Perturb and Simplify: Multi-level Boolean Network Optimizer

Shih-Chieh Chang, Malgorzata Mark-Sadowska and Kwang-Ting Cheng

Abstract

In this paper, we present logic optimization techniques for multi-level combinational networks. Our techniques apply a sequence of perturbations which result in simplification of the circuit. The perturbation and simplification is achieved through wires/gates addition and removal which are guided by the Automatic Test Pattern Generation (ATPG) based reasoning. The main operations of our approaches are incremental transformations of the circuit (such as adding wires/gates and changing gate's functionality) to remove some particular wire. At each iteration, a summary information of such wires/gates addition and removal is precomputed first. Then, a transformation is chosen to remove several wires at once. We have performed experiments on MCNC benchmarks and compared the results to those of misII[5] and RAMBO[8]. Experimental results are very encouraging.

1 Introduction

The technology independent multi-level logic minimization is crucial for logic synthesis. In this paper, we discuss the problem of multi-level logic optimization for a combinational Boolean network. Previous multi-level optimization approaches can be categorized into two classes. The first class locally collapses and optimizes a circuit using techniques like factorization, decomposition, kernel extraction, cube extraction, etc. (e.g.: misII [5]). The second class introduces a perturbation, usually in a form of adding wires, to a network and results in potential removal of some redundant gates or wires (e.g.: global flow [3], transduction method [18], and RAMBO [8]). Our proposed approach falls into the second class.

1. Shih-Chieh Chang is with Institute of Computer Science and Information Engineering National Chung Cheng University, Chiayi, Taiwan. Malgorzata Marek-Sadowska and Kwang-Ting Cheng are with the Electircal and Computer Engineering Department, University of California at Santa Barbara, Santa Barbara, CA 93106, USA.

2. This work was supported in part by the National Science Foundation under Grant MIP 9419119 and in part by AT&T Bell Laboratories and Xilinx through the California MICRO program.

We now take a closer look at the approaches falling into the second class. In [3], techniques of data flow analysis are used. A summary information about the circuit is gathered in the form of implicants which result in identification of wires to be connected or disconnected without affecting the external behavior of the network. At each iteration of the algorithm, one signal net is restructured. Another group of techniques are descendants of the transduction method. The basic scheme is as follows. To optimize a circuit, for each gate, a set of permissible functions is calculated. Then a gate is replaced by another gate within the range of permissible functions. Gate and wire transformations are executed next. In the transduction method both calculation of permissible functions and gate transformations are computationally very expensive. Several heuristics have been proposed to speed up the transduction method. One idea was to use an easy computable subset of permissible functions, what resulted in a weaker optimization capability. Another approach was to determine, for a node in a network, a superset of permissible functions and later check the validity of the perturbation and/or transformation [14]. Selection of a gate to be perturbed as well as the resulting changes are determined based on the structure of the network and on the permissible functions of the nodes.

An efficient approach to perturbing and simplifying a network was proposed in [8], [11] and implemented in RAMBO. The idea was that the perturbation-simplification process of network optimization can be viewed as wire addition and removal which can be efficiently computed using automatic test pattern generation techniques [1]. If adding a wire w_a and removing another wire w_r, the circuit's functionality remains unchanged, we say w_a is a *single-alternative* wire for w_r. An essential contribution of [8], [11] was how to identify the single-alternative wires. Based on the concept of single-alternative wire, this optimization philosophy is best explained on the following example. In Fig. 1a, the dotted wire g_5->g_9 is a single-alternative wire for the bold wires g_1->g_4 and g_6->g_7. We may remove both g_1->g_4 and g_6->g_7 by adding g_5->g_9. The result is shown in Fig. 1b. Therefore, by iteratively removing redundant wires caused by adding their common alternative wire, logic optimization can be achieved. In [6] we applied the ideas of test pattern generation guided wire additions and removals to alleviate FPGA routing.

In this paper we carry further the idea of perturbing-simplifying a circuit applying ATPG techniques. In particular, we discuss how the single-alternative wire procedure [8] [11] can be improved both in its quality and efficiency. Then we propose several circuit transformations to remove a wire for optimization. These circuit transformations include adding multiple wires/gates. We also show how to identify gates which are good candidates for local functionality change. In addition, we discuss the problem of adding and removing two wires, none of which alone is redundant, but when jointly added/removed they do not change the functionality of a network. We have observed that the ATPG based optimization may get stuck at local minimum. A guided perturbation which can lift such ATPG based technique out of local minimum is also proposed. It is based on the analysis of internal don't cares migration after wire/gate addition and removal.

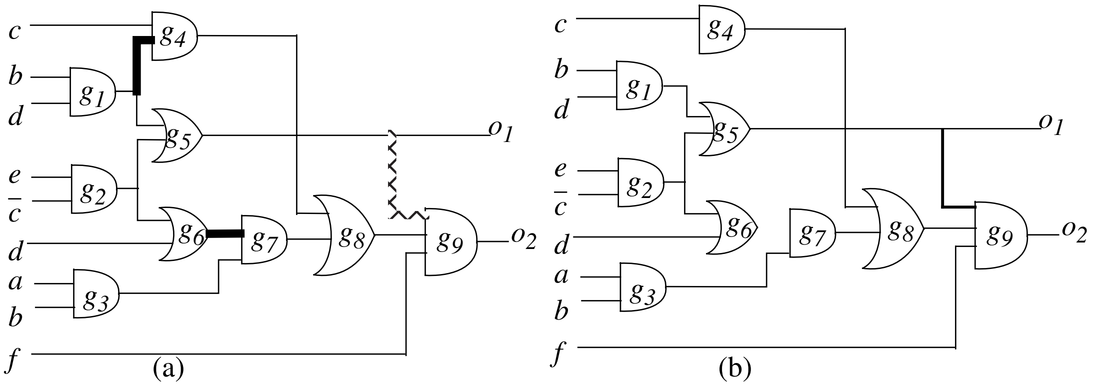


Fig. 1: Example of wire addition and removal

All the transformations we mentioned in this paper, can be also considered as incremental transformations for the wiring control. An incremental removal of particular wires by adding some wires/gates is very useful to control circuit structure. In [6], we mentioned several applications of such incremental transformations.

The remainder of this paper is organized as follows. Section 2 overviews our approach. Section 3 provides necessary definitions. Section 4 reviews the single alternative wire procedure. Section 5 shows an improved version of the procedure to identify the single alternative wires. Sections 6 to 10 show a series of new transformations and the use of don't cares. Section 11 summarizes our overall algorithm. Finally we show experimental results and give conclusions.

2 Overview of our approach

For ease of explanation, we particularly contrast our philosophy with that of RAMBO [8], [11]. Our new techniques not only result in smaller networks than those produced by RAMBO, but also speed up the optimization process considerably. In this section, we give an intuitive interpretation of the reasons why we are able to improve in these two aspects.

The speeding up over RAMBO was possible by observing that while one wire is tested for redundancy, information about irredundancy of some other wires is obtained as a side effect. For example, after performing a redundancy check for the stuck-at-0 fault at wire g_2->g_6 in Fig. 1a, we conclude that g_2->g_6 and also the new connections, not present in the circuit, g_1->g_7, g_1->g_9, c->g_7, c->g_9 are irredundant. This information of irredundancy of new connections eliminates many trials and is important factor to speed up over RAMBO.

Throughout this paper, all the perturbations applied to the optimized network, like wire or gate addition and removal, the change of gate's functionality, etc., will be such that the external behavior of the network remains intact. When we say that a certain gate can be added/removed to/from the network, we mean that this gate can be added/removed without affecting the primary outputs of the network.

