Node Addition and Removal in the Presence of Don't Cares

Yung-Chih Chen and Chun-Yao Wang
Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan
{ycchen, wcyao}@cs.nthu.edu.tw

ABSTRACT

This paper presents a logic restructuring technique named node addition and removal (NAR). It works by adding a node into a circuit to replace an existing node and then removing the replaced node. Previous node-merging techniques focus on replacing one node with an existing node in a circuit, but fail to replace a node that has no substitute node. To enhance the node-merging techniques on logic restructuring and optimization, we propose an NAR approach in this work. We first present two sufficient conditions that state the requirements of added nodes for safely replacing a target node. Then, an NAR approach is proposed to fast detect the added nodes by performing logic implications based on these conditions. We also apply the NAR approach to circuit minimization together with two techniques: redundancy removal and mandatory assignment reuse. We conduct experiments on a set of IWLS 2005 benchmarks. The experimental results show that our approach can enhance the state-of-the-art ATPG-based node-merging approach. Additionally, our approach has a competitive capability of circuit minimization with 44 times speedup compared to a SAT-based node-merging approach.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Optimization*

General Terms

Algorithms

Keywords

Logic implication, node merging, node addition and removal, observability don't care

1. INTRODUCTION

Recently, node-merging techniques have been proposed and enhanced in [4] [5] [7] [12] [14]. They work by merging two nodes – replacing one node with another node – in a logic circuit with don't cares. Two nodes can be correctly merged when they are functionally equivalent or their functional differences are never observed at any primary output (PO). Because the replaced node can be removed and the replacement may result in additional redundancies, the resultant circuit is minimized.

The effectiveness and efficiency of node-merging techniques for circuit minimization have been shown in the previous works. The SAT-based approaches [12] [14] have a great capability of circuit minimization. As reported in [14] and [12], an average of 15.6% nodes can be merged in a benchmark circuit and an average of additional 4.9% circuit size reduction can be achieved for a benchmark circuit after optimized by a synthesis engine [3] [10], respectively. However, the efficiency is a major concern for these SAT-based approaches due to the expense of observability don't care computation and SAT solving calls. On the other hand, the ATPG-based approach [5] is much faster, although its capability is not as good as that of the SAT-based approaches for circuit minimization. The experimental results in [5] show that a large benchmark circuit having more than 70,000 nodes can be optimized in approximately one minute.

However, these previous works only focus on searching and merging two nodes that originally exist in a circuit. Given a target node in a circuit that possesses no substitute node, the node-merging approaches fail to replace the target node. In fact, we observe that a target node without any substitute node could be replaced with a newly-added node. That is, we could add a node into the circuit to replace the target node. For the objective of circuit optimization, once more than one node is removed due to the addition of a new node, the circuit size is reduced as well. We name this technique *Node Addition and Removal* (NAR).

NAR can be considered an improved verison of node-merging technique, which also merges two nodes with don't cares. The difference between NAR and node merging is that NAR uses an added node rather than an existing node to replace the target node. Because more nodes can be replaced by an added node, NAR can enhance the results of node merging in logic optimization.

In this work, we propose an efficient approach for NAR using logic implications. The approach works based on two sufficient conditions that state the requirements of added nodes for correctly replacing a target node. If a given target node possesses no substitute node from the circuit, the approach further identifies an added node to replace it. We also apply the NAR approach to circuit size reduction. Two techniques, redundancy removal and mandatory assignment reuse, are engaged to enhance the performance. Redundancy removal detects redundant nodes without extra effort when the approach identifies substitute nodes. Mandatory assignment reuse is a method for reusing the logic implication results such that the number of required logic implications can be saved.

We conduct experiments on a set of IWLS 2005 benchmarks [15] and compare to the node-merging approaches in [5] and [12]. For replaceable node identification, as compared to the ATPG-based approach [5], an average of 28% more nodes can be identified replaceable in a benchmark circuit by using NAR. For circuit size reduction, the proposed approach has a better capability with a ratio of 1.277 compared to the ATPG-based approach [5], with an overall CPU time overhead of only 4 minutes. Additionally, our optimization capability is competitive with that of the SAT-based approach [12], which is highly time-consuming.

The rest of this paper is organized as follows: Section 2 uses an example to demonstrate the NAR technique and formulates the problem considered in this paper. Section 3 reviews the related concepts in VLSI testing and the ATPG-based node-merging approach [5]. Section 4 presents the proposed algorithm for NAR. The application of NAR for circuit size reduction is introduced in Section 5. Finally, the experimental results and conclusion are presented in Sections 6 and 7.

2. AN EXAMPLE OF NAR

We use an example in Fig. 1 to demonstrate the difference between node merging and NAR. For ease of discussion, the circuits considered in this paper are presented as And-Inverter

*This work was supported in part by the National Science Council of R.O.C. under Grants NSC 98-2220-E-007-015 and NSC 98-2220-E-007-023.

DAC'10, June 13-18, 2010, Anaheim, California, USA

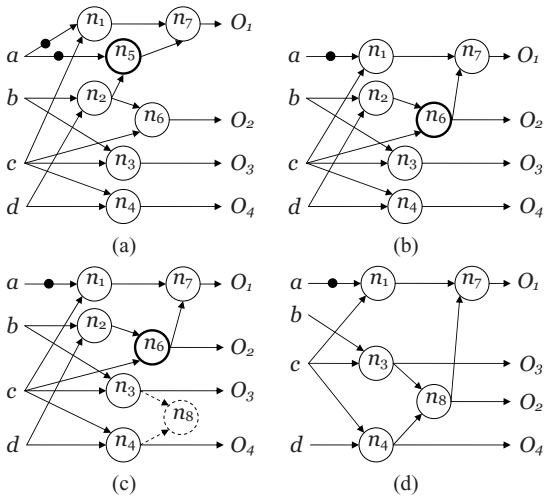


Figure 1: An example for demonstrating node merging and NAR.

