# An Implicit Approach to Minimizing Range-Equivalent Circuits

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

***Abstract*—Simplifying a combinational circuit while preserving its range has a variety of applications, such as combinational equivalence checking and random simulation. Previous approaches use the *binary decision diagram* (BDD) technique to compute the range of one circuit and then reconstruct the circuit using the computed range. Although the size of the new circuit is significantly reduced due to the range rearrangement, this method suffers from the BDD blowup problems for large circuits since performing range computation using BDD is memory intensive. Thus, in this paper, we propose a new method for simplifying combinational circuits without explicit range computation. We first introduce a new concept of a stuck-at fault test for a circuit's range, showing that a range untestable stuck-at fault on a primary input (PI) indicates that this PI is range redundant, i.e., it can be removed without affecting the circuit's range. We then present a procedure to determine if a given range stuck-at fault on a PI is untestable. Our method iteratively identifies and removes range-redundant PIs to simplify a combinational circuit without performing range computation. Accordingly, large circuits that BDD-based methods cannot deal with can be handled using our method. We conduct experiments on a set of ISCAS'85 and MCNC benchmarks, and the experimental results show that our approach can minimize circuits such that fewer PIs are left. On average, our approach gets 37.06% reduction in terms of the number of PIs and 36.31% reduction in terms of the node counts.**

***Index Terms*—Range-preserving simplification, range-redundant primary input (PI).**

## I. Introduction

THE RANGE of a combinational circuit is the set of all possible output combinations [14]. Simplifying a combinational circuit while preserving its range has a variety of applications, e.g., combinational equivalence checking and random simulation.

*Binary decision diagram* (BDD)-based [6] cut-point verification is a widely used approach in combinational logic equivalence checking [4], [10], [13]. These methods partition the specification and implementation circuits into many subcircuits to avoid huge memory requirements in verifying the equivalence of the circuits. However, these approaches cause false negative problems if they simply replace the cut nets with free variables, because they lose the correlations with the free variables. To deal with this problem, one can add

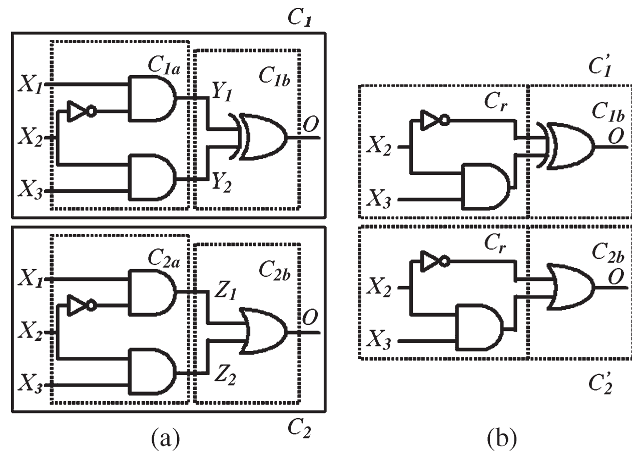



Fig. 1. (a) False negative problem: $C_{1b}$ is not equivalent to $C_{2b}$, but $C_1$ and $C_2$ are equivalent. (b) Add a simplified range-equivalent circuit $C_r$ to avoid the false negative problem.

a simplified range-equivalent circuit connected to the cuts to drive the subcircuits [14], [15]. For example, in Fig. 1(a) (taken from [15]), $C_1$ and $C_2$ are designs under verification (DUV). The cut-point-based approaches partition $C_1$ and $C_2$ into two subcircuits to avoid memory explosion during verification. Therefore, the equivalence checking for $C_1$ and $C_2$ is now transformed to verifying the equivalence of $C_{1a}$ and $C_{2a}$ as well as the equivalence of $C_{1b}$ and $C_{2b}$. In this example, $C_{1a}$ and $C_{2a}$ are equivalent, but $C_{1b}$ is not equivalent to $C_{2b}$. However, $C_1$ and $C_2$ are actually functionally equivalent, demonstrating that this method can give falsely negative results. The reason for this problem is that these approaches regard $Y_1$, $Y_2$ and $Z_1$, $Z_2$ as free variables when verifying the equivalence of $C_{1b}$ and $C_{2b}$. In fact, $Y_1(Z_1)$ and $Y_2(Z_2)$ cannot be simultaneously assumed to equal 1. To deal with this problem, we can add a simplified range-equivalent circuit of $C_{1a}$ and $C_{2a}$, i.e., $C_r$, to drive $C_{1b}$ and $C_{2b}$ [as shown in Fig. 1(b)] and then verify the equivalence of $C_1'$ and $C_2'$. Since the number of primary inputs (PIs) in the simplified range-equivalent circuit is usually reduced, it is more likely that the equivalence check for $C_1'$ and $C_2'$ can be conducted by BDD-based approaches.

Random simulation is an efficient way to detect design errors. However, applying illegal random vectors to the DUV is meaningless for verification. Thus, the concept of constraint-based random simulation that can generate legal patterns for simulation is proposed [19]–[21]. However, the knowledge of generating these constraints is carried by designers according to the DUV specification. As a result, designers have to be involved in writing the input constraints for random pattern generation, increasing the possibility of manual errors. Another

Manuscript received July 19, 2007; revised February 3, 2008 and May 20, 2008. Current version published October 22, 2008. This work was supported in part by the National Science Council of R.O.C. under Grants NSC 96-2220-E-007-009 and NSC 96-2220-E-007-049. This paper was recommended by Associate Editor S. Nowick.

The authors are with the Department of Computer Science, National Tsing Hua University, Hsinchu 300, Taiwan (e-mail: ycchen@cs.nthu.edu.tw; wcyao@cs.nthu.edu.tw).

Digital Object Identifier 10.1109/TCAD.2008.2006088

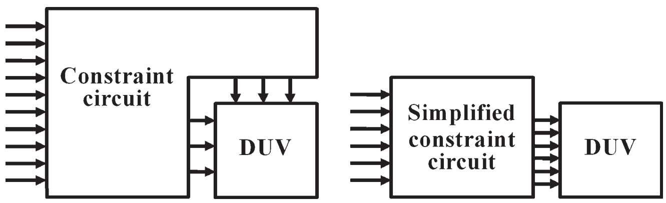



Fig. 2. Replace the constraint circuit by a simplified (with fewer PIs) but range-equivalent circuit.

approach to dealing with the problem of illegal random simulation is to construct a constraint circuit for generating legal random vectors for simulation. This idea is as follows. In general, the inputs of the DUV are driven by the outputs of other modules that are within the same system. Thus, these modules, also called the environment, can be seen as constraint circuits for the DUV. The outputs of these modules can always feed legal vectors to the DUV. The main concern of this approach is verification efficiency. Verification engineers hope that the random vectors can be evenly distributed to reach more design corners. However, the randomness of input vectors in the DUV can suffer from the nonuniformity constraint circuit outputs. For example, many random input vectors in the constraint circuit will generate the same output vector, which will be the random input vector for the DUV. Thus, if the constraint circuit can be replaced by a simplified (with fewer PIs) but range-equivalent circuit, the verification process will be accelerated. This idea is shown in Fig. 2. Note that the constraint circuit is limited to a combinational circuit in this application. For a sequential circuit, this idea still works if the sequential circuit can be translated into a combinational one by setting reachable state values to flip-flops.

To simplify a combinational circuit without changing its range, several works have been proposed [2], [3], [11], [14], [22]. These methods first compute and present the range of the circuit using BDDs. They then synthesize the computed range into a range-equivalent circuit. These range-computation-based methods can obtain significant reductions on the size of the circuit. Unfortunately, they could suffer from BDD blowup for large circuits since performing range computation using BDD is memory intensive. As a result, these methods fail to simplify a circuit whose range cannot be computed and presented efficiently.

On the other hand, a *normalized function* method [15] is proposed to simplify a circuit without performing range computation. This method partitions all PIs into two groups. One group is *fan-out-free* PIs, which are PIs in the transitive fan-in cone of only one primary output (PO). The other group is fan-out PIs, which are PIs in the intersection of the transitive fan-in cones of more than one PO. This method is capable of removing all fan-out-free PIs, but retains the fan-out PIs. As a result, although the method can minimize some large circuits that the range-computation-based methods cannot, the reduction is limited. In particular, for circuits with only fan-out PIs, the algorithm fails.

Thus, in this paper, we propose a new method to simplify a circuit by removing range-redundant PIs. A *range-redundant* PI is a PI which is not responsible for the circuit's range. We can replace a range-redundant PI with a constant value, 1 or 0, without changing the circuit's range. Our approach is based on the concept of mandatory assignments (MAs), which is used in the identification of functional redundant wires [7], [8], [17]. With this approach, we iteratively identify and remove the range-redundant PIs without performing range computation. Thus, our method also can deal with large circuits that the range-computation-based methods cannot. We conduct experiments on a set of ISCAS'85 and MCNC benchmarks. The experimental results show that our approach can minimize circuits so that fewer PIs are left. Our approach gets an average of 19.74% more reduction than that of the normalized function method [15] in terms of the number of PIs. Note that our approach and that in [15] both focus on reducing the number of PIs in simplification. This is because fewer PIs in the simplified circuit improves performance in various applications. Additionally, although our approach focuses on reducing the number of PIs, the total node count is also reduced when we simplify a circuit by removing the range-redundant PIs. The experimental results show that our approach on average gets 36.31% reduction in terms of the number of nodes.

In contrast to the range-computation-based methods, the simplification capability of our method may not be as good as that of them, but our method is more scalable and can simplify a circuit having an enormous range. Additionally, since the proposed algorithm is not limited to specific applications, any application using the technique of range-preserving simplification can take advantage of this work.

This paper is organized as follows. Section II reviews the related concepts in very large scale integration (VLSI) testing used in this paper. Section III describes our fundamental method of identifying range-redundant PIs. Section IV introduces two procedures to improve the method mentioned in Section III. Section V shows the overall algorithm. Section VI presents the experimental results. Finally, Section VII concludes this paper.

## II. Preliminaries

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 is stuck at either 1 (stuck-at 1) or 0 (stuck-at 0). A stuck-at fault is testable if there exists a test vector that can generate the different output values in the fault-free and faulty circuits. Otherwise, the fault is untestable.

In a combinational circuit, an untestable stuck-at fault does not affect the functionality of the circuit. Thus, an untestable fault on a wire indicates that the wire is redundant. The circuit will still be functionally equivalent if the redundant wire is replaced with the stuck value. Here, we review an approach of identifying functional redundant wires using the concept of MAs [17]. Additionally, we assume that circuits only consist of AND, OR, and INV gates for simplicity. Complex gates can be decomposed into these gates.

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 *output-controlling value* of $g$ is the output value with respect to the input-controlling value. For example, the input-controlling value of a NAND (NOR) gate is 0 (1), and the output-controlling value is 1 (0). The inverse of the input-controlling value is called *input-noncontrolling value*, and the inverse of the output-controlling value is called *output-noncontrolling value*. Furthermore, a gate $g$ or a wire $w$ is in the *transitive fan-out cone* of a wire $w_s$ if there exists a path from $w_s$ to $g$ or $w$, and $w_s$ is in the *transitive fan-in cone* of $g$ or $w$.

The *dominators* [9] of a wire $w$ are a set of gates $G$ such that all paths from $w$ to any PO have to pass through all gates in $G$. Consider the dominators $G$ of a wire $w$. The *side inputs* of $G$ are the inputs of $G$ that are not in the transitive fan-out cone of $w$.