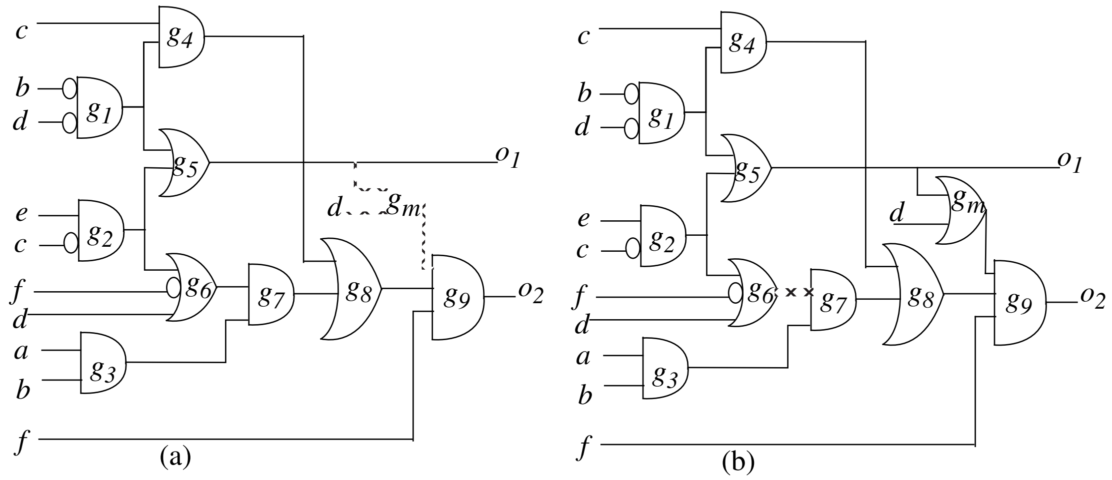


Fig. 2: Adding gates to remove wires(gates)

The new transforms of adding one or several redundant *gates* to a circuit are illustrated with an example in Fig. 2. In Fig. 2a, the OR gate g_m (dotted in the figure) with inputs from g_5 and d can be added (without affecting the behavior of the network). Then, the originally irredundant wire g_6->g_7 (dotted in the Fig. 2b) becomes redundant. This in turn allows us to remove the 3-input gate g_6.

Network perturbation can be also achieved by changing the functionality of selected gates. For example, gate g_5 in Fig. 3a may change from an AND to an XNOR as shown in Fig. 3b. As a result, the wire g_6->g_7 (dotted in Fig. 3b) and the 3-input gate g_6 can be removed from the circuit. Applying these new techniques, along with the earlier technique of adding one wire, allows us to explore a larger solution space and to obtain a substantial network reduction.

We also propose a general algorithm to detect pairs of simultaneously redundant wires. We say that two wires (w_a, w_b) are *simultaneously redundant* if each wire alone is irredun-

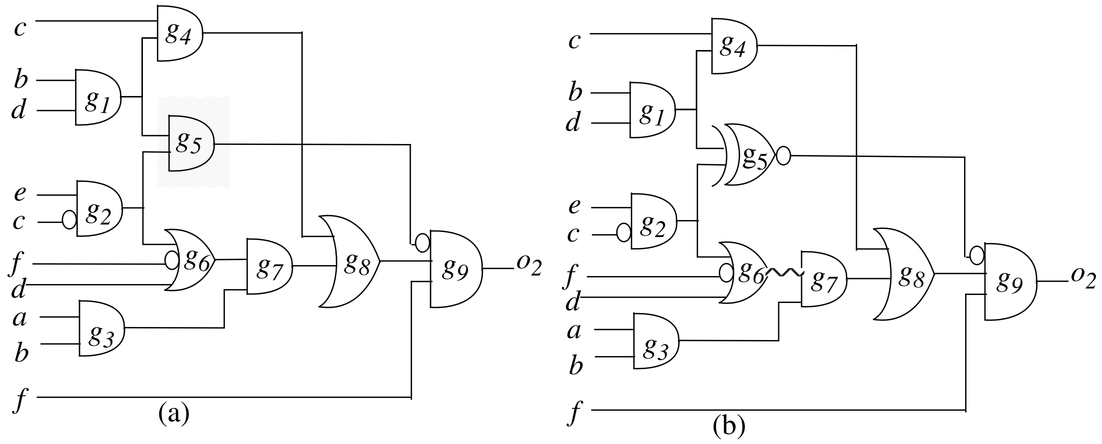


Fig. 3: Changing gates' functionality to remove wires(gates)

dant but simultaneously adding/removing w_a and adding/removing w_b will not change the network's functionality. For example, in Fig. 4a the wire *a->*g_1 (thick line) and a new connection *c->*g_2 are each irredundant; however simultaneous removal of *a->*g_1 and addition of *c->*g_2, shown in Fig. 4b, does not change the network's functionality.

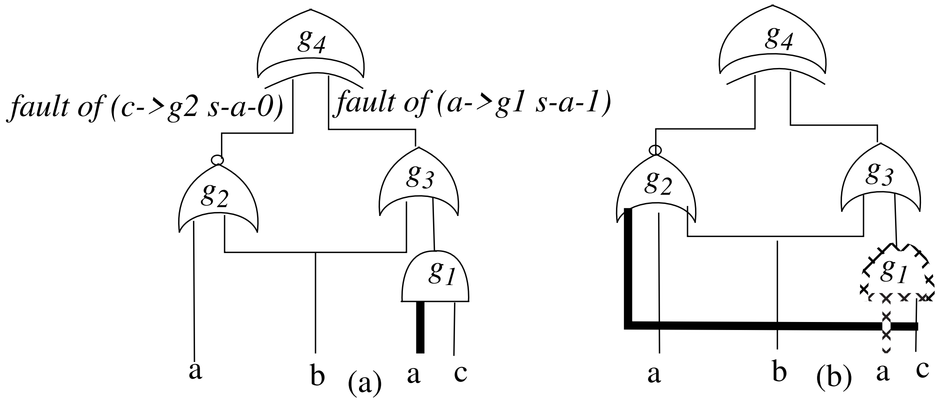


Fig. 4: Two simultaneously redundant wires

Note that none of the transformations shown in Fig. 2 through Fig. 4 can be obtained by adding a single wire and eliminating redundancies. Therefore none of these transformations would be possible in RAMBO.

3 Definitions

In the following, we review logic synthesis terminology and ATPG related concepts used in this paper.

A Boolean network is a Directed Acyclic Graph(DAG) where each node is associated with a Boolean function f_i, and a Boolean variable y_i. There is an edge directed from a node n_i to a node n_j if the function f_j depends on the variable y_i. The node n_i is a fanin of the node n_j and the node n_j is a fanout of the node n_i. If there is a directed path from a node n_i to a node n_j then we say that n_i is in transitive fanin of n_j and that n_j is in transitive fanout of n_i.

The *absolute dominators* (dominators) [15] of a wire *W* is the set of gates *G* such that all paths from wire *W* to any primary output have to pass through all gates in *G*. For example, the dominators of g_1->g_4 in Fig. 1a are g_4, g_8, *and* g_9.

The value of an input to a gate is said to be *controlling* if it determines the value of the gate's output regardless of the values of the other inputs; the controlling value is 1 for an OR or NOR gate, and 0 for an AND or an NAND gate. The inverse of the controlling value is called the *non-controlling* value or *sensitizing* value.

Consider the absolute dominators of a wire *W*. The *side inputs* of a dominator are its inputs not in the transitive fanout of the wire *W*. To generate a test for a stuck-at fault at wire *W*, all side inputs of the wire *W*'s dominators must be assigned to their noncontrolling values. For example, in Fig. 1a, to test wire (g_1->g_4) s-a-1, we must assign 1 to *c*, 0 to g_7, and 1 to *f*.

Assume the stuck-at-0(1) fault at wire w_r is under consideration. The *faulty* circuit is referred to as the circuit in which w_r is replaced by a constant 0(1). An input combination *v* is a *test vector* if the outputs of the good circuit and the faulty circuit are different when applying *v*. If no such test vector exists, then the *stuck-at fault* is redundant.

When we perform ATPG for a wire stuck-at fault, a test vector must satisfy two conditions: it must activate the fault and it must propagate the fault to at least one primary output. For example, in Fig. 1a to test g_1->g_4 s-a-1 (stuck-at 1), any test vector must satisfy g_1=0 for fault activation and must satisfy $\{c=1,\ g_7=0,\ f=1\}$ (side inputs to some dominators) for the fault propagation.

The *mandatory assignments* are the value assignments required for a test to exist and they must be satisfied by any test vector. The process of computing mandatory assignments and checking their consistency with the previously determined assignments is referred to as *implication* [1]. A complete implication process is known to be an NP complete problem. In other words, to compute the entire set of mandatory assignments may require exponential time in term of the number of nodes in a circuit. We adopt the same principle as in [8] [11] and complete the Set of Mandatory Assignments SMA(f) whose members can be found via the implication process described in [15] [19]. A more complex approach [16] which requires more CPU time may be applied to obtain more mandatory assignments.

During the implication process, if the mandatory assignments for a stuck-at fault cannot be consistently justified, the fault is untestable and therefore, the wire is redundant. A wire to be removed is referred to as the *target wire*. The corresponding stuck-at fault is called the *target fault*. Since our goal is to remove the target wire w_r, we term a logic transformation *candidate* if, after the transformation, SMA(w_r stuck-at fault) becomes inconsistent.

4 Single alternative wire

In this section, we review a method of adding single alternative wire to make a target wire redundant. This method finds and adds one redundant wire to make the target fault untestable and therefore redundant. The idea was originally proposed in [8] [11].

