y, 1] \in impl[N, a]$, $impl[N, a] = impl[N, a] \cup (\bigcap_{i=1}^{m} impl[S_i, 1, t])$

if $[Y, 0] \in impl[N, a]$, $impl[N, a] = impl[N, a] \cup (\bigcup_{i=1}^{m} impl[S_i, 0, t])$

4. Extended backward implication: For gate G in time frame t with m unspecified input nodes S_i and a specified output node Y,

if G is an AND gate:

if $[Y, 0] \in impl[N, a]$ and $[Y, 0]$ is unjustified by gate inputs S_i, then

$impl[N, a] = impl[N, a] \cup (\bigcap_{i=1}^{m} Forward\_Imply(impl[N, a] \cup impl[S_i, 0, t]))$

Forward_Imply is a procedure performing forward implications on a set of node assignments.

if G is an OR gate:

if $[Y, 1] \in impl[N, a]$ and $[Y, 1]$ is unjustified by gate inputs S_i, then

$impl[N, a] = impl[N, a] \cup (\bigcap_{i=1}^{m} Forward\_Imply(impl[N, a] \cup impl[S_i, 1, t]))$

5. Transitive law: If $[M, w] \rightarrow [N, v, t_1]$ AND $[N, v] \rightarrow [L, y, t_2]$, then $[M, w] \rightarrow [L, y, t_1 + t_2]$. In set notation, if $[N, v, t_1] \in impl[M, w]$ and $[L, y, t_2] \in impl[N, v]$, then $[L, y, t_1 + t_2] \in impl[M, w]$.

6. Contrapositive law: If $[M, w] \rightarrow [N, v, t]$, then $[N, \overline{v}] \rightarrow [M, \overline{w}, -t]$. In set notation, if $[N, v, t] \in impl[M, w]$, then $[M, \overline{w}, -t] \in impl[N, \overline{v}]$. This law enables the algorithm to discover unilateral indirect implications [2].

7. <u>Conflicting assignments</u>: If $[M, w] \rightarrow [N, v, t]$ AND $[M, w] \rightarrow [N, \overline{v}, t]$, then $[M, w]$ is an impossible setting. In other words, M will permanently hold the value $\overline{w}$. This law enables the algorithm to detect those nodes with constant values. Our algorithm includes conflict checking. If conflicts are not checked, the false values will create many useless new implications during execution of the algorithm, thus affecting the performance.

The contrapositive law discovers at trivial cost many indirect implications that would cost at least one recursion depth to be discovered using recursive learning approach [15]. Figure 1 shows two examples of this advantage.

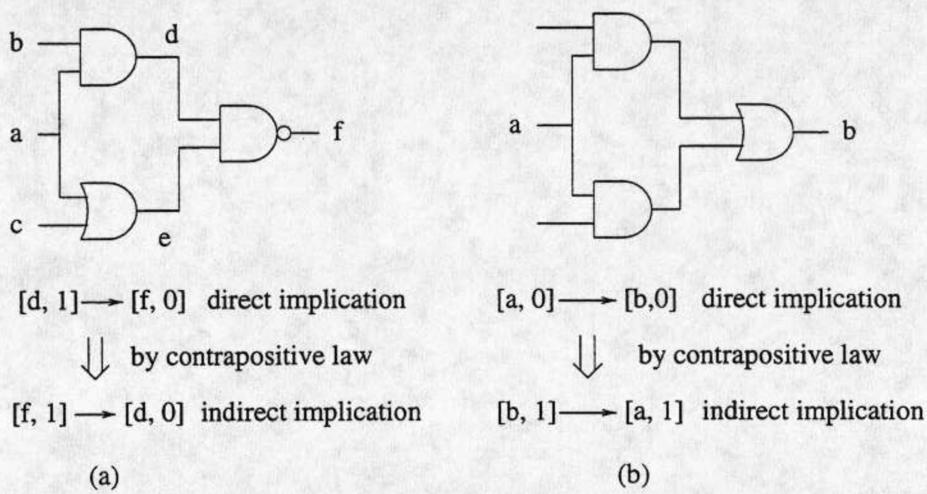


Figure 1: Contrapositive implication example.

Extended backward implication further discovers some indirect implications that cannot be discovered by simply applying the transitive and contrapositive laws.