*Logic implication* is a process of computing unique logic values based on known logic values of nodes in a Boolean network. Given a logic value assigned at one node, the value can be propagated forward or backward until no more logic values can be determined. Recursive learning [12], a learning method in automatic test pattern generation, can be used to perform logic implications more completely.

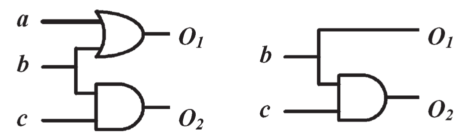
The MAs are the unique value assignments to nodes required for a test to exist. Consider a stuck-at fault test on a wire $w$. The MAs can be obtained by setting $w$ to the fault-activating value and by setting the side inputs of dominators of $w$ to the fault-propagating values. Then, these MAs can be propagated forward or backward to infer more MAs by performing logic implication. If the MAs of the fault are inconsistent, the fault is untestable, and therefore, $w$ is redundant [17].

In this paper, our fundamental idea of identifying range-redundant PIs is derived from that of identifying functional redundant wires using inconsistent MAs.

## III. Range-Redundant PI Identification

In this section, we first present a new concept of a stuck-at fault test for a circuit's range and show that, if a stuck-at fault on a PI is range untestable, this PI is range redundant. We then derive the MAs for a test vector based on a specific type of don't care. The don't care we consider is based on range equivalence, and it is different from the traditional controllability/observability don't cares used in Boolean network minimization [1], [5], [16]. The don't cares that these works consider are based on functionality equivalence. Finally, when the MAs are inconsistent, the fault is range untestable.

Before the detailing description of our approach, we use a simple example to demonstrate the intention of this paper. Fig. 3(a) shows a three-input two-output circuit. Its truth table is also shown in Fig. 3(a). The range of this circuit, denoted as $R$, is $(O_1, O_2) = \{(0,0),(1,0),(1,1)\}$. We observe that the output combinations of the last four minterms $\{a = 1, b = -, c = -\}$ ($-$ denotes irrelevance of value) is the subset of the first four minterms $\{a = 0, b = -, c = -\}$. Thus, the range-equivalent circuit can be constructed using only the first four minterms $\{a = 0, b = -, c = -\}$. Furthermore, since $a = 0$ is the input-noncontrolling value of the OR gate in Fig. 3(a), the OR gate can be removed as well, and the resultant range-equivalent circuit is shown in Fig. 3(b). In this example, we must examine if the minterms with $a = 1$ only generate output values that are the subset of the range of minterms with $a = 0$.




$R = \{(0, 0), (1, 0), (1, 1)\}$ $R' = \{(0, 0), (1, 0), (1, 1)\}$

| a b c | $O_1$ $O_2$ |
|---|---|
| 0 0 0 | 0 0 |
| 0 0 1 | 0 0 |
| 0 1 0 | 1 0 |
| 0 1 1 | 1 1 |
| 1 0 0 | 1 0 |
| 1 0 1 | 1 0 |
| 1 1 0 | 1 0 |
| 1 1 1 | 1 1 |

(a)

| b c | $O_1$ $O_2$ |
|---|---|
| 0 0 | 0 0 |
| 0 1 | 0 0 |
| 1 0 | 1 0 |
| 1 1 | 1 1 |

(b)

Fig. 3. (a) Stuck-at 0 fault on PI $a$ is range untestable. (b) The resultant circuit by replacing $a$ with a constant 0 in (a).

Therefore, we set a stuck-at 0 fault on the PI $a$ and check if the fault effect affects the circuit's range. Note that the traditional stuck-at fault test considers the fault effect to the circuit's functionality; however, we consider if the fault effect affects the circuit's range in this work.

### A. Stuck-at Fault Test for a Circuit's Range

*Definition 1: A stuck-at fault test while considering a circuit's range is a process to find a test (vector) such that the output value of this vector is in the range of a fault-free circuit but not in that of a faulty circuit. Given a stuck-at fault $f$, if there exists such a test, $f$ is said to be range testable; otherwise, $f$ is range untestable.*

In the previous example in Fig. 3(a), the circuit's range $R$ is $(O_1, O_2) = \{(0,0),(1,0),(1,1)\}$. If we consider the stuck-at 0 fault test of the PI $a$, the range of the faulty circuit, denoted as $R'$, is also $(O_1, O_2) = \{(0,0),(1,0),(1,1)\}$ as seen in Fig. 3(b). Since each output value in $R$ is also in $R'$, we cannot find a test vector that produces an output value only in the range of the fault-free circuit. Thus, this fault is range untestable. The following theorem shows that a range untestable fault of a PI indicates that it is a range-redundant PI.

*Theorem 1: A PI is range redundant and can be replaced with a constant value $v$($v$ denotes a logical value 0 or 1), if the stuck-at $v$ fault of this PI is range untestable.*

*Proof:* According to Definition 1, the stuck-at $v$ fault of a PI is range untestable means that there is no output value that is only in the range of the fault-free circuit. Therefore, replacing the PI with a constant value $v$ will not change the circuit's range. Hence, the PI is range redundant. ■

For example, in Fig. 3(a), the stuck-at 0 fault of PI $a$ is range untestable as explained. Hence, we can replace $a$ with a constant 0 without changing the circuit's range. The resultant circuit is shown in Fig. 3(b).

Note that our approach does not use explicit means to examine the range of a circuit, e.g., truth tables or BDDs, and then determine if any PI is range redundant. Instead, we use the idea of inconsistent range MAs (RMA) to achieve this. The details of our method will be presented in the following sections.

### B. RMAs

As mentioned in Section II, for a traditional stuck-at fault test, the MAs are the unique assignments to nodes required for a test to exist. Similarly, for a stuck-at fault to be range testable, some nodes in the circuit have to be set to certain values for a test vector. These assignments are named RMAs. The formal definition of RMA is as follows:

*1) Definition 2: Let $f$ be a stuck-at fault in a combinational circuit $C$, and let $T$ be a set of all vectors that can detect $f$ for $C$'s range. A node $n$ in $C$ has an RMA $m$ if $n$ is assigned as the value $m$ for all vectors in $T$.*

Computing all RMAs of a stuck-at fault is with exponential time complexity. This is because the process involves finding all vectors that can detect the fault. In this paper, instead of finding all RMAs, we present two types of value assignments that can be derived in practice, and then we show that they are some RMAs for a stuck-at fault test on a PI.

*Type 1 Assignments:* The *activating assignment*, which is necessary for activating the fault effect, and the *propagating assignments*, which are necessary for propagating the fault effect.

Consider a stuck-at $v$ fault on a PI pi, the activating assignment is pi $= \bar{v}$ ($\bar{v}$ denotes the inverse value of $v$), and the propagating assignments can be obtained by setting the side inputs of dominators of pi to input-noncontrolling values. These assignments are also the MAs for a stuck-at fault test as mentioned in Section II.

Before introducing another type of assignment, we define a concept that a node with a value assignment is *observable* at a PO. This definition is different from that previously defined and used in Boolean network minimization [1], [5], [16]. In the previous definition, a node $n$ is observable if a change at its value is perceived at one of the circuit's outputs. If $n$ is a PI, its observability is identical in our definition and the previous definition. However, if $n$ is an internal node, its observability may be different. The previous definition only considers whether the value change at $n$ can be observed at the POs in the transitive fan-out cone of $n$. In our approach, however, we also consider whether the value change at $n$ (that made by some value changes at PIs) causes a value change at a PO which is not in the transitive fan-out cone of $n$.

*Definition 3: A node $n$ is observable if all possible value changes at PIs in the transitive fan-in cone of $n$ that can change the value of $n$ make a value change at a PO. An input pattern $t$ makes $n$ observable if $n$ is observable under the condition that $t$ is applied to the circuit. Otherwise, $t$ makes $n$ unobservable.*

In Definition 3, we only consider the value changes at PIs that are in the transitive fan-in cone of $n$. This is because only they are responsible for the value change at $n$. In addition, if $n$ is a PI, the PI we consider in its transitive fan-in cone is $n$ itself. We use a simple example in Fig. 4 to clarify Definition 3 and to explain its difference with the previous definition. In this example, $a = 0$ is observable since a change at its value flips $O_1$'s value. Similarly, $b = 0$ and $d = 1$ are observable as well. However, $c = 1$ is not observable since $b = 0$ blocks the effect of $c$'s value change. Next, let us consider the observability of the internal node $g_1$. In the previous definition, $g_1 = 0$ is not observable because $d = 1$ blocks the effect of its value change.

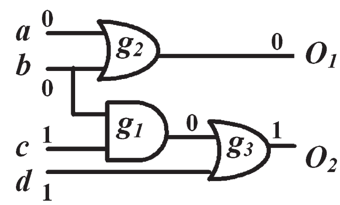
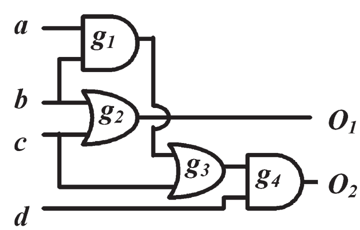



Fig. 4. Example of node assignments that are observable/unobservable.




| | COAs | NCOAs |
|---|---|---|
| $a$ | $b$=1, $c$=0, $d$=1 | $b$=1, $c$=0, $d$=1 |
| $b$ | ∅ | ∅ |
| $c$ | ∅ | ∅ |
| $d$ | $g_3$=1 | $g_3$=1 |
| $g_1$ | ∅ | $b$=1, $c$=0, $d$=1 |
| $g_2$ | ∅ | ∅ |
| $g_3$ | ∅ | ∅ |
| $g_4$ | ∅ | $g_3$=1 |

Fig. 5. Example of observability assignments.

However, $g_1 = 0$ is observable according to Definition 3. The value change at $g_1$, $g_1$ becomes 1, must be caused by the value change at $b$, $b$ becomes 1, and $b$ becoming 1 flips the value of $O_1$. As a result, $g_1 = 0$ is observable. Furthermore, let us consider the observability of $g_1$ when the input pattern is $\{a = 0, b = 1, c = 1, d = 1\}$. In this case, $g_1 = 1$ is not observable according to Definition 3. Three possible value changes at $b$ and $c$ can cause $g_1 = 0$, $\{b = 0, c = 1\}$, $\{b = 1, c = 0\}$, and $\{b = 0, c = 0\}$, but one of them, $\{b = 1, c = 0\}$, is not observable. Hence, $g_1 = 1$ is not observable.

For a value assignment in a circuit to be observable at a PO, some nodes in the circuit have to be set at certain values. These assignments are named *observability assignments*.

*Type 2 Assignments:* The *observability assignments*, which are necessary for making type 1 or 2 assignments themselves observable at a PO.

To compute type 2 assignments, we first calculate the observability assignments of each PI and internal gate and then collect them based on these assignments.

In a combinational circuit, the observability assignments of a PI or an internal gate are further classified into *controlling observability assignments* (COAs) and *noncontrolling observability assignments* (NCOAs). The COAs and NCOAs of a PI are identical, and they are the same with its propagating assignments. For example, in Fig. 5, $a$'s propagating assignments are $\{b = 1, c = 0, d = 1\}$, and these assignments are also $a$'s COAs and NCOAs. $b$'s COAs and NCOAs are empty set $\emptyset$. This is because $b$ has no dominators. This is also true for the PI $c$. $d$'s COAs and NCOAs are $\{g_3 = 1\}$.