This procedure of finding single alternative wires consists of three steps. In the first step, we calculate the mandatory assignments for the target fault. Then, we identify a set of candidate connections to be added. Each candidate connection when added to the circuit will make the target fault untestable and therefore the target wire redundant. However, adding such a candidate connection may change the circuit's behavior. Therefore, in the final step, we check whether a candidate connection is redundant or not. If a candidate connection is redundant, we conclude that it is an alternative wire for the target wire. The following example illustrates the process.

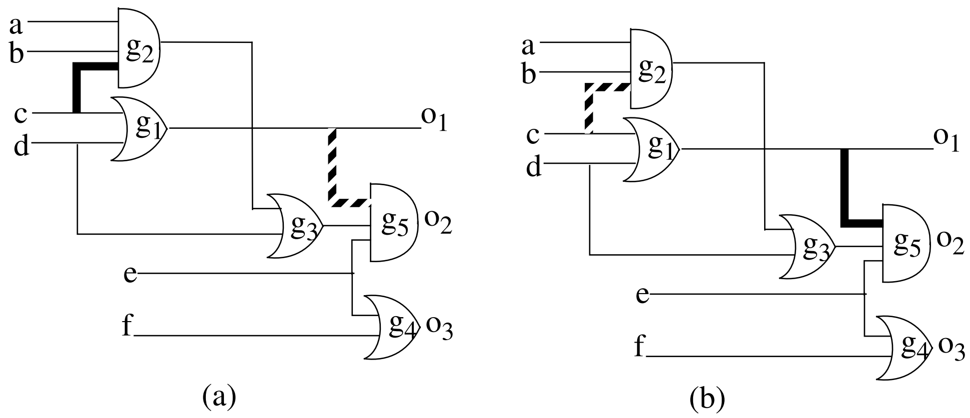
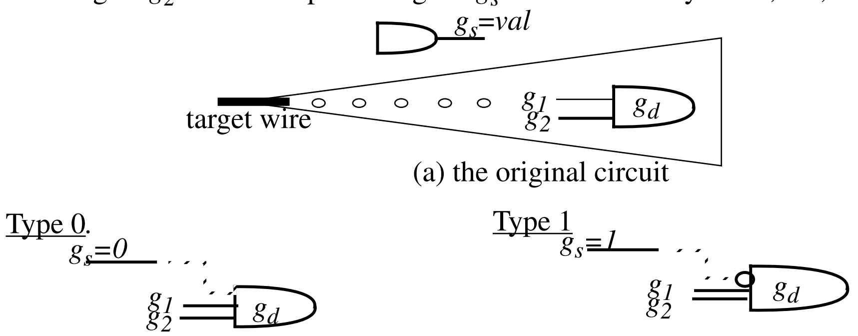


Fig. 5: Example of single alternative wire

Example 1: Consider the circuit in Fig. 5a. Let $c\text{->}g_2$ be the target wire to be removed. First, we compute the SMA(c->g_2 s-a-1). We have SMA = {a=1, b=1, c=0, d=0, g_1=0, e=1}. Note that g_1 is outside the transitive fanouts of the target wire and has a mandatory value 0. If we connect g_1 to g_5, a dominator, it will cause inconsistency of the mandatory assignment at g_1. It is because if the new wire $g_1\text{->}g_5$ is added, g_1 becomes a side input of the dominator g_5. It requires a non-controlling value 1 at g_1 to propagate the fault effect of the target fault to primary outputs. This additional requirement causes a conflict in the MA. The process, thus, suggests wire $g_1\text{->}g_5$ as a candidate connection. Finally, we check whether $g_1\text{->}g_5$ is redundant by checking if it is testable when s-a-1 fault is assumed there. The SMA of this fault is inconsistent and therefore, wire $g_1\text{->}g_5$ is an alternative wire for wire $c\ \text{->}g_2$.

Of the three steps, the first and third steps can be efficiently performed by implication and checking the consistency of the SMA. Step 2 is detailed in the following. Given a target wire to be removed, Fig. 6 shows two types of candidate transformations. The dotted wires in Fig. 6 are the candidate connections. Both types of candidate connections in Fig. 6 follow the same principle that after making any transformation, the SMA of the target wire becomes inconsistent and the target wire becomes redundant. The destination gate g_d is a dominator of the target wire. The source gate g_s has a mandatory assignment and we also impose the restriction that g_s is not in the transitive fanout of the target wire. This is because if g_s is in the transitive fanout of the target wire, adding $g_s\text{->}g_d$ may alter the mandatory assignment at g_s. The candidate connection of type 0 adds a wire from g_s to g_d. The candidate connection of type 1 adds a wire and an inverter from g_s to g_d.

The gate g_d is a dominator. The gate g_1 is in the fault propagating paths.
The gate g_2 is a side input. The gate g_s has a mandatory value, val, for the target fault.



(b) two types of transformations

Fig. 6: Types of transformations

Both candidate connections when added will make the MA of the target wire inconsistent. For example, after the type 0 transformation, g_s becomes a side input to the dominator g_d and requires a noncontrolling value 1 for the fault propagation. This new requirement at g_s after the transformation will conflict with the original mandatory assignment 0. As a result, after this type 0 transformation, the MA for the target fault becomes inconsistent and the target wire becomes redundant.

Fig. 7 shows the pseudo code for finding the single-alternative wires of a target wire. Note that this is a subset of the entire set of single-alternative wires because only subset of mandatory assignments are found. For each candidate connection (g_s->g_d), we consider both types of circuit transformations suggested in Fig. 6.

```
single_wire_ alternative(wire, stuck_type) {
    compute_SMA(wire, stuck_type);
    for (each gate g_i in the circuit){
            if (g_i has mandatory assignment) insert g_i into the source_array.
    }       if (g_i is a dominator) insert g_i into the destination_array.
    for (each g_s in the source_array) {
        for (each g_d in the destination_array) {
            if(verify_redundancy(g_s, g_d, type_0 or 1))
                insert_into_alternative_wire_array(g_s, g_d, type_0 or1);
        }
    }
}
```

Fig. 7: Pseudo code for finding single-wire alternative

5 An improved single-alternative wire procedure

In Section 4 we only considered candidate connections that connect to dominators of the target wire. In this section, we show other possible candidate connections.

In Fig. 5 we choose the wire g_1->g_5 as a possible alternative wire for c->g_2. The reason is that g_5 is a dominator of c->g_2 and adding a wire input to the dominator g_5 blocks the fault propagation path of g_1->g_5. Such a way of blocking the fault propagation at target wire's dominator is the main idea of previous single-wire alternative procedure. There are other transformations that can make the target fault untestable not necessarily by blocking the wire's fault propagation at its dominator. Again, consider the same example in Fig. 5a. There, we add g_1->g_5 to remove the wire c->g_2. Since alternative wires are mutual alternatives of each other, we can also add wire c->g_2 to remove g_1->g_5 in Fig. 5b. The gate g_2 is not a dominator of g_1->g_5. Below, we discuss how to find such alternative wires.

We distinguish two types of mandatory assignments: (1) backward MA, (2) forward MA. If a mandatory assignment of a gate g_i is obtained by backward implications from g_i's fanout, the mandatory assignment is a backward MA. If a mandatory assignment of a gate g_i is obtained by a forward implication from g_i's fanin, the mandatory assignment is a forward MA. During the computation of mandatory assignments for the target fault, we can easily mark a gate's mandatory assignment as forward or backward depending on the direction of the implication.

Example 2: Consider the s-a-1 fault at g_1->g_5 in Fig. 5b. We have MA = {g_2=1, g_4=1, ...}. The mandatory assignment 1 on g_2 is a backward MA. This MA is from g_2's fanout. The mandatory assignment 0 on g_4 is a forward MA. This MA is from g_4's fanins.

To identify more candidate connections, we may consider a candidate connection from some gate with a mandatory assignment to another gate with a backward (justified) mandatory assignment. In the improved procedure, the line 5 of Fig. 7 changes to " if ((g_i is a dominator) || (g_i has a backward mandatory assignment)) insert g_i into the destination_array." The following example demonstrates such a candidate connection.

Example 3: Suppose we would like to find the single alternative wire for g_1->g_5 for the circuit shown in Fig. 5b. For g_1->g_5 s-a-1, the SMA is {g_2=1, g_4=1, c=0, ...}. Since g_2 has a backward (justified) mandatory assignment 1 and the primary input c has the mandatory assignment 0, the connection c->g_2 is a candidate connection for the target wire g_1->g_5.

6 A more general procedure for finding single-alternative wire

Now we discuss how a wire can be removed by adding not only a wire but also a two-input gate. Because the search space is much larger when adding a two-input gate, in this section, we show how to reduce the solution domain. It is possible by making use of an observation that while one wire is tested for redundancy, information about irredundancy of other wires is obtained as a side effect. This observation allows us to eliminate many unnecessary redundancy tests.