B Data structure — implication graph

a. Graph representation

A directed graph is used to represent the implication relationship in the circuit. We call this graph *implication graph*. Each graph node corresponds to a circuit node assignment. Each directed edge represents an implication. In implication graphs of sequential circuits, each edge has a weight that indicates the time distance (i.e. the number of time frames) that this implication spans. Figure 2 shows an example of the implication graph of a sequential circuit.

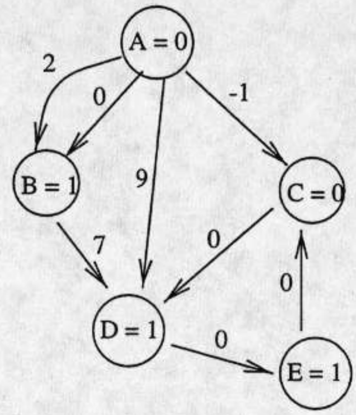


Figure 2: An implication graph example.

The weight of edge is an integer. Its range depends on the time frame constraint of the implication procedure. In our implementation, we restrict the implication propagation within 21 frames (10 backward time frames, 10 forward time frames, and the current time frame). So the edge weight ranges from -10 to 10.

The transitivity nature of the implication relationship is also reflected in the implication graph. For example, in Figure 2, since $[A, 0] \rightarrow [B, 1, 2]$ and $[B, 1] \rightarrow [D, 1, 7]$, implication $[A, 0] \rightarrow [D, 1, 9]$ can be derived by transitive law. Therefore we define the implication relationship in an implication graph as follows:

Definition 1 *Graph node A implies graph node B with time distance t if there exists a path of length t from A to B in the implication graph.*

Note that the length of a path could be negative.

b. Graph reduction

By transitive law the implications of a circuit node assignment (i.e. a graph node) can be collected by traversing from the corresponding graph node, in other word, the implications are contained in the transitive closure of the graph node. Therefore, this graph representation has great potential in reducing the storage space for implications, by deriving the simplest version of the implication graph without changing the transitive closure of the graph. This procedure is known as *transitive reduction* [19] and defined as follows:

Definition 2 *A transitive reduction of a directed graph G = (V, E) is defined to be any graph G' = (V, E') with as few edges as possible, such that the transitive closure of G' is equal to the transitive closure of G.*

Transitive reduction can be done in a much easier way if the graph is acyclic. However, this is not the case for the implication graph discussed here, in which there may exist many cycles or strongly connected components. A strongly connected component actually forms an equivalence class, in which all nodes are mutually implied and therefore equivalent in the sense of logic implication. So we first identify those strongly connected components, merge them into single nodes, and then perform the transitive reduction procedure on the graph. As an example, Figure 3 shows how the implication graph in Figure 2 is reduced to its simplest version.

Algorithms invloved in this 3-step reduction procedure will be discussed in detail in a seperate section later.

c. Graph traversal

The implications of a node assignment reside in the transitive closure of the corresponding graph node and are collected by traversing from the graph node. Therefore, graph traversal is a key step in the implication procedure. There are two major ways to traverse a graph: depth first search (DFS) and breadth first search (BFS). In this work, depth first search is used in traversal.

d. Graph initialization Graph initialization is performed at the beginning of the static implication procedure. It is a procedure that maps the functions of the circuit elements to

