FIRE: A Fault-Independent Combinational Redundancy Identification Algorithm

Mahesh A. Iyer and **Miron Abramovici**

Abstract

FIRE is a novel Fault-Independent algorithm for combinational REdundancy identification. The algorithm is based on a simple concept that a fault which requires a conflict as a necessary condition for its detection is undetectable and hence redundant. FIRE does not use the backtracking-based exhaustive search performed by fault-oriented automatic test generation algorithms, and identifies redundant faults without any search. Our results on benchmark and real circuits indicate that we find a large number of redundancies, much faster than a test-generation-based approach for redundancy identification. However, FIRE is not guaranteed to identify all redundancies in a circuit.

Index terms: Redundancy identification, automatic test generation, logic synthesis

1. Introduction

An automatic test generation (ATG) algorithm spends a large portion of its time dealing with undetectable faults. A fault is *undetectable* if there exists no test to detect it. A fault is identified as undetectable only when the test generator fails to generate a test to detect it, after exhausting the search space. Any ATG algorithm relying on exhaustive search can identify all single undetectable faults, if given enough time. Practical limits imposed to keep the computation time within reasonable bounds may result in aborting the search process for "difficult" target faults. Most abandoned faults are undetectable. Thus, the test generator exhibits its worst-case behavior for undetectable faults.

In a combinational circuit, an undetectable stuck fault is always caused by a redundancy. Such a circuit can be simplified by removing the redundant region associated with the undetectable (redundant) fault. In addition to complicating ATG, redundancies have many other detrimental effects. The presence of a redundant fault may preclude the detection of other faults in the circuit and may convert a complete detection test set into an incomplete one [1]. Redundancies increase the chip area, the power consumption and often the propagation delays in the circuit. Redundancies may also unnecessarily reduce the yield of the IC manufacturing process [2]. For example, many combinationally redundant faults become detectable with I_{DDQ} testing. Although the circuit remains fully operational in the presence of a redundant fault, I_{DDQ} testing will reject that faulty circuit and will result in a yield loss. Another problem is that the presence of abandoned faults may preclude achieving the desired fault coverage.

Redundancy identification (RID) using ATG is *indirect* and is a byproduct of test generation. In contrast, a *direct* RID technique finds redundancies without using the exhaustive search process of ATG. Direct RID techniques can be classified as *static* or *dynamic*. Static RID techniques work as preprocessing to ATG, whereas dynamic RID techniques work during ATG. These low-cost direct techniques can significantly reduce the CPU time used in ATG.

Static RID techniques based on an analysis of reconvergent fanout structures have been proposed in [3,4]. Their methods are based on stem analysis and they determine undetectable faults by performing an explicit analysis of dominators. Their method is based on implications and they explicitly analyze every reconvergent gate of every stem to discover controlling value implications that cause undetectability. In contrast, the analysis performed by FIRE is much simpler, and dominators and reconvergent gates are analyzed only implicitly.

Most indirect RID methods [5,6,7,8,9,10] attempt to accelerate ATG using several preprocessing and dynamic techniques. More recently, other improved combinational test generators use powerful implication procedures [11] or boolean satisfiability formulations [12]. Nevertheless, all these approaches are fault-oriented and require a branch-and-bound search to accomplish RID as a byproduct.

Once a redundant region associated with a redundant fault is removed, new redundancies may be introduced in the circuit. In earlier work [13], we proposed a dynamic RID technique to identify a subset of these newly created redundancies. This technique was based on analyzing combinations of values that become *illegal* as a result of the removal and was used to avoid repeated runs of ATG and fault simulation. We described a general method to find faults for which a given combination of values (on a set of lines) is a necessary condition for detection.

In this paper, we extend this method for static RID. Some results of this work were reported in [14,15]. FIRE assumes combinational circuits with AND, NAND, OR, NOR and NOT gates as primitives. FIRE identifies redundant faults for which a conflict on a single line in the circuit is necessary for their detection. This approach is radically different from any ATG-based approach for RID which identifies a fault as redundant, if all possible ways to detect that fault end up with conflicts. In contrast, our approach starts with a possible conflict (which is always the root cause of combinational undetectability) and finds faults for which that conflict is necessary for detection. Its key advantages are that RID can be accomplished without any search and that several redundant faults may be identified by analyzing the same conflict. FIRE has polynomial-time complexity. Hence it is not guaranteed to identify all redundancies in a circuit, since the RID problem is NP-complete [16].

FIRE can be used as a preprocessor to an ATG program, which can avoid targeting the faults identified as redundant and thus save the large computational effort associated with them. Identifying redundancies may provide useful feedback for designers, helping them in locating design errors or finding ways to simplify the circuit. RID is also very useful in synthesis, by providing the basis for redundancy removal.