During the discussion, we define a wire w_f to be a *fault propagating wire* of the target wire w_r if there is a path from wire w_r to wire w_f.

6.1 Finding single-alternative wires by adding gates

We will now extend the discussion of Section 5 and address the following general problem. Given a source gate g_s which has a MA for the target wire w_r and a destination gate g_d which is a dominator of w_r, we are asking what are the possible types of logic transformations to connect g_s to g_d such that w_r becomes redundant. For simplicity, we assume g_d is a two-input gate.

We can view these logic transformations as replacement of g_d by a 3-input gate fed by g_s, g_1 and g_2 (the dotted box in Fig. 8a). There are 2^{2^3} (256) possible 3-input functions. We have filtered these 256 three-input Boolean functions and found that only sixteen of them are candidate logic transformations, which have the desired property of blocking the target fault propagation. In Fig. 8(b)(c), we list two of these sixteen 3-input functions, named Type 0 and Type 1 transformations.

The Type 0 transformation (Fig. 8b) is the one suggested in the original RAMBO procedure. Consider the Type 1 transformation in Fig. 8c. Let g_1 be a fault propagating wire and g_s be a wire with mandatory assignment 0 outside the transitive fanout of w_r. Because g_d is a dominator of w_r, the added gate g_n is also a dominator of w_r. Since g_s is a side input to the dominator g_n, g_s must be assigned a non-controlling value 1, which causes a conflict with the original mandatory assignment of 0. Therefore, this Type 1 transformation is also a candidate logic transformation.

g_s has a mandatory assignment 0, g_d is a dominator (AND gate) and g_1 is a fault propagating wire.

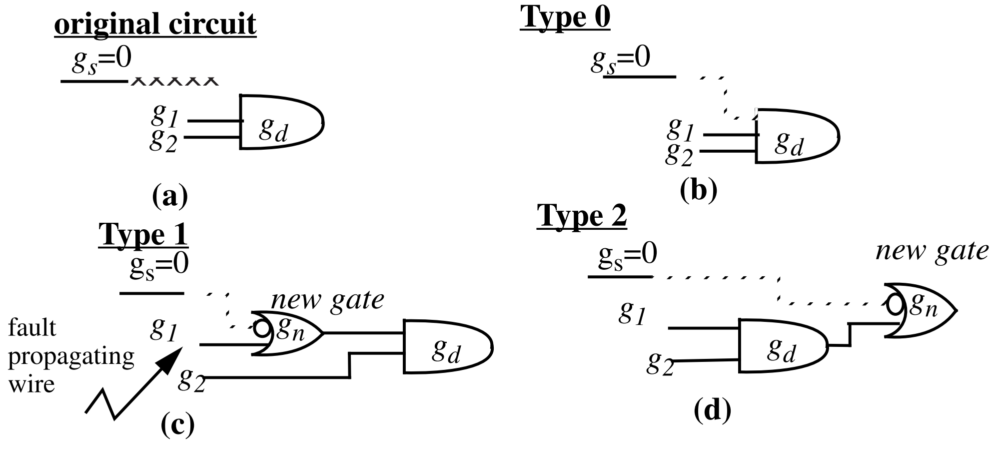


Fig. 8: Circuit transformations when a gate has mandatory assignment 0 and a dominator is an AND gate.

Now we will show that only Type 0 and Type 1 transformations in Fig. 8 need to be checked and the other fourteen candidate transformations need not to be examined. In Fig. 8d, we show another possible candidate transformation named Type 2. The following theorem guarantees that if the added wire/gate of the Type 1 transformation is irredundant, then the added wire/gate of the Type 2 transformation, when applied to the same g_s and g_d, is also irredundant. Therefore, the Type 2 transformation need not be examined if the Type 1 transformation has been performed. We omit a similar discussion when g_d is an OR gate, or when g_s has a mandatory assignment 1, and the cases of the remaining thirteen adequate logic transformations.

Theorem 1: Consider the transformations of Type 1 and Type 2 in Fig. 8 applied to the same g_s and g_d. If a new wire added to the network as suggested by the Type 1 transformation is irredundant, then so is the wire added by the Type 2 transformation.

Proof: We prove that an input vector that can test g_s->g_n s-a-0 in the Type 1 transformation can also test g_s->g_n s-a-0 in the Type 2 transformation. Suppose the Type 1 transformation has been performed. Let v be a test vector for the stuck-at-1 fault of the dotted wire in Type 1 transformation. Applying v to the circuit causes g_1=0 and g_2 =1 (g_1 and g_2 are the side inputs to the dominator g_d). If we apply v to the inputs of the circuit with the Type 2 transformation, it will cause g_d=0 because g_1=0 and g_2 =1. Therefore, v is also a test vector for g_s->g_n s-a-1 under the Type 2 transformation since the fault can also be propagated though g_d.

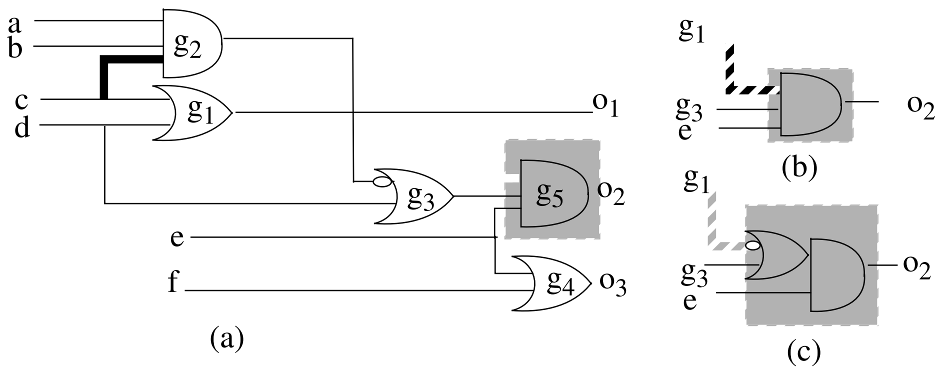


Fig. 9: Though there are 16 candidate functions when replacing the 3-input box (highlighted) can make c-> g_2 redundant in (a), only two candidate functions (b) and (c) need to be examined for redundancy

Example 4: In Fig. 9a, even though there are 16 candidate functions which when replace the 3 input box (highlighted) will make the target wire c->g_2 redundant, only the two candidate functions shown in Fig. 9(b,c) need to be examined. If these two functions are not redundant, all the remaining 14 functions are not redundant either.

6.2 Further exclusion of unnecessary redundancy checks

In the following, we show that, depending on the value at the dominator gate g_d for testing the target fault, the added wire/gate of either Type 0 or Type 1 transformations must be irredundant. Based on this knowledge, we can further prune the space of redundancy checks.

We define $b_g(g_i)$ to be a binary *fault-free* value at the output of gate g_i, and $b_f(g_i)$ be the *faulty* value at g_i.

Example 5: Consider c->g_2 s-a-1 in Fig. 9a. For vector (a,b,c,d,e,f) = (1,1,0,0,1,1), $b_g(g_2)$=0, $b_f(g_2)=1$, $b_g(g_3)$=1, $b_f(g_3)$=0, $b_g(g_5)$=1, *and* $b_f(g_5)$=0.

Theorem 2: Let w_r be an irredundant wire in a network. Let g_d be a dominator of w_r. For *the* w_r stuck-at fault, if there exists a test vector v_t which causes that $b_g(g_d)=1$ and $b_f(g_d)$=0, then, any candidate connection suggested by the Type 0 transformation is irredundant. If there exists a test vector v_t which causes $b_g(g_d)=0$ and $b_f(g_d)$=1, then the candidate connection suggested by the Type 1 transformation must be irredundant.

Proof: Consider the case that under test vector v_t, $b_g(g_d)=1$ and $b_f(g_d)$=0. After the Type 0 transformation, under the same vector v_t, $b_g(g_d)$ will be 0 and thus the good circuit behavior has been changed for this vector. Therefore, Type 0 transformation must be irredundant.

Corollary 1: Suppose that all test vectors for w_r stuck-at fault cause $b_g(g_d)=0$ and $b_f(g_d)$=1, then only the Type 0 transformation needs to be considered. If $b_g(g_d)=1$ and $b_f(g_d)$=0, then, only the Type 1 transformation needs to be considered. If some test vectors cause $b_g(g_d)=0$ and $b_f(g_d)$=1 and others cause $b_g(g_d)=1$ and $b_f(g_d)$=0, then neither of the transformations needs to be considered.