Graphs (AIGs) [8], which are an efficient and scalable representation for Boolean networks. Circuits with complex gates can be handled by transforming them into AIGs first. In the circuit of Fig. 1(a), a, b, c, and d are primary inputs (PIs). $O_1 \sim O_4$ are POs. $n_1 \sim n_8$ are 2-input AND gates. Their connectivities are presented by directed edges. A dot marked on an edge indicates that an inverter (INV) is in between two nodes.

First, let us review the node-merging technique. In Fig. 1(a), n_5 and n_6 have different functionalities. However, their values only differ when $n_2 = 1$ and $a = c$. Because $a = c$ further implies $n_1 = 0$, which is an input-controlling value of n_7, the value of n_5 is prevented from being observed at O_1. This situation makes the different values of n_5 with respect to n_6 never observed. Thus, n_5 can be replaced with n_6 without altering the overall functionality of the circuit. The resultant circuit is shown in Fig. 1(b). Here, n_5 is considered a target node and n_6 is a substitute node of n_5.

Next, let us consider n_6 in Fig. 1(b). Suppose n_6 is a target node to be replaced. Because n_6 does not have any substitute node, the node-merging technique fails to replace it. However, we can add a new node into the circuit to replace it. When we add n_8 into the circuit as shown in Fig. 1(c), the functionality of the circuit is unchanged, because n_8 does not drive any node. Additionally, n_8 can correctly replace n_6. The resultant circuit is shown as Fig. 1(d), where n_8 drives n_7 and O_2. This example demonstrates that a node which has no substitute node still can be replaced by a newly-added node and the resultant circuit might be minimized. Thus, the NAR technique can replace a node which cannot be replaced by the node-merging technique, and can optimize a circuit as well.

The problem formulation of this work is as follows: Given a target node n_t in a circuit, find a node n_a which can correctly replace n_t after it is added into the circuit. Here, we name n_a an *added* substitute node to distinguish from a substitute node because n_a is absent in the original circuit.

3. PRELIMINARIES

3.1 Background

This subsection reviews some terminologies used in logic synthesis and related concepts used in VLSI testing.

An input of a gate g has an *input-controlling value* of g if this value determines the output value of g regardless of the other inputs. The inverse of the input-controlling value is called the *input-noncontrolling value*. For example, the input-controlling value of an AND gate is 0 and its input-noncontrolling value is 1. A gate g is in the *transitive fanout cone* (TFO) of a gate g_s if there exists a path from g_s to g.

The *dominators* [6] of a gate g are a set of gates G such that all paths from g to any PO have to pass through all gates in G. Consider the dominators of a gate g: the *side inputs* of a dominator are its inputs that are not in the TFO of g.

In VLSI testing, a *stuck-at fault* is a fault model used to represent a manufacturing defect within a circuit. The effect of the fault is as if the faulty wire or gate were stuck at either 1 (stuck-at 1) or 0 (stuck-at 0). A stuck-at fault test is a process to find a test which can generate the different output values in the fault-free and faulty circuits. Given a stuck-at fault f, if there exists such a test, f is said to be testable; otherwise, f is untestable. To make a stuck-at fault on a wire or gate testable, a test needs to activate and propagate the fault effect to a PO. In a combinational circuit, an untestable stuck-at fault on a wire or gate indicates that the wire or gate is redundant and can be replaced with a constant value 0 or 1.

The *mandatory assignments* (MAs) are the unique value assignments to nodes necessary for a test to exist. Consider a stuck-at fault on a gate g; the assignments obtained by setting g to the fault-activating value and by setting the side inputs of dominators of g to the fault-propagating values are MAs. These assignments can be further propagated forward or backward to infer additional MAs by performing logic implications. Computing all MAs of a stuck-at fault requires an exponential time complexity. To compute more MAs with reasonable CPU time overhead, a recursive learning technique [9] with the recursive depth 1 can be used to perform logic implications more completely. If the MAs of a stuck-at fault on a gate are inconsistent, the fault is untestable, and therefore, the gate is redundant [13].

3.2 ATPG-based node merging

The work in [5] proposed a node-merging algorithm by using logic implications. It models a node replacement as a misplaced-wire error [1]. When the error is undetectable, the replacement is safe and correct. Based on the observation, the work proposes a sufficient condition, as presented in Condition 1, that renders a misplaced-wire error undetectable.

Condition 1 [5]*: Let f denote an error of replacing n_t with n_s. If $n_s = 1$ and $n_s = 0$ are MAs for the stuck-at* 0 *and stuck-at* 1 *fault tests on n_t, respectively, f is undetectable.*

The condition works because when it is held, no input pattern that can detect the error of replacing n_t with n_s exists. As a result, n_t can be correctly replaced with n_s.

Based on Condition 1, the proposed algorithm in [5] requires only two MA computations to identify the substitute nodes of a target node n_t: one is for computing the MAs of the stuck-at 0 fault on n_t and the other one is for computing the MAs of the stuck-at 1 fault on n_t.

We use the above example in Fig. 1(a) to demonstrate the algorithm. Suppose n_5 is a target node. The MAs of the stuck-at 0 fault on n_5 are $\{n_5 = 1, n_1 = 1, \boldsymbol{n_2 = 1}, b = 1, d = 1, a = 0, c = 1, \boldsymbol{n_6 = 1}, n_3 = 1, n_4 = 1, \boldsymbol{n_7 = 1}\}$. These values can be computed by setting $n_5 = 1$ to activate the fault effect, setting $n_1 = 1$ to propagate the fault effect, and by performing logic implications to derive additional MAs. In addition, the MAs of the stuck-at 1 fault on n_5 are $\{n_5 = 0, n_1 = 1, a = 0, c = 1, \boldsymbol{n_2 = 0}, \boldsymbol{n_6 = 0}, \boldsymbol{n_7 = 0}\}$. As a result, both n_2 and n_6 are the substitute nodes of n_5 due to the satisfaction of Condition 1. Note that although n_7 also satisfies Condition 1, it is excluded from being a substitute node of n_5. This is because n_7 is in the TFO of n_5, and replacing n_5 with n_7 will result in a cyclic combinational circuit.