For an internal gate $g$, its COAs and NCOAs, denoted as COAs$(g)$ and NCOAs$(g)$, are the necessary assignments for making $g$ observable at a PO when $g$ has its output-controlling and -noncontrolling values, respectively. They can be derived from the COAs and NCOAs of $g$'s inputs. Theorems 2 and 3 show the methods. Here, we suppose $g$ has $k$ inputs. In addition, icv$(g)$, incv$(g)$, ocv$(g)$, and oncv$(g)$ denote the

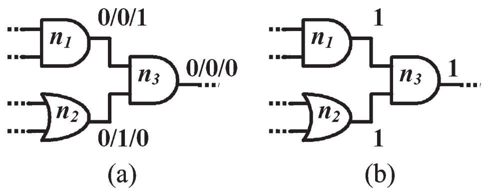



Fig. 6. (a) COAs($n_3$) are the intersection of COAs($n_1$) and NCOAs($n_2$). (b) NCOAs($n_3$) are the union of NCOAs($n_1$) and COAs($n_2$).

input-controlling, input-noncontrolling, output-controlling, and output-noncontrolling values of $g$, respectively.

*Theorem 2: Let $g_{\text{in\_}j}$ be the $j$th input of $g$* and OAs_C($g_{\text{in\_j}}$) *denote the set of observability assignments necessary for making $g_{\text{in\_}j} = \text{icv}(g)$ observable. Each value assignment in $\bigcap_{j=1}^{k}$* OAs_C($g_{\text{in\_j}}$) *must be necessary for making $g = \text{ocv}(g)$ observable.*

*Proof:* Suppose $n = v$ is a value assignment in $\bigcap_{j=1}^{k}$ OAs_C($g_{\text{in\_j}}$), and $t$ is an input pattern that generates $g = \text{ocv}(g)$ but $n = \bar{v}$. Since $n = v$ is necessary for each input of $g$ having icv($g$) to be observable, $t$ generating $n = \bar{v}$ makes all of them unobservable. Let us consider the value change at $g$; it must be caused by all inputs of $g$ becoming incv($g$). Since all of them cannot be observed at a PO, $g = \text{ocv}(g)$ is unobservable. Thus, an input pattern generating $n = \bar{v}$ must make $g = \text{ocv}(g)$ unobservable, and therefore, $n = v$ is necessary for $g = \text{ocv}(g)$ to be observable. ■

We use the example in Fig. 6 to demonstrate Theorem 2. In Fig. 6(a), $n_1$ and $n_2$ are the inputs of $n_3$. Suppose $n = v$ is a value assignment in OAs_C($n_1$) $\bigcap$ OAs_C($n_2$), and $t$ is an input pattern that generates $n_3 = 0$ but $n = \bar{v}$. When we apply $t$ to the circuit, there are three possible combinations on the values of $n_1$ and $n_2$ : $\{n_1 = 0, n_2 = 0\}$, $\{n_1 = 0, n_2 = 1\}$, and $\{n_1 = 1, n_2 = 0\}$, and $t$ makes both $n_1 = 0$ and $n_2 = 0$ unobservable. Next, let us consider the value change at $n_3$. It must be caused by the values of $n_1$ and $n_2$ becoming 1. Since they cannot be observed at a PO, $n_3 = 0$ is unobservable. Thus, $t$ does not make $n_3 = 0$ observable, and hence, $n = v$ is a necessary assignment.

Consider computing COAs($g$). We can compute them by intersecting the COAs or NCOAs of all of $g$'s inputs. For example, in Fig. 6(a), $\text{ocv}(n_3) = \text{icv}(n_3) = \text{ocv}(n_1) = \text{oncv}(n_2) = 0$. The observability assignments necessary for making $n_1 = \text{icv}(n_3) = \text{ocv}(n_1) = 0$ observable are COAs($n_1$), and the observability assignments necessary for making $n_2 = \text{icv}(n_3) = \text{oncv}(n_2) = 0$ observable are NCOAs($n_2$). Thus, COAs($n_3$) are the intersection of COAs($n_1$) and NCOAs($n_2$). In the example, in Fig. 5, COAs($g_1$) are the intersection of COAs($a$) and COAs($b$), and they are $\emptyset$.

*Theorem 3: Let $g_{\text{in\_}j}$ be the $j$th input of $g$ and* OAs_NC($g_{\text{in\_j}}$) *denote the set of observability assignments necessary for making $g_{\text{in\_}j} = \text{incv}(g)$ observable. Each value assignment in $\bigcup_{j=1}^{k}$* OAs_NC($g_{\text{in\_j}}$) *must be necessary for making $g = \text{oncv}(g)$ observable.*

*Proof:* Suppose $n = v$ is a value assignment in $\bigcup_{j=1}^{k}$ OAs_NC($g_{\text{in\_j}}$), and $t$ is an input pattern that generates $g = \text{oncv}(g)$ but $n = \bar{v}$. Since $n = v$ is necessary for at least one input $g_{\text{in}}$ of $g$ having incv($g$) to be observable, $t$ makes $g_{\text{in}} = \text{incv}(g)$ unobservable. Let us consider the value change at $g$; it may be caused by $g_{\text{in}}$ becoming icv($g$). Since the value change at $g_{\text{in}}$ cannot be observed at a PO, $t$ makes $g = \text{oncv}(g)$ unobservable. Thus, an input pattern generating $n = \bar{v}$ must make $g = \text{oncv}(g)$ unobservable, and therefore, $n = v$ is necessary for $g = \text{oncv}(g)$ to be observable. ■

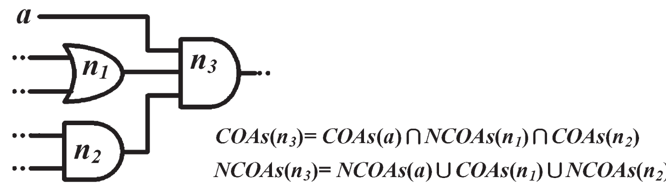



Fig. 7. Example of (1) and (2).

We use the example in Fig. 6 to demonstrate Theorem 3. In Fig. 6(b), $n_1$ and $n_2$ are the inputs of $n_3$. Suppose $n = v$ is necessary for $n_1 = \text{incv}(n_3)$ to be observable, and $t$ is an input pattern that generates $n_3 = 1$ but $n = \bar{v}$. When we apply $t$ to the circuit, we obtain $n_1 = 1$ and $n_2 = 1$. Next, let us consider the value change at $n_3$. It may be caused by $n_1$'s value becoming 0. Since the value change at $n_1$ cannot be observed at a PO, $n_3 = \text{incv}(n_3)$ is unobservable. Thus, $t$ does not make $n_3 = 1$ observable, and hence, $n = v$ is a necessary assignment. Similarly, each value assignment necessary for $n_2 = \text{incv}(n_3)$ to be observable is necessary for $n_3 = 1$ to be observable as well.

Next, consider computing NCOAs($g$). NCOAs($g$) are the union of the COAs or NCOAs of all of $g$'s inputs. Consider the example in Fig. 6(b). $\text{oncv}(n_3) = \text{incv}(n_3) = \text{oncv}(n_1) = \text{ocv}(n_2) = 1$. The observability assignments necessary for making $n_1 = \text{incv}(n_3) = \text{oncv}(n_1) = 1$ observable are NCOAs($n_1$), and the observability assignments necessary for making $n_2 = \text{incv}(n_3) = \text{ocv}(n_2) = 1$ observable are COAs($n_2$). Thus, NCOAs($n_3$) are the union of NCOAs($n_1$) and COAs($n_2$). In Fig. 5, NCOAs($g_1$) are the union of NCOAs($a$) and NCOAs($b$), and they are $\{b = 1, c = 0, d = 1\}$.

More formally, COAs($g$) and NCOAs($g$) can be calculated using (1) and (2) as follows. We denote $g_{\text{in\_}j}$ as the $j$th input of $g$. If $\text{icv}(g) = \text{ocv}(g_{\text{in\_}j})$, the notation OAs_C($g_{\text{in\_}j}$) represents COAs($g_{\text{in\_}j}$); otherwise, it represents NCOAs($g_{\text{in\_}j}$). If $\text{incv}(g) = \text{ocv}(g_{\text{in\_}j})$, the notation OAs_NC($g_{\text{in\_}j}$) represents COAs($g_{\text{in\_}j}$); otherwise, it represents NCOAs($g_{\text{in\_}j}$). For example, in Fig. 7, $a$ is a PI, and $n_1$, $n_2$, and $n_3$ are internal nodes in a circuit. $\text{icv}(n_3) = \text{ocv}(n_3) = 0$ is oncv($n_1$) and ocv($n_2$). Thus, COAs($n_3$) are the intersection of COAs($a$), NCOAs($n_1$), and COAs($n_2$). Additionally, $\text{incv}(n_3) = \text{oncv}(n_3) = 1$ is ocv($n_1$) and oncv($n_2$). Thus, NCOAs($n_3$) are the union of NCOAs($a$), COAs($n_1$), and NCOAs($n_2$)

$$\text{COAs}(g) = \bigcap_{j=1}^{k} \text{OAs\_C}(g_{\text{in\_}j}) \tag{1}$$

$$\text{NCOAs}(g) = \bigcup_{j=1}^{k} \text{OAs\_NC}(g_{\text{in\_}j}). \tag{2}$$

Finally, in the example of Fig. 5, by (1) and (2), COAs($g_2$) and NCOAs($g_2$) are $\emptyset$. COAs($g_3$) and NCOAs($g_3$) are $\emptyset$ as well. Furthermore, COAs($g_4$) is $\emptyset$, and NCOAs($g_4$) is $\{g_3 = 1\}$.

With each gate's COAs and NCOAs, we can calculate type 2 assignments for a PI's stuck-at fault test. Consider $a$'s stuck-at 1 fault test in Fig. 5. Type 1 assignments are $a = 0$ to activate the fault effect and $\{b = 1, c = 0, d = 1\}$ to propagate the fault effect. For type 2 assignments, $g_3 = 1$ is COAs$(d)$ and NCOAs$(d)$; therefore, $g_3 = 1$ is necessary to make $d = 1$ observable. Furthermore, since $g_3$ is an OR gate and COAs$(g_3)$ is $\emptyset$, there does not exist any assignment necessary for making $g_3 = 1$ observable. Therefore, the type 2 assignment is $g_3 = 1$.

To summarize the procedure for computing type 1 and 2 assignments for a given fault test on a PI, we first compute the activating assignment and each PI's propagating assignments in order to get type 1 assignments. Second, based on these propagating assignments, we calculate each gate's COAs and NCOAs. Finally, we collect type 2 assignments according to the assignments computed in the first two steps.

*Theorem 4: For a stuck-at fault test on a PI, type 1 and 2 assignments are some RMAs.*

*Proof:* Let $f$ be a stuck-at $v$ fault on a PI pi in a combinational circuit $C, T$ be the set of all input vectors that can detect $f$ for $C$'s range, and $R$ be the set of output values with respect to $T$. By Definition 1, each output value in $R$ must be in the range of the fault-free circuit but not in that of the faulty circuit.