Fig. 10 enumerates all the transformations which are generalized from Type 0 and Type 1 where g_d could be a two-input AND or OR gate, g_s has a mandatory assignment either 1 or 0, and g_d has a mandatory assignment of either 1/0 or 0/1.

7 Multiple-wire addition and gate function substitution

In this section, we extend the idea of adding *one* wire(gate) to adding *multiple* wires (gates). In addition, for the purpose of removing a wire, we also allow gates to change their functionality.

The three-tuple in each Type refers to
(the mandatory assignment of g_s, function of g_d, b_g/b_f for g_d)

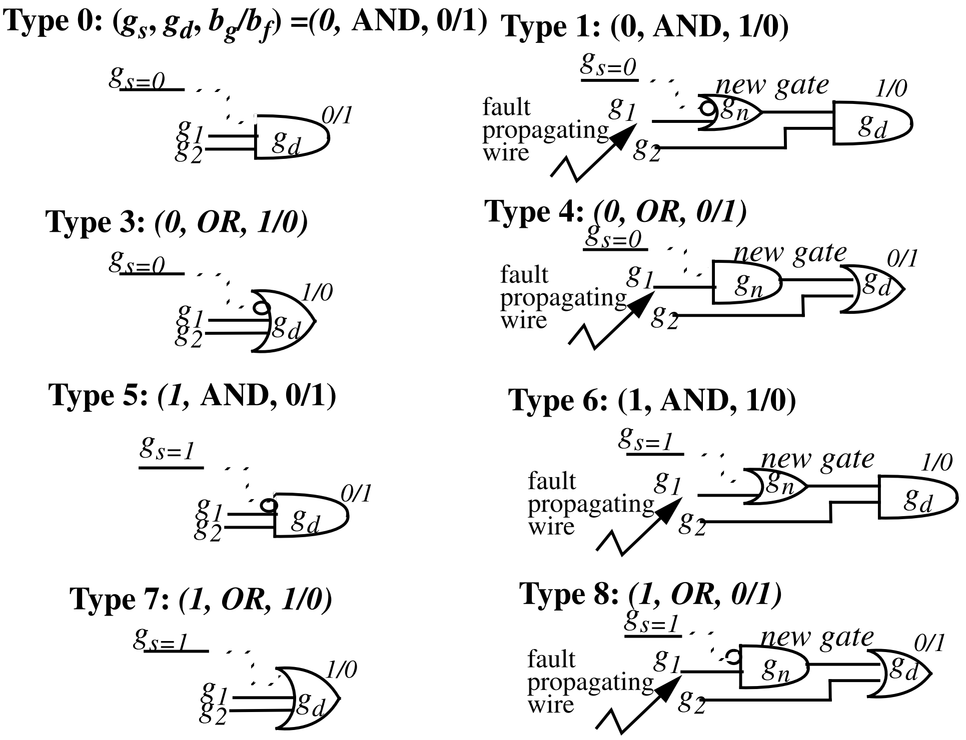


Fig. 10: The complete set of transformations that need to be checked

7.1 Multiple-wire addition

Removing a redundant fault may result in the deletion of multiple wires and/or gates. For example, when g_6->g_7 in Fig. 2 is removed, the 3-input gate g_6 can be also removed. This in turn leaves gate g_7 with a single input, therefore a direct connection g_3 ->g_8 is possible and g_7 can be deleted. If deleting a wire can result in the removal of more than *2* wires or gates from the network, we refer to such a wire as a *large_redunction* wire. When optimizing a circuit, we give higher priority to removing large_redunction wires. In case that adding one redundant wire(gate) cannot remove a large_redunction wire, we may add more than one wire (gate) to delete the wire in question. For example, in Fig. 2, we add a 2-input gate g_m and a wire g_m->g_9 to remove the large_redunction wire g_6->g_7. However, arbitrarily adding many wires as in [6] to remove a large_redunction wire is computationally expensive. We have developed an efficient approach for the multiple-wire addition.

We will explain the principle of the multiple-wire addition by analyzing the example in Fig. 2. Suppose our objective is to remove the wire g_6->g_7. First, we attempt the one-wire substitution using the approach explained in Section 5. We compute the SMA(g_6->g_7 s-a-1) = {g_5 =0, g_3=1, g_1=0, g_4=0, a=1, b =1, d =0, f =1}. Then, we determine a set of candidate connections. A candidate connection is expressed by a triple: (source node, destination node, type). Consider the candidate connection (g_5, g_9, Type 0). To make sure that it will not affect the functionality of the network, we need to verify its redundancy. Since the SMA(g_5->g_9 s-a-1) = {g_5 =0, g_3=1, g_1=0, g_4=0, a=1, b =1, d =1, f =1, g_6 =1} is consistent, we cannot conclude that the g_5->g_9 is redundant. Therefore, the target wire g_6->g_7 cannot be removed by adding just the candidate connection (g_5, g_9, Type 0).

The candidate connection (g_5, g_9, Type 0) causes the wire g_6->g_7 to be redundant, but it itself is irredundant. We may still add it to the network and seek another redundant wire or gate to be added which would make (g_5, g_9, Type 0) redundant, or which would replace g_5->g_9 by some redundant structure. To achieve this, we now consider simultaneously the SMA(g_6->g_7 s-a-1) and SMA(g_5->g_9 s-a-1). We note that $d=0$ in SMA(g_6->g_7 s-a-1) but $d=1$ in SMA(g_5->g_9 s-a-1). Suppose a 2-input OR gate g_m(with inputs from d and g_5) is added as shown in dotted line in Fig. 2a. We will prove that the added wire g_m->g_9 is redundant and the addition of g_m->g_9 makes the wire g_6->g_7 redundant too. To verify the redundancy, we need to compute the SMA(g_m->g_9 s-a-1). The only difference between SMA(g_m->g_9 s-a-1) and the SMA(g_5->g_9 s-a-1) is that we have one more assignment, namely $d=0$ in the SMA(g_m->g_9 s-a-1). Since $d=1$ is in the SMA(g_5->g_9 s-a-1), the SMA(g_m->g_9 s-a-1) is inconsistent, and g_m->g_9 is redundant. As a result, we can add the new wire g_m->g_9 as shown in Fig. 2b. without changing the circuit's functionality. Furthermore, the addition of g_m->g_9 also guarantees that the target wire g_6->g_7 becomes redundant. It is so because on one hand d=0 and g_5=0 in the SMA(g_6->g_7 s-a-1) so that g_m=0. On the other hand, the mandatory assignment of g_m needs to be 1 because it is a side input of the dominator g_9. The addition of wire g_m->g_9 causes inconsistency of the SMA(g_6->g_7 s-a-1). Therefore after adding the gate g_m and the wire g_m->g_9, we can remove the wire g_6->g_7.

The above example demonstrates our basic philosophy of adding multiple wires so as to cause an originally irredundant candidate connection to become redundant. Suppose we wish to remove a large_reduction wire, and all the single-wire candidate connections are irredundant. Then, we try to add multiple wires. The procedure is as follows. We compute and store the SMA(w_r stuck-at fault). Then, we pick a candidate connection, w_c, (g_{s1}, g_d, type) and compute the SMA(candidate wire stuck-at fault). Then, we look for a gate denoted as g_{s2} such that it is in both the SMA(w_r stuck-at fault) and the SMA(w_c stuck-at fault) but has different mandatory assignments. In the example of Fig. 2, the gate d appears with the assignment 0 in the SMA(g_6->g_7 s-a-1) and with the assignment 1 in the SMA(g_5->g_9 s-a-1) so the gate d is our g_{s2}. Finally, we add a gate g_m, and a wire g_m->g_d where g_d is the dominator of the candidate connection, using the following rule. The gate g_m is an OR(AND) gate, if the gate g_d is an AND(OR) gate. The inputs to g_m are g_{s1} and g_{s2}. If g_{s1} =1(0) in SMA(w_r stuck-at fault) and g_m is an OR(AND) gate, we invert the input of g_{s1} to g_m. The same rule is applied to g_{s2}. In our example, g_m is an OR gate and $g_{s1} = g_5$ and $g_{s2} = d$. Both $d=0$ and g_5=0 in the SMA(g_6->g_7 s-a-1) so we do not invert the input phase for d and g_5 to g_m.

7.2 Changing the gate's functionality

In this section, we discuss how to change a gate's function to remove a particular wire. For example, in Fig. 3, we can change the gate g_5 from an AND to an XNOR without changing the circuit functionality. After changing g_5 to an XNOR gate, the wire g_6->g_7 becomes redundant. Two issues need to be addressed; namely, how to check if a given gate can change its functionality, and which gate should be changed to make a target wire redundant.