The rest of the paper is organized as follows. Section 2 reviews our earlier work in RID. Section 3 describes our new algorithm for static RID. Section 4 presents our results and section 5 concludes the paper.

2. RID Using Illegal Combinations of Values

This section reviews our previous technique to identify redundancies for which an illegal combination of values (on a set of lines in the circuit) is necessary for detection. Assume that $\{X = a, Y = b, Z = c\}$ is an illegal (impossible) combination of values in the fault-free circuit. Faults for which this combination of values is necessary for detection are undetectable and hence redundant. We decompose the problem of finding such faults, by finding the faults for which each condition is *individually* necessary for detection. Let S_X, S_Y and S_Z be the sets of faults that require $X = a$, $Y = b$, and $Z = c$, respectively, for detection. Then the faults that require $X = a$ and $Y = b$ and $Z = c$ for detection are in the set $S_{XYZ} = S_X \cap S_Y \cap S_Z$.

The faults in S_X become undetectable when X cannot assume value a (or when X is uncontrollable for value a). To find these faults, we use an improved version of the uncontrollability and unobservability analysis introduced in [17]. In this analysis, $\overline{0}(\overline{1})$ denotes the status of a line that is uncontrollable for value 0(1). Note that this uncontrollability is conditional and occurs only when X cannot have value a. For example, to compute the set S_X, the algorithm propagates $X = \overline{a}$ and determines the faults that become undetectable if X cannot have value a; that is, $X = a$ is a necessary condition for the detection of these faults.

Figure 1 illustrates the propagation rules of the uncontrollability indicators. Uncontrollability can propagate forward and backward, as shown by the arrows. For example, the output of a NAND gate is $\overline{0}$ if at least one input is $\overline{1}$, and is $\overline{1}$ iff all its inputs are $\overline{0}$. Similar rules apply to other gate types. Propagation of uncontrollability may result in some lines becoming unobservable. If a gate input cannot be set to the noncontrolling value of the gate, all the other inputs become unobservable. For example in Figure 1, $a = \overline{1}$ implies that b is unobservable (unobservable lines are marked with a "*"). This is because $a = 1$ is necessary to observe a value on b. The unobservability status propagates from a gate output backwards to all its inputs; in Figure 1, b being unobservable makes c and d unobservable as well. The meaning of these implications is that $a = 1$ is necessary to observe fault effects from b, c, and d. When all fanout branches (FOBs) of a stem are marked as unobservable, we use the following lemma to decide whether the stem can also be marked as unobservable.

Lemma 1: A stem s with all its FOBs marked as unobservable may also be marked as unobservable if for each FOB f of s, there exists at least one set of lines $\{l_f\}$ such that the following conditions are satisfied:

1. f is unobservable because of uncontrollability indicators on every line in $\{l_f\}$; and
2. every line in $\{l_f\}$ is unreachable from s.

Proof: We use Figure 2 for the proof. Assume that a value $\overline{v}$ from some line l implies $a = \overline{1}$ and $b = \overline{0}$, and that a and b are unreachable from s. $a = \overline{1}$ makes $f1$ unobservable and $b = \overline{0}$ makes $f2$ unobservable. To observe stem s, at least one of $\{ a = 1, b = 0 \}$ must be achieved. (Both a and b are unreachable from s, and a fault-effect from s cannot propagate on either of them.) This is possible iff $l = v$. Hence $l = v$ is necessary for the detection of s. □

In general, a stem may be observed even if none of its FOBs is. The following example illustrates how such a stem will not satisfy the conditions of Lemma 1.

Example 1: Consider the AND gate in Figure 3. Assume that $a = \overline{1}$ which causes $a1 = a2 = b = \overline{1}$. $a1 = 1$ is necessary to observe $a2$ and $a2 = 1$ is necessary to observe $a1$. However, neither of these is necessary to observe a. Lemma 1 is not satisfied because both $a1$ and $a2$ are reachable from a. □

While determining uncontrollable and unobservable lines, we can also identify the faults that become undetectable. These are the faults that cannot be activated (s-a-0 on $\overline{1}$ lines and s-a-1 on $\overline{0}$ lines) and the faults that cannot be propagated (both faults on unobservable lines). The conditions for which we identify undetectable faults that cannot be activated are obvious. However, identifying undetectable faults on unobservable lines requires the following justification. The requirements for observing fault-effects may be either in the faulty or the fault-free circuit. Consider for example, the AND gate in Figure 4. To observe a $0/1$ fault-effect (fault-free value/faulty value) on a at a primary output (PO), a 1 must be justified on b in the faulty circuit. However, to observe a $1/0$ fault-effect on a, a 1 must be justified on b in the fault-free circuit. If b had a $\overline{1}$, our rules for unobservability propagation claim that a $1/1$ on b is necessary to observe both fault-effects on a. This is true if any vector which justifies a 1 on b (to observe a fault-effect on a), in either the fault-free or the faulty circuit, will drive a $1/1$ on b. This condition will be invalidated only if multiple fault-effects arrive at the the inputs of this gate. Lemma 1 in fact pessimistically ensures that this situation does not occur.