3.3 Notation

For convenience and concision, we use the notations in Table 1 to represent certain objects throughout the paper.

4. NODE ADDITION AND REMOVAL

In this section, we first discuss the relationship between the node-merging and the NAR techniques. Based on this relationship, we derive two sufficient conditions for correctly replacing one node with an added node. Finally, an NAR algorithm is presented.

Table 1: Notations used.

Notation	Description
n_t	a target node
n_s	a substitute node of n_t
n_a	an added substitute node of n_t
n_{f1}	one fanin node of n_a
n_{f2}	the other fanin node of n_a different from n_{f1}
T	the set of input patterns that can detect the stuck-at 1 fault on n_t
$T_{n_{f1}=0}$	the set of input patterns in T that generate $n_{f1}=0$
$T_{n_{f1}=1}$	the set of input patterns in T that generate $n_{f1}=1$
$imp(A)$	the set of value assignments logically implied from a set of value assignments A
$MAs(n=sav)$	the set of MAs for the stuck-at v (v is a logical value 0 or 1) fault test on a node n

4.1 Sufficient conditions for NAR

As mentioned in Section 2, the node-merging technique replaces a target node with an existing node while the NAR technique uses an added one. These two techniques have one thing in common – performing node replacement. Thus, when a node is added into the circuit, we can exploit Condition 1 to check if it is an added substitute node. For example, in Fig. 1(c), n_6 is a target node and n_8 is a node added into the circuit. We find that $n_8=1$ and $n_8=0$ are MAs for the stuck-at 0 and stuck-at 1 fault tests on n_6, respectively. Thus, we can conclude that n_8 is an added substitute node for n_6.

Although we can use Condition 1 to check if an added node is a substitute node, it is not efficient to add all possible nodes into the circuit first and then identify which are substitute nodes for the target node. Thus, we transform the problem of finding an added substitute node into finding its two fanin nodes. It is more appropriate to find two nodes that are originally in the circuit.

Our objective now becomes finding two nodes such that the added node driven by them will satisfy Condition 1. For convenience, let n_t denote a target node and n_a denote an added node driven by two nodes n_{f1} and n_{f2}. For ease of discussion, we first suppose that n_a is directly driven by n_{f1} and n_{f2} without any INV in between them. That is, the functionality of n_a is $n_{f1} \wedge n_{f2}$. Next, we present two sufficient conditions for such n_a. Finally, we also extend the sufficient conditions for all eight different types of added nodes. The first condition is presented in Condition 2.

Condition 2: If both $n_{f1}=1$ and $n_{f2}=1$ are MAs for the stuck-at 0 fault test on n_t, $n_a=1$ is an MA for the same test as well.

Because n_a equals $n_{f1} \wedge n_{f2}$, $\{n_{f1}=1, n_{f2}=1\}$ implies $n_a=1$. Thus, if both $n_{f1}=1$ and $n_{f2}=1$ are MAs, $n_a=1$ must be an MA as well by logic implication.

In fact, when Condition 2 is held, n_a satisfies one half of Condition 1 that $n_a=1$ is an MA for the stuck-at 0 fault test on n_t. If we can further show that $n_a=0$ is an MA for the stuck-at 1 fault test on n_t, we can conclude that n_a is an added substitute node of n_t. Thus, the next sufficient condition as presented in Condition 3 is proposed to make n_a satisfy the other half of Condition 1. Here, let $imp(A)$ denote the set of value assignments logically implied from a set of value assignments A, and $MAs(n_t=sav)$ denote the set of MAs for the stuck-at v fault test on n_t, where v is a logical value 0 or 1.

Condition 3: If $n_{f2}=0$ is a value assignment in $imp((n_{f1}=1) \cup MAs(n_t=sa1))$, $n_a=0$ is an MA for the stuck-at 1 fault test on n_t.

To determine whether $n_a=0$ is an MA for the stuck-at 1 fault test on n_t, we can check if all input patterns that can detect the fault generate $n_a=0$. If so, $n_a=0$ is an MA. Let T denote the set of input patterns that can detect the stuck-at 1 fault on n_t. According to the value of n_{f1}, we can classify T into two subsets: The first one, $T_{n_{f1}=0}$, and the second one, $T_{n_{f1}=1}$, which consist of the patterns generating $n_{f1}=0$ and $n_{f1}=1$, respectively. Because $n_{f1}=0$ implies $n_a=0$, all patterns in $T_{n_{f1}=0}$ generate $n_a=0$. As for $T_{n_{f1}=1}$, $imp((n_{f1}=1) \cup MAs(n_t=sa1))$ is also the set of unique value assignments that all patterns in $T_{n_{f1}=1}$ generate. If $n_{f2}=0$ is a value assignment in $imp((n_{f1}=1) \cup MAs(n_t=sa1))$, all patterns in $T_{n_{f1}=1}$ must generate $n_{f2}=0$, implying that $n_a=0$ as well. As a result, when Condition 3 is held, each pattern in T generates $n_a=0$. Hence, $n_a=0$ is an MA for the stuck-at 1 fault test on n_t.

In summary, when Conditions 2 and 3 are held simultaneously, $n_a=1$ and $n_a=0$ are MAs for the stuck-at 0 and stuck-at 1 fault tests on n_t, respectively, and n_a is an added substitute node of n_t.

Note that none of n_{f1} and n_{f2} represents a particular fanin node of n_a. When one fanin node of n_a is determined as n_{f1}, the other fanin node is n_{f2}. Thus, although $n_{f1}=0 \in imp((n_{f2}=1) \cup MAs(n_t=sa1))$ is also a sufficient condition for $n_a=0$ to be an MA for the stuck-at 1 fault test on n_t, we do not state it in Condition 3. We can ignore it by always selecting the node having a value 1 as n_{f1}.

