Logic Restructuring Using Node Addition and Removal

Yung-Chih Chen and Chun-Yao Wang, *Member, IEEE*

***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 paper. 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 *quickly* detect the added nodes by performing logic implications based on these conditions. We apply the NAR approach to circuit minimization together with two techniques: redundancy removal and mandatory assignment reuse. We also apply it to satisfiability (SAT)-based bounded sequential equivalence checking (BSEC) to reduce the computation complexity of SAT solving. The experimental results show that our approach can enhance our prior automatic test pattern generation-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. For BSEC, our approach can work together with other optimization technique to save a total of approximately 39-h verification time for all the benchmarks.**

***Index Terms*—Logic implication, node addition and removal, node merging, observability don't care.**

I. Introduction

LOGIC restructuring techniques have been widely developed during the last 20 years. Popular methodologies like redundancy addition and removal (RAR) [9]–[13], [17]–[19], [25], [32], [34], [35], irredundancy removal and addition [24], automatic test pattern generation (ATPG)/diagnosis-based design rewiring [31], error cancelation-based rewiring [36], and rewriting [26] have demonstrated their effectiveness on logic synthesis and optimization.

Recently, a novel logic restructuring technique named node merging was proposed and enhanced in [7], [14], [15], [21], [28], and [37]. This methodology works by merging two nodes—replacing one node with another node—in a logic circuit with don't cares. When two nodes are functionally equivalent or their functional differences are never observed at any primary output (PO), they can be correctly merged. 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 the node-merging technique for circuit minimization has been shown in the previous works. The satisfiability (SAT)-based approaches [28], [37] have a great capability of circuit minimization. As reported in [37] and [28], 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], [26], respectively. However, the efficiency is a major concern for these SAT-based approaches due to the expense of observability don't care (ODC) computation and SAT solving calls.

On the contrary, our prior ATPG-based approach [14] is much faster, although its capability of circuit minimization is not as good as that of the SAT-based approaches. The experimental results in [14] show that a large benchmark circuit having more than 70 000 nodes can be optimized in approximately 1 min.

However, these previous works only focus on searching and merging two nodes that originally exist in a circuit. They fail to replace a target node that possesses no substitute 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, when 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) [16]. Because more nodes can be replaced by an added node, NAR can enhance the results of node merging in logic optimization.

In this paper, we extend our prior node-merging technique [14] to consider NAR and propose an efficient approach by 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 (MA) reuse, are engaged to enhance the performance. Redundancy removal detects redundant nodes without extra effort when the ap-

Manuscript received February 21, 2011; revised June 24, 2011; accepted August 14, 2011. Date of current version January 20, 2012. This work was supported in part by the National Science Council of Taiwan, under Grants NSC 99-2628-E-007-096, NSC 99-2220-E-007-003, NSC 100-2628-E-007-031-MY3, and NSC 100-2628-E-007-008. This paper was recommended by Associate Editor S. Nowick.

Y.-C. Chen is with the Department of Electronic Engineering, Chung Yuan Christian University, Taoyuan 32023, Taiwan (e-mail: ycchen@cycu.edu.tw).

C.-Y. Wang is with the Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan (e-mail: wcyao@cs.nthu.edu.tw).

Digital Object Identifier 10.1109/TCAD.2011.2167327

proach identifies substitute nodes. MA 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 [38] and compare to the node-merging approaches in [14], [28], and [37]. For replaceable node identification, as compared to our prior ATPG-based approach [14], an average of 28% more nodes can be identified replaceable in a benchmark circuit by using NAR. Additionally, an average of 72% of replaceable nodes cannot be found by the SAT-based node-merging approach with a bounded depth $k = 5$ [37], and thus, the proposed approach could complement it.

For circuit size reduction, the proposed approach reduces 2873 more nodes than our prior ATPG-based node-merging approach [14] for all the benchmarks, with an overall CPU time overhead of only 4 min. Additionally, the optimization capability is competitive with that of the SAT-based node-merging approach [28], which is highly time-consuming.

Moreover, we apply the proposed method to SAT-based bounded sequential equivalence checking (BSEC) [8], [30], [33]. Our method is used as a preprocess to reduce the computation complexity of SAT solving. Not only the variable count that the SAT solver deals with is minimized due to logic optimization but also the relationships among the variables become tighter by logic restructuring. Thus, the process of SAT solving could be facilitated. The effectiveness of the proposed method is compared with a logic optimization technique *resyn2* [26]. The experimental results show that when we use *resyn2* to facilitate SAT-based BSEC, a total of approximately 25 h are saved for verifying all the benchmarks. However, when we integrate the proposed method with *resyn2*, we can save approximately 39 h.

A related work studying SAT-controlled RAR rather than NAR is presented in [32]. The work proposes a SAT-based method for finding redundancies to replace a target wire. Although the work aims to replace a wire rather than a node, it also introduces a similar sufficient condition for identifying an added node to replace a wire. However, the objective of our paper is different from that of [32]. Additionally, our method is an ATPG-based approach and is more complete, since we consider a total of eight types of added nodes. Moreover, we also propose an efficient NAR-based algorithm for circuit optimization and SAT-based BSEC facilitation.

The remainder of this paper is organized as follows. Section II uses an example to demonstrate the NAR technique and formulates the problem considered in this paper. Section III reviews the related concepts in very large-scale integrated (VLSI) testing and our prior ATPG-based node-merging approach [14]. Section IV presents the proposed algorithm for NAR. The application of NAR for circuit size reduction is introduced in Section V. Section VI shows the application of the NAR method on SAT-based BSEC. Finally, the experimental results and conclusion are presented in Sections VII and VIII.

II. 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 graphs (AIGs) [22], 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, and $n_1 \sim n_8$ 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.