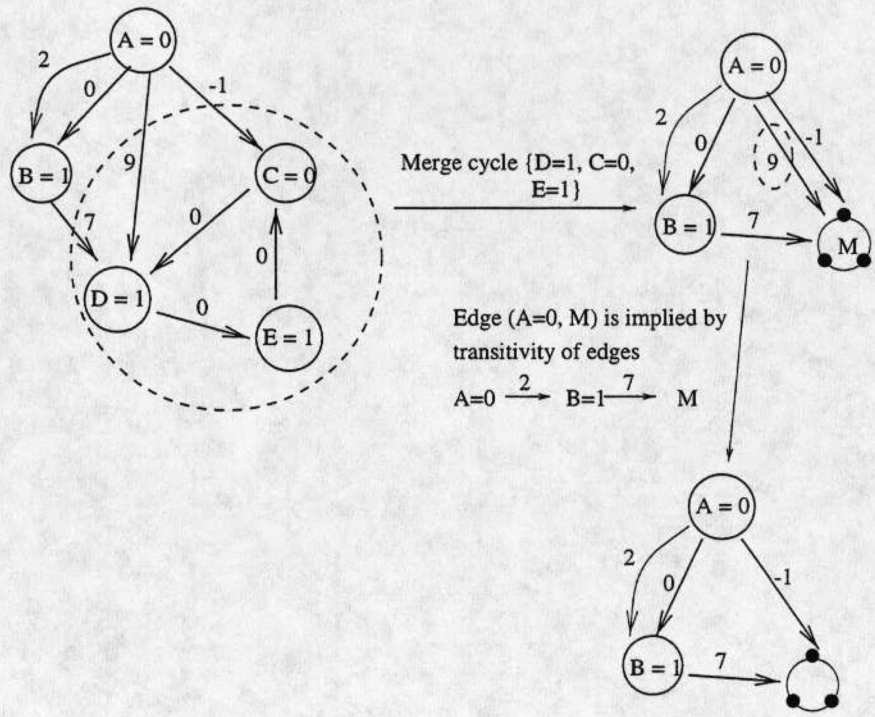


Figure 3: A graph reduction example.

a graph representation. There are two major things done in this procedure

. Create the graph nodes. Each node represents a circuit node assignment.

. Add the direct implications local to the gates in the circuit.

Since the purpose of the graph approach is to reduce memory space consumption, transitively implied edges should be avoided as early as in the initialization phase. Figure 4 shows an example of graph initialization. The original circuit is shown in Figure 4(a), and the initial version of the implication graph, using only local implications, is shown in Figure 4(b).

e. Implication generation

The implication engine searches for new implications by iteratively performing forward and backward implications. Forward implication is incorporated within the graph traversal procedure. Figure 5 shows an example of how forward implication is performed in graph

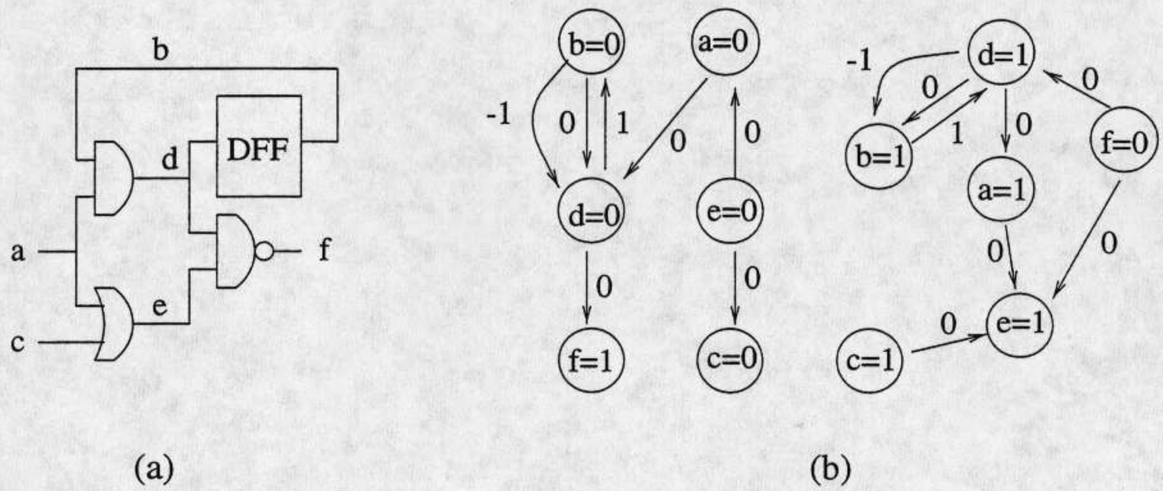
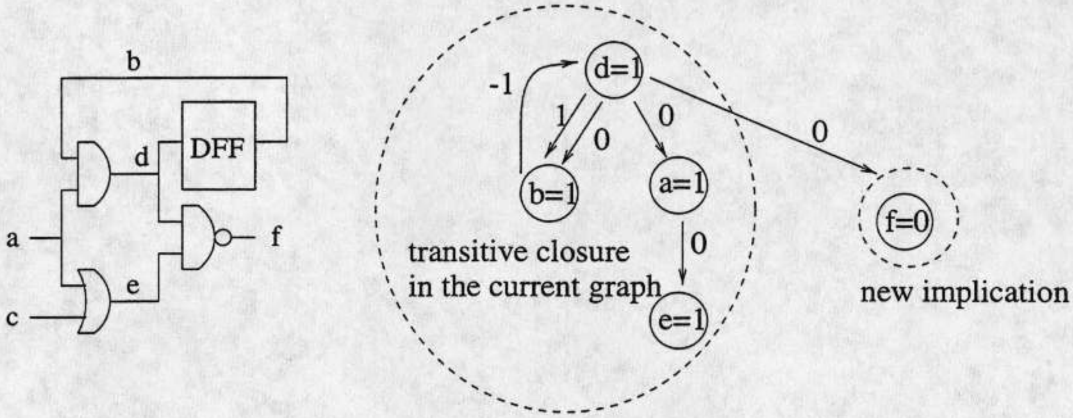