We refer to the process of propagating uncontrollability and unobservability to determine undetectable faults as *implication*. In summary, the procedure to find the redundant faults caused by an illegal combination $\{l_1 = v_{1,}\ l_2 = v_{2,}\ ...\}$ is :

1. For every $l_i = v_i$, imply $l_i = \overline{v_i}$ to determine all lines becoming uncontrollable or unobservable. Let S_i be the set of the corresponding faults. (Note that the implications are done separately for each i.)
2. The redundant faults caused by the given illegal combination are in the set $S = \bigcap_i S_i$.

This is a general procedure to find faults for which a given combination of values is a necessary condition for detection. If the combination is illegal, then the faults so found are redundant.

Example 2: Consider the two subcircuits shown in Figure 5. Assume that the function of the subcircuit CKT1 precludes the combination of values $a = 1$, $b = 0$ and $c = 1$. The corresponding implications are summarized in Table 1. (A *s-a-v* fault on a line l is represented as l_v, and we use a collapsed set of faults based on functional equivalence.) The faults in the set S_a, for example, are those which require $a = 1$ as a necessary condition for their detection.

Note that when $c = \overline{1}$, both $a1$ and $a2$ are unobservable and hence a is also unobservable, because c is unreachable from a and $c = 1$ is necessary to observe a. (The information that e is a dominator for a is implicitly computed.) Thus, $S_a \cap S_b \cap S_c = \{ d_{1,}\ a2_0 \}$, which identifies d *s-a*-1 and $a2$ *s-a*-0 as undetectable. It can be verified that $a = 1$, $b = 0$ and $c = 1$ is a necessary condition for the detection of these faults. □

3. A Novel Extension for Static RID

The previous section outlined a procedure to identify redundant faults by processing an illegal combination of values on a set of lines, $\{l_1 = v_1, l_2 = v_2, ..., l_n = v_n\}$. The novel extension we propose here is to process *conflicting* values on the same line, l, considering $l = 0$ and $l = 1$ as a special case of an illegal combination.

In principle, the analysis could be performed for every signal in the circuit. Conflicts can occur only in the presence of reconvergent fanout structures, and the signal leads where conflicts occur are fanout stems. The following example illustrates how redundant faults can be identified by processing conflicts on stems.

Example 3: Consider the circuit in Figure 6. We process a conflict on stem c by propagating $c = \overline{0}$ and $c = \overline{1}$ and determining the faults that require $c = 1$ and $c = 0$, respectively, as necessary conditions for detection. The resulting implications are summarized in Table 2. The redundant faults identified are $\{ i_0, f_0, g_0, b_0, d_0, a_1 \}$. □

In general, a fault could be redundant because of a conflict on at least one stem from a group of stems. The detection of such a fault does not require a conflict on any single stem as a necessary condition, and the fault will not be identified by the above analysis. Our observation is that such conflicts often map as conflicts on reconvergent inputs of reconvergent gates.

Example 4: Consider the circuit in Figure 7 [3] and the fault f *s*-*a*-0. Clearly, the activation of this fault requires $c = d = a1 = b1 = a = b = 1$. Also, to propagate the fault-effect from f, e has to be 0 which requires *either a or b* to have value 0. Thus f *s*-*a*-0 requires a conflict on either a or b and processing either a or b individually for conflicts will not find this fault as redundant. However, as summarized in Table 3, processing a conflict on line f (a reconvergent input of a reconvergent gate common to both stems) identifies f *s*-*a*-0 as redundant. □

The preceding examples show that the entire analysis to determine redundant faults is fault-independent and analyzing a single conflict can identify several redundant faults. Our Fault-Independent REdundancy identification algorithm (FIRE) is outlined in Figure 8. Although, in general, processing conflicts on reconvergent inputs of reconvergent gates in addition to fanout stems can find more redundant faults, our algorithm processes conflicts only on fanout stems. This follows from our empirical observation that processing conflicts only on stems finds most of the redundancies in the circuit and saves a lot of computation time. All the results reported in Section 4 were obtained by processing only stem conflicts.