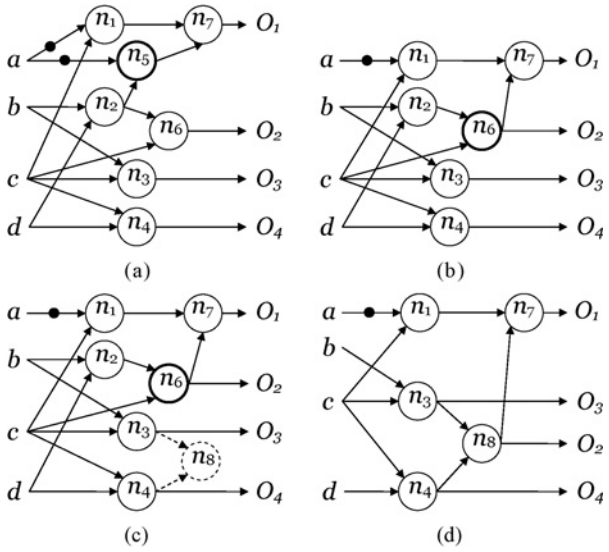

Fig. 1. Example for demonstrating node merging and NAR. (a) Original circuit. (b) Resultant circuit of replacing n_5 with n_6. (c) Resultant circuit of adding n_8. (d) Resultant circuit of replacing n_6 with n_8.

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. Here, because n_2 only drives n_6, when n_6 is replaced, n_2 can be removed as well. This example demonstrates that a node which has no substitute node still can be replaced by a newly added node, and the resultant circuit can be minimized if the replaced node has one or more single-fanout fanin (SFoFi) nodes. Thus, the NAR technique can replace a node which cannot be replaced by the node-merging technique, and can optimize a circuit as well. Note that although n_6 and n_8 are functionally equivalent ($n_6 = (b * d) * c = (b * c) * (d * c) = n_8$)

in this example, it does not indicate that the proposed NAR technique can only identify a functionally equivalent added node. In fact, the proposed NAR technique can also find a functionally different added node to replace a target node.

The problem formulation of this paper 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 it from a substitute node because n_a is absent in the original circuit.

III. Preliminaries

A. Background

This section 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* [20] 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 detect a stuck-at fault on a wire or gate, 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 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 [23] 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 [29].

B. ATPG-Based Node Merging

Our previous work in [14] presented 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, this paper proposes a sufficient condition, as presented in Condition 1, that renders a misplaced-wire error undetectable.

Condition 1 [14]*:* 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.

When Condition 1 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 [14] 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 performing logic implications to derive additional MAs. On the other hand, 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.

TABLE I
NOTATIONS USED

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

C. Notation

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

IV. Node Addition and Removal

In this section, we first derive two sufficient conditions for correctly replacing one node with an added node. Next, we present a method for finding added substitute nodes based on these conditions.

A. Sufficient Conditions for NAR

Because an NAR technique performs node replacement as the node-merging technique, we can exploit Condition 1 to check if an added node 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 n_8 satisfies Condition 1 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.

However, it is not efficient to add all possible nodes into the circuit first and then exploit Condition 1 to 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 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 is $n_{f1} \wedge n_{f2}$, $\{n_{f1} = 1, n_{f2} = 1\}$ 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.

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. Thus, 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. Based on this idea, 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 $\mathrm{MAs}(n_t = sav)$ 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 $imp((n_{f1} = 1) \cup \mathrm{MAs}(n_t = sa1))$, $n_a = 0$ 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 the 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. Based on the value of n_{f1}, we 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 $T_{n_{f1}=1}$, because $imp((n_{f1} = 1) \cup \mathrm{MAs}(n_t = sa1))$ is the set of unique value assignments that all patterns in $T_{n_{f1}=1}$ generate, if $n_{f2} = 0$ is a value assignment in $imp((n_{f1} = 1) \cup \mathrm{MAs}(n_t = sa1))$, all patterns in $T_{n_{f1}=1}$ must generate $n_{f2} = 0$, which implies $n_a = 0$. As a result, when Condition 3 is held, each pattern in T generates $n_a = 0$, and $n_a = 0$ is an MA for the stuck-at 1 fault test on n_t.

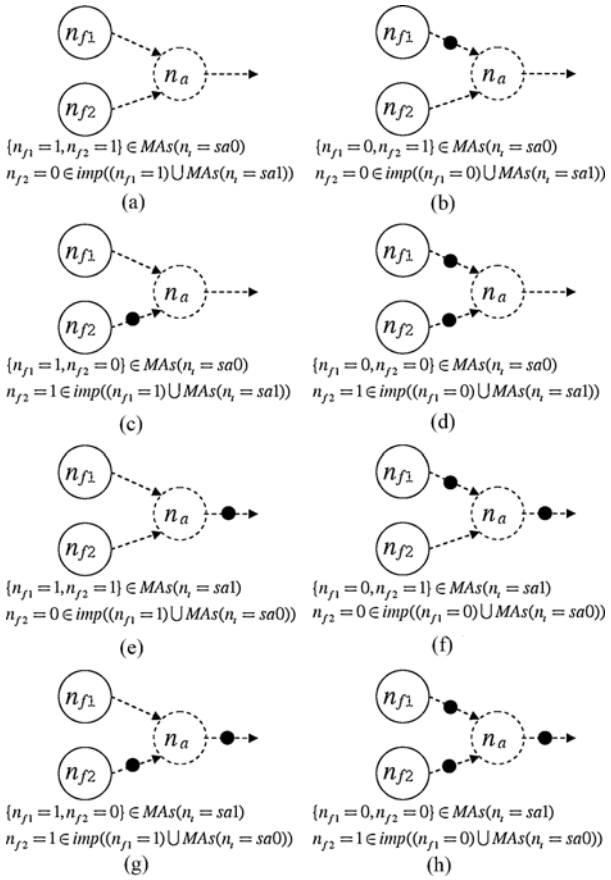

Fig. 2. Eight different types of added substitute nodes and their corresponding sufficient conditions. (a) Type 1. (b) Type 2. (c) Type 3. (d) Type 4. (e) Type 5. (f) Type 6. (g) Type 7. (h) Type 8.