Figure 4: A graph initialization example.

traversal. In Figure 5(b), implication is currently performed on $[d,1]$. The current implication set of $[d,1]$, $\{[d,1,0],[b,1,0],[b,1,1],[a,1,0],[e,1,0],[f,0,0]\}$, is contained in the transitive closure of $[d,1]$. $[d,1,0]$ and $[e,1,0]$ are both present in the current implication set. Therefore $[f,0,0]$ is learned by evaluating the NAND gate in the circuit. In our implementation, the evaluation procedure is event-driven, i.e. evaluation on a gate is performed when the number of inputs with known values reaches the threshold value that make the gate ready for evaluation. For common gate types, such as AND and OR, the threshold value is the number of gate inputs instead of 1, since controlling value propagation is reflected in the initialized graph.

The forward implication procedure basically does graph traversal while keeping an eye on circuit nodes ready for evaluation and adding new implications to the graph conditionally. Also, the contrapostive law is applied whenever a new implication is added to the graph. Many indirect implications are discovered through this way at trivial time cost. Graph traversal combined with forward implication can also be viewed as an independent dynamic learning procedure.



(a) circuit (b) implication generation on d=1

Figure 5: A forward implication example.

III The Algorithms

In this section we present several procedures involved in this graph-based static implication algorithm. The algorithm is called $GRAPH\_SIMP$.

Figure 6 shows the outline of the main function. In each main iteration, graph reduction is first performed and then implication generation.

```
GRAPH_SIMP()
  Graph_Initialize();       // Implication graph initialization
  While implications found  // main iteration
    Graph_Reduce();         // graph reduction
    For each circuit node N
      Imply(N,0);           // implication generation
      Imply(N,1);
```

Figure 6: Main function $GRAPH\_SIMP$.

Figure 7 shows the outline of the implication generation procedure — $Imply$. The *AddNew* procedure (Figure 8) adds new implications and applies contrapositive law as well. Procedure $Forward\_Imply$, which is also frequently called during extended backward implication, is shown in Figure 9. The extended backward implication is described in the fourth basic law in Section II.

```
Imply(N: node; V: logic-value)
  If [N,V] or [N,V̄] is a constant
    return;
  else
    Forward_Imply(N, V);
    AddNew();
    For each unjustified implication [M,w,t]
      Extended_Backward_Imply(M,w,t);
      AddNew();
```

Figure 7: Procedure *Imply*.

```
AddNew( )
  For every new implication [x,a,t] found
    impl[N,v] = impl[N,v] U {[X,a,t]};
    impl[X,ā] = impl[X,ā] U [N,v̄,-t]
    If [X,ā,t] also belongs to impl[N,v]
      Then mark [N,v] as impossible;
           return;
```

Figure 8: AddNew.

Figure 10 shows the outline of the procedure *Graph_Reduce*.

Procedure *Graph_Reduce* consists of two major steps: strongly connected component identification and merging (The merged node is considered as a single node thereafter.), and removal of transitively implied edges. Procedure *Find_Cycle*[20], which identifies the strongly connected components, is shown in Figure 11.