A Note on the Time Complexity of FIRE

Let n be the number of lines in the circuit. The worst-case for FIRE would occur when all n lines in the circuit are analyzed for conflicts, and the implications for uncontrollability and unobservability propagate through the entire circuit. The conditions for marking stem unobservability (Lemma 1) are checked only when all FOBs of a stem are marked as unobservable. This situation is quite seldom. Hence, ignoring the computations required to check for conditions of Lemma 1, the worst-case complexity of FIRE is $O(n^2)$. However, in practice only stems are analyzed for conflicts and the implications propagate only through a small subcircuit surrounding the analyzed stem. Moreover, in practice, these implications for a stem remains the same as the circuit size increases. Therefore the worst-case situation never occurs in practice. Thus, *in practice, FIRE identifies redundancies in linear time* and this is very important, considering that the RID problem is NP-complete [16]. However, FIRE is not guaranteed to find all the redundancies in the circuit. The experimental results reported in the next section attest the efficiency of the algorithm.

4. Results

FIRE was implemented in C in a prototype program. To compare FIRE with a fault-oriented approach for RID, we used the LTG test generation/ fault simulation package [6]. Faults proved redundant by FIRE were passed as targets to LTG with a backtrack limit of 100. LTG also has the advantage of dynamic RID techniques [18]. It is true that in a free run, LTG would, in general, identify more redundancies than FIRE. However, the purpose of these experiments is to testify how much time LTG would save by not targeting the faults proved redundant by FIRE. FIRE was run on the ISCAS85

Table 4 shows the comparison for the ISCAS85 combinational benchmark circuits [19] and for the full-scan versions of the ISCAS89 sequential benchmark circuits [20]. These results are for a collapsed set of faults. We show only the the circuits where redundancies were identified by FIRE. *#Red.* represents the number of identified redundancies. The *CPU times* are in seconds for a SUN sparc2. *#Abo.* represents the number of targets aborted by LTG. Speed-up ratio represents (LTG time)/(FIRE time). For almost all the circuits, FIRE performed much better than LTG. For S35932, FIRE found 3984 redundant faults in 236 secs. LTG found all of them to be redundant in 2502 secs. Thus, in this case, FIRE is about 11 times faster than LTG. Furthermore, for S9234, LTG could not solve all the target faults identified as redundant by FIRE.

Table 5 shows the results obtained for some real circuits. *Gates*, *FFs*, *PIs*, and *POs* refer to the number of gates, flip-flops, primary inputs and primary outputs respectively. FIRE performed much better than LTG for all the circuits. A speed-up ratio of up to 37 was achieved for these circuits.

Table 6 shows the percentage of redundancies identified by FIRE in the benchmark circuits. Since TRAN [12] is a state-of-the-art combinational test generator that reports all the single-fault redundant faults in the benchmark circuits, we used it for our comparisons. The TRAN CPU times are also for a SUN sparc2 and include times for random test generation, transitive closure algorithm and fault simulation. Note that TRAN can find more redundancies at the expense of more CPU time. In some cases (like S35932), FIRE did a complete job of finding all the redundancies. However, in some other cases (like S1238) FIRE did not perform as well. We believe that the reasons for such a behavior are (i) Our implication procedure is not complete; (ii) there could be redundant faults that do not require a single conflict as a necessary condition for detection, and (iii) Lemma 1 to mark unobservability on a stem is pessimistic and provides only a sufficient condition. We also show the percentage of FIRE CPU time of the total CPU time used by TRAN to identify these redundancies. For example, in S38417, FIRE identified 60% of the combinational redundancies in 0.9% of the total time used by TRAN to identify all the redundancies. FIRE found 5876 (about 80%) of the total 7313 redundant faults in these circuits. In summary, FIRE identified a range of 8.3% to 100% of the redundant faults identified by TRAN and FIRE CPU times ranged from 0.9% to 66.7% of TRAN CPU times. Note that in every case when FIRE time was more than 15% of the TRAN time, the FIRE time was under 12 seconds. The results in Table 6 indicate that FIRE can be an useful preprocessor even to state-of-the-art test generators like TRAN.

5. Conclusions

This paper has presented FIRE, a new fault-independent combinational redundancy identification algorithm. The algorithm identifies faults for which conflicts on fanout stems are necessary for detection. The process is based on logic implications, as opposed to the exhaustive search done by ATG algorithms. By processing a single conflict, FIRE may identify several redundant faults that would require separate targeting by an ATG-based approach.

Our results indicate a speed-up of up to about 37 times when compared to a fault-oriented conventional ATG algorithm that targeted only the faults identified by FIRE. Although FIRE is not guaranteed to identify all redundancies in a circuit, it did identify 80% of the redundant faults in the benchmark circuits. Thus FIRE could be used as a fast preprocessor to ATG and to obtain significant savings in computations. Since FIRE is based on logic implications, any improved procedure to compute global value assignments [11,12] is expected to improve the performance of FIRE.

Logic synthesis procedures for redundancy removal are very time-consuming if they rely on ATG for RID, because many faults have to be repeatedly targeted. FIRE can provide significant advantages over ATG in this application, since repeated use of FIRE for RID is much more efficient than repeated use of ATG.