Consider an AND gate $g_x(g_{i1}, g_{i2})$ with two inputs g_{i1} and g_{i2}. If the output of g_x is a don't care when $(g_{i1}, g_{i2})=(0, 0)$, the function of $g_x(g_{i1}, g_{i2})$ can change from AND to XNOR. We first show a procedure to verify whether ($v_1, v_2,...,v_n$) is a don't care minterm for a gate $g_x(g_{i1}, g_{i2},..., g_{in})$. It is based on checking consistency of a certain SMA.

To verify whether a minterm ($v_1, v_2,...,v_n$) is a don't care to $g_x(g_{i1}, g_{i2},..., g_{in})$, we first set $g_{i1}=v_1$, $g_{i2}=v_2$,...., $g_{in}=v_n$ and include this assignment in a SMA. Then, we treat the output of g_x as stuck-at the value produced by that minterm and compute the appropriate SMA. Therefore, this SMA includes the MAs for justifying the minterms plus the MAs to sensitize a path from g_x to any primary output. In the following theorem, we show that if the SMA(f) is inconsistent, the minterm is a don't care.

Theorem 3: Consider the SMA induced by setting the gate's inputs to a minterm in question and treating the gate's output as stuck-at the value produced by the minterm. If this SMA cannot be consistently justified, then the minterm $(g_{i1}, g_{i2},..., g_{in})$ is a don't care of the $g_x(g_{i1}, g_{i2},..., g_{in})$ embedded in the network [4].

For example, consider the gate g_5 in Fig. 3. We will verify that $(g_1, g_2)=(0, 0)$ is a don't care of g_5. We set g_1=0, g_2=0 and assume that there is a fault at the output of g_5 in the network. Since g_9 is a dominator of g_5, we need the following mandatory assignments at the side inputs: g_8=1 and f=1. This yields the SMA = (g_1=0, g_2=0, g_8=1, f=1, g_4=0, g_7=1, g_6=1, g_3=1, a=1, b=1, d=1,g_1=1). Since g_1 is inconsistent, $(g_1, g_2)=(0, 0)$ is a don't care of g_5.

Theorem 3 suggests how to verify quickly if a gate can switch its functionality without affecting the network's behavior.

The sets of mandatory assignments, SMAs, computed to justify particular conditions in the network depend on gates' functionality. We may change some gates' functionality to achieve SMA's inconsistency, and therefore create a redundancy. In Fig. 3, suppose we wish to delete the connection g_6 ->g_7. The SMA(g_6 ->g_7 s-a-1) is (g_1=0, g_2=0, ..., and g_5=0). Now, changing g_5 from an AND to an XNOR causes that g_5 = 1 because g_1=0 *and* g_2=0. But on the other hand g_5 needs to be 0 for the fault propagation. Therefore, we conclude that if g_5 can be changed to an XNOR gate, then SMA(g_6 ->g_7 s-a-1) is inconsistent and g_6 ->g_7 is redundant.

For the large_redunction wires, if the single wire addition approach does not work, we try either to add multiple wires or to make a functional change of some gates. These two new techniques substantially extend the power of redundancy addition and removal.

8 Simultaneous addition and removal of two wires (gates)

We say that two wires (w_a, w_b) are simultaneously redundant if each wire is itself irredundant but simultaneously adding/removing w_a and adding/removing w_b does not change the circuit's functionality. That is, the double fault w_a and w_b is redundant. For example, in Fig. 4 the wire a-> g_1 and a new connection c->g_2 are each irredundant. Therefore we cannot remove(add) a->g_1(c->g_2). However, we can simultaneously remove the wire a->g_1 and add c->g_2 without changing the network's functionality. We have developed an efficient algorithm to identify two simultaneously redundant wires.

Multiple stuck-at faults have been studied extensively [13]. The existence of two simultaneously redundant wires is analyzed as follows. Let V_t(w_r stuck-at fault) denote all the input vectors that can test the w_r stuck-at fault. That is any vector in V_t(w_r stuck-at fault) can distinguish the original (good) circuit from the faulty one. Consider another wire w_a. If both sets, V_t(w_r stuck-at fault) and V_t(w_a stuck-at fault) are the same, and the fault effects of these two faults are propagated through the same XOR (XNOR) gate (from different inputs of the gate), they cancel out at the output of the XOR (XNOR) gate. Thus, neither V_t(w_r stuck-at fault) nor V_t(w_a stuck-at fault) can detect the double fault. In the example of Fig. 4, $V_t(a\text{->}g_1$ s-a-1) is the same as $V_t(c\text{->}g_2$ s-a-0)= {(0,0,1)} and both faults propagate through the same XOR gate. For the $a\text{->}g_1$ s-a-1, the fault b_g/b_f at g_3 is 0/1 and for the $c\text{->}g_2$ s-a-0, the fault b_g/b_f at g_4 is 1/0. When primary inputs = (0,0,1), the XOR output of the original circuit is 1 in the good and in the double faulty network.

Theorem 4: Let g_r be a fanin of XOR(XNOR) gate g_x and let g_a be another gate. If (g_a, g_r) =(0,1) and (g_a, g_r) =(1,0) are don't cares in the network, we can replace g_r->g_x by g_a->g_x.

Proof: Any input vector v_t that can distinguish between the original circuit and the faulty circuit (circuit after wire replacement) is the vector that causes (g_a, g_r) =(0,1) or (g_a, g_r) =(1,0). In addition, the discrepancies between the good and faulty networks after applying v_t should propagate to a primary output. If the set of such vectors is empty then the good and faulty circuits are functionally equivalent. The above procedure verifies that indeed such vectors do not exist.

Note that when a dominator g_d is an XOR(XNOR) gate, the network transformations in Fig. 10 are no longer valid and cannot be used to remove a particular wire. It is because some of the test vectors cause g_d to assume b_g/b_f = 1/0, and others cause g_d to assume b_g/b_f =0/1. According to Corollary 1, none of the transformations can be applied. Therefore, the redundancy addition and removal technique cannot be applied when a dominator is an XOR(XNOR) gate.

The wire replacement technique based on identification of two simultaneously redundant wires is very useful when there are XOR(XNOR) gates in the network. It can be applied to remove large_reduction wires which are inputs of XOR(XNOR) gates.

9 A greedy optimization based on incremental transformations

In this section, we discuss how to apply the transformation technique for logic optimization. The global optimization strategy is similar to that used in the program RAMBO [8] [11]. A node g_i is chosen randomly. For each wire w_r which shares at least one dominator with g_i, we compute the alternative wires for w_r. Then, we form a list of such alternative relationships. For example, we record that adding wire *x* or *y* may make wire *a* redundant, adding wire *x*, or *z* may make wire *b* redundant, and adding wire *x* may make wire *d* redundant. Therefore, we can conclude that adding wire *x* may make wires *a, b,* and *d* redundant. Note that in RAMBO, when adding a wire to a node g_i, they only consider removal of the wires in the input cone of g_i. In our algorithm, we consider all wires that share a common dominator with g_i.

This greedy approach adds one or several wires/gates to remove more wires and gates from a circuit. It is very efficient because at each iteration only a subset of wires is considered. In addition, whenever some wires/gates are added to a circuit, some wires/gates can be removed. On the other hand, such a greedy approach may easily be stuck at a local minimum. In the following we discuss a perturbation technique which can lift such a greedy approach out of a local minimum.

10 Internal don't care migration after wire substitution

The single-wire substitution technique can be viewed as a perturbation. Replacing one wire by another wire can alter the circuit structure without increasing its size. Repeatedly applying single wire substitutions may potentially lead to a better circuit structure for subsequent optimization. Instead of randomly applying such single-wire substitutions, we carefully guide the process. In this section, we analyze the internal don't care migration after single-wire substitution. Based on the analysis, we can guide the substitution to achieve efficient optimization.

The degree of difficulty to propagate a fault effect from a wire(gate) to any primary output is expressed in terms of its observability don't cares. Intuitively, it is easy (difficult) to propagate a fault effect to any primary output when the don't care set is small (large). We use the term *observable* to describe how difficult it is to propagate the fault. Therefore, if a gate has a fanout to a primary output, the gate is fully observable and has no observability don't cares. Typically, if a gate has more fanouts, it is easy to be observed. This is because we may have more paths to primary outputs. However, it is not always true because multiple fault effects may cancel each other due to reconvergence of fanouts. In the following, we discuss how observability don't cares can be redistributed when one wire is replaced by another wire.