In our implementation, in merging a strongly connected component, one node in the component is selected as the representative of the component, and all incoming and outgoing edges of the nodes in the component are hooked to this representative. The original nodes within the component are then kept in a seperate record. During graph traversal, if a merged

```
Forward_Imply(N: node; V:logic-value )
  // An evaluation queue is maintained to hold gates ready for
  // evaluation. Each event in queue has the form [N, T], indicating
  // that circuit node N in time frame T is ready for evaluation

  While there are untraversed outgoing edges from [N,V]
  [ Traverse-Watch(); // Traverse from these untraversed edges,
                      // keep watching for gates that become
                      // ready for evaluation and add them
                      // to the evaluation queue.
    For each event [N,T] in the evaluation queue
      [ Evaluate(N,T);  // Evaluate gate N in time frame T;
      [ AddNew();
```

Figure 9: Procedure $Forward\_Imply$.

```
Graph_Reduce()
  Find_Cycle();// Identify strongly connected components and
               // merge them into single nodes
  Remove_Implied_Edge();  // Remove transitively implied edges.
```

Figure 10: Procedure $Graph\_Reduce$.

```
Find_Cycle()
  // This procedure consists of two rounds of depth-first-search's (DFS).

  Depth_First_Search(); // First depth-first-search
                        // In this round, the finishing order
                        // of search is recorded.
  Inverse_Depth_First_Search();
                        // Second depth-first-search
                        // In this round, search is :
                        //   1. performed on the nodes in decreasing
                        //      finishing order in  the first DFS.
                        //   2. along the reverse directions of the edges.
                        // Each tree formed in this traversal corresponds
                        // to a strongly connected component,

  Merge_Cycles(); // Merge the strongly connected components identified
                  // in the second DFS into single nodes.
```

Figure 11: Procedure $Find\_Cycle$[20].

node is reached, the original nodes in the component are visited first and then traversal proceeds from the representative node.

To simplify the problem, only the combinational strongly connected components , i.e. those strongly connected components in which there is a path of length 0 between each pair of nodes, are identified and merged.

IV Sequential Redundancy Identification Using Sequential Implications

One useful application of sequential implication is sequential redundant fault identification. Our previous work [21] illustrated that applying our algorithm SIMP (a combinational implication algorithm) to FIRE[5] , (a combinational redundancy identifier) finds more combinational redundancies than reported in [5]. In this section, we briefly review FIRES, a sequential *c-cycle redundancy* identifier, developed by Iyer et al.[6]. A c-cycle redundant fault, is a fault for which no test sequence exists after powering up the faulty circuit and applying c clock cycles[6].

The FIRES algorithm proposed in [6] is a fault-independent redundancy identification algorithm for sequential circuits. It identifies faults which require a conflict on a stem (a gate with two or more fanouts) as a necessary condition for detection. Since a node in a circuit can only achieve one value at a time, these faults are redundant. The algorithm works by first applying a '0' to a stem and collecting faults which are either not activated or not propagated. Unactivated faults are found through implication analysis. Unpropagated faults are found by finding unobservable lines caused by controlling values. Then the algorithm applies a '1' to the stem and determines faults which are not activated or not propagated in the same manner. Common faults between the two tests are the redundant faults. The outline of the FIRES algorithm is shown in Figure 12.

We applied our implication results to FIRES. One important issue involved in fault

```
FIRES( )
  S_untestable = empty;
  For each circuit node N
  [ Sequentially imply on N=0
      S_0 = all lines that become uncontrollable or unobservable
            under assignment N=0;
    Sequentially imply on N=1
      S_1 = all lines that become uncontrollable or unobservable
            under assignment N=1;
    S_untestable = S_untestable ∪ (S_0 ∩ S_1);
```

Figure 12: FIRES procedure.

collection in FIRES is unobervability validation for those stems that have all fanouts marked unobservable during the fault collection. As we know, a stem maybe observable even if all its fanouts are unobservable due to the fact that the faulty effects may be propagated onto multiple fanout branches and then reconverge, making the fault on the stem observable. This is also known as multiple path sensitization issue and often happens on reconvergent gates.

In FIRES, the unobservability propagates onto a stem s^i (the copy of line l at time i) if

1. The fanouts of s^i are marked as unobservable at time i.
2. For every fanout f^i of s^i, there exists at least one set of lines $\{p^i\}$, such that
   - . f^i is unobservable because of uncontrollability indicators on every line in $\{p^i\}$; and
   - . there is no sequential path from s^k, $i \leq k \leq j$, to any line in $\{p^i\}$.