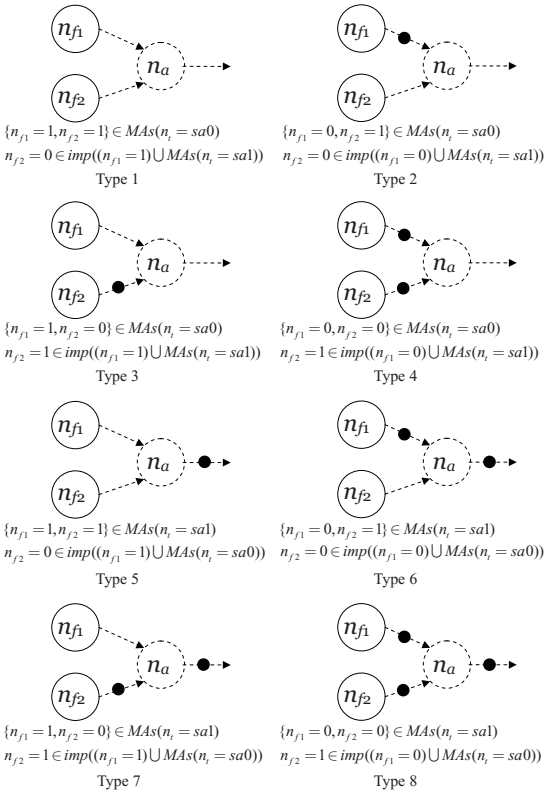


Figure 2: Eight different types of added substitute nodes and their corresponding sufficient conditions.

4.2 Types of added substitute nodes

In the last subsection, we suppose that an added node is directly driven by two nodes without any INV in between them, and then derive Conditions 2 and 3. In fact, these conditions can be modified by reversing the values of n_{f1}, n_{f2}, or the stuck-at fault for different types of added substitute nodes. We present eight types of added substitute nodes and their corresponding sufficient conditions in Fig. 2.

For example, Type 1 is the original added node we consider before. By reversing the value of n_{f1} in Conditions 2 and 3, we have Type 2, n_a equals $\neg n_{f1} \wedge n_{f2}$. Similarly, if we reverse the value of n_{f2}, we have Type 3, n_a equals $n_{f1} \wedge \neg n_{f2}$. For Type 4, n_a equals $\neg n_{f1} \wedge \neg n_{f2}$, we can reverse the values of n_{f1} and n_{f2} simultaneously. For Types 5 $\sim$ 8, they are corresponding to Types 1 $\sim$ 4, respectively. We can reverse the stuck-at fault values in Types 1 $\sim$ 4 to obtain Types 5 $\sim$ 8.

Find_Added_Substitute_Node(Node n_t)
1. Compute $MAs(n_t = sa0)$.
2. Compute $MAs(n_t = sa1)$.
3. For each MA $n = v$ in $MAs(n_t = sa0)$
   (a) Let n be n_{f1}.
   (b) Compute $imp((n_{f1} = v) \cup MAs(n_t = sa1))$.
   (c) The n_{f2} set $\longleftarrow$ Nodes that have different values in MAs($n_t = sa0$) and $imp((n_{f1} = v) \cup MAs(n_t = sa1))$.
4. The n_a set of Types $1 \sim 4$ $\longleftarrow$ Nodes driven by n_{f1} and n_{f2}.
5. For each MA $n = v$ in $MAs(n_t = sa1)$
   (a) Let n be n_{f1}.
   (b) Compute $imp((n_{f1} = v) \cup MAs(n_t = sa0))$.
   (c) The n_{f2} set $\longleftarrow$ Nodes that have different values in. MAs($n_t = sa1$) and $imp((n_{f1} = v) \cup MAs(n_t = sa0))$.
6. The n_a set of Types $5 \sim 8$ $\longleftarrow$ Nodes driven by n_{f1} and n_{f2}.

Figure 3: The algorithm for finding added substitute nodes.

In this work, the proposed algorithm will consider all these possible added substitute nodes when performing NAR.

4.3 NAR algorithm

Given a target node n_t, we can exploit Conditions 2 and 3 to find its added substitute nodes. Based on Condition 2, we always select an MA in $MAs(n_t = sav)$ as a candidate n_{f1}, and then use the n_{f1} and Condition 3 to find n_{f2}. The proposed algorithm is shown in Fig. 3. In the first two steps, the algorithm computes $MAs(n_t = sa0)$ and $MAs(n_t = sa1)$, respectively. In step 3, the algorithm starts to find the added substitute nodes of Types $1 \sim 4$. It iteratively selects an MA $n = v$ from MAs($n_t = sa0$) and sets n_{f1} to n. Then, it computes $imp((n_{f1} = v) \cup MAs(n_t = sa1))$ by performing logic implications of $n_{f1} = v$ associated with $MAs(n_t = sa1)$. Finally, the nodes that have different values in MAs($n_t = sa0$) and $imp((n_{f1} = v) \cup MAs(n_t = sa1))$ can be n_{f2}. Thus, in step 4, the nodes that driven by n_{f1} and n_{f2} are the added substitute nodes of Types $1 \sim 4$. In steps 5 and 6, the algorithm uses a similar method to find the added substitute nodes of Types $5 \sim 8$.

Note that the algorithm in Fig. 3 is designed to find all added substitute nodes. If the objective is to identify one added substitute node or check if a target node is replaceable, we can terminate the algorithm once it finds an n_{f1} and n_{f2} pair. Additionally, we will ensure that an added substitute node is not in the TFO of the target node and has at least one different fanin node from that of the target node.