1) *Activating assignment*. Because each output value in $R$ is not in the range of the faulty circuit, each input vector in $T$ must have pi $= \bar{v}$. Therefore, the activating assignment is an RMA for $f$.
2) *Propagating assignment*. Suppose $t$ is an input vector that has pi $= \bar{v}$, and pi $= \bar{v}$ is unobservable when we apply $t$ to $C$. Let $o$ be the output value with respect to $t$. Since $t$ does not make pi $= \bar{v}$ observable, the value change at pi cannot be observed at a PO. Thus, we can find another input vector $t'$ with pi $= v$, and $t'$ can generate $o$ as well. Thus, in order to generate the output values only in the range of the fault-free circuit, each input vector in $T$ must make pi $= \bar{v}$ observable at a PO. Therefore, the propagating assignments are also RMAs for $f$.
3) *Observability assignment*. Suppose $t$ is an input vector that, when we apply $t$ to $C$, pi $= \bar{v}$ is observable, but there exists at least one propagating assignment whose value is unobservable. Let $n_p$ be the propagating assignment and $n_p = v_p$ be the unobservable value. Since $n_p = v_p$ is unobservable, we can find an input vector $t'$ that generates $n_p = \bar{v_p}$ and outputs the same output value. When $n_p$ is assigned to $\bar{v_p}$, the value pi $= \bar{v}$ is blocked and becomes unobservable. As a result, $t'$ cannot make pi $= \bar{v}$ observable. In the earlier paragraph, we have proved that a test vector for $f$ must make pi $= \bar{v}$ observable. Thus, $t'$ is not a test vector, and hence, $t$ is not a test vector as well. Similarly, an input vector that cannot make all observability assignments' values observable is not a test vector for $f$. This is because we can find an input vector that makes at least one propagating assignment's value unobservable while generating the same output value. Thus, each input vector in $T$ must make pi $= \bar{v}$, each propagating assignment's value, and each observability assignment's value observable. Therefore, the observability assignments are also RMAs for $f$. ■

Thus, the MAs for a traditional stuck-at fault test are the subset of the RMAs of a stuck-at fault test on a PI for the circuit's range. When a PI's traditional stuck-at fault test is untestable, there is no input pattern that can produce a different output value between the faulty and fault-free circuit; the PI's stuck-at fault is range untestable as well. As a result, a functional redundant PI is also range redundant. However, a range-redundant PI may not be functional redundant.

Given a stuck-at fault on a PI, we can compute the RMAs of the fault test for the circuit's range and then perform logic implications to obtain more RMAs. If the RMAs are inconsistent, the fault is range untestable. This is because no input pattern can generate a conflict value on a node in the circuit. Thus, we can replace the PI with a constant value. For example, in Fig. 5, consider $a$'s stuck-at 1 fault test. The RMAs are $\{a = 0, b = 1, c = 0, d = 1, g_3 = 1\}$. After performing logic implication, we can find a conflict on $g_3$ because $a = 0$ and $c = 0$ imply $g_3 = 0$. Thus, $a$ is a range-redundant PI, and it can be replaced with a constant 1.

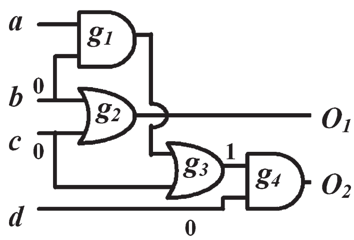
## IV. More RMA Identification

In Section III, we introduce a simple method to compute the RMAs for a given stuck-at fault on a PI. The method, in fact, can only find a subset of all possible RMAs. However, the number of found RMAs significantly affects our capability of identifying range-redundant PIs. In this section, we propose two approaches for finding more RMAs so that more range-redundant PIs can be identified. Finally, we summarize the proposed range-redundant PI identification algorithm.

### A. Propagating Assignment Identification

Given a stuck-at fault on a PI pi, pi's propagating assignments can be obtained by setting the side inputs of the dominators of pi to input-noncontrolling values. For example, in Fig. 5, the propagating assignments of the fault on $a$ are $\{b = 1, c = 0, d = 1\}$. However, this method only works when pi has dominators and side inputs. In Fig. 5, $c$ has no dominators, therefore, no propagating assignment for the fault on $c$ can be found. However, $g_1 = 0$ is necessary to propagate the fault effect, and it is a propagating assignment indeed.

A new method to finding more propagating assignments for a given fault on a multiple fan-out PI is to individually consider the fault propagating path of each fan-out. For example, in Fig. 5, consider the stuck-at fault on $c$. The fault effect can be propagated through either $g_2$ or $g_3$. First, we suppose that the effect is propagated through $g_2$. We then find that the propagating assignment is $b = 0$, and $b = 0$ implies $g_1 = 0$. Next, we suppose that the effect is propagated through $g_3$. We then find that the propagating assignments are $\{g_1 = 0, d = 1\}$. Finally, since $g_1 = 0$ is necessary for propagating the fault effect through either $g_2$ or $g_3$, it is a propagating assignment of the stuck-at fault on $c$. Thus, given a stuck-at fault on a multiple fan-out PI pi, its propagating assignments can be obtained by intersecting the propagating assignments of each of pi's fan-out.




| | *COAs* | *NCOAs* |
|---|---|---|
| *a* | *disabled* | *disabled* |
| *b* | *c*=0 | *c*=0 |
| *c* | *b*=0 | *b*=0 |
| *d* | $g_3$=1 | $g_3$=1 |
| $g_1$ | *c*=0 | *disabled* |
| $g_2$ | Ø | *c*=0, *b*=0 |
| $g_3$ | *b*=0 | *c*=0, *b*=0 |
| $g_4$ | Ø | *b*=0, $g_3$=1 |

Fig. 8. Example of observability assignments by considering the fault effect $d = 0$.

## B. COA and NCOA Identification

To review the approach discussed in Section III for identifying each gate's COAs and NCOAs, we first compute each PI's propagating assignments. Next, based on these propagating assignments, we identify each gate's COAs and NCOAs using (1) and (2).

With these COAs and NCOAs, we can find the RMAs for a given stuck-at fault on a PI. For example, in Fig. 5, consider $d$'s stuck-at 1 fault test for the circuit's range. $d = 0$ is the activating assignment, and $g_3 = 1$ is its propagating assignment. Since $g_3$ is an OR gate and COAs$(g_3)$ is $\emptyset$, we cannot find any observability assignment for $g_3 = 1$. Thus, the RMAs of $d$'s stuck-at 1 fault test are $\{d = 0, g_3 = 1\}$. However, when $d = 0$, any effect through $g_4$ will be blocked such that $b = 0$ becomes necessary to make $g_3 = 1$ observable at $O_1$, and the effect propagating path is from $g_3$, $c$, $g_2$ to $O_1$. Thus, $b = 0$ is a COA of $g_3$ as we consider $d$'s stuck-at 1 fault test.

To find more RMAs for a given stuck-at fault, we should consider the values of computed RMAs and then recalculate each gate's COAs and NCOAs. For example, consider $d$'s stuck-at 1 fault test in Fig. 5. We have the RMAs $\{d = 0, g_3 = 1\}$. Since $d = 0$ blocks any effect through $g_4$, we can assume that no path from $g_3$ to $g_4$ exists for effect propagation, and then, we recalculate each gate's COAs and NCOAs. The results are shown in Fig. 8. In Fig. 8, we show COAs$(a)$ and NCOAs$(a)$ as disabled. This means that both $a = 0$ and $a = 1$ are unobservable since no path from $a$ to a PO exists. COAs$(b)$ and NCOAs$(b)$ become $\{c = 0\}$, and COAs$(c)$ and NCOAs$(c)$ become $\{b = 0\}$. The intersection and union rules for a disabled set are as follows. The intersection of a set $S$ and a disabled set results in $S$, and the union of $S$ and disabled results in disabled. By these rules, COAs$(g_1)$ become $\{c = 0\}$, and NCOAs$(g_1)$ become disabled. NCOAs$(g_2)$ become $\{c = 0, b = 0\}$. COAs$(g_3)$ become $\{b = 0\}$, and NCOAs$(g_3)$ become $\{c = 0, b = 0\}$. NCOAs$(g_4)$ become $\{b = 0, g_3 = 1\}$.

Again, consider $d$'s stuck-at 1 fault test. Before recalculating each gate's COAs and NCOAs, we have the RMAs $\{d = 0, g_3 = 1\}$. Since COAs$(g_3)$ become $\{b = 0\}$, $b = 0$ is necessary to make $g_3 = 1$ observable at $O_1$. Furthermore, $c = 0$ is necessary to make $b = 0$ observable at $O_1$ since $c = 0$ is a COA and an NCOA of $b$. Finally, the RMAs for $d$'s stuck-at 1 fault test are $\{d = 0, g_3 = 1, b = 0, c = 0\}$. After performing logic implication, we can find a conflict on $g_3$ because $b = 0$ and $c = 0$ imply $g_3 = 0$. Thus, $d$ is a range-redundant PI, and it can be replaced with a constant 1. In this example, it is clear that we cannot identify $d$ as a range-redundant PI if we do not consider the values of computed RMAs and recalculate each gate's COAs and NCOAs.

---

**void Range_Redundant_PI** (**Circuit** $C$, **PI** $pi$, **Fault** $v$)
$R$ denotes the set of RMAs. Initially, $R$ = Ø.

1. Compute Type 1 assignments:
   (a) Add $pi = \bar{v}$ to $R$.
   (b) Compute propagating assignments of $pi$ and add them to $R$.
2. Compute Type 2 assignments using **Type2_Assignment**($C$, $R$).
3. Perform logic implications to learn more RMAs.
   (a) If there exists a conflict in $R$, replace $pi$ with a constant $v$ and **return**.
4. Re-perform step 2 while considering the value assignments in $R$.
5. Re-perform logic implications to learn more RMAs.
   (a) If there exists a conflict or *disabled* assignment in $R$, replace $pi$ with a constant $v$.

---

Fig. 9. Range-redundant PI identification algorithm.

---

**Set Type2_Assignment**(**Circuit** $C$, **Set** $R$)

1. For each primary input $i$ in $C$, compute *COAs(i)* and *NCOAs(i)*.
2. For each internal gate $g$ in $C$, compute *COAs(g)* and *NCOAs(g)* using Equation (1) and (2).
3. For each assignment $n = v_n$ in $R$, if $v_n = ocv(n)$, then add *COAs(n)* to $R$, else add *NCOAs(n)* to $R$.
4. **return** $R$.

---

Fig. 10. Type 2 assignment identification algorithm.

## C. Range-Redundant PI Identification Algorithm

Our algorithm to identifying and removing a range-redundant PI is shown in Fig. 9. Given a PI pi and the stuck-at $v$ fault on pi, in step 1, we compute type 1 assignments. They are the activating assignment pi $= \bar{v}$ and the propagating assignments of pi. If pi has a single fan-out, we compute its propagating assignments by setting the side inputs of dominators of pi to input-noncontrolling values. However, if pi has multiple fan-outs, we compute them using the method mentioned in Section IV-A. The method individually considers the propagating path of each fan-out of pi to compute its propagating assignments. In step 2, we compute type 2 assignments using algorithm Type2_Assignment shown in Fig. 10. First, we compute COAs$(i)$ and NCOAs$(i)$ for each PI $i$. Both of them are identical to the propagating assignments of $i$. Next, we compute COAs$(g)$ and NCOAs$(g)$ for each internal gate $g$ using (1) and (2). Finally, we collect RMAs according to the computed COAs and NCOAs. In step 3, we perform logic implications to learn more RMAs. If there exists a conflict in RMAs, we replace pi with a constant $v$ and stop. Otherwise, in step 4, we reperform step 2 to calculate more type 2 assignments by considering the value assignments in RMAs. The method is mentioned in Section IV-B. In step 5, we perform logic implications again to check if there exists a conflict or *disabled* assignment in RMAs. If so, we replace pi with a constant $v$.