In Fig. 11 two types of single wire substitutions are depicted. They are referred to as fanout and fanin substitution. We say that fanout substitution occurs when a fanout wire (g_a->g_d) of a gate is replaced by another fanout (g_a->g_b). This case is shown in Fig. 11a. Here, we restrict our discussion to the case when g_b is a dominator of g_d and g_b has the same gate type as g_d. We will discuss how the internal don't cares are redistributed during fanout substitution. Referring to Fig. 11a, let Cone II contain all the wires in the intersection of the transitive fanout cones of g_d and in the transitive fanin cones of g_b.

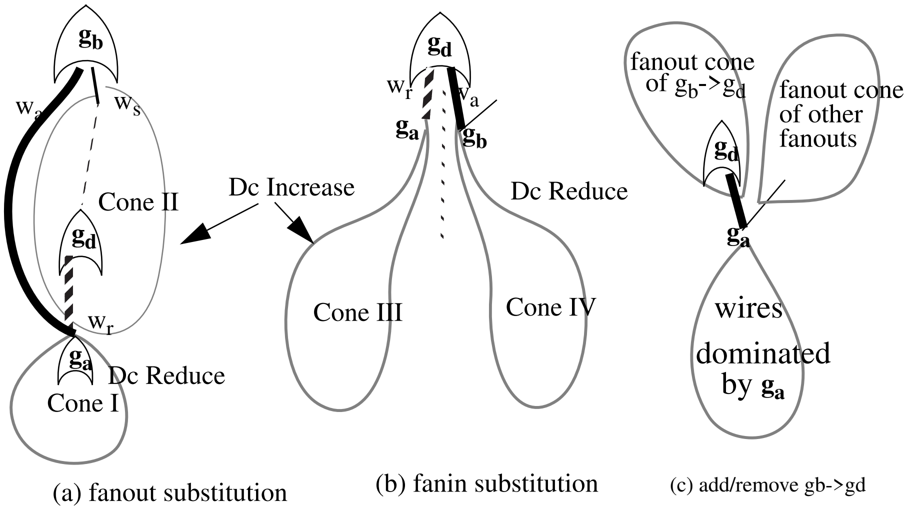


Fig. 11: Internal don't cares migration

Theorem 5: Suppose wire g_a->g_d is replaced by g_a->g_b and w is a wire in Cone II. A test pattern which detects w stuck-at fault in the circuit after substitution also detects the fault in the circuit before substitution.

Proof: Consider testing w stuck-at fault in the circuit after substitution. Because g_b is a dominator for w, the side input g_a must be assigned to noncontrolling value. This assignment at g_a is independent of both wires g_a->g_d and g_a->g_b. Therefore any test pattern which can detect w stuck-at fault for the circuit after substitution is also a test pattern for the same fault in the circuit before substitution. Q.E.D.

Corollary: The size of don't care set for the wires in Cone II increases after replacing g_a->g_d by g_a->g_b.

The increase of don‘t cares for those wires in Cone II will be beneficial for circuit optimization. However such replacement may also potentially reduce the don't cares for other parts of the circuit. Consider the situation of replacing g_a->g_d by g_a->g_b. Let Cone I (in Fig. 11a) include all the wires that are in the transitive fanin of g_a. The size of the don't care sets for the wires in Cone I may potentially decrease. The intuition behind this observation is as follows. For simplicity, assume that g_a->g_d is the only fanout and is replaced by g_a->g_b. Since g_b is a dominator of g_d, the *distance* from g_a to the primary output is now shorter. It is therefore easier to *observe* g_a after replacement. If the observability of g_a is increased, so are the gates in the input cone of g_a. Therefore the observability don't cares for wires in the input cone of g_a may decrease.

To summarize the effect of such a replacement, we have the don't care set size increasing for the wires in Cone II and potentially decreasing in the Cone I. If g_a is a primary input, the input cone of g_a is empty. The replacement g_a->g_d by g_a->g_b is always beneficial. Suppose now that g_a has a fanout to a primary output. Replacing g_a->g_d (g_d is another fanout of g_a) with g_a->g_b is also good for optimization. This is because g_a is already fully observable since one of its fanouts is a primary output. It is not possible to increase its observability anymore. Intuitively, we can gain some benefit if g_a is close to primary inputs or primary outputs and g_b is far away from g_d.

We now focus on the other type of single wire substitution. First, we have the following observation. In Fig. 11c, let g_a->g_d be a fanin wire of g_d. Suppose the fanout cone of wire g_a->g_d does not intersect the fanout cone of g_a's other fanouts. Adding (deleting) g_a->g_d decreases (increases) the don't cares for wires that are dominated by g_a.

The intuition behind is as follows. If a gate has more fanouts, it is easier to observe it. As a result, the don't cares associated with this gate decrease and so happens with the wires that are dominated by this gate. The restriction on the empty intersection of the fanout cones is to preclude the possibility of reconvergence. Reconvergent fanouts may potentially cause fault effect cancellation and invalidate the above theorem.

Based on the above observation, we consider the following single wire substitution in Fig. 11b. Let the fanin wire g_a->g_d of a gate g_d be replaced by another fanin wire g_b->g_d. According to the above theorem, since one fanout wire of the gate g_a is deleted, the don't cares of the wires (marked as Cone III in the figure) dominated by g_a possibly increase. And since one fanout wire of the gate g_b is added, the don't cares of those wires (marked as Cone IV) dominated by g_b possibly decrease. For logic optimization purposes, we would like the area of Cone III to be larger than the area of Cone IV.

The observation of the single wire substitution effect on the internal don't cares migration can be used to guide wire selection for replacement to improve the optimization results. As shown in Fig. 11a and Fig. 11b, the don't care sets of wires in Cones II and III increase and are beneficial for logic optimization and the don't cares of wires in Cones I and IV decrease and are detrimental. In general, keeping track of how much don't cares increase/decrease is quite time consuming. Our heuristic uses the number of dominators of a gate as cost function. After substitution, if a gate has more (less) dominators, then most likely that it has less (more) don't cares than before.

We now discuss the perturbation algorithm. Both fanin and fanout substitutions mentioned above are applied. For each wire w_1 (g_a->g_d) in the circuit, we check the fanout substitution first. If there exists a fanout substitution w_2 (g_a->g_b) for wire w_1, and if the size of Cone II is greater than the size of Cone I in Fig. 11a, we calculate the difference of the number of g_a's dominators before and after the substitution. If the increase of the number of g_a's dominators is smaller than some threshold, we replace w_1 (g_a->g_d) by w_2 (g_a->g_b). Otherwise, we try the fanin substitution. If there exists a fanin wire w_2 (g_b->g_d), and the size of Cone III is greater than the size of Cone IV in Fig. 11b, we count the difference of the number of g_b's and g_a's dominators before and after wire substitution. If the decrease of number of g_a's dominators is greater than the increase of number of g_b's dominators, we perform the fanin substitution. The pseudo code for perturbing the circuit is shown Fig. 12.

11 The overall algorithm

The overall algorithm of our optimization scheme is shown in Fig. 13. If the circuit is a sequential circuit, we calculate the unreachable states and use them as external don't cares for optimizing the combinational parts of the network. Then, we perform *n* iterations of the following two steps: *greedy_optimization* described in section 9 and *perturb_the circuit* described in Fig. 12.

```
Perturb_the_circuit {
    for each wire w1 (ga->gd) in the circuit {
        if (there is a fanout substitution w2=gb->gd) {
          if (Cone II > Cone I &&
              # of dominators of ga increase < THRESHOLD)
                replace_w1_by_w2();
        }
        else if (there is a fanin substitution w2=gb->gd) {
          if (Cone III > Cone IV &&
              decrease of # of dominators of ga> increase of # of dominators of gb)
                replace_w1_by_w2();
        }
}   }
```

Fig. 12: The perturbation algorithm

12 Experimental results

We applied the algorithm in Fig. 13 to MCNC and ISCAS combinational and sequential circuits and the parameter *n_iterations* was chosen to be 2.

TABLE 1 shows results for combinational circuits. For these circuits, we minimize circuits using s*cript.algebraic* of misII first, followed by our algorithm and RAMBO. For

```
perturb_simplify(n_iterations)
   {
        if (sequential_circuit) {
              cal_unreachable_states();
              construct_external_dont_cares();
         }
        for (i =1 to n_iterations ) {
             greedy_optimzation(); /* shown in Section 10 */
             perturb_the_circuit(); /* shown in Section 11 */
        }
 }
```

Fig. 13: The overall algorithm

comparison with misII, we also run *script.boolean* of misII (for consistency, we don't use *script.rugged* because some examples cannot be run due to space/time limitation). For all cases, we mapped the circuits into 2-input gates by invoking the *map* command in misII. Our result is shown in column Perturb/Simplify. For all three algorithms: misII, RAMBO and Perturb/Simplify, we use the number of 2-input gates as a measure. Columns 2, 3, and 4 of TABLE 1 show the number of 2-input gates and literal counts. On the average for the listed examples, our results are 19% better than misII and 16% better than RAMBO in terms of number of 2-input gates. Our algorithm has also a low memory requirement. For example, C7552 uses only 6 Mbytes of memory. This makes our approach very attractive for large circuits. All our results have been verified using the circuit verification command in misII.