While combinational ATG is a problem where current state-of-the-art algorithms can efficiently deal with VLSI circuits, the same cannot be said about the much more complex sequential ATG problem, which may require extremely long run-times. Recent extensions of the techniques presented in this paper to sequential circuits produced efficient algorithms to identify sequentially untestable and redundant faults [15,21,22,23] up to several orders of magnitude faster than a conventional sequential test generator.

Acknowledgements

We thank V. D. Agrawal, W. T. Cheng, D. E. Long, and D. T. Miller for useful discussions and comments. We also thank the anonymous reviewers for their useful suggestions. The support provided by S. Davidson, A. Dunlop, J. Dussault, H. Nham, and S. Wu made this work possible and is gratefully acknowledged.

REFERENCES

[1] A. D. Friedman, ‘‘Fault Detection in Redundant Circuits,’’ *IEEE Trans. on Electronic Computers*, vol.EC-16, pp. 99-100, February 1967.

[2] S. Davidson, ‘‘Is I_{DDQ} Yield Loss Inevitable ?,’’ *Proc. Intn’l. Test Conf.*, pp. 572-579, October 1994.

[3] M. Harihara and P. R. Menon, ‘‘Identification of Undetectable Faults in Combinational Circuits,’’ *Proc. Intn’l. Conf. on Computer Design*, pp. 290-293, October 1989.

[4] P. R. Menon and H. Ahuja, ‘‘Redundancy Removal and Simplification of Combinational Circuits,’’ *Digest of Papers, IEEE VLSI Test Symp.*, pp. 268-273, April 1992.

[5] H. Fujiwara and T. Shimono, ‘‘On the acceleration of test generation algorithms,’’ *IEEE Trans. on Computers*, vol. C-32, pp. 1137-1114, December 1983.

[6] M. Abramovici, J. J. Kulikowski, P. R. Menon, and D. T. Miller, ‘‘SMART and FAST: Test Generation for VLSI scan-design circuits,’’ *IEEE Design and Test of Computers*, pp. 53-54, August 1986.

[7] T. Krikland and M. R. Mercer, ‘‘A topological search algorithm for ATPG,’’ *Proc. 24th. ACM/IEEE Design Automation Conf.*, pp. 502-508, June 1987.

[8] M. H. Schulz and E. Auth, ‘‘Improved deterministic test pattern generation with applications to redundancy identification,’’ *IEEE Trans. on Computer-Aided Design*, vol. 8, pp. 811-816, July 1989.

[9] J. Rajski and H. Cox, ‘‘A method to calculate necessary assignments in algorithmic test pattern generation,’’ *Proc. Intn’l. Test Conf.*, pp. 25-34, September 1990.

[10] J. Giraldi and M. L. Bushnell, ‘‘Search State Equivalence for Redundancy Identification and Test Generation,’’ *Proc. Intn’l. Test Conf.*, pp. 184-193, October 1991.

[11] W. Kunz and D. K. Pradhan, ‘‘Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits,’’ *Proc. Intn’l. Test Conf.*, pp. 816-825, September 1992.

[12] S. T. Chakradhar, V. D. Agrawal, and S. G. Rothweiler, ‘‘A Transitive Closure Algorithm for Test Generation,’’ *IEEE Trans. on Computer-Aided Design*, vol. 12, no. 7, pp. 1015-1028, July 1993.

[13] M. Abramovici and M. A. Iyer, ‘‘One-Pass Redundancy Identification and Removal,’’ *Proc. Intn’l. Test Conf.*, pp. 807-815, September 1992.

[14] M. A. Iyer and M. Abramovici, ‘‘Low-Cost Redundancy Identification for Combinational Circuits,’’ *Proc. 7th. Intn’l. Conf. on VLSI Design*, pp. 315-318, January 1994.

[15] M. A. Iyer, ‘‘On Redundancy and Untestability in Sequential Circuits,’’ *Ph.D. Thesis, ECE Department, Illinois Institute of Technology*, Chicago, IL - 60616, July 1995.

[16] O. H. Ibarra and S. K. Sahni, ‘‘Polynomially Complete Fault Detection Problems,’’ *IEEE Trans. on Computers*, vol. C-24, pp. 242-249, March 1975.

[17] M. Abramovici, J. J. Kulikowski, and R. K. Roy, ‘‘The Best Flip-Flops to Scan,’’ *Proc. Intn’l. Test Conference*, pp. 166-173, October 1991.

[18] M. Abramovici, D. T. Miller, and R. K. Roy, ‘‘Dynamic Redundancy Identification in Automatic Test Generation,’’ *IEEE Trans. on Computer-Aided Design*, pp. 404-407, March 1992.