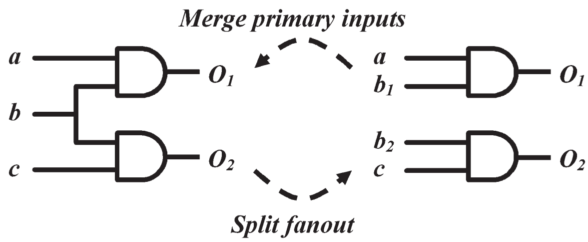



Fig. 11. Example of splitting fan-out and merging PIs.

## V. Range-Preserving Simplification

In Sections III and IV, we present a method for range-redundant PI identification, which is of the basis of range-equivalent circuit minimization. Next, in this section, we first discuss two operations, *split fan-out* and *merge PIs*. These two operations can be used to remove more PIs. We then present our overall algorithm.

### A. Split Fan-Out and Merge PIs

In previous sections, we propose an approach for finding RMAs for a stuck-at fault test on a PI. A shortcoming of this approach is that only a few RMAs can be found when each PI only has a few propagating assignments. To solve this problem, we can change the structure of PIs to potentially generate more propagating assignments for each PI while preserving the circuit's range. *Split fan-out* is an operation to split a PI with multiple fan-outs into two PIs. For example, in Fig. 11, the PI $b$ is split into $b_1$ and $b_2$, and the circuit's range is intact.

Given a combinational circuit, *split fan-out* may create extra output values that do not belong to the original circuit. Let $C$ be the original circuit and $C'$ be the resultant circuit of splitting the fan-out of a PI $a$ into $a_1$ and $a_2$. If the ranges of $C$ and $C'$ are different, there must exist an output value $o$ that is in the range of $C'$ but not in that of $C$. Since $o$ is in the range of $C'$ only, the input vector that can generate $o$ must be the one with $a_1 \neq a_2$. For such an input vector and output value $o$ to exist, there must exist unique assignments to nodes in $C'$. These MAs are the propagating assignments necessary for propagating the effects of $a_1 \neq a_2$ and the observability assignments necessary for making the propagating assignments observable at a PO. The method for finding these MAs is the same as that mentioned in Section III. If these MAs are inconsistent, we can split $a$ into $a_1$ and $a_2$ without changing the circuit's range.

Our algorithm to splitting a given PI pi is shown in Fig. 12. For each fan-out of pi, we first split it into $\mathrm{pi}_1$ and $\mathrm{pi}_2$. In step 1(b), we compute the propagating assignments of $\mathrm{pi}_1$ and $\mathrm{pi}_2$, respectively, and they are RMAs. In step 1(c), we compute type 2 assignments using algorithm Type2_Assignment shown in Fig. 10. In step 1(d), we perform logic implications to learn more RMAs. If there exists a conflict or same value assignments on $\mathrm{pi}_1$ and $\mathrm{pi}_2$ in RMAs, we stop and continue to consider the next fan-out of pi. Otherwise, in step 1(e), we reperform step 1(c) to calculate more type 2 assignments by considering the value assignments in RMAs. In step 1(f), we perform logic implications again to check if there exists a conflict, same value assignments on $\mathrm{pi}_1$ and $\mathrm{pi}_2$, or a *disabled* assignment in RMAs.

---

**void Split_Fanout** (**Circuit** $C$, **PI** $pi$)
$R$ denotes the set of RMAs.

1. For each fanout of $pi$:
   (a) Split the fanout into $pi_1$ and $pi_2$, and initialize $R = \emptyset$.
   (b) Compute propagating assignments of $pi_1$ and $pi_2$ respectively, and add them to $R$.
   (c) Compute Type 2 assignments using **Type2_Assignment**($C$, $R$).
   (d) Perform logic implications to learn more RMAs.
      i. If there exists a conflict or same value assignments on $pi_1$ and $pi_2$ in $R$, **continue**.
   (e) Re-perform step 1.(c) while considering the value assignments in $R$.
   (f) Re-perform logic implications to learn more RMAs.
      i. If there exists a conflict, same value assignments on $pi_1$ and $pi_2$, or a *disabled* assignment in $R$, **continue**.
   (g) Merge $pi_1$ and $pi_2$.

---

Fig. 12. Algorithm of splitting fan-out.

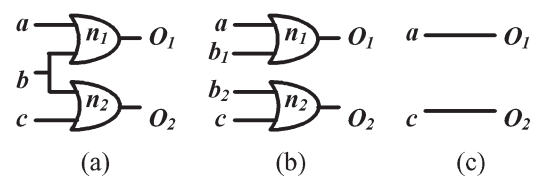



Fig. 13. (a) Stuck-at 0 fault on $b$ is range untestable. (b) Split $b$ into $b_1$ and $b_2$. (c) The resultant circuit by replacing $b_1$ and $b_2$ with a constant 0 in (b).

If so, we stop and continue to consider the next fan-out of pi. Otherwise, in step 1(g), we merge $\mathrm{pi}_1$ and $\mathrm{pi}_2$.

For example, in Fig. 13(a), $b$ is a range-redundant PI and can be replaced with a constant 0. However, our method mentioned in Section III cannot identify that the stuck-at 0 fault on $b$ is range untestable. This is because there is no propagating assignment for $b = 1$. To solve this problem, we can split $b$ into $b_1$ and $b_2$ and then remove them. To determine if $b$ can be split and the circuit's range remains intact, we first split $b$ into $b_1$ and $b_2$, as shown in Fig. 13(b). We then check if there exists an output value $o$ that belongs to the circuit in Fig. 13(b) but not to the circuit in Fig. 13(a). The MAs for $o$ to exist are $\{a = 0, c = 0\}$, which are necessary to propagate the effect of $b_1 \neq b_2$. Type 2 assignments for $o$ to exist are $\{b_1 = 0, b_2 = 0\}$, which are necessary to make $\{a = 0, c = 0\}$ observable. As a result, a conflict occurs. Therefore, $o$ does not exist, and we can split $b$ into $b_1$ and $b_2$ without changing the circuit's range. After splitting $b$, we further identify that both the stuck-at 0 faults on $b_1$ and $b_2$ are range untestable. We then replace them with a constant 0 and obtain a simplified but range-equivalent circuit as shown in Fig. 13(c).

*Merging PIs* can be seen as an inverse operation of split fan-out. Let $C$ be the original circuit with two PIs $a_1$ and $a_2$, and $C'$ be the resultant circuit by merging $a_1$ and $a_2$ into a PI $a$. Similarly, if the ranges of $C$ and $C'$ are different, there must exist an output value $o$ which is in the range of $C$ but not in that of $C'$. We also can compute the MAs for $o$ to exist. If the MAs are inconsistent, merging $a_1$ and $a_2$ does not change the circuit's range. When two range-irredundant PIs can be merged into one PI, the number of PIs is reduced.

**void Merge_PIs** (**Circuit** $C$, **PI** $pi_1$, **PI** $pi_2$)
$R$ denotes the set of RMAs. Initially, $R = \emptyset$.

1. Compute propagating assignments of $pi_1$ and $pi_2$ respectively and add them to $R$.
2. Compute Type 2 assignments using **Type2_Assignment**($C$, $R$).
3. Perform logic implications to learn more RMAs.
   (a) If there exists a conflict or same value assignments on $pi_1$ and $pi_2$ in $R$, merge $pi_1$ and $pi_2$ and then **return**.
4. Re-perform step 2 while considering the value assignments in $R$.
5. Re-perform logic implications to learn more RMAs.
   (a) If there exists a conflict, same value assignments on $pi_1$ and $pi_2$, or a *disabled* assignment in $R$, merge $pi_1$ and $pi_2$.

Fig. 14. Algorithm of merging PIs.

**void Range_Equivalent_Minimization** (**Circuit** $C$)

1. For each primary input $pi$ having multiple fanouts, perform **Split_Fanout**($C$, $pi$).
2. For each primary input $pi$:
   (a) Perform **Range_Redundant_PI**($C$, $pi$, 1).
   (b) If $pi$ is not replaced, perform **Range_Redundant_PI**($C$, $pi$, 0).
3. For each pair of primary inputs $pi_1$ and $pi_2$, perform **Merge_PIs**($C$, $pi_1$, $pi_2$).

Fig. 15. Algorithm of range-equivalent circuit minimization.

Our algorithm to merging two PIs $\mathrm{pi}_1$ and $\mathrm{pi}_2$ is shown in Fig. 14. The algorithm is similar to that of splitting fan-out as shown in Fig. 12. They perform the same checking method but different operations.

### B. Algorithm

We have described the operations for identifying range-redundant PIs, splitting fan-out, and merging PIs. Our overall algorithm based on these three operations is shown in Fig. 15. Given a combinational circuit, we first split PIs that can be split without changing the circuit's range. We then remove all range-redundant PIs. Finally, we merge the PIs that can be merged while preserving the circuit's range.

## VI. Experimental Results

In this section, we provide the experimental results of our algorithm on a set of ISCAS'85 and MCNC combinational benchmarks. The experiments are conducted within a SIS [18] environment on a Sun Blade 2500 workstation with 4-GB memory. The benchmarks are in Berkeley Logic Interchange Format, which is a netlist level design description. For simplicity, we also decompose complex gates into AND2, OR2, and INV gates in the experiments using the *decomp_tech_network* function in SIS.

The experiments include three parts: The first one is to compare the simplification capabilities of our approach and the normalized function method [15], the second one is to present the effect of using different orders to remove range-redundant PIs on the size of a simplified circuit, and the third one is to show the benefits of a simplified circuit to constraint-based random simulation.