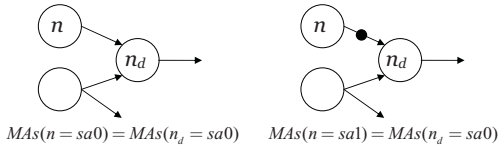
We use the example in Fig. 1 to demonstrate the algorithm. Let us consider finding an added substitute node of n_6 in the circuit of Fig. 1(b). First, we compute the MAs for the stuck-at 0 fault on n_6. To activate the fault effect, n_6 is set to 1. We then perform logic implications to derive additional MAs. They are $n_2 = 1$, $c = 1$, $b = 1$, $d = 1$, $n_3 = 1$, and $n_4 = 1$. Thus, $MAs(n_t = sa0)$ includes $\{n_6 = 1, \boldsymbol{n_2 = 1}, c = 1, b = 1, \boldsymbol{d = 1}, n_3 = 1, \boldsymbol{n_4 = 1}\}$. Second, we use the same method to compute the MAs for the stuck-at 1 fault on n_6. They are $\{n_6 = 0, n_7 = 0\}$. Third, suppose we select n_3 as n_{f1} and compute $imp((n_3 = 1) \cup MAs(n_6 = sa1))$. The implication results have $\{n_6 = 0, n_7 = 0, n_3 = 1, b = 1, c = 1, \boldsymbol{n_2 = 0}, \boldsymbol{d = 0}, \boldsymbol{n_4 = 0}\}$. Finally, n_2, d, and n_4 all can be n_{f2} due to the satisfaction of Conditions 2 and 3. If we select n_4 as n_{f2}, n_8 driven by n_3 and n_4 is an added substitute node of n_6 as shown in Fig. 1(c).

5. CIRCUIT SIZE REDUCTION

In this section, we present an NAR-based algorithm for circuit size reduction. The node-merging technique [5] is also included in the algorithm to quickly replace a node having a substitute node. In addition, two techniques, redundancy removal and MA reuse, are engaged to enhance the performance of the algorithm.

5.1 Node merging

Let us review the proposed NAR algorithm as shown in Fig. 3. The algorithm computes $MAs(n_t = sa0)$ and $MAs(n_t = sa1)$ in the first two steps. As mentioned in Section 3.2, the ATPG-based node-merging approach [5] only requires $MAs(n_t = sa0)$ and $MAs(n_t = sa1)$ to find substitute nodes. Thus, we can combine the node-merging approach with the NAR algorithm for circuit size reduction. Given a target node, after computing $MAs(n_t = sa0)$ and $MAs(n_t = sa1)$, we use the node-merging approach to find its substitute nodes for replacement. If there is no substitute node, we continue to find its added substitute nodes. This approach saves the effort of finding an added substitute node when there is a substitute node.



(a) (b)

Figure 4: The rules for MA reuse.

5.2 Redundancy removal

As mentioned in Section 3.1, MAs are the unique value assignments to nodes necessary for a test to exist. Given a stuck-at fault on a node, when the MAs are inconsistent, the fault is untestable and the node is redundant. The NAR algorithm computes $MAs(n_t = sa0)$ and $MAs(n_t = sa1)$, and hence can simultaneously find untestable faults. Once we find the assignments in $MAs(n_t = sa0)$ are inconsistent, we replace n_t with a constant value 0 and use 0 to drive all the wires originally driven by n_t. Similarly, if the assignments in $MAs(n_t = sa1)$ are inconsistent, we replace n_t with a constant value 1. Thus, for circuit size reduction, we can identify these redundancies and remove them without extra effort.

5.3 MA reuse

MA reuse is a method to reuse the computed MAs such that the number of performed logic implications can be reduced and the optimization process is accelerated. The idea comes from the concept of fault collapsing [2] that two equivalent stuck-at faults have the same test set. Based on this concept, when two stuck-at faults are equivalent, their corresponding MAs are identical as well. Thus, we can reuse the computed MAs when optimizing a circuit. Here, we simply derive two rules for MA reuse as shown in Fig. 4.

Suppose n drives only n_d and $MAs(n_d = sa0)$ has been computed. Let us consider computing the MAs for the stuck-at fault tests on a node n with MA reuse. As shown in Fig. 4(a), if there exists no INV between n and n_d, we can directly set $MAs(n = sa0)$ to $MAs(n_d = sa0)$ rather than re-compute the same MA set. Otherwise, if there exists an INV between n and n_d as shown in Fig. 4(b), we can directly set $MAs(n = sa1)$ to $MAs(n_d = sa0)$.

In summary, for each node n_d, only $MAs(n_d = sa0)$ can be reused. Additionally, it is reused when n_d has a fanin node n which drives only n_d.

5.4 Overall algorithm

During the optimization process, each node in a circuit is considered a target node, one at a time. We first find the target node's substitute nodes for replacement using the node-merging technique [5]. However, if there is no substitute node, we then consider performing NAR. In order to ensure that each node replacement can reduce the circuit size, we only perform NAR for the target nodes that have a fanin node driving only one node. In this situation, when the target node is replaced, the fanin node can be removed as well. Thus, adding one node removes at least two nodes.