[19] F. Brglez and H. Fujiwara, ‘‘Neutral netlist of ten combinational benchmark circuits and a target translator in FORTRAN,’’ *Proc. IEEE Intn’l. Symp. on Circuits and Systems*, June 1985.

[20] F. Brglez, D. Bryan, and K. Kozminski, ‘‘Combinational profiles of sequential benchmark circuits,’’ *Proc. 1989 Intn’l. Symp. on Circuits and Systems*, pp. 1929-1934, May 1989.

[21] M. A. Iyer and M. Abramovici, ‘‘Sequentially Untestable Faults Identified Without Search (Simple Implications Beat Exhaustive Search !),’’ *Proc. Intn’l. Test Conf.*, pp. 259-266, October 1994.

[22] D. E. Long, M. A. Iyer, and M. Abramovici, ‘‘Identifying Sequentially Untestable Faults Using Illegal States,’’ *13th. IEEE VLSI Test Symp.*, pp. 4-11, May 1995.

[23] M. A. Iyer, D. E. Long, and M. Abramovici, ‘‘Identifying Sequential Redundancies Without Search,’’ Technical Report No. IIT-ECE-TR-017-0495, *ECE Department, Illinois Institute of Technology*, Chicago, IL - 60616, April 1995.

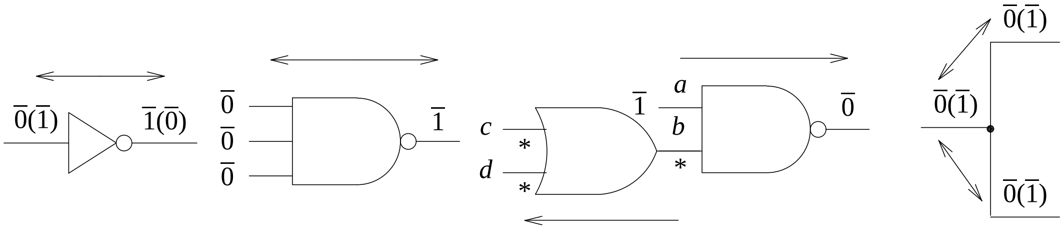

Figure 1. Propagation of uncontrollability and unobservability

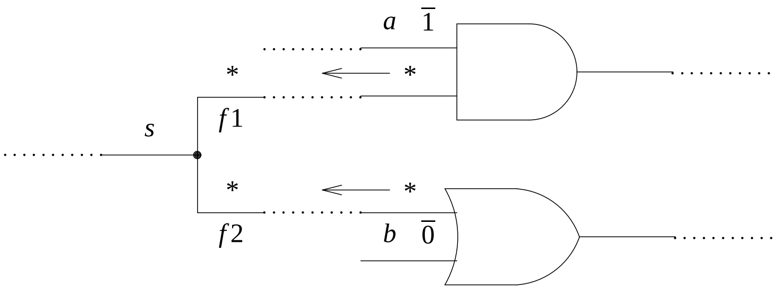

Figure 2. Conditions for marking stem unobservability

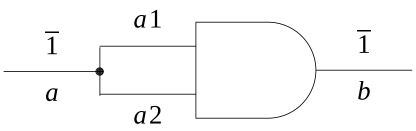

Figure 3. An example circuit

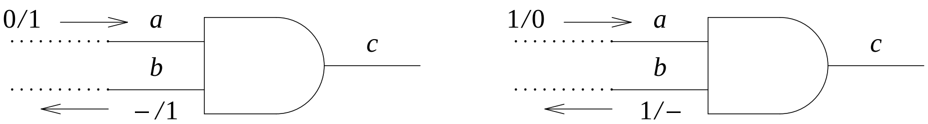

Figure 4. Necessary conditions for fault-effect observation

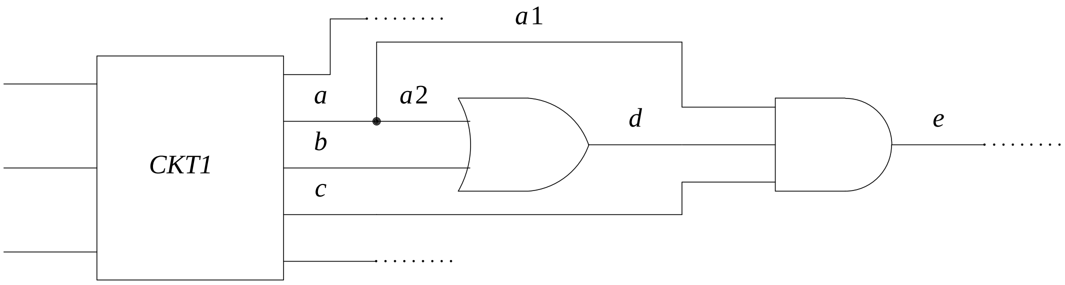

Figure 5. An example circuit

Table 1. Implications for Example 1