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 $n_{f1} = 0 \in imp((n_{f2} = 1) \cup \mathrm{MAs}(n_t = sa1))$ 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 ignore it by always selecting the node having a value 1 as n_{f1}.

B. Types of Added Substitute Nodes

In the last section, 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,

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}.

Fig. 3. Algorithm for finding added substitute nodes.

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 reverse the values of n_{f1} and n_{f2} simultaneously. For Types 5–8, they are corresponding to Types 1–4, respectively. We reverse the stuck-at fault values in Types 1–4 to obtain Types 5–8.

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

C. NAR Algorithm

Given a target node n_t, we exploit Conditions 2 and 3 to find its added substitute nodes. Based on Condition 2, we always select a 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 1–4. It iteratively selects an MA $n = v$ from MAs($n_t = sa0$) and sets n_{f1} to n. Then, it computes $imp((n_{f1} = v) \cup \text{MAs}(n_t = sa1))$ 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 $imp((n_{f1} = v) \cup \text{MAs}(n_t = sa1))$ 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 1–4. In steps 5 and 6, the algorithm uses a similar method to find the added substitute nodes of Types 5–8.

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 $imp((n_3 = 1) \cup \text{MAs}(n_6 = sa1))$. 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).

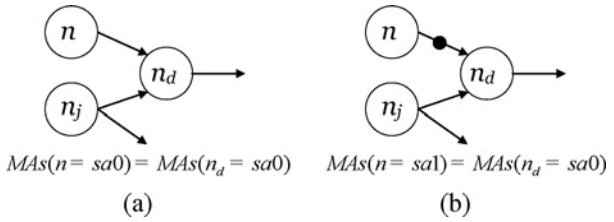

Fig. 4. Rules for MA reuse. (a) MAs($n = sa0$) = MAs($n_d = sa0$). (b) MAs($n = sa1$) = MAs($n_d = sa0$).

V. Circuit Size Reduction

In this section, we present an NAR-based algorithm for circuit size reduction. Our prior node-merging technique [14] 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.

A. Node Merging

As mentioned in Section III-B, our prior ATPG-based node-merging approach [14] only requires MAs($n_t = sa0$) and MAs($n_t = sa1$) to find substitute nodes. Because the proposed NAR algorithm also computes MAs($n_t = sa0$) and MAs($n_t = sa1$) as shown in Fig. 3, 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 method saves the effort of finding an added substitute node when there is a substitute node.

B. Redundancy Removal

As mentioned in Section III-A, 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.

C. MA Reuse

MA reuse is a method to reuse the computed MAs such that the number of performed logic implications can be reduced

Calculate_MAs(Node n, **LogicalValue** v)
IF n drives only one node n_d THEN
IF v == **Compl**(n_d, n) THEN
MAs$(n = sav)$ = MAs$(n_d = sa0)$;
ELSE
Compute MAs$(n = sav)$;
ELSE
Compute MAs$(n = sav)$;

Fig. 5. Algorithm for computing MAs with MA reuse.

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, only one MA computation is required for them. Here, we simply derive two rules for MA reuse as shown in Fig. 4.

First, consider computing MAs(n_d = $sa0$) in Fig. 4(a). To activate the fault effect, n_d is set to 1. To propagate the fault effect, the side inputs of dominators of n_d are set to their corresponding input-noncontrolling values. For simplicity, we use P to denote these fault-propagating assignments. Then, MAs(n_d = $sa0$) can be obtained by performing a logic implication of {n_d = 1, P}. They are {n_d = 1, n = 1, n_j = 1, $imp(P)$}. Next, consider computing MAs(n = $sa0$). n = 1 is the fault-activating assignment. Because n drives only n_d, the dominators of n_d and n_d itself are dominators of n. Thus, the fault-propagating assignments are {n_j = 1, P}. MAs(n = $sa0$) then can be obtained by performing a logic implication of {n = 1, n_j = 1, P}. They are {n_d = 1, n = 1, n_j = 1, $imp(P)$}, which are identical to MAs(n_d = $sa0$). Thus, when we compute MAs(n = $sa0$), we can reuse MAs(n_d = $sa0$). Based on the same method, we also find that MAs(n = $sa1$) equals to MAs(n_d = $sa0$) in Fig. 4(b).

According to these two rules, for each node n_d, only MAs(n_d = $sa0$) could be reused. Additionally, it is reused when n_d has a fanin node n which drives only n_d.

Fig. 5 shows the algorithm for computing the MAs of the stuck-at v fault on a node n with MA reuse. If n drives more than one node, the algorithm directly computes MAs(n = sav). On the other hand, if n drives only one node n_d, the algorithm then checks whether v equals **Compl**(n_d, n). Here, **Compl**(n_d, n) returns 1 if there is an INV between n and n_d; otherwise, it returns 0. When v does not equal **Compl**(n_d, n), the algorithm computes MAs(n = sav) as well. However, if v equals **Compl**(n_d, n), the algorithm reuses MAs(n_d = $sa0$) and MAs(n = sav) equals MAs(n_d = $sa0$).

D. 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 [14]. 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.

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 $SubstituteNodes \neq \varnothing$, 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 $n_{f2} \neq \varnothing$, 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 $n_{f2} \neq \varnothing$, replace n_t with a node driven by n_{f1} and the n_{f2} that is closest to PIs, and then **break**.

Fig. 6. Overall algorithm for circuit size reduction.

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 paper, we follow the optimization order of selecting a target node and a substitute node used by the node-merging algorithm in [14] for fair comparison. A target node is selected from POs to PIs in the depth-first search (DFS) order 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. 6 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.