As for the optimization order, although the orders of selecting a target node, a substitute node, and an added substitute node can significantly affect the optimization results, it is difficult to evaluate the most effective optimization order. Additionally, this evaluation process might be time-consuming or fruitless. Thus, in this work, we follow the optimization order of selecting a target node and a substitute node used by the node-merging algorithm in [5] for fair comparison. A target node is selected from POs to PIs in the depth-first search (DFS) order

Circuit_Size_Reduction(Circuit C)
For each node n_t in C in the DFS order from POs to PIs
1. Compute $MAs(n_t = sa0)$ with MA reuse.
   (a) If the MAs in MAs($n_t = sa0$) are inconsistent, replace n_t with 0, and then **continue**.
   (b) If n_t has a fanin node which drives only n_t, store MAs($n_t = sa0$) for further reuse.
2. Compute $MAs(n_t = sa1)$ with MA reuse.
   (a) If the MAs in MAs($n_t = sa1$) are inconsistent, replace n_t with 1, and then **continue**.
3. $SubstituteNodes \longleftarrow$ nodes having the different values in MAs($n_t = sa0$) and MAs($n_t = sa1$).
4. If $SubstituteNodes \neq \{\}$, replace n_t with a node that is in $SubstituteNodes$ and closest to PIs, and then **continue**.
5. If n_t has no fanin node which drives only n_t, **continue**.
6. For each MA $n = v$ in $MAs(n_t = sa0)$ in a topological order
   (a) Let n be n_{f1}.
   (b) Compute $imp((n_{f1} = v) \cup MAs(n_t = sa1))$.
   (c) The n_{f2} set $\longleftarrow$ Nodes that have different values in MAs($n_t = sa0$) and $imp((n_{f1} = v) \cup MAs(n_t = sa1))$.
   (d) If the set of $n_{f2} \neq \{\}$, replace n_t with a node driven by n_{f1} and the n_{f2} that is closest to PIs, and then **break**.
7. If n_t is replaced, **continue**.
8. For each MA $n = v$ in $MAs(n_t = sa1)$ in a topological order
   (a) Let n be n_{f1}.
   (b) Compute $imp((n_{f1} = v) \cup MAs(n_t = sa0))$.
   (c) The n_{f2} set $\longleftarrow$ Nodes that have different values in MAs($n_t = sa1$) and $imp((n_{f1} = v) \cup MAs(n_t = sa0))$.
   (d) If the set of $n_{f2} \neq \{\}$, replace n_t with a node driven by n_{f1} and the n_{f2} that is closest to PIs, and then **break**.

Figure 5: The overall algorithm for circuit size reduction.

and is replaced with a substitute node that is closest to PIs. Additionally, we replace a target node once we find an added substitute node due to the inefficiency of finding all added substitute nodes. When we search an added substitute node, each MA node is selected as n_{f1} in a topological order to identify the n_{f2} that is closest to PIs.

Fig. 5 shows the overall algorithm for circuit size reduction. Given a circuit C, the algorithm iteratively selects a target node n_t in the DFS order from POs to PIs and replaces it if applicable. At each iteration, in step 1, the algorithm computes MAs($n_t = sa0$). If the MAs are inconsistent, it replaces n_t with 0 and continues to consider the next target node. Otherwise, if n_t has a fanin node that drives only n_t, the algorithm stores the computed $MAs(n_t = sa0)$ for further reuse. Next, in step 2, the algorithm computes $MAs(n_t = sa1)$. Similarly, if the MAs in $MAs(n_t = sa1)$ are inconsistent, it replaces n_t with 1 and continues to consider the next target node. Otherwise, the algorithm starts to find substitute nodes.

In step 3, the nodes that have the different values in $MAs(n_t = sa0)$ and $MAs(n_t = sa1)$ are the substitute nodes of n_t. The algorithm selects one substitute node which is closest to PIs to replace n_t and continues to consider the next target node. However, if n_t has no substitute node, the algorithm starts to perform NAR when n_t has one fanin node which drives only n_t. From steps 6 $\sim$ 8, the algorithm finds an added substitute node to replace n_t by using the method as presented in Fig. 3.

6. EXPERIMENTAL RESULTS

We implemented our algorithm in C language within an ABC [3] environment. For comparison, we also reimplemented the node-merging algorithm in [5]. The experiments were conducted on a 3.0 GHz Linux platform (CentOS 4.6). The benchmarks are from the IWLS 2005 suite [15]. Each benchmark is initially transformed to an AIG format and we only consider its combinational portion. Additionally, to balance quality and efficiency, the recursive learning technique [9] is applied with the recursion depth 1 in the algorithms. The experimental setup and parameters are the same with that in [5] for fair comparison.

The experimental results consist of two parts: The first one shows the logic restructuring capability of our approach combining the node-merging and the NAR techniques. The second one shows the efficiency and effectiveness of our approach for circuit size reduction.

Table 2: The experimental results of finding replaceable nodes by using the node-merging approach [5] and our approach.

benchmark	N	[5]			our approach			impr.
		N_{rep}	%	T(s)	N_{rep}	%	T(s)	%
C3540	**1038**	**29**	**2.8**	**0.3**	**328**	**31.6**	**0.5**	**28.8**
rot	1063	42	4.0	0.2	384	36.1	0.3	32.8
simple_spi	1079	26	2.4	0.1	234	21.7	0.3	19.3
i2c	1306	80	6.1	0.2	528	40.4	0.4	34.3
pci_spoci_ctrl	1451	170	11.7	0.6	630	43.4	1.5	31.7
dalu	1740	217	12.5	1.0	885	50.9	3.1	38.4
C5315	1773	33	1.9	0.2	279	15.7	0.3	13.9
s9234	1958	175	8.9	0.4	827	42.2	0.7	33.3
C7552	2074	60	2.9	0.4	691	33.3	0.7	30.4
C6288	2337	2	0.1	0.5	932	39.9	1.4	39.8
i10	2673	626	23.4	1.4	1493	55.9	2.8	32.4
s13207	2719	159	5.9	0.6	891	32.8	1.7	26.9
systemcdes	3190	147	4.6	1.5	1355	42.5	2.6	37.9
i8	3310	1533	46.3	3.8	2522	76.2	7.2	29.9
spi	4053	65	1.6	3.4	950	23.4	6.6	21.8
des_area	4857	80	1.7	5.6	891	18.3	13.3	16.7
alu4	5270	206	3.9	54.9	2852	54.1	83.6	50.2
s38417	9219	173	1.9	1.4	2319	25.2	2.4	23.3
tv80	9609	496	5.2	17.2	3415	35.5	41.6	30.4
b20	12219	849	7.0	17.3	4424	36.2	34.6	29.3
s38584	12400	549	4.4	17.0	4385	35.4	66.2	30.9
b21	12782	1094	8.6	19.3	5249	41.1	39.5	32.5
systemcaes	13054	202	1.6	17.7	2888	22.1	36.8	20.6
ac97_ctrl	14496	98	0.7	3.2	1428	9.9	7.5	9.2
mem_ctrl	15641	1537	9.8	98.8	3443	22.0	178.0	12.2
usb_funct	15894	370	2.3	6.3	3430	21.6	16.7	19.3
b22	18488	1047	5.7	25.0	6497	35.1	53.8	29.5
aes_core	21513	452	2.1	15.2	8076	37.5	39.9	35.4
pci_bridge32	24369	309	1.3	21.7	3700	15.2	47.2	13.9
wb_conmax	48429	5608	11.6	28.2	13492	27.9	116.0	16.3
b17	52920	1565	3.0	174.5	17473	33.0	533.8	30.1
des_perf	79288	2505	3.2	51.4	34376	43.6	82.7	40.2
average			**6.5**			**34.4**		**27.9**
total				**589.3**			**1423.7**	
ratio			**1**			**5.26**		