Process	Uncontrollability	Unobservability	Implied Faults
$a = \overline{1}$	$a = a1 = a2 = e = \overline{1}$	$d, a2, b, c$	$S_a = \{a_0, a2_0, d_1, b_0, c_1, e_0\}$
$b = \overline{0}$	$b = d = \overline{0}$	$a2$	$S_b = \{d_1, a2_0\}$
$c = \overline{1}$	$c = e = \overline{1}$	$d, a2, b, a1, a$	$S_c = \{d_1, b_0, a2_0, a1_1, a_0, a_1, e_0\}$
Redundant faults: $S_a \cap S_b \cap S_c = \{d_1, a2_0\}$			

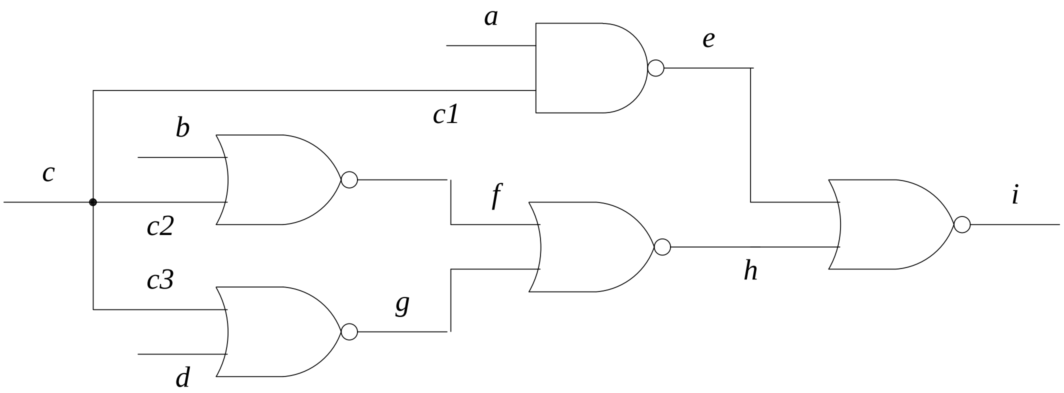

Figure 6. An example circuit

Table 2. Implications for Example 2

Process	Uncontrollability	Unobservability	Implied Faults
$c = \overline{0}$	$c = c1 = c2 = c3 = \overline{0}$ $f = g = \overline{1}$ $h = \overline{0}, i = \overline{1}$	$b, d, e, a, c1$	$S_0 = \{c1_1, f_0, g_0, i_0, b_0, d_0, e_0, a_1\}$
$c = \overline{1}$	$c1 = c2 = c3 = \overline{1}$ $e = \overline{0}, i = \overline{1}$	$a, h, f, g, b, c2, c3, d$	$S_1 = \{c2_0, c3_0, i_0, h_0, f_0, g_0, b_0, d_0, a_1\}$
Redundant faults: $S_0 \cap S_1 = \{i_0, f_0, g_0, b_0, d_0, a_1\}$			

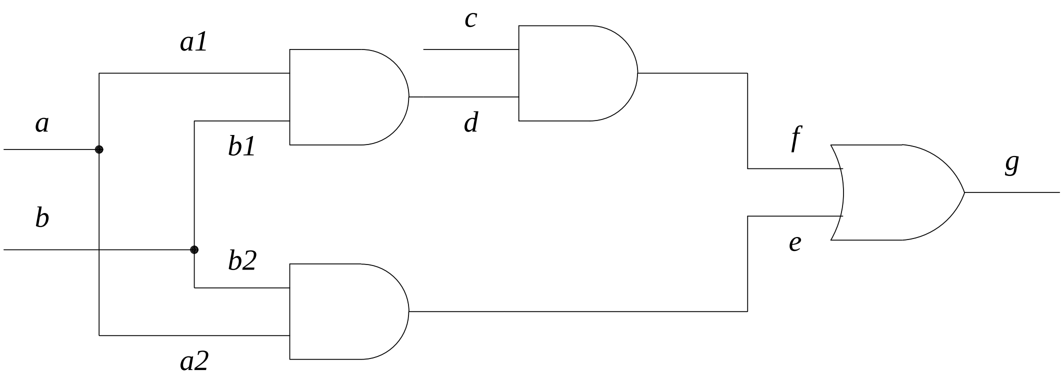

Figure 7. An example circuit [3]

Table 3. Implications for Example 3

Process	Uncontrollability	Unobservability	Implied Faults
$f = \overline{0}$	$c = d = a1 = b1 = a = b = a2 = b2 = e = g = \overline{0}$	$f, c, d, a1, b1, e, b2, a2$	$S_0 = \{ c_1, d_1, a1_1, b1_1, a_1, a2_1, b_1, b2_1, g_1, f_0, e_0 \}$
$f = \overline{1}$	$f = \overline{1}$	-	$S_1 = \{ f_0 \}$
Redundant faults: $S_0 \cap S_1 = \{ f_0 \}$			