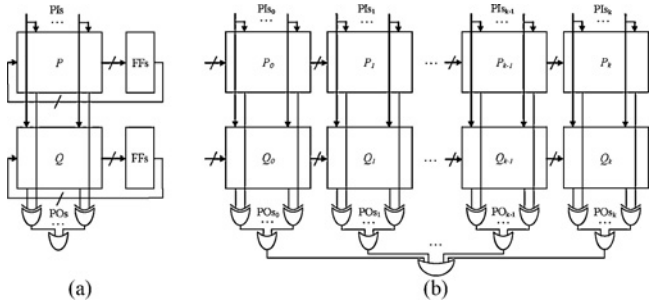

Fig. 7. (a) Miter. (b) BSEC model.

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. In steps 6–8, the algorithm finds an added substitute node to replace n_t by using the method presented in Fig. 3.

VI. SAT-Based BSEC Facilitation

SAT-based BSEC is a special case of bounded equivalence checking [4]–[6]. As shown in Fig. 7(a), two sequential circuits, P and Q, to be checked are first constructed as a miter by connecting their corresponding POs with additional XOR gates and connecting these XOR gates to an OR gate. Then, the miter is unfolded to a bounded depth k of timeframes, and all the inserted OR gates of each timeframe are connected to an additional OR gate as shown in Fig. 7(b). Finally, the output value of the OR gate determines the equivalence of P and Q within the bounded depth k. If the value is 1, it means P and Q are nonequivalent. This is because there exists at least one input pattern that produces different output values on one pair of the POs of P and Q within k timeframes.

To determine whether or not P and Q are equivalent within k timeframes, the BSEC model can be transformed into a conjunctive normal form (CNF), and then be solved by using a SAT solver. When the answer is unsatisfiable, the output of the OR gate is a constant 0, and P and Q are equivalent; otherwise, they are nonequivalent.

However, as the bounded depth increases, the variable count and the size of the constructed CNF grow as well, making the SAT solving problem become computationally inefficient. To reduce the complexity, performing logic optimization before SAT solving is one of the effective methods. Additionally, modifying the variable relationships by logic restructuring could be a possible solution to speedup SAT solving, especially when the answer is unsatisfiable. This is because an unsatisfiable result means there exists at least one variable conflict in the CNF. How fast a SAT solver can detect a conflict determines the required solving time. Thus, if we can make it be detected more easily by changing variable relationships, the SAT solving can be facilitated.

We use the example in Fig. 1 to demonstrate our motivation. Suppose the circuit in Fig. 1(b) is a part of a BSEC model,

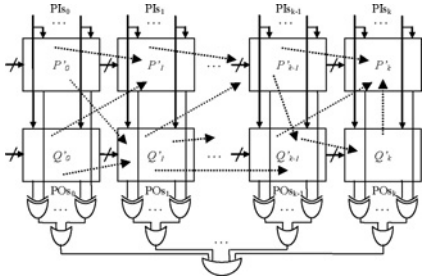

Fig. 8. Optimized BSEC model.

and $n_7 = 1$ and $n_4 = 0$ are the sources that cause a conflict in it. To detect this conflict, a SAT solver may need to learn that $n_7 = 1$ implies $n_6 = 1$, $n_6 = 1$ implies $c = 1$ and $n_2 = 1$, $n_2 = 1$ implies $d = 1$, and then $c = 1$ and $d = 1$ imply $n_4 = 1$. However, if we add n_8 into the circuit and use it to replace n_6 as shown in Fig. 1(d), then the SAT solver only needs to learn that $n_7 = 1$ implies $n_8 = 1$, and $n_8 = 1$ implies $n_4 = 1$, to detect this conflict. As a result, the SAT solving process is accelerated.

In a BSEC model, the conflict sources may come from different circuits under equivalence checking and distribute at different timeframes. The NAR and node-merging techniques could make their relationships tighter by adding a new node driven by two nodes locating at different timeframes, and merging two nodes locating at different timeframes as well. For example, Fig. 8 demonstrates our intent. The dotted lines can be considered the new relationships constructed by node merging and NAR, which further tighten the variable relationships as compared to the original BSEC model. Additionally, the BSEC model size is minimized. Thus, the computation complexity of SAT solving could be reduced.

The proposed optimization flow for a BSEC model is as follows. After constructing a miter for two circuits under equivalence checking, we first optimize its combinational portion. Next, we unfold the simplified miter and construct a BSEC model. Again, we optimize the BSEC model. Here, the (added) substitute node that is farther from the target node in terms of the number of timeframes has a higher priority to be selected to replace the target node. Finally, the simplified BSEC model is transformed into a CNF and solved by a SAT solver.

VII. Experimental Results

We implemented our algorithm in C language within an ABC [3] environment. The experiments were conducted on a 3.0 GHz Linux platform (CentOS 4.6). The benchmarks are from the IWLS 2005 suite [38]. Each benchmark is initially transformed to an AIG and we only consider its combinational portion. Additionally, to balance quality and efficiency, the recursive learning technique [23] is applied with the recursion depth 1 in the algorithms. The experimental setup and parameters are the same with that in [14] for fair comparison.