TABLE I
Comparison of Our Approach and the Normalized Function Method [15]

| Circuit | $\lvert PO\rvert$ | $\lvert PI\rvert$ | NC | Ours | | | | [15] | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | $\lvert PI_d\rvert$ | $\frac{\lvert PI_d\rvert}{\lvert PI\rvert}\%$ | $NC_d$ | $\frac{NC_d}{NC}\%$ | $\lvert PI_d\rvert$ | $\frac{\lvert PI_d\rvert}{\lvert PI\rvert}\%$ | $NC_d$ |
| cm151a | 2 | 12 | 37 | 11 | 91.67 | 33 | 89.19 | 0 | 0 | 0 |
| cordic | 2 | 23 | 118 | 6 | 26.09 | 24 | 20.34 | 0 | 0 | 0 |
| frg1 | 3 | 28 | 820 | 25 | 89.29 | 814 | 99.27 | 22 | 78.57 | 795 |
| comp | 3 | 32 | 132 | 27 | 84.38 | 114 | 86.36 | 0 | 0 | 0 |
| m2 | 3 | 6 | 16 | 4 | 66.67 | 9 | 56.25 | 0 | 0 | 0 |
| m1 | 3 | 6 | 15 | 3 | 50 | 7 | 46.67 | 0 | 0 | 0 |
| cm85a | 3 | 11 | 58 | 8 | 72.73 | 52 | 89.66 | 0 | 0 | 0 |
| cmb | 4 | 16 | 67 | 3 | 18.75 | 10 | 14.93 | 3 | 18.75 | 9 |
| cm162a | 5 | 14 | 64 | 6 | 42.86 | 18 | 28.13 | 4 | 28.57 | 15 |
| cm163a | 5 | 16 | 66 | 11 | 68.75 | 56 | 84.85 | 10 | 62.5 | 51 |
| i4 | 6 | 192 | 444 | 186 | 96.88 | 432 | 97.3 | 186 | 96.88 | 432 |
| i3 | 6 | 132 | 264 | 126 | 95.45 | 252 | 95.45 | 126 | 95.45 | 252 |
| x2 | 7 | 10 | 73 | 4 | 40 | 17 | 23.29 | 0 | 0 | 0 |
| C432 | 7 | 36 | 215 | 4 | 11.11 | 4 | 1.86 | 0 | 0 | 0 |
| cm138a | 8 | 6 | 40 | 2 | 33.33 | 4 | 10 | 0 | 0 | 0 |
| pcle | 9 | 19 | 90 | 3 | 15.79 | 4 | 4.44 | 0 | 0 | 0 |
| term1 | 10 | 34 | 535 | 10 | 29.41 | 130 | 24.3 | 5 | 14.71 | 77 |
| cu | 11 | 14 | 97 | 5 | 35.71 | 19 | 19.59 | 4 | 28.57 | 4 |
| pm1 | 13 | 16 | 102 | 6 | 37.5 | 39 | 38.24 | 0 | 0 | 0 |
| sct | 15 | 19 | 226 | 2 | 10.53 | 7 | 3.1 | 2 | 10.53 | -5 |
| i1 | 16 | 25 | 78 | 9 | 36 | 17 | 21.79 | 2 | 8 | -10 |
| count | 16 | 35 | 178 | 19 | 54.29 | 146 | 82.02 | 18 | 51.43 | 129 |
| unreg | 16 | 36 | 164 | 20 | 55.56 | 132 | 80.49 | 2 | 5.56 | -23 |
| dalu | 16 | 75 | 1543 | 1 | 1.33 | 304 | 19.7 | 1 | 1.33 | -1699 |
| pcler8 | 17 | 27 | 122 | 9 | 33.33 | 10 | 8.2 | 0 | 0 | 0 |
| c8 | 18 | 28 | 264 | 2 | 7.14 | 35 | 13.26 | 0 | 0 | 0 |
| lal | 19 | 26 | 234 | 8 | 30.77 | 62 | 26.5 | 7 | 26.92 | 10 |
| cc | 20 | 21 | 120 | 1 | 4.76 | 7 | 5.83 | 1 | 4.76 | -4 |
| b9 | 21 | 41 | 202 | 19 | 46.34 | 64 | 31.68 | 10 | 24.39 | -19 |
| C3540 | 22 | 50 | 1254 | 10 | 20 | 10 | 0.8 | 0 | 0 | 0 |
| C880 | 26 | 60 | 440 | 6 | 10 | 69 | 15.68 | 2 | 3.33 | -545 |
| *C1355 | 32 | 41 | 583 | 1 | 2.44 | 9 | 1.54 | 0 | 0 | 0 |
| cht | 36 | 47 | 300 | 11 | 23.4 | 228 | 76 | 6 | 12.77 | 30 |
| apex7 | 37 | 49 | 342 | 6 | 12.24 | 24 | 7.02 | 0 | 0 | 0 |
| i9 | 63 | 88 | 900 | 13 | 14.77 | 13 | 1.44 | 1 | 1.14 | -3055 |
| i5 | 66 | 133 | 556 | 67 | 50.38 | 424 | 76.26 | 67 | 50.38 | 424 |
| example2 | 66 | 85 | 440 | 22 | 25.88 | 51 | 11.59 | 17 | 20 | -570 |
| i6 | 67 | 138 | 885 | 68 | 49.28 | 701 | 79.21 | 67 | 48.55 | 459 |
| i7 | 67 | 199 | 1093 | 128 | 64.32 | 902 | 82.53 | 128 | 64.32 | 835 |
| *i8 | 81 | 133 | 2196 | 5 | 3.76 | 5 | 0.23 | 7 | 5.26 | -2393 |
| *rot | 107 | 135 | 1365 | 20 | 14.81 | 91 | 6.67 | 11 | 8.15 | -1784 |
| *C7552 | 108 | 207 | 2804 | 75 | 36.23 | 90 | 3.21 | 0 | 0 | 0 |
| *C5315 | 123 | 178 | 2355 | 11 | 6.18 | 68 | 2.89 | 8 | 4.49 | -8461 |
| **C2670** | **140** | **233** | **1073** | **87** | **37.34** | **429** | **39.98** | **3** | **1.29** | **-2362** |
| *i10 | 224 | 257 | 2814 | 26 | 10.12 | 176 | 6.25 | 7 | 2.72 | -13310 |
| Average | | | | | 37.06 | | 36.31 | | 17.32 | |

random simulation.

### A. Comparison of Simplification Capabilities

To show our simplification capabilities, we reimplement the normalized function method [15] for comparison. In the experiments, we repeatedly apply the method [15] to the benchmark circuit until the circuit can no longer be simplified. Additionally, to perform logic implications more completely, recursive learning technique [12] is applied with the recursion depth $= 1$ by our approach in the experiments.

Table I summarizes the experimental results of our approach and that in [15] over the number of reduced PIs and nodes. Column 1 lists the benchmarks. The benchmarks marked with "*" mean that their ranges are enormous and cannot be presented by constructing their characteristic function BDDs. Therefore, they cannot possibly be handled by the range-computation-based methods. Additionally, all benchmarks can be simplified by our approach, but not by that in [15]. Column 2 lists the number of POs in each benchmark |PO|. Column 3 lists the number of PIs in each benchmark |PI|. Column 4 lists the total node counts in each benchmark NC. Columns 5 and 9 list the number of reduced PIs $|\mathrm{PI}_d|$ of these two approaches, respectively. Columns 6 and 10 list the reduction percentage of the number of PIs. Column 7 lists the number of reduced nodes by our approach $\mathrm{NC}_d$. Column 8 lists the reduction percentage of the number of nodes. Column 11 lists the number of reduced nodes by the method [15]. In Column 11, a negative value indicates that the node count increases after the benchmark is simplified. For example, the *C2670* benchmark has 140 POs, 233 PIs, and 1073 nodes, and our approach can remove 87 PIs and 429 nodes without influencing its range. The reduction percentage is 37.34% in terms of the number of PIs and 39.98% in terms of the node counts. However, the method in [15] only can remove three PIs, and the reduction percentage is 1.29%. Furthermore, the simplified benchmark has more node count than the original one with the increase of 2362.

Table I shows that the node counts of some benchmarks simplified by the method in [15] increase. This is because [15] first partitions an $n$-output circuit into $n$ subcircuits and then simplifies each subcircuit individually; a lot of nodes are duplicated in different subcircuits. When we merge all subcircuits to obtain the simplified circuit by connecting the same PIs, there may exist many redundant nodes in it. Thus, it is possible that the method in [15] reduces the number of PIs but increases the number of nodes when simplifying a benchmark. In the experiments, we apply the *com_redundancy_removal* function in SIS to remove the redundant nodes in the benchmarks simplified by the approach in [15] and then measure the node counts as shown in Column 11.

According to Table I, our approach provides more reduction in terms of the number of PIs in most benchmarks as compared to that of the method in [15]. In particular, 17 benchmarks which cannot be simplified by the approach in [15] can be simplified by our approach. On average, our approach can reduce 37.06% of PIs, but that in [15] only can reduce 17.32% of PIs. As a result, our approach reduces the PIs by 19.74% more than that of [15] on average. Furthermore, our approach gets an average of 36.31% reduction in terms of the node counts.

Note that the experimental results of our approach and that in [15] are variant from one benchmark to another. The results strongly depend on the functionalities of circuits. Since both approaches only eliminate minterms with repeated outputs and do not rearrange the range of each circuit, the amount of feasible reduction could be limited. It is an inherent limitation of both approaches. On the other hand, a 100% reduction of PIs is impossible since each output combination in the range has at least one corresponding input combination. Thus, a reasonable objective of this paper is to obtain a simplified circuit with the same number of PIs and POs. The reason is that an $n$-output circuit has at most $2^n$ output combinations and hence requires at least $2^n$ corresponding minterms ($n$-input). For all benchmarks shown in Table I, if the suggested objective is achieved, the average reduction percentage is 48.74% in terms of the number of PIs. The results of our approach are significantly closer to this objective than that of [15].

TABLE II
EXPERIMENTAL RESULTS OF SIMPLIFYING THE *cu* BENCHMARK USING DIFFERENT ORDERS TO REMOVE RANGE-REDUNDANT PIs

| PI | Fault type | #RMAs Order 1 ($a\sim o$) | #RMAs Order 2 ($o\sim a$) |
|---|---|---|---|
| *a* | s-a-1 | 15 | 18 |
| *a* | s-a-o | 15 | 18 |
| *b* | s-a-1 | 16 | 14 |
| *b* | s-a-o | 15 | 11 |
| *c* | s-a-1 | 7 | 7 |
| *c* | s-a-o | 28 | 28 |
| *d* | s-a-1 | 1 | 1 |
| *d* | s-a-o | 46 | 46 |
| *e* | s-a-1 | 12 | 12 |
| *e* | s-a-o | 18 | 18 |
| *f* | s-a-1 | 28 | 25 |
| *f* | s-a-o | 6 | 6 |
| *g* | s-a-1 | 7 | 7 |
| *g* | s-a-o | 1 | 1 |
| *i* | s-a-1 | 62 | 63 |
| *i* | s-a-o | *59 | *57 |
| *j* | s-a-1 | 75 | *63 |
| *j* | s-a-o | 79 | — |
| *k* | s-a-1 | 75 | *79 |
| *k* | s-a-o | 83 | — |
| *l* | s-a-1 | 75 | 74 |
| *l* | s-a-o | 84 | *85 |
| *m* | s-a-1 | 75 | 76 |
| *m* | s-a-o | 84 | *88 |
| *n* | s-a-1 | 43 | 43 |
| *n* | s-a-o | 54 | 55 |
| *o* | s-a-1 | 12 | 13 |
| *o* | s-a-o | 15 | 15 |

### B. Effect of Different Orders of Removed PIs

The number of RMAs that we can compute is heavily related to a circuit's structure. In particular, when each PI only has a few propagating assignments, each internal gate will have a few observability assignments as well, and hence, the number of RMAs that we can compute will be small. Additionally, the number of computed RMAs significantly affects our capability of identifying range-redundant PIs. As a result, using different orders to remove range-redundant PIs may result in different numbers of RMAs that we can compute and may even affect the size of a simplified circuit.

We conducted the experiments to show this phenomenon. For each benchmark listed in Table I, we use two different orders to simplify it separately. Order 1 is from the first PI to the last one, and Order 2 is from the last PI to the first one. For each PI, we first set stuck-at 1 fault on it and check if it is range untestable. If so, we remove it and then check the next PI. Otherwise, we set stuck-at 0 fault on it.