FIRE() {

For every stem s **in the circuit {**

Imply $s = \overline{0}$ to determine all lines becoming uncontrollable or unobservable. Let S_0 be the set of the corresponding faults

Imply $s = \overline{1}$ to determine all lines becoming uncontrollable or unobservable. Let S_1 be the set of the corresponding faults

The redundant faults are in the set $S = S_0 \cap S_1$

}

}

Figure 8. Outline of the FIRE algorithm

Table 4. Results for benchmark circuits

Circuit Name	FIRE # Red.	FIRE CPU secs.	LTG (Bt. Limit=100) # Red.	LTG # Abo.	LTG CPU secs.	Speed-up Ratio
C1908	6	1.8	6	0	5.0	2.8
C2670	29	1.5	29	0	3.3	2.2
C3540	93	11.9	93	0	9.7	0.8
C5315	20	2.8	20	0	10.0	3.6
C6288	33	1.3	33	0	25.9	19.9
C7552	30	4.7	30	0	17.3	3.7
S349	2	0.4	2	0	0.7	1.8
S444	11	0.2	8	0	0.7	3.5
S713	32	0.1	32	0	1.0	10.0
S1238	6	1.9	6	0	1.6	0.8
S1423	5	0.3	5	0	2.6	8.7
S1494	1	1.1	1	0	1.4	1.3
S5378	34	3.7	34	0	15.8	4.3
S9234	165	20.6	158	7	55.6	2.7
S13207	55	23.2	55	0	30.4	1.3
S15850	234	20.5	234	0	157.8	7.7
S35932	3984	235.6	3984	0	2502.4	10.6
S38417	99	45.4	99	0	275.2	6.1
S38584	1037	156.0	1036	0	1264.7	8.1

Table 5. Results for real circuits

Circuit Name	Circuit Parameters				FIRE		LTG			Speed-up Ratio
	# Gates	# FFs	# PIs	# POs	# Red.	CPU secs.	# Red.	# Abo.	CPU secs.	
Test7	829	31	40	41	94	4.6	94	0	7.3	1.6
AMD2910	915	87	20	16	102	4.5	102	0	12.3	2.7
Chip17	1182	22	29	39	113	2.3	113	0	22.1	9.6
Chip58	1242	83	123	128	6	2.2	6	0	2.6	1.2
Test132	2272	24	162	88	336	4.7	336	0	158.7	33.8
Test117	3204	82	110	141	295	5.8	295	0	217.0	37.4
Test149	4249	88	176	187	674	38.3	666	8	454.9	11.9
Test198	4257	87	147	141	396	35.8	396	0	268.6	7.5
Test105	5598	162	230	343	737	29.8	737	0	615.3	20.7
Test154	5781	108	245	240	517	41.1	517	0	410.4	10.0
Test122	7179	1	270	303	952	145.9	938	14	1431.0	9.8
Test66	7614	162	193	326	305	63.2	305	0	594.2	9.4
Test119	12635	183	398	405	988	215.9	988	0	2432.2	11.3
C42825	18504	0	1179	1180	2630	1013.0	2630	0	2050.8	2.0

Table 6. Percentage of redundancies identified by FIRE

Circuit Name	FIRE		TRAN		% FIRE of Total	% FIRE time of TRAN time
	# Red.	CPU sec.	# Red.	CPU sec.		
C1908	6	1.8	7	13.0	85.7	13.9
C2670	29	1.5	115	95.2	25.2	1.6
C3540	93	11.9	131	24.9	71.0	47.8
C5315	20	2.8	59	32.3	33.9	8.7
C6288	33	1.3	34	38.0	97.1	3.4
C7552	30	4.7	131	308.0	22.9	1.5
S349	2	0.2	2	0.3	100.0	66.7
S444	11	0.2	14	0.4	78.6	50.0
S713	32	0.1	38	3.1	84.2	3.2
S1238	6	1.9	69	17.4	8.7	10.9
S1423	5	0.3	14	8.5	35.7	3.5
S1494	1	1.1	12	3.7	8.3	29.7
S5378	34	3.7	40	73.0	85.0	5.1
S9234	165	20.6	452	803.7	36.5	2.6
S13207	55	23.2	151	806.5	36.4	2.9
S15850	234	20.5	389	1177.5	60.2	1.7
S35932	3984	235.6	3984	1617.0	100.0	14.6
S38417	99	45.4	165	5078.2	60.0	0.9
S38584	1037	156.0	1506	2483.8	68.9	6.3