Stem unobservability validation in FIRES aims to verify there is no sequential path from s^k, $i \leq k \leq j$, to any line in $\{p^i\}$. The original paper didn't give the concrete implementation of this validation step. As we think this validation step plays a critical role in the fault collection — it determines whether the unobservability can be progapated further backward, we present our approach here. We solve this problem in a conservative way. Our method filters out those stems that have greater-than-zero chance to be observed. This approach guarantees that after filtering the remaining stems are unobservable. The combinational stem analysis we used in implementing FIRE [5](combinational redundancy identification) is shown in Figure 13. Our sequential stem analysis procedure is based on the similar working principle. It marks s^k $(i \leq k)$ and their fanouts as "affected" and proceeds the analysis in increasing order of circuit level and time frame. It also distinguishes between the nodes affected by s^i and the nodes affected by stems s^k $(k > i)$ in subsequent time frames so as to terminate the procedure when the faulty effect on s^i cannot be propagated further.

```
STEM_ANALYSE(S)
  // A queue is maintained for each level in the circuit
  Mark S and all its fanouts as "affected";
  Insert the successors of S into queues corresponding to their levels

  Go through each queue Q in increasing level order
    While Q is not empty
      Take a node A from the head of Q
      If A is a primary output
        Return(OBSERVABLE); // The faulty effect could possibly
                            // affect primary output A
      else
        If every input of A is either marked "affected" or
          at non-controlling values
          Mark A and its fanouts as "affected";
          Insert the successors of A into corresponding queues;

  Return(UNOBSERVABLE);// Stem S has passed examination
                       // and is guaranteed unobservable.
```

Figure 13: Combinational stem unobservability validation procedure

We also applied our implication results to FUNTEST[22], a sequential untestable fault identifier based on the single fault ATPG theorem provided in [23]. FUNTEST is simliar to FIRES in structure. The main difference between them is that FUNTEST doesn't cross the time boundaries in fault collection whereas FIRES does. We also achieved better results than reported in [22].

V Experimental Results

This section presents the experimental results for ISCAS89 sequential benchmark circuits. Both the proposed sequential circuit implication algorithm and the sequential redunancy identification procedure were implemented in C++. Experiments were run on an HP 9000 workstation.

Table 1 shows the results of our static sequential implication algorithm GRAPH_SIMP. For each circuit, the total number of implications that can be derived from the generated implication graph (*#impl.*), the actual number of edeges in the graph (*#edge*), the maximum edge weight in the graph (*max |edge weight|*), the number of graph nodes in the original graph right after initialization (*#nodes(original)*), the number of graph nodes after equivalence merging (*#nodes(after merging)*), the number of constants (*#Cons.*) identified, and the CPU time are shown. Constants are not counted as implications in these results. We do not discriminate between stems and fanout branches; therefore, they are considered to be the same node. Compared with our previous work which stores the implications for each node in a seperate set, the memory consumption is very low for this graph-based implication engine. The percentage reduction can be approximated by $\frac{(\#impl+\#nodes(original))-(\#edge+\#nodes(aftermerging))}{\#impl+\#nodes(original)}$. In this experiment, the percentage reduction ranges from 92.3% to 99.6%.

max |edge weight| indicates the maximum time offset of the implications shown in the graph (*not* including those implied edges). In our implementation, we restrict the implication propagation within 10 backward and 10 forward time frames. It is interesting to see that quite a few circuits have maximum edge weight of 10 *even after transitive reduction.* The maximum edge weight for these circuits may go even beyond 10 if we set the time offset contraint larger.

Table 1: Graph-based static implication results on ISCAS89 circuits

Ckt	**#impl.**	**#edge**	**max |edge weight|**	**#nodes (original)**	**#nodes (after merging)**	**#Cons.**	**time**
s208	39588	1227	10	246	158	0	13.6s
s298	19238	891	8	284	158	3	17.1s
s344	14682	947	4	390	236	5	1.8s
s349	14682	947	4	392	236	6	1.9s
s382	53085	1875	10	376	226	0	137.3s
s386	32574	1255	3	358	234	3	12.0s
s400	58799	1977	10	388	234	1	150.2s
s420	262565	3618	10	506	334	0	99.6s
s444	74353	2419	10	422	254	2	252.3s
s510	44916	2932	4	486	408	0	31.3s
s526	50054	2122	10	446	286	1	83.4s
s641	64866	1576	10	914	310	0	1.0s
s713	66432	1726	10	940	310	16	1.6s
s820	62058	3040	3	662	472	0	43.3s
s838	1310185	8569	10	1026	686	0	695.3s
s953	244118	5061	4	926	706	0	87.3s
s1196	73562	5141	1	1150	836	0	10.2s
s1238	74764	5745	1	1108	912	0	12.6s
s1423	143198	4851	10	1506	1072	0	67.4s
s1488	154286	10066	2	1372	1076	0	146.4s
s1494	154550	10049	2	1360	1090	0	162.8s
s5378	2899860	11476	10	6084	1711	404	1347.4s
s9234	4531017	25229	10	11766	3818	26	5.5h
s13207.1	8146713	41780	10	17748	5509	296	3.7h
s15850.1	15604841	50208	10	21092	7486	76	2.0h
s35932	10866538	98047	3	36296	26846	0	3.8h
s38417	29811195	106218	10	48334	19339	131	7.5h
s38584	54544728	165901	10	42350	23739	254	7.5h

