Random Generation of Test Instances for Logic Optimizers

Kazuo Iwama Kensuke Hino
Department of Computer Science and Communication Engineering
Kyushu University, Hakozaki, Fukuoka 812, Japan
{iwama, hinoken}@csce.kyushu-u.ac.jp

Abstract **The attempt of using random test circuits for evaluating the performance of logic optimizers like SIS is apparently new. To generate "reasonable" random circuits, we propose the random applications of several transformation rules to an initial circuit instead of the obvious method, random placement of connections. A preliminary experiment has been conducted on SIS's responses against such random circuits. SIS shows considerably different performances for different circuits generated from the same original circuit.**

1. Introduction

There have been millions of papers on logic optimizers and also have been even more papers on generating test-cases for detecting faults of logic circuits. However, there have been few papers on how to generate test-cases for *evaluating the performance of algorithms* for logic optimizers. In this paper, we present a new way of providing those test-cases, a random generation of test circuits, for logic optimizers such as MIS [1], SIS [6] and Transduction Method [4].

Since it is little meaningful for this kind of algorithms to use the worst-case upper bound such as $O(n2^{n^{(1+\alpha)}})$, we are forced to depend on so-called the empirical performance-evaluation. To do so, there has been virtually no other way than using benchmarks; for example, MCNC[8] is the most standard benchmark set that includes more than 70 combinational multi-level circuits (and also other types of circuits). Those benchmark circuits are carefully selected with a wide variety and appear to have been accepted in the field. Even so, it is still true that we cannot give reasonable answers to the following naive question: Is the algorithm that is good for the benchmarks good for every circuit? It is also hard to figure out the general tendency of the performance when the size of instances increases. Moreover, we cannot even deny the possibility of cheating, or unnatural tune-ups of the algorithms only for benchmarks, in theory.

In these circumstances, it is quite reasonable to use random instances as well as benchmarks. That is actually very common in many graph problems [7] and in the satisfiability problem for CNF predicates (SAT) [3, 5]. In the case of (multi-level) logic circuits, however, there is no obvious way of generating random circuits. Consider, for example, the following way motivated by the standard generation of random graphs: (i) We first introduce some number of logic gates, (ii) and then between each pair of the input of one gate and the output of another, a connection is drawn with some probability. Unfortunately, it is questionable if such "circuits" can be accepted as logic circuits in the usual sense. Furthermore, if we provide logic optimizers with such random circuits, we would have very little information about what kind of (simplified) circuits should be output by the optimizers.

Our answer to those questions is as follows: Our generator does not make random connections between gates but applies a sequence of *random transformations* into an initial circuit. Each single transformation should not change the logic function realized by the circuit and it should be computed quickly. In more detail: (i) Circuits are described by strings in a usual way. (ii) The equivalent transformation is given as a rewriting rule over the strings, which can be applied in polynomial time. (iii) A complete set of these transformations (rules) is developed, where "complete" means that from any circuit to any (other) equivalent circuit there exits at least one sequence of transformations. (iv) A test circuit is obtained from an initial circuit by applying the rules (principally) at random. Many test circuits of different sizes can be obtained from the single initial circuits. The initial circuit can be good information on "correct" answers of the optimizer. These details will be give in Sections 2-4.

The first version of the generator has been implemented. As a preliminary experiment, the majority function of six variables, the 4-bit adder and so on have been chosen as the initial circuits. A number of different test circuits have been generated from each initial circuit. For those test circuits SIS produced the simplified circuits which are considerably different, twice to four times differ-

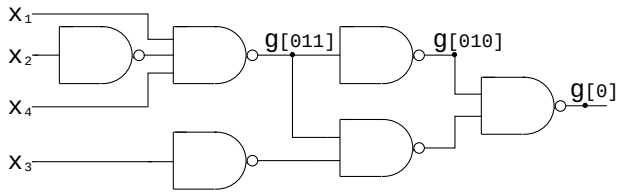
ent, in size and depth. This suggests that our motivation and goal are not bad and the project has the promising future.