Table II summarizes the experimental results of the *cu* benchmark on the number of computed RMAs and the removed PIs. Column 1 lists the PIs. Column 2 lists the type of stuck-at fault. s-a-1 denotes stuck-at 1 fault, and s-a-0 denotes stuck-at 0 fault. Columns 3 and 4 list the number of computed RMAs using Orders 1 and 2, respectively. The numbers of RMAs marked with "*" mean that there exists a conflict among them and the fault with respect to the PI is range untestable. "—" in column 4 means that the PI has been identified and that its stuck-at 1 fault is range untestable, and hence, we do not set stuck-at 0 fault on it (we check stuck-at 1 fault first in

TABLE III
EXPERIMENTAL RESULTS OF SIMULATING A BENCHMARK AND ITS SIMPLIFIED ONE FOR 100 s

| Circuit | \|PO\| | \|PI\|/\|PI$_s$\| | NC/NC$_s$ | Outputs | | |
|---|---|---|---|---|---|---|
| | | | | Original | Simplified | Ratio (S/O) |
| term1 | 10 | 34/24 | 535/405 | 530 | 627 | 1.18 |
| sct | 15 | 19/17 | 226/219 | 8262 | 8340 | 1.01 |
| i1 | 16 | 25/16 | 78/61 | 1878 | 2418 | 1.29 |
| count | 16 | 35/16 | 178/32 | 54326 | 65536 | 1.21 |
| unreg | 16 | 36/16 | 164/32 | 61853 | 65536 | 1.06 |
| dalu | 16 | 75/74 | 1543/1239 | 2856 | 3516 | 1.23 |
| pcler8 | 17 | 27/18 | 122/112 | 641 | 711 | 1.11 |
| c8 | 18 | 28/26 | 264/229 | 71995 | 91457 | 1.27 |
| lal | 19 | 26/18 | 234/172 | 8835 | 10432 | 1.18 |
| cc | 20 | 21/20 | 120/113 | 17398 | 27850 | 1.60 |
| b9 | 21 | 41/22 | 202/138 | 5498 | 9511 | 1.73 |
| C3540 | 22 | 50/40 | 1254/1244 | 18155 | 18330 | 1.01 |
| C880 | 26 | 60/54 | 440/371 | 42586 | 48092 | 1.13 |
| C1355 | 32 | 41/40 | 583/574 | 55821 | 56735 | 1.02 |
| **cht** | **36** | **47/36** | **300/72** | **69884** | **876775** | **12.55** |
| apex7 | 37 | 49/43 | 342/318 | 59260 | 63143 | 1.07 |
| i9 | 63 | 88/75 | 900/887 | 36488 | 36885 | 1.01 |
| i5 | 66 | 133/66 | 556/132 | 76585 | 554142 | 7.24 |
| example2 | 66 | 85/63 | 440/389 | 94149 | 104433 | 1.11 |
| i6 | 67 | 138/70 | 885/184 | 35017 | 148212 | 4.23 |
| i7 | 67 | 199/71 | 1093/191 | 35618 | 287425 | 8.07 |
| i8 | 81 | 133/128 | 2196/2191 | 11507 | 12125 | 1.05 |
| rot | 107 | 135/115 | 1365/1274 | 26805 | 29050 | 1.08 |
| C7552 | 108 | 207/132 | 2804/2714 | 9109 | 9535 | 1.05 |
| C5315 | 123 | 178/167 | 2355/2287 | 11931 | 12511 | 1.05 |
| **C2670** | **140** | **233/146** | **1073/644** | **37264** | **68993** | **1.85** |
| i10 | 224 | 257/231 | 2814/2638 | 9759 | 11168 | 1.14 |
| Average | | | | | | 2.20 |

both orders). The experimental results show that using different orders results in a different number of computed RMAs for most PIs and results in different simplified circuits. When using Order 2 to simplify the *cu* benchmark, we can remove five PIs but only one PI using Order 1.

Although the order of removed PIs affects the simplified result of the *cu* benchmark, it is not true for all benchmarks. Only 6 out of 45 benchmarks listed in Table I have different simplified results when applying Orders 1 and 2 to simplify them, respectively. They are *cm151a*, *cu*, *b9*, *rot*, *C2670,* and *i10*.

In this paper, we do not consider the order of removed PIs to simplify a circuit. This is because it is difficult to identify which PI, if removed first, can have more range-redundant PIs later. Thus, as we identify a range-redundant PI, we remove it right away. Considering the order of removed PIs to obtain better simplifications could be our future work.

### C. Generation of Output Combinations

In Section I, we mentioned that a constraint circuit can be replaced by a simplified but range-equivalent one to accelerate the verification process. We conducted the experiments to show the advantages of this work for constraint-based random simulation.

First, we serve each benchmark as a constraint circuit, and its circuit simplified by our approach is a simplified constraint circuit. We then simulate each circuit by repeatedly applying random input patterns and record the number of generated output combinations. For comparison of various aspects, we conducted two experiments as follows.

1) *Same simulation time*. We simulate each circuit and its simplified version for 100 s to compare the number of generated output combinations.
2) *Same number simulation patterns*. We simulate each circuit and its simplified version for 100 000 random input patterns to compare the number of generated output combinations and the simulation time.

The experimental results are summarized in Tables III and IV, respectively. Since the smaller (fewer number of PIs or POs) circuits can easily apply all input combinations or produce all output combinations, they are not considered in the experiments.

In Table III, column 1 lists the benchmarks. Column 2 lists the number of POs in each benchmark. Column 3 lists the number of PIs, and column 4 lists the number of nodes in each benchmark and its simplified one. Columns 5 and 6 list the number of generated output combinations by these two range-equivalent benchmarks, respectively. Column 7 lists the ratio of generated output combinations. For example, the *C2670* benchmark originally has 140 POs, 233 PIs, and 1073 nodes, and can generate 37 264 output combinations for 100-s simulation time. After being simplified by our approach, the *C2670* benchmark has 146 PIs and 644 nodes. For the same simulation time, it can generate 68 993 output combinations, and the ratio normalized with respect to that of the original benchmark is 1.85.

The experimental results in Table III show that each simplified benchmark generates more output combinations than the original one. In particular, the ratio of the *cht* benchmark is up to 12.55. On average, a simplified benchmark generates the output combinations 120% more than that of the original one for 100-s random simulation time. Let us consider constraint-based random simulation. For the same simulation time, since the simplified constraint circuit can generate more output combinations, it increases the probability of detecting design errors. Thus, replacing a constraint circuit with a simplified one can accelerate the verification process.

Table IV shows the results of the second experiment. Unlike Table III, columns 6 and 9 list the CPU time measured in seconds of each benchmark and its simplified one, respectively. Column 8 lists the improvement in terms of the generated output combinations of each simplified benchmark as compared to its original one. Furthermore, column 10 lists the reduction in terms of the CPU time. For example, the *i6* benchmark originally has 67 POs, 138 PIs, and 885 nodes. After simulating 100 000 random patterns, it generates 74 997 output combinations, and the simulation time is 214.51 s. Furthermore, the simplified *i6* benchmark has 70 PIs and 184 nodes. For the same number of simulation patterns, it can exactly generate 100 000 output combinations and only requires 68.13 s. As compared with the original one, the simplified *i6* benchmark can save 68% CPU time and obtain 33% improvement in terms of the generated output combinations.

Table IV shows that each simplified benchmark requires less CPU time than its original one. Furthermore, most simplified benchmarks generate the same number or more

TABLE IV
EXPERIMENTAL RESULTS OF SIMULATING A BENCHMARK AND ITS SIMPLIFIED ONE FOR 100 000 RANDOM PATTERNS

| | | | | Original | | Simplified | | | |
|---|---|---|---|---|---|---|---|---|---|
| Circuit | $\lvert PO\rvert$ | $\lvert PI\rvert/\lvert PI_s\rvert$ | $NC/NC_s$ | Outputs | CPU | Outputs | Impr.(%) | CPU | Redu.(%) |
| term1 | 10 | 34/24 | 535/405 | 565 | 152.28 | 626 | 10.796 | 112.40 | 26.19 |
| sct | 15 | 19/17 | 226/219 | 7650 | 60.61 | 7744 | 1.229 | 58.52 | 3.45 |
| i1 | 16 | 25/16 | 78/61 | 1607 | 15.39 | 2377 | 47.915 | 12.60 | 18.13 |
| count | 16 | 35/16 | 178/32 | 34951 | 43.80 | 51401 | 47.066 | 4.56 | 89.59 |
| unreg | 16 | 36/16 | 164/32 | 44706 | 39.63 | 51302 | 14.754 | 4.56 | 88.49 |
| **dalu** | **16** | **75/74** | **1543/1239** | **12834** | **509.92** | **11748** | **-8.462** | **385.38** | **24.42** |
| pcler8 | 17 | 27/18 | 122/112 | 559 | 28.31 | 591 | 5.725 | 26.33 | 6.99 |
| c8 | 18 | 28/26 | 264/229 | 53428 | 69.62 | 61141 | 14.436 | 58.41 | 16.10 |
| **lal** | **19** | **26/18** | **234/172** | **7069** | **59.75** | **6250** | **-11.586** | **43.44** | **27.30** |
| cc | 20 | 21/20 | 120/113 | 7326 | 28.20 | 11744 | 60.306 | 23.17 | 17.84 |
| b9 | 21 | 41/22 | 202/138 | 4473 | 49.53 | 7928 | 77.241 | 33.69 | 31.98 |
| C3540 | 22 | 50/40 | 1254/1244 | 47385 | 387.42 | 47804 | 0.884 | 376.55 | 2.81 |
| **C880** | **26** | **60/54** | **440/371** | **47113** | **115.59** | **46305** | **-1.715** | **95.53** | **17.35** |
| C1355 | 32 | 41/40 | 583/574 | 99996 | 179.33 | 99999 | 0.003 | 176.85 | 1.38 |
| cht | 36 | 47/36 | 300/72 | 49950 | 72.38 | 100000 | 100.200 | 9.78 | 86.49 |
| apex7 | 37 | 49/43 | 342/318 | 50417 | 85.04 | 50465 | 0.095 | 80.17 | 5.73 |
| i9 | 63 | 88/75 | 900/887 | 89660 | 247.77 | 89895 | 0.262 | 245.21 | 1.03 |
| i5 | 66 | 133/66 | 556/132 | 100000 | 132.63 | 100000 | 0 | 18.07 | 86.38 |
| example2 | 66 | 85/63 | 440/389 | 95372 | 102.36 | 95435 | 0.066 | 91.31 | 10.80 |
| **i6** | **67** | **138/70** | **885/184** | **74997** | **214.51** | **100000** | **33.339** | **68.13** | **68.24** |
| i7 | 67 | 199/71 | 1093/191 | 100000 | 283.09 | 100000 | 0 | 35.25 | 87.55 |
| i8 | 81 | 133/128 | 2196/2191 | 90593 | 772.30 | 90613 | 0.022 | 752.10 | 2.62 |
| rot | 107 | 135/115 | 1365/1274 | 100000 | 375.28 | 100000 | 0 | 346.27 | 7.73 |
| C7552 | 108 | 207/132 | 2804/2714 | 100000 | 1072.70 | 100000 | 0 | 1026.48 | 4.31 |
| C5315 | 123 | 178/167 | 2355/2287 | 100000 | 811.98 | 100000 | 0 | 785.18 | 3.30 |
| C2670 | 140 | 233/146 | 1073/644 | 100000 | 269.15 | 100000 | 0 | 145.70 | 45.87 |
| i10 | 224 | 257/231 | 2814/2638 | 100000 | 1007.47 | 100000 | 0 | 883.69 | 12.29 |
| Average | | | | | | | 14.540 | | 29.42 |

output combinations. Compared to the original benchmark, a simplified one saves an average of 29.42% of the CPU time and obtains an average of 14.54% improvement in terms of the generated output combinations for 100 000 random patterns.