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

TABLE II
EXPERIMENTAL RESULTS OF FINDING REPLACEABLE NODES BY USING THE NODE-MERGING APPROACH [14] AND OUR APPROACH

Benchmark	N	[14]		Our Approach				
		Repl.	T (s)	Repl.	T (s)	Impr.	$k > 5$	SFoFi
C3540	**1038**	**2.8**	**0.3**	**31.6**	**0.5**	**28.8**	**96.3**	**13.9**
rot	1063	4.0	0.2	36.1	0.3	32.1	83.6	29.9
simple_spi	1079	2.4	0.1	21.7	0.3	19.3	74.8	15.4
i2c	1306	6.1	0.2	40.4	0.5	34.3	65.9	29.0
pci_spoci_ctrl	1451	11.7	0.6	43.4	1.5	31.7	59.7	31.9
dalu	1740	12.5	1.0	50.9	3.1	38.4	60.2	25.5
C5315	1773	1.9	0.2	15.7	0.3	13.8	66.7	16.7
s9234	1958	8.9	0.4	42.2	0.7	33.3	82.7	16.1
C7552	2074	2.9	0.4	33.3	0.7	30.4	71.3	15.8
C6288	2337	0.1	0.5	39.9	1.4	39.8	99.8	0.0
i10	2673	23.4	1.4	55.9	2.8	32.5	55.9	43.0
s13207	2719	5.8	0.6	32.8	1.7	27.0	78.0	19.6
systemcdes	3190	4.6	1.5	42.5	2.6	37.9	75.1	21.3
i8	3310	46.3	3.8	76.2	7.2	29.9	37.5	57.9
spi	4053	1.6	3.4	23.4	6.6	21.8	88.4	11.2
des_area	4857	1.6	5.6	18.3	13.3	16.7	87.1	17.3
alu4	5270	3.9	54.9	54.1	83.6	50.2	80.1	61.2
s38417	9219	1.9	1.5	25.2	2.4	23.3	84.7	12.4
tv80	9609	5.2	17.2	35.5	41.6	30.3	75.5	15.9
b20	12 219	6.9	17.3	36.2	34.6	29.3	59.4	11.3
s38584	12 400	4.4	17.0	35.4	66.2	31.0	87.4	13.1
b21	12 782	8.6	19.3	41.1	39.5	32.5	54.8	10.6
systemcaes	13 054	1.5	17.7	22.1	36.8	20.6	96.7	8.5
ac97_ctrl	14 496	0.7	3.2	9.9	7.5	9.2	85.2	7.8
mem_ctrl	15 641	9.8	98.8	22.0	178.0	12.2	22.9	22.9
usb_funct	15 894	2.3	6.3	21.6	16.7	19.3	62.6	14.0
b22	18 488	5.7	25.0	35.1	53.8	29.4	62.6	10.4
aes_core	21 513	2.1	15.2	37.5	39.9	35.4	90.6	9.2
pci_bridge32	24 369	1.3	21.7	15.2	47.2	13.9	86.0	8.2
wb_conmax	48 429	11.6	28.2	27.9	116.0	16.3	37.5	33.3
b17	52 920	3.0	174.5	33.0	533.8	30.0	48.7	9.6
des_perf	79 288	3.2	51.4	43.4	82.7	40.2	86.3	20.2
Average		**6.5**		**34.4**		**27.8**	**72.0**	**19.8**
Total			**589.3**		**1423.7**			
Ratio		**1**		**5.26**				

one shows the effectiveness of the proposed approach on facilitating SAT-based BSEC.

A. Replaceable Node Identification

In the experiments, we compare the proposed approach with our prior node-merging approach [14]. Each node in a benchmark is considered a target node one at a time. We separately use the node-merging approach and the proposed 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, the proposed approach first finds its substitute nodes. If the proposed approach fails to do so, it further finds the added substitute nodes. Additionally, to demonstrate that the proposed approach could complement the local ODC-based node-merging approach [37], we measure how many replaceable nodes that the local ODC-based node-merging approach with a bounded depth $k = 5$ cannot find.

Table II summarizes the experimental results. Column 1 lists the benchmarks. Column 2 lists the number of nodes in each benchmark represented as an AIG N. Columns 3 and 4 list the results of our prior node-merging approach. They are the percentage of the number of replaceable nodes with respect to N, and the CPU time T, respectively. Columns 5 and 6 list the corresponding results of the proposed approach. Column 7 shows the improvements of the proposed approach on the percentage of replaceable nodes. Let $N_{\text{rep\_}k>5}$ denote the number of replaceable nodes that cannot be found by the reimplemented local ODC-based node-merging approach with a bounded depth $k = 5$. Column 8 lists the percentage of $N_{\text{rep\_}k>5}$ with respect to the number of replaceable nodes. Additionally, let $N_{\text{rep\_}k>5\_\text{SFoFi}}$ denote the number of replaceable nodes in $N_{\text{rep\_}k>5}$ that have at least one SFoFi node. Column 9 lists the percentage of $N_{\text{rep\_}k>5\_\text{SFoFi}}$ with respect to $N_{\text{rep\_}k>5}$. When a node having a SFoFi node is replaced, the fanin node can be removed as well. Thus, the resultant circuit is reduced, even though we add one node into the circuit.

For example, the benchmark C3540 has 1038 nodes. Our prior node-merging approach found substitute nodes for 2.8% of nodes with a CPU time of 0.3 s. The proposed approach found that 31.6% of nodes have substitute nodes or added substitute nodes with a CPU time of 0.5 s. Thus, the proposed approach can find 28.8% more replaceable nodes. Additionally, 96.3% replaceable nodes cannot be found by the reimplemented local ODC-based node-merging approach with a bounded depth $k = 5$. Among these nodes, 13.9% of them have at least one SFoFi node.

According to Table II, our prior 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 s. As for the proposed 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 s.

As compared with our prior node-merging approach, the proposed approach can find more replaceable nodes with a reasonable CPU time overhead. The average number of replaceable nodes is 27.8% more with a ratio 5.26, and the CPU time overhead is only 834.4 s for all benchmarks. Because the proposed approach identifies much more replaceable nodes, it has a better logic restructuring capability than that of the node-merging approach.

Additionally, an average of 72.0% of the replaceable nodes that identified by the proposed approach cannot be found by the reimplemented local ODC-based node-merging approach with a bounded depth $k = 5$. An average of 19.8% of these nodes have at least one SFoFi node. Thus, it can be expected that the proposed approach could complement the local ODC-based node-merging approach [37]. They could work together to obtain a better quality.