6.1 Replaceable node identification

In the experiments, we compare our approach with the node-merging approach [5]. Each node in a benchmark is considered a target node one at a time. We separately use the node-merging approach and our approach to check how many nodes in a benchmark are replaceable. A node is considered replaceable if it has a substitute node or an added substitute node. Given a target node, our approach first finds its substitute nodes. If our approach fails to do so, it further finds the added substitute nodes.

Table 2 summarizes the experimental results. Column 1 lists the benchmarks. Column 2 lists the number of nodes in each benchmark represented by AIG N. Columns 3 to 5 list the results of the node-merging approach. They contain the number of replaceable nodes N_{rep}, the percentage of N_{rep} with respect to N, and the CPU time T, respectively. Columns 6 to 8 list the corresponding results of our approach. The improvements of our approach on the number of replaceable nodes are listed in the last column.

For example, the benchmark $C3540$ has 1038 nodes. The node-merging approach found substitute nodes for 29 out of 1038 nodes, or 2.8%, with a CPU time of 0.3 seconds. Our approach found that 328 nodes, or 31.6%, have substitute nodes or added substitute nodes with a CPU time of 0.5 seconds. Thus, our approach can find 28.8% more replaceable nodes.

According to Table 2, the node-merging approach can find substitute nodes for an average of 6.5% of nodes in a benchmark. The overall CPU time for all benchmarks is 589.3 seconds. As for our approach, it can find substitute nodes or added substitute nodes for an average of 34.4% of nodes in a benchmark. The overall CPU time is 1423.7 seconds.

As compared with the node-merging approach, our approach can find more replaceable nodes with a reasonable CPU time overhead. The average number of replaceable nodes is 27.9% more with a ratio 5.26, and the CPU time overhead is only 834.4 seconds for all benchmarks. Because our approach identifies much more replaceable nodes, it has a better logic restructuring capability than that of the node-merging approach.

Table 3: The experimental results of circuit size reduction by using the approaches in [12] and [5], and our approach.

benchmark	N	[12]		[5]			our approach		
		%	T(s)	N_r	%	T(s)	N_r	%	T(s)
pci_spoci_ctrl	878	9.2	6	782	10.9	0.2	757	13.8	0.4
i2c	941	3.2	3	923	1.9	0.1	894	5.0	0.2
dalu	1057	12.0	10	985	6.8	0.3	979	7.4	0.5
C5315	1310	0.7	2	1304	0.5	0.1	1297	1.0	0.1
s9234	1353	1.2	8	1331	1.6	0.2	1323	2.2	0.2
C7552	1410	3.4	8	1371	2.8	0.3	1356	3.8	0.3
i10	1852	1.3	12	1755	5.2	0.6	1742	5.9	1.0
s13207	2108	1.8	17	2063	2.1	0.5	2043	3.1	0.8
alu4	2471	22.9	64	1941	21.5	5.3	1878	24.0	9.9
systemcdes	2641	4.7	9	2600	1.6	0.9	2580	2.3	1.2
spi	3429	1.3	84	3411	0.5	2.7	3383	1.3	5.6
tv80	7233	7.1	1445	6960	3.8	10.6	6813	5.8	20.3
s38417	8185	1.0	275	8136	0.6	1.2	8105	1.0	1.5
mem_ctrl	8815	18.0	738	7257	17.7	6.8	7287	17.3	13.8
s38584	9990	0.8	223	9846	1.4	11.4	9836	1.5	15.1
ac97_ctrl	10395	2.0	188	10379	0.2	2.0	10364	0.3	3.1
systemcaes	10585	3.8	360	10521	0.6	13.1	10386	1.9	30.7
usb_funct	13320	1.4	681	13026	2.2	5.9	12868	3.4	11.4
pci_bridge32	17814	0.1	1134	17729	0.5	12.0	17599	1.2	19.7
aes_core	20509	8.6	1620	20371	0.7	13.2	20195	1.5	22.7
b17	34523	1.6	5000	33979	1.5	72.4	33204	3.8	205.5
wb_conmax	41070	6.2	5000	39266	4.4	31.9	38880	5.3	48.4
des_perf	71327	3.7	5000	70081	1.8	62.6	69421	2.7	84.7
average		**5.0**			**3.9**			**5.0**	
total			**21887**			**254.3**			**497.1**
ratio		**1.27**	**44.03**		**1**	**0.51**		**1.27**	**1**

6.2 Circuit size reduction

In the experiments, we compare our approach with the ATPG-based node-merging approach [5] as well as the SAT-based node-merging approach [12] for circuit size reduction. To have a fair comparison with the SAT-based node-merging approach, which focuses on post-synthesis optimizations, we initially optimize each benchmark by using the *resyn2* script in the ABC package as performed by [12], which performs local circuit rewriting optimization [10]. Note that although we have the same initialization, the initial number of nodes in each benchmark is still a little different from that reported in [12]. The reason might be that the structures of the original benchmarks are not completely identical.