Let us consider the differences between a circuit $C$ and its simplified one $C'$. Since $C'$ is obtained by simplifying $C$, $C'$ must have fewer PIs and nodes than $C$. Thus, the simulation time of $C'$ must be less than that of $C$ as well. Additionally, since $C'$ has fewer PIs, the number of all input combinations of $C'$ is less than that of $C$. When we apply the same number of input patterns to $C$ and $C'$, it is more possible that $C'$ generates more output combinations than $C$. In Table IV, 17 out of 27 simplified benchmarks really generate more output combinations than their original ones. However, although simplifying a circuit can reduce the number of all input combinations, we cannot ensure that the probability for each output combination to be generated will increase. It is possible that the probabilities of many output combinations decrease and only a few output combinations' probabilities increase. This phenomenon may cause a simplified circuit to generate fewer output combinations than the original one for the same number of simulation patterns. In Table IV, the simplified benchmarks of *dalu*, *lal*, and *C880* are such examples that generate fewer output combinations than their original ones. Nevertheless, they still generate more output combinations for the same simulation time as shown in Table III.

Next, we show the experimental results of performing random simulation using a simplified constraint circuit over the number of required input patterns and CPU time. In the experiments, we serve the *C2670* benchmark and its simplified one by our approach as the constraint circuits. Additionally, we select some benchmarks in Table I as DUV. The method of applying the input patterns generated by the constraint circuit to a DUV is as follows. We assume that a DUV has $n$ PIs. Since *C2670* only has 140 POs, if $n \leq 140$, only the first $n$ POs of the constraint circuit will drive the DUV. Conversely, if $n > 140$, we additionally apply the values of the first $(n-140)$ POs of the constraint circuit to drive the last $(n-140)$ PIs of the DUV.

In the experiments, for each DUV, we first inject an error to it by adding an INV to the fan-out of a randomly selected gate. Next, we iteratively apply random patterns to the constraint circuit to generate legal patterns for the DUV. Finally, the verification process stops when the DUV outputs a wrong output value. We repeatedly run 100 iterations for each DUV and measure the average results over the number of required input patterns and CPU time in seconds. The experimental results are shown in Table V.

In Table V, column 1 lists the benchmarks. Columns 2 and 3 list the average number of required input patterns and CPU time $AP_o$ and $AT_o$, respectively, with respect to the original constraint circuit. Columns 4–7 list the results with respect to the simplified constraint circuit. Among them, the average number of required input patterns $AP_s$ and the reduction percentage as compared to $AP_o$ appear in columns 4 and 5. Furthermore, the average CPU time $AT_s$ and the reduction percentage as

TABLE V
EXPERIMENTAL RESULTS OF PERFORMING RANDOM SIMULATION USING *C2670* AND ITS SIMPLIFIED CIRCUIT AS THE CONSTRAINT CIRCUITS

| | Original | | Simplified | | | |
|---|---|---|---|---|---|---|
| Circuit | $AP_o$ | $AT_o$ | $AP_s$ | Redu.(%) | $AT_s$ | Redu.(%) |
| frg1 | 1273.41 | 12.70 | 882.28 | 30.72 | 5.90 | 53.53 |
| rot | 30.27 | 0.44 | 22.93 | 24.25 | 0.26 | 41.57 |
| i3 | 99.95 | 0.50 | 87.47 | 12.49 | 0.23 | 54.82 |
| i4 | 201.27 | 1.25 | 165.38 | 17.83 | 0.56 | 54.81 |
| i5 | 18.43 | 0.14 | 5.16 | 72.00 | 0.02 | 82.04 |
| i6 | 7.83 | 0.08 | 3.47 | 55.68 | 0.02 | 69.57 |
| i7 | 2.94 | 0.04 | 2.69 | 8.50 | 0.02 | 33.60 |
| i8 | 368.62 | 8.78 | 6.91 | 98.13 | 0.14 | 98.37 |
| C432 | 6.77 | 0.04 | 5.96 | 11.96 | 0.02 | 55.00 |
| C499 | 17.63 | 0.17 | 15.75 | 10.66 | 0.10 | 40.53 |
| C880 | 8.64 | 0.06 | 6.72 | 22.22 | 0.03 | 53.26 |
| C1355 | 23.19 | 0.20 | 9.32 | 59.81 | 0.05 | 73.76 |
| C1908 | 52.20 | 0.49 | 32.94 | 36.90 | 0.21 | 58.10 |
| C2670 | 32.39 | 0.38 | 28.30 | 12.63 | 0.24 | 37.60 |
| C3540 | 26.91 | 0.40 | 25.69 | 4.53 | 0.29 | 27.05 |
| C5315 | 4.31 | 0.11 | 4.01 | 7.96 | 0.09 | 20.00 |
| C6288 | 1.65 | 0.05 | 1.57 | 4.84 | 0.04 | 16.34 |
| C7552 | 151.33 | 4.79 | 12.03 | 92.54 | 0.32 | 93.30 |
| Average | | | | 32.37 | | 53.51 |

compared to $AT_o$ appear in columns 6 and 7. For example, consider the *frg1* benchmark. When we use the original *C2670* benchmark as the constraint circuit, on average, we require 1273.41 input patterns and 12.70 s to detect the error. However, when we use the simplified *C2670* benchmark as the constraint circuit, on average, we only require 882.28 input patterns and 5.90 s. The reductions are 30.72% in terms of the number of required input patterns and 53.53% in terms of the CPU time.

Table V shows that using a simplified constraint circuit can accelerate the random simulation process for each benchmark on average. Additionally, for all benchmarks, we obtain an average of 32.37% reduction in terms of the number of required input patterns and 53.51% reduction in terms of the CPU time.

## VII. CONCLUSION

In this paper, we propose a new method for simplifying a circuit while preserving its output range. This is the first work that introduces the concept of a stuck-at fault test on a PI for a circuit's range and proposes a procedure to determine if a given fault is range untestable. We also show that a range-untestable fault on a PI indicates that this PI is range redundant. Furthermore, a range-redundant PI can be replaced with a constant value without changing the circuit's range. Since our method does not perform range computation, iteratively identifying and removing the range-redundant PIs instead, the method can handle large circuits that the range-computation-based methods cannot.

The experimental results show that our approach gets 36.31% reduction in terms of the node counts on average. Furthermore, as compared with the normalized function method [15], a non-BDD-based method, our approach provides an average of 19.74% more reduction in the number of PIs. Additionally, each benchmark simplified by our approach can generate more output combinations than the nonsimplified one for the same random simulation time. This work certainly can be applied to the application of constraint-based random simulation to accelerate the verification process.

## REFERENCES

[1] K. A. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "Multi-level logic minimization using implicit don't cares," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 7, no. 6, pp. 723–740, Jun. 1988.
[2] J. Baumgartner, "Automatic structural abstraction techniques for enhanced verification," Ph.D. dissertation, Univ. Texas, Austin, TX, Dec. 2002. chapter 7.
[3] J. Baumgartner and H. Mony, "Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies," in *Proc. Int. Conf. Correct Hardware Design Verification Methods*, 2005, pp. 222–237.
[4] C. L. Berman and L. H. Trevillyan, "Functional comparison of logic designs for VLSI circuits," in *Proc. Int. Conf. Comput.-Aided Des.*, 1989, pp. 456–459.
[5] D. Bostick, G. D. Hachtel, R. Jacoby, M. R. Lightner, P. Moceyunas, C. R. Morrison, and D. Ravenscroft, "The boulder optimal logic design system," in *Proc. Int. Conf. Comput.-Aided Des.*, 1987, pp. 62–65.
[6] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.
[7] C. W. Jim Chang, M. F. Hsiao, and M. M. Sadowska, "A new reasoning scheme for efficient redundancy addition and removal," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 22, no. 7, pp. 945–952, Jul. 2003.
[8] S. C. Chang, L. P. P. P. Van Ginneken, and M. M. Sadowska, "Circuit optimization by rewiring," *IEEE Trans. Comput.*, vol. 48, no. 9, pp. 962–970, Sep. 1999.
[9] T. Kirkland and M. R. Mercer, "A topological search algorithm for ATPG," in *Proc. Des. Autom. Conf.*, 1987, pp. 502–508.
[10] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proc. Des. Autom. Conf.*, 1997, pp. 263–268.
[11] J. H. Kukula and T. R. Shiple, "Building circuits from relations," in *Proc. Int. Conf. Comput. Aided Verification*, 2000, pp. 131–143.
[12] W. Kunz and D. K. Pradhan, "Recursive learning: An attractive alternative to the decision tree for test generation in digital circuits," in *Proc. Int. Test Conf.*, 1992, pp. 816–825.
[13] Y. Matsunaga, "An efficient equivalence checker for combinational circuits," in *Proc. Des. Autom. Conf.*, 1996, pp. 629–634.
[14] I. H. Moon, H. H. Kwak, J. Kukula, T. Shiple, and C. Pixley, "Simplifying circuits for formal verification using parametric representation," in *Proc. Int. Conf. Formal Methods Comput.-Aided Design*, 2002, pp. 52–69.
[15] J. Moondanos, C. H. Seger, Z. Hanna, and D. Kaiss, "CLEVER: Divide and conquer combinational logic equivalence VERification with false negative elimination," in *Proc. Int. Comput. Aided Verification Conf.*, 2001, pp. 131–143.
[16] H. Savoj, R. K. Brayton, and H. J. Touati, "Extracting local don't cares for network optimization," in *Proc. Int. Conf. Comput.-Aided Des.*, 1991, pp. 514–517.
[17] M. H. Schulz and E. Auth, "Advanced automatic test pattern generation and redundancy identification techniques," in *Proc. Int. Fault-Tolerant Comput. Symp.*, 1988, pp. 30–35.
[18] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Electron. Res. Lab., Univ. California, Berkeley, CA, May 1992. Tech. Rep. UCB/ERL M92/41.
[19] K. Shimizu and D. L. Dill, "Deriving a simulation input generator and a coverage metric from a formal specification," in *Proc. Des. Autom. Conf.*, 2002, pp. 801–806.
[20] J. Yuan, K. Albin, A. Aziz, and C. Pixley, "Constraint synthesis for environment modeling in functional verification," in *Proc. Des. Autom. Conf.*, 2003, pp. 296–299.
[21] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz, "Modeling design constraints and biasing in simulation using BDDs," in *Proc. Int. Conf. Comput.-Aided Des.*, 1999, pp. 584–589.
[22] F. Zaraket, J. Baumgartner, and A. Aziz, "Scalable compositional minimization via static analysis," in *Proc. IEEE Int. Conf. Comput.-Aided Des.*, 2005, pp. 1060–1067.

**Yung-Chih Chen** received the B.S. and M.S. degrees in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 2003 and 2005, respectively, where he is currently working toward the Ph.D. degree in the Department of Computer Science.

His research interests include logic synthesis and design verification.

**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. His research interests include logic synthesis, design verification, and very large scale integration testing.