Table 2 compares the results of applying our static implication results to FIRES and the results of the original FIRES implementation. The number of c-cycle redundancies identified by each procedure, the number of 0-cycle redundancies, and the maximum c, are shown in the table for each circuit. Again, the large number of implications found by our implication algorithm leads to the superior performance over the orginal FIRES.

Table 2: Results of c-cycle redundancy identification

	FIRES[6]				w/ GRAPH_SIMP			
Circuit	**Red.**	**(sec)**	**0-cycle**	**Max. c**	**Red.**	**(sec)**	**0-cycle**	**Max. c**
s298	-	-	-	-	3	0.2	2	1
s344	-	-	-	-	5	0.2	4	1
s349	2	0.3	2	0	7	0.2	4	1
s382	-	-	-	-	4	0.4	3	1
s386	27	0.6	0	2	60	0.4	60	0
s400	1	1.2	0	2	8	0.5	8	0
s444	11	1.5	11	0	16	0.6	13	1
s526	-	-	-	-	6	0.5	5	1
s713	32	0.8	32	0	32	0.6	32	0
s953	-	-	-	-	5	2.2	5	0
s1238	6	2.8	6	0	12	1.3	12	0
s1423	5	1.5	5	0	9	1.5	9	0
s1494	1	1.7	1	0	1	2.0	1	0
s5378	366	69.3	48	11	796	151.2	224	3
s9234	270	142.8	165	6	911	209.2	892	1
s13207.1	-	-	-	-	391	171.4	232	1
s15850.1	-	-	-	-	320	471.1	290	1
s35932	3984	684.8	3984	0	3984	986.3	3984	0
s38417	147	386.2	115	1	343	577.8	333	1
s38584	1437	272.0	1052	3	1460	2505.1	1145	1

Table 3 compares the results of applying our static implication results to the FUNTEST procedure and the results of the original FUNTEST implementation. The number of untestable faults identified by each procedure is shown in the table for each circuit. "-" represents "data not available", i.e. result for the corresponding circuit was not reported in [22]. Again, the large number of implications found in the static learning phase leads to the superior performance over the orginal FUNTEST.

Table 3: Results of untestable fault identification using FUNTEST

	FUNTEST[22]		w/ SIM P	
Circuit	**Unt.**	**(sec)**	**Unt.**	**(sec)**
s298	-	-	3	1.2
s344	-	-	3	1.0
s349	2	0.2	5	1.0
s382	-	-	4	3.4
s386	27	0.5	60	2.6
s400	1	0.6	8	3.8
s444	8	0.5	16	5.0
s526	-	-	2	3.9
s713	32	0.3	32	4.0
s953	-	-	5	17.7
s1238	6	3.0	12	7.1
s1423	5	0.7	9	9.74
s1494	1	1.8	1	13.4
s5378	210	25.6	772	421.9
s9234	277	126.1	923	697.8
s13207.1	-	-	376	992.5
s15850.1	-	-	317	2385
s35932	3984	340.6	3984	2939
s38417	125	66.9	332	2601

VI Conclusion

This paper has presented a new graph-traversal based framework of sequential implication for use in many applications such as c-cycle redundancy identification. By iterative method, contrapositive law, and extended backward implication, our implication procedure discovers at low cost a large number of indirect implications. To prevent the storage space requirement for the large number of indirect implications found from becoming the bottleneck of this implication algorithm, a graph reduction step, which consists of equivalence class merging and transitive reduction, is incorporated into the implication generation process.

To show the efficiency of this algorithm, the static implication results were applied to sequential c-cycle redundancy identification. Incorporating the implication algorithm proposed here in the c-cycle redundant fault identification achieved better results than previous work[6].