B. Circuit Size Reduction

In the experiments, we compare the proposed approach with our prior ATPG-based node-merging approach [14] as well as the SAT-based node-merging approach [28] 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 [28], which

TABLE III
EXPERIMENTAL RESULTS OF CIRCUIT SIZE REDUCTION BY USING THE APPROACHES IN [14], [28], AND OUR APPROACH

Benchmark	N	[28]		[14]			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	7334	16.8	7.8	7318	17.0	14.8
s38584	9990	0.8	223	9846	1.4	11.4	9836	1.5	15.1
ac97_ctrl	10 395	2.0	188	10 379	0.2	2.0	10 364	0.3	3.1
systemcaes	10 585	3.8	360	10 521	0.6	13.1	10 386	1.9	30.7
usb_funct	13 320	1.4	681	13 026	2.2	5.9	12 868	3.4	11.4
pci_bridge32	17 814	0.1	1134	17 729	0.5	12.0	17 599	1.2	19.7
aes_core	20 509	8.6	1620	20 371	0.7	13.2	20 195	1.5	22.7
b17	34 523	1.6	5000	33 979	1.5	72.4	33 204	3.8	205.5
wb_conmax	41 070	6.2	5000	39 266	4.4	31.9	38 880	5.3	48.4
des_perf	71 327	3.7	5000	70 081	1.8	62.6	69 421	2.7	84.7
Average		**5.0**			**3.9**			**5.0**	
Total			**21 887**	**266 094**		**255.3**	**263 221**		**498.1**
Ratio			**43.94**			**0.51**			**1**

TABLE IV
COMPARISON OF EXPERIMENTAL RESULTS AMONG $(Ours + resyn2) \times 3$, $resyn2 \times 6$, AND $Ours \times 6$

	$(Ours + resyn2) \times 3$		$resyn2 \times 6$		$Ours \times 6$	
	%	T (s)	%	T (s)	%	T (s)
Average	8.6		4.3		5.9	
Total		1453.1		157.1		2691.2

TABLE V
EXPERIMENTAL RESULTS OF SAT-BASED BSEC FACILITATION

Benchmark	FFs	k	Original	Opti. by *resyn2*			Opti. by *(resyn2+Ours)*		
			Total	SAT	Total	Saved	SAT	Total	Saved
b04	**132**	**12**	**1615.7**	**10.3**	**10.9**	**1604.8**	**0.1**	**3.8**	**7.1**
ss.	174	48	2947.8	353.6	356.2	2591.5	0.3	63.3	292.9
usb.	196	40	2627.2	972.1	975.0	1652.2	89.1	164.9	810.1
simple.	264	30	36000.0	1610.0	1615.6	34384.4	213.7	355.1	1260.5
s5378	328	26	1224.8	265.3	269.9	954.9	84.7	140.7	129.2
syst.	380	10	36000.0	36000.0	36000.0	0.0	120.0	358.2	35641.8
s9234	422	19	2570.6	329.0	333.1	2237.5	21.8	47.5	285.6
b22	1470	5	22943.4	796.6	801.8	22141.7	224.5	310.6	491.2
aes.	1060	5	36000.0	12204.6	12222.9	23777.1	182.7	677.5	11545.4
Total			**141929.5**	**52541.5**	**52585.4**	**89344.1**	**936.9**	**2121.6**	**50463.8**

performs local circuit rewriting optimization [26]. 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 [28]. 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 the proposed approach as shown in Fig. 6 and our prior ATPG-based node-merging approach. Finally, we also apply an equivalence checking tool, *cec* [27], in the ABC package to verify the correctness of the optimization.

Table III summarizes the experimental results. Columns 1 and 2 list the benchmarks and the number of nodes in each benchmark represented as an AIG, respectively. Columns 3 and 4 list the results of the SAT-based node-merging approach reported in [28], 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 s, which is the CPU time limit set by the work. Columns 5–7 list the results of our prior 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 8–10 list the corresponding results of the proposed approach.

The experimental results in Table III show that the proposed approach is 43.94 times faster than the SAT-based node-merging approach and has a competitive capability of circuit size reduction. Additionally, the capability is better than that of our prior ATPG-based node-merging approach by saving 2873 more nodes for all the benchmarks. The overall CPU time overhead is only 242.8 s.

Because the proposed approach is highly efficient, we think it could be used as a preprocess to optimize a circuit before applying the SAT-based node-merging approach.

Moreover, to demonstrate the feasibility of combining our approach with other technique for circuit size optimization, we optimize the benchmarks listed in Table III by repeatedly using our approach followed by the *resyn2* script three times—(*Ours*+*resyn2*)×3. The average circuit size reduction is 8.6% and the CPU time is 1453.1 s. However, if we optimize these benchmarks by repeatedly using the *resyn2* script six times—*resyn2*×6, the average circuit size reduction is only 4.3% with a CPU time of 157.1 s. In addition, the average circuit size reduction is 5.9% and the CPU time is 2691.2 s by repeatedly using our approach six times—*Ours*×6. The experimental results are summarized in Table IV. According to the experimental results, we find 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.

C. SAT-Based BSEC Facilitation

In the experiments, we focus on verifying two functionally equivalent circuits. Each benchmark is first optimized by using the *resyn2* script, and then it is connected to its original one as a miter for equivalence checking. After constructing the miter, we unfold it to a bounded depth k to form a BSEC model. Finally, we use the SAT solver, MiniSat [39], to solve the BSEC model and measure the CPU time. For comparison, we also execute two different optimization procedures. In the first procedure, we use the *resyn2* script to optimize the miters and then the BSEC models. In the second procedure, we use the *resyn2* script followed by our approach to optimize the miters and then the BSEC models. The second procedure is

used to demonstrate that our approach can work together with the *resyn2* script to obtain a better quality. Finally, we also measure the spent CPU time by the SAT solver for solving these optimized BSEC models.