For MCNC and ISCAS sequential benchmarks circuits, we computed the unreachable states and use them as external don't cares. Then, the circuits were minimized by using *script.algebraic* in misII [5]. Next, we applied our logic optimizer to the resulting circuits. The results of some sequential benchmarks are shown in TABLE 2. We compared them with those produced by *script.rugged* in misII [5]. The misII results also include the use of the unreachable states as external don't cares. Similarly to TABLE 1, Columns 2 and 3 show the number of 2-input gates and the literal count. The average improvement of our results to misII's is around 21%.

Again, our algorithm shows a very low memory usage. For example running the circuit *scf* requires only 3 Mbyte of memory and running *s1196* requires 10 Mbytes. We do not show several large sequential circuits because the state reachability computation causes time/space violation in our system. However, for logic optimization, the complete reachability is not required. Heuristics for finding subsets of unreachable states [9] can be used.

13 Conclusions

In this paper, we presented an ATPG based approach to simplify multilevel Boolean networks. In particular, we have proposed several new techniques to add one or more redundant gates/wires to remove other gates/wires from the network. We have shown how to identify gates which are good candidates for local functionality change for simplifying a network. In addition, we discuss the problem of adding and removing two wires, none of which alone is redundant, but when jointly added/removed they do not affect functionality of the network. These new techniques allow us to simplify substantially the networks. Our experimental results have demonstrated usefulness of our approach.

14 References

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

[2] K.A. Bartlett et al, “Multilevel Logic Minimizing Using Implicit Don’t cares,” IEEE Tran. on CAD-7(6), pp. 723-740, June 1988.

[3] C. L. Berman and L. H. Trevillyan. “Global Flow Optimization in Automatic Logic Design,” IEEE Trans. CAD 10, pp. 557-564, May 1991.

[4] D. Bostick, G.D. Hachtel, R. Jacoby, M.R. Lightner, P. Moceyunas, C.R. Morrison, and D. Ravenscroft, “The Boulder Optimal Logic Design System,” *Proc. ICCAD*, pp. 62-65, Nov. 1987.

[5] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang, “MIS: Multi-level Interactive Logic Optimization System,” IEEE Trans. on CAD, CAD-6(6), pp. 1062-1081, Nov. 1989.

[6] S.C. Chang, K-T Cheng, Nam-Sung Woo and M. Marek-Sadowska, “Layout Driven Logic Synthesis for FPGA,” *Proc*. DAC, *pp*. 308-313, June 1994.

[7] S.C. Chang, and M. Marek-Sadowska, “Perturb and Simplify: Multi-level Boolean Network Optimizer,” *Proc*. ICCAD, *pp*. 2-4, Nov. 1994.

[8] K.T. Cheng and L.A. Entrena, “Multi-Level Logic Optimization by Redundancy Addition and Removal,” *Proc*. European Conference On Design Automation, pp. 373-377, Feb. 1993.

[9] H. Cho, G.D. Hachtel, E.Macii, B.Pleasier, F. Somenzi, “Algorithms for Approximate FSM Traversal,” *Proc*. European Design Automation Conference, pp. 200-204, Feb. 1993

[10] M.Damiani, J.C.Y.Yang and G.De Micheli, “Optimization of Combinational Logic Circuits Based on Compatible Gates”, *Proc*. DAC, pp.631-636, June 1993.

[11] L.A. Entrena and K. T. Cheng, “Sequential Logic Optimization By Redundancy Addition and Removal”, *Proc*. ICCAD, pp. 310-315, Nov. 1993.

[12] E. Detjens, G. Gannot, R. Rudell, A. L. Sangiovanni-Vincentelli and A. Wang, “Technology Mapping in MIS,” *Proc. ICCAD*, pp. 116-119, Nov. 1987.

[13] J.W. Gault, J.P.Robinson, and S.M. Reddy, “Multiple Fault Detection in Combinational Networks,” IEEE Transactions on Computer, pp. 31-37, 1972.

[14] M. Higashida, J. Ishikawa, M. Hiramine, K. Nomura, “Multi-level Logic Optimization Based on Pseudo Maximum Sets of Permissible Functions,” *Proc*. Euro Design Automation Conf., pp. 386-391, Feb. 1993.

[15] T.Kirkand and M.R. Mercer, “A Topological Search Algorithm For ATPG,” *Proc*. DAC, pp. 502-508, June 1987.

[16] W. Kunz and D.K. Pradhan, “Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation for Digital Circuits”, in *Proc*. Int’l Test Conference, pp. 816-825, October 1992.

[17] C.E.Leiserson, F.M.Rose, and J.B.Saxe, “Optimizing synchronous circuit by retiming”, *Proc*. Third Caltech Conf. on VLSI, 1983.

[18] S. Muroga, Y. Kambayashi, H. C. Lai and J. N. Culliney, “The Transduction Method-Design of Logic Networks Based on Permissible Functions,” IEEE Transaction. on Computer C38(10). pp. 1404-1424, Oct. 1989.

[19] M.Schulz and E.Auth, “Advanced Automatic Test Pattern Generation and Redundancy Identification Techniques,” *Proc*. Fault Tolerant Computing Symposium, pp. 30-34 June 1988.

TABLE 1 Combinational optimization results.

Circuit	misII gates/literals	RAMBO gate/literals	Perturb/ Simplify gates/literals	CPU of Perturb/ Simplify (sec)
5xp1	117(231)	111(221)	66(131)	34.5
9sym-hdl	96(192)	100(200)	39(78)	19.7
C3540	1073(2145)	988(1976)	938(1876)	5692.8
C5315	1452(2871)	1458(2883)	1321(2631)	2236.7
C6288	2619(5237)	2334(4666)	1883(3766)	2124.8
C7552	1757(3513)	1761(3521)	1426(2851)	3668.6
alu2	383(765)	366(731)	281(562)	1127.4
alu4	687(1373)	700(1399)	555(1110)	4171.5
apex6	632(1260)	647(1291)	543(1086)	568.9
b9_n2	102(200)	96(188)	79(156)	17.4
comp	137(273)	119(273)	84(168)	51.9
des	3048(6095)	3073(6145)	2859(5718)	31507.6
duke2	366(727)	314(626)	246(491)	648.5
f51m	120(239)	116(231)	78(155)	4.7
misex3	434(868)	468(936)	317(634)	978.8
my_adder	160(320)	160(320)	116(232)	29.1
pcler8	80(151)	80(151)	64(128)	29.7
rot	575(1135)	569(1131)	452(902)	256
sao2-hdl	195(390)	199(398)	104(208)	119.6

Circuit	misII gates/literals	RAMBO gate/literals	Perturb/ Simplify gates/literals	CPU of Perturb/ Simplify (sec)
term1	203(403)	203(404)	113(225)	56.2
ttt2	184(365)	174(247)	118(236)	57.8
x3	629(1253)	617(1231)	552(1104)	472.0
total	15059 (30006)	14653 (29169)	12234 (24448)	53874.2

TABLE 2 Sequential optimization results

circuit	script.rug (2-input gates/ literals)	Perturb/Simplify (2-input gates/literals)	cpu time perturb-simplify (sec)
bbara	51/101	43/86	15.7
dk14	90/180	76/152	50.3
dk16	256/512	216/432	591.2
ex2	126/252	103/206	90.0
ex3	67/134	51/102	20.1
ex4	66/129	57/114	23.3
ex5	61/102	44/88	14.5
ex6	77/153	58/116	22.9
ex7	56/112	41/82	16.1
keyb	222/444	178/356	455.1
opus	64/128	49/98	22.5
planet1	445/890	325/650	1260.4
s1	309/618	221/442	534.3
s1a	311/620	240/468	709.1
sand	440/880	377/754	1851.3
scf	665/1319	575/1146	5855.2
sse	105/213	75/150	55.6
styr	475/914	339/678	1452.3
tbk	693/1386	564/1128	7202.7
train4	16/32	10/20	1.0
s1196	475/947	384/766	5864.4
s1238	464/925	389/766	6154.1
s1488	540/1079	409/818	2549.3
s1494	538/1075	409/818	2916.9
s382	115/223	91/177	297.2
s386	92/182	73/146	50.1
s400	112/221	89/172	317.3
s444	112/215	92/178	672.2
s820	247/485	183/366	391.4
s832	239/467	177/354	378.5
total	7290/14471	5761/11475	39456