After the initialization, we separately optimize each benchmark by using our approach as shown in Fig. 5 and the ATPG-based node-merging approach. Finally, we also apply an equivalence checking tool, *cec* [11], in the ABC package to verify the correctness of the optimization.

Table 3 summarizes the experimental results. Columns 1 and 2 list the benchmarks and the number of nodes in each benchmark represented by AIG, respectively. Columns 3 to 4 list the results of the SAT-based node-merging approach reported in [12], the percentage of circuit size reduction in terms of node count and the CPU time, respectively. The maximal CPU time in Column 4 is 5000 seconds, which is the CPU time limit set by the work. Columns 5 to 7 list the results of the ATPG-based node-merging approach. They contain the number of nodes in each resultant benchmark N_r, the percentage of circuit size reduction, and the CPU time, respectively. Columns 8 to 10 list the corresponding results of our approach.

The experimental results in Table 3 show that our approach is 44.03 times faster than the SAT-based node-merging approach and has a competitive capability of circuit size reduction. Additionally, our capability is better than that of the ATPG-based node-merging approach with a ratio of 1.27. The overall CPU time overhead is only 242.8 seconds.

Moreover, because our approach is highly efficient, we can combine it with the *resyn2* script to achieve more circuit size reduction. We optimized the benchmarks listed in Table 3 by repeatedly using our approach followed by the *resyn2* script 3 times – (*ours+resyn2*)x3. The average circuit size reduction is 8.6% and the CPU time is 1453.1 seconds. On the other hand, if we optimize these benchmarks by repeatedly using the *resyn2* script 6 times – *resyn2*x6, the average circuit size reduction is only 4.3% with a CPU time of 157.1 seconds. In addition, the average circuit size reduction is 5.9% and the CPU time is 2691.2 seconds by repeatedly using our approach 6 times – *ours*x6. The experimental results are summarized in Table 4. According to the experimental results, we can conclude that the efficiency and the logic restructuring capability of our approach can make the integration of our approach and other optimization techniques such as *resyn2* possible.

Table 4: The comparison of experimental results among (*ours+resyn2*)x3, *resyn2*x6, and *ours*x6.

	(*ours+resyn2*)x3		*resyn2*x6		*ours*x6	
	%	T(s)	%	T(s)	%	T(s)
average	**8.6**		**4.3**		**5.9**	
total		**1453.1**		**157.1**		**2691.2**

7. CONCLUSION

In this paper, we propose an ATPG-based NAR approach that can efficiently find an added node to replace a node in a circuit. The NAR approach can replace a target node that a node-merging approach cannot handle, thus enhancing the capability of circuit restructuring. The proposed approach is based on two sufficient conditions that state the requirements of added nodes for correctly replacing a target node. It can quickly identify added substitute nodes by using logic implications.

Moreover, we also propose an efficient algorithm for circuit size reduction based on the NAR approach. The techniques of redundancy removal and MA reuse are engaged to make the algorithm more efficient and effective.

The experimental results show that the proposed algorithm enhances an ATPG-based node-merging approach. Additionally, it has a competitive capability of circuit size reduction and expends much less CPU time compared to a SAT-based node-merging approach. The experimental results also show that the proposed algorithm can be integrated with an optimization technique to obtain a better circuit size reduction. All these results show the efficiency and effectiveness of the proposed approach.

8. REFERENCES

[1] M. S. Abadir, J. Ferguson, and T. E. Kirkland, "Logic Design Verification via Test Generation," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 138-148, Jan. 1988.

[2] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Design for Testability*, IEEE Press, 1990.

[3] Berkeley Logic Synthesis and Verification Group, "ABC: A System for Sequential Synthesis and Verification," http://www.eecs.berkeley.edu/~alanmi/abc/.

[4] M. Case, V. Kravets, A. Mishchenko, and R. Brayton, "Merging Nodes Under Sequential Observability," in *Proc. Design Automation Conf.*, 2008, pp. 540-545.

[5] Y. C. Chen and C. Y. Wang, "Fast Detection of Node Mergers Using Logic Implications," in *Proc. Int. Conf. on Computer-Aided Design*, 2009, pp. 785-788.

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

[7] A. Kuehlmann, "Dynamic Transition Relation Simplification for Bounded Propery Checking," in *Proc. Int. Conf. on Computer-Aided Design*, 2004, pp. 50-57.

[8] A. Kuehlmann, V. Paruthi, F Krohm, and M. K. Ganai, "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification," in *IEEE Trans. Computer-Aided Design*, vol. 21, pp. 1377-1394, Dec. 2002.

[9] W. Kunz and D. K. Pradhan, "Recursive Learning: An New Implication Technique for Efficient Solutions to CAD Problems - Test, Verification, and Optimization," in *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1143-1158, Sep. 1994.

[10] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis," in *Proc. Design Automation Conf.*, 2006, pp. 532-536.

[11] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to Combinational Equivalence Checking, " in *Proc. Int. Conf. on Computer-Aided Design*, 2006, pp. 836-843.

[12] S. Plaza, K. H. Chang, I. Markov, and V. Bertacco, "Node Mergers in the Presence of Don't Cares," in *Proc. Asia South Pacific Design Automation Conf.*, 2007, pp. 414-419.

[13] M. H. Schulz amd E. Auth, "Advanced Automatic Test Pattern Generation and Redundancy Identification Techniques," in *Proc. Int. Fault-Tolerant Computing Symp.*, 1988, pp. 30-35.

[14] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT Sweeping with Local Observability Don't Cares," in *Proc. Design Automation Conf.*, 2006, pp. 229-234.

[15] http://iwls.org/iwls2005/benchmarks.html.