The experimental results are summarized in Table V. Columns 1 and 2 list the benchmarks and the number of flip-flops (FFs) in each benchmark, respectively. Column 3 lists the bounded unfolding depth k. Column 4 lists the spent CPU time by the SAT solver for solving an original BSEC model. The CPU time limit is 36 000 s. Columns 5–7 list the results of using the *resyn2* script to optimize the BSEC models. They are the spent CPU time for SAT solving, the total CPU time, and the saved CPU time compared to the CPU time shown in Column 4. Columns 8–10 list the corresponding results of using the *resyn2* script followed by our approach to optimize the BSEC models. Here, the saved CPU time is compared to the total CPU time shown in Column 6.

For example, the benchmark b04 has 132 FFs. The SAT solver spent 1615.7 s to solve its BSEC model which is unfolded to 12 timeframes. When we optimized the BSEC model by using the *resyn2* script, the SAT solver spent 10.3 s to solve it, and the total CPU time is 10.9 s. Thus, we saved 1604.8 s ($1615.7 - 10.9 = 1604.8$). However, when we optimized the BSEC model by using the *resyn2* script followed by our approach, the SAT solver spent only 0.1 s and the total CPU time is 3.8 s. Thus, we further saved 7.1 s ($10.9 - 3.8 = 7.1$).

The experimental results show that logic optimization and restructuring could be a preprocess before SAT-based BSEC for reducing the verification complexity. When using the *resyn2* script to optimize BSEC models, we can save a total of 89344.1 s (approximately 25 h) for all the benchmarks. Furthermore, when using our approach to complement the *resyn2* script, we can further save a total of 50463.8 s (approximately 14 h) for all the benchmarks. Additionally, the time overhead of the BSEC model optimization is less than 20 min ($2121.6 - 936.9 = 1184.7$). Thus, although the *resyn2* script is effective for most of the benchmarks, our approach can be combined with it to achieve a better speedup.

VIII. Conclusion

In this paper, we proposed 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.

We also proposed an efficient algorithm for circuit size reduction based on the NAR approach. The techniques of redundancy removal and MA reuse were engaged to make the algorithm more efficient and effective. Moreover, we applied the algorithm to facilitate SAT-based BSEC by reducing the computation complexity.

The experimental results showed that the proposed algorithm enhanced our prior ATPG-based node-merging approach and could complement a SAT-based node-merging approach. Additionally, it has a competitive capability of circuit size reduction and expends much less CPU time compared to another SAT-based node-merging approach. The experimental results also showed that the proposed algorithm can be integrated with other optimization technique to obtain a better circuit size reduction. For SAT-based BSEC, the proposed algorithm can work together with other optimization technique to save much equivalence checking time for all the benchmarks. All these results showed the efficiency and effectiveness of the proposed approach.

References

[1] M. S. Abadir, J. Ferguson, and T. E. Kirkland, "Logic design verification via test generation," *IEEE Trans. Comput.-Aided Des.*, vol. 7, no. 1, pp. 138–148, Jan. 1988.

[2] M. Abramovici, M. A. Breuer, and A. D. Friedman, "Digital systems testing and testable design," in *Design for Testability*. Piscataway, NJ: IEEE Press, 1990.

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

[4] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Proc. Des. Automat. Conf.*, 1999, pp. 317–320.

[5] A. Biere, A. Cimatti, E. M. Clarke, O. Strichmanm, and Y. Zhu, "Bounded model checking," *Adv. Comput.*, vol. 58, no. 3, pp. 118–149, 2003.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. Tools Algorithms Construct. Anal. Syst.*, 1999, pp. 193–207.

[7] M. Case, V. Kravets, A. Mishchenko, and R. Brayton, "Merging nodes under sequential observability," in *Proc. Des. Automat. Conf.*, 2008, pp. 540–545.

[8] L. C. L. Chang, C. H. P. Wen, and J. Bhadra, "Speeding up bounded sequential equivalence checking with cross-timeframe state-pair constraints from data learning," in *Proc. Int. Test Conf.*, 2009, pp. 1–8.

[9] C. W. J. Chang, M. F. Hsiao, and M. M. Sadowska, "A new reasoning scheme for efficient redundancy addition and removal," *IEEE Trans. Comput.-Aided Des.*, vol. 22, no. 7, pp. 945–952, Jul. 2003.

[10] S. C. Chang, K. T. Cheng, N. S. Woo, and M. Marek-Sadowska, "Post-layout logic restructuring using alternative wires," *IEEE Trans. Comput.-Aided Des.*, vol. 16, no. 6, pp. 587–596, Jun. 1997.

[11] S. C. Chang, L. P. P. P. Van Ginneken, and M. Marek-Sadowska, "Circuit optimization by rewiring," *IEEE Trans. Comput.*, vol. 48, no. 9, pp. 962–970, Sep. 1999.

[12] S. C. Chang, M. Marek-Sadowska, and K. T. Cheng, "Perturb and simplify: Multi-level boolean network optimizer," *IEEE Trans. Comput.-Aided Des.*, vol. 15, no. 12, pp. 1494–1504, Dec. 1996.

[13] Y. C. Chen and C. Y. Wang, "An Improved approach for alternative wire identification," in *Proc. Int. Conf. Comput. Des.*, 2005, pp. 711–716.

[14] Y. C. Chen and C. Y. Wang, "Fast detection of node mergers using logic implications," in *Proc. Int. Conf. Comput.-Aided Des.*, 2009, pp. 785–788.

[15] Y. C. Chen and C. Y. Wang, "Fast node merging with don't cares using logic implications," *IEEE Trans. Comput.-Aided Des.*, vol. 29, no. 11, pp. 1827–1832, Nov. 2010.

[16] Y. C. Chen and C. Y. Wang, "Node addition and removal in the presence of don't cares," in *Proc. Des. Automat. Conf.*, 2010, pp. 505–510.

[17] K. T. Cheng and L. A. Entrena, "Multi-level logic optimization by redundancy addition and removal," in *Proc. Eur. Conf. Des. Automat.*, 1993, pp. 373–377.