The implication framework proposed in this paper can also be applied to circuits with tri-state elements. The flexible structure of this framework allows easy extension to circuits with new gate types and multiple-value logic. Our implication algorithm can be efficiently applied to many other processes as well as redundancy identification. In our future work, we will investigate the effects of including this implication engine into ATPG and logic verification.

References

[1] M. Schulz and E. Auth, "Improved Deterministic Test Pattern Generation with Applications to Redundancy Identification," *IEEE Transactions on Computer-Aided Design*, vol. 8, pp. 811–816, July 1989.

[2] W. Kunz and D. Pradhan, "Accelerated Dynamic Learning for Test Pattern Generation in Combinational Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 684–694, May 1993.

[3] M. Schulz, E. Trischler, and T. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Transactions on Computer-Aided Design*, vol. 7, pp. 126–137, Jan. 1988.

[4] S. Chakradhar, V. Agrawal, and S. Rothweiler, "A Transitive Closure Algorithm for Test Generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 1015–1028, July 1993.

[5] M. Iyer and M. Abramovici, "FIRE: A Fault-independent Combinational Redundancy Identification Algorithm," *IEEE Transactions on VLSI Systems*, vol. 4, pp. 295–301, June 1996.

[6] M. Iyer, D. Long, and M. Abramovici, "Identifying Sequential Redundancies without Search," in *Proceedings of the 33rd Design Automation Conference*, pp. 457–462, 1996.

[7] P. Menon and H. Ahuja, "Redundancy Removal and Simplification of Combinational Circuits," in *Proceedings of IEEE VLSI Test Symposium*, pp. 268–273, 1992.

[8] L. Entrena and K. Cheng, "Combinational and Sequential Logic Optimization by Redundancy Addition and Removal," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, pp. 909–916, July 1995.

[9] M. C. D. Pradhan and W. Kunz, "LOT: Logic Optimization with Testability – New Transformations Using Recursive Learning," in *Proceedings of IEEE International Conference on Computer-Aided Design*, pp. 318–325, 1995.

[10] W. Kunz and P. Menon, "Multi-level Logic Optimization by Implication Analysis," in *Proceedings of IEEE International Conference on Computer-Aided Design*, pp. 6–13, 1994.

[11] W. Kunz, D. Pradhan, and S. Reddy, "A Novel Framework for Logic Verification in a Synthesis Environment," *IEEE Transactions on Computer Aided Design*, vol. 15, pp. 20–32, Jan. 1996.

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

[13] S. T. Chakradhar and V. D. Agrawal, "A Transitive Closure Based Algorithm for Test Generation," *Proc. ACM/IEEE Design Automation Conf.*, pp. 353–358, June 1991.

[14] D. Stoffel, W. Kunz, and S. Gerber, "And/Or Reasoning Graphs for determining Prime Implicants in Multi-level Combinational Networks," *Asia and South Pacific Design Automation Conference*, pp. 529-538, Jan. 1997.

[15] W. Kunz and D. Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits," *Proc. Int. Test Conf.*, pp. 816-825, Sept. 1992.

[16] P. Tafertshofter, A. Ganz, and M. Henftling, "A SAT-Based Implication Engine for Efficient ATPG, Equivalence Checking, and Optimization of Netlists", *Proceedings of IEEE/ACM International Conference on Computer-aided Design*, pp. 648-655, Nov. 97.

[17] Wolfgang Kunz, "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning," *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp 538-543, 1993.

[18] P. R. Menon and M. Harihara, "Redundancy Identification and Removal in Combinational Circuits," *Proceedings of IEEE International Conference on Computer Design*, pp 290-293, 1989.

[19] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, pp 219-220, 1974.

[20] T. H. Corman, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, pp 488-493, 1994.

[21] J. K. Zhao, E. Rudnick, and J. Patel, "Static Logic Implication with Application to Redundancy Identification," in *Proceedings of the 15th IEEE VLSI Test Symposium*, 1997.

[22] M. Iyer and M. Abramovici, "Sequentially untestable faults identified without search," *Proceedings of IEEE International Test Conference*, pp 259-266, 1994.

[23] V. D. Agrawal and S. T. Chakradhar, "Combinational ATPG Theorems for Identifying Untestable Faults in Sequential Circuits," *European Test Conference*, pp 249-253, 1993.