[18] F. S. Chim, T. K. Lam, and Y. L. Wu, "On improved scheme for digital circuit rewiring and application on further improving FPGA technology mapping," in *Proc. Asia South Pacific Des. Automat. Conf.*, 2009, pp. 197–202.

[19] L. A. Entrena and K. T. Cheng, "Combinational and sequential logic optimization by redundancy addition and removal," *IEEE Trans. Comput.-Aided Des.*, vol. 14, no. 7, pp. 909–916, Jul. 1995.

[20] T. Kirkland and M. R. Mercer, "A topological search algorithm for ATPG," in *Proc. Des. Automat. Conf.*, 1987, pp. 502–508.

[21] A. Kuehlmann, "Dynamic transition relation simplification for bounded propery checking," in *Proc. Int. Conf. Comput.-Aided Des.*, 2004, pp. 50–57.

[22] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. Comput.-Aided Des.*, vol. 21, no. 12, pp. 1377–1394, Dec. 2002.

[23] W. Kunz and D. K. Pradhan, "Recursive learning: A new implication technique for efficient solutions to CAD problems-test, verification, and optimization," *IEEE Trans. Comput.-Aided Design*, vol. 13, no. 9, pp. 1143–1158, Sep. 1994.
[24] C. C. Lin and C. Y. Wang, "Rewiring using irredundancy removal and addition," in *Proc. Des., Automat. Test Eur.*, 2009, pp. 324–327.
[25] W. H. Lo and Y. L. Wu, "Improving single-pass redundancy addition and removal with inconsistent assignments," in *Proc. Int. Symp. VLSI Des., Automat. Test*, 2006, pp. 175–178.
[26] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *Proc. Des. Automat. Conf.*, 2006, pp. 532–536.
[27] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *Proc. Int. Conf. Comput.-Aided Des.*, 2006, pp. 836–843.
[28] S. Plaza, K. H. Chang, I. L. Markov, and V. Bertacco, "Node mergers in the presence of don't cares," in *Proc. Asia South Pacific Des. Automat. Conf.*, 2007, pp. 414–419.
[29] M. H. Schulz amd E. Auth, "Advanced automatic test pattern generation and redundancy identification techniques," in *Proc. Int. Fault-Tolerant Comput. Symp.*, 1988, pp. 30–35.
[30] D. Stoffel, M. Wedler, P. Warkentin, and W. Kunz, "Structural FSM traversal," *IEEE Trans. Comput.-Aided Des.*, vol. 23, no. 5, pp. 598–619, May 2004.
[31] A. Veneris and M. S. Abadir, "Design rewiring using ATPG," *IEEE Trans. Comput.-Aided Des.*, vol. 21, no. 12, pp. 1469–1479, Dec. 2002.
[32] C. A. Wu, T. H. Lin, S. L. Huang, and C. Y. Huang, "SAT-controlled redundancy addition and removal: A novel circuit restructuring technique," in *Proc. Asia South Pacific Des. Automat. Conf.*, 2009, pp. 191–196.
[33] W. Wu and M. S. Hsiao, "Mining global constraints with domain knowledge for improving bounded sequential equivalence checking," *IEEE Trans. Comput.-Aided Des.*, vol. 27, no. 1, pp. 197–201, Jan. 2008.
[34] Y. L. Wu, W. Long, and H. Fan, "A fast graph-based alternative wiring scheme for boolean networks," in *Proc. Int. VLSI Des. Conf.*, 2000, pp. 268–273.
[35] Y. L. Wu, C. C. Cheung , D. I. Cheng, and H. Fan, "Further improve circuit partitioning using GBAW logic perturbation techniques," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 11, no. 3, pp. 451–460, Jun. 2003.
[36] X. Q. Yang, T. K. Lam, and Y. L. Wu, "ECR: A low complexity generalized error cancellation rewiring scheme," in *Proc. Des. Automat. Conf.*, 2010, pp. 511–516.
[37] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT Sweeping with local observability don't cares," in *Proc. Des. Automat. Conf.*, 2006, pp. 229–234.
[38] *IWLS 2005 Benchmarks* [Online]. Available: http://iwls.org/iwls2005/benchmarks.html
[39] MiniSat [Online]. Available: http://minisat.se

Yung-Chih Chen received the B.S., M.S., and Ph.D. degrees from the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan, in 2003, 2005, and 2011, respectively.

He is currently an Assistant Professor with the Department of Electronic Engineering, Chung Yuan Christian University, Taoyuan, Taiwan. From June 2010 to March 2011, he was a Visiting Student with the Department of Computer Science and Engineering, Pennsylvania State University, University Park. His current research interests include logic synthesis, design verification, and design automation for emerging technologies.

Chun-Yao Wang (S'00–M'03) received the B.S. degree from the Department of Electronics Engineering, National Taipei University of Technology, Taipei, Taiwan, in 1994, and the Ph.D. degree from the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, in 2002.

Since 2003, he has been an Assistant Professor with the Department of Computer Science, National Tsing Hua University, Hsinchu. In 2010, he was a Visiting Professor with the Department of Computer Science and Engineering, Pennsylvania State University, University Park. He is currently an Associate Professor with the Department of Computer Science, National Tsing Hua University. He is the inventor on six patents. His current research interests include logic synthesis, optimization, and verification for very large-scale integrated/system-on-chip designs and emerging technologies. He has published over 40 technical papers in these areas.

Dr. Wang received the Distinguished Teaching Award from the College of Electrical Engineering and Computer Science, National Tsing Hua University, in 2007. In 2008, he was also awarded the Distinguished Teaching Award from National Tsing Hua University. Only 10–12 out of about 800 faculties of National Tsing Hua University can be awarded per year. In 2009, he was awarded the Excellent Young Electrical Engineer Medal from the Chinese Institute of Electrical Engineering, Taipei. Two of his research results were nominated as the Best Papers in the 2009 IEEE Asia and South Pacific Design Automation Conference and the 2010 IEEE/ACM Design Automation Conference, respectively.