2. Definition of Circuits

Within this paper we restrict ourselves to logic circuits using only NAND gates of unlimited fan-in and unlimited fan-out. Circuits of AND, OR and NOT gates have been also discussed under the same framework [2]. Restriction of fan-in and fan-out will be an important future research. A (NAND) circuit is given as a set of equations, e.g., as follows:

$$g[0] = (g[010], (g[011], (x_3)))$$
$$g[010] = (g[011])$$
$$g[011] = (x_1, (x_2), x_4)$$

Namely, a circuit is divided into one or more subcircuits such as $g[01]$, $g[010]$ and $g[011]$. This circuit is illustrated as follows using conventional diagrams.



Definition 1. A *partial circuit* (*p-circuit*) is a string over alphabets $\{\mathbf{0}, \mathbf{1}, 0, 1, x, g, (,), [,], , \}$ defined recursively as follows:

(1) **0** and **1** are p-circuits.

(2) $x[\ell]$ is a p-circuit where ℓ is a string over $\{0,1\}$,i.e., $\ell \in \{0,1\}^*$.

(3) $g[\ell]$ is a p-circuit where $\ell \in \{0,1\}^*$.

(4) Suppose that $S_1, S_2, \cdots, S_m (m \geq 1)$ are p-circuits. Then $(S_1, S_2, \cdots, S_m)$ is also a p-circuit.

0 and **1** denote logical false and true, respectively. To avoid confusion, we use different symbols 0 and 1 for binary strings used in (2) and (3). $x[\ell]$ is an input variable. Again for simplicity we assume that if n variables are needed then those are $x[1], x[10], x[11], \cdots, x[B_n]$ (B_n is the binary number for n). $x[B_i]$ may be denoted by x_i if no confusion occurs. $g[\ell]$ is called a *label*.

Definition 2. A *proper circuit* is a set $\{W_1, W_2, \cdots, W_t\}$, where:

(1) Each W_i is a string of the form

$$g[\ell] = \text{p-circuit},$$

which is called the *definition* of p-circuit $g[\ell]$.

(2) If a label $g[\ell]$ appears in some p-circuit, then its definition must exist.

(3) The definition of p-circuit $g[\ell]$ must be unique for each p-circuit.

In this paper all circuits are proper, so proper circuits are simply called circuits. Suppose that a circuit C has n inputs and m outputs. Then C includes $x_1, \cdots, x_n$ and, without loss of generality, $g[B_0], g[B_1], \cdots, g[B_m]$ as C's outputs. (Namely, $g[0]$ must exist if C is a circuit of single output under this rule.)

3. Transformation Rules

A *transformation rule*, or simply a *rule*, is denoted by $g \Longrightarrow h$. The following set of rules are denoted by $\Re$.

(1) $(1) \Longleftrightarrow \mathbf{0}$ **(2)** $(0) \Longleftrightarrow \mathbf{1}$
(3) $(x,x) \Longleftrightarrow (x)$ **(4)** $(x,(x)) \Longleftrightarrow \mathbf{1}$
(5) $x,((y,z)) \Longleftrightarrow x,y,z$ **(6)** $x,y \Longleftrightarrow y,x$
(7) $(x,\mathbf{1}) \Longleftrightarrow (x)$ **(8)** $((x)) \Longleftrightarrow x$
(9) $(x,(y,z)) \Longleftrightarrow (((x,(y)),(x,(z))))$

(10) If $g[\ell] = f$ is the definition of p-circuit $g[\ell]$ then $g[\ell] \Longleftrightarrow f$.

(11) If $g[\ell]$ is neither an output of the circuit nor does not appear in right-hand side of the definition of any p-circuit, then the definition of $g[\ell]$ is removed. (This rule is called a *deletion*.)

(12) If the definition of label $g[\ell]$ does not exist in the right-hand side of any definition, then $g[\ell] = C$ is added, where C can be any p-circuit whose length is bounded polynomially. This rule is called a *creation*.

$f \Longleftrightarrow g$ stands for $f \Longrightarrow g$ and $g \Longrightarrow f$. f (and g also) is a string (not necessarily a p-circuit, see e.g., rule (5)) including special symbols x, y and z. Suppose that we wish to transform a circuit C_1 to C_2 by applying a rules $f \Longrightarrow g$. Then we seek a substring s of C_1 that "matches" f. In this pattern matching the special symbol x (y, z also) matches any substring s_1 of C_1 if s_1 meets the condition of p-circuits (such s_1 is called a *subcircuit* of C_1). Circuit C_2 is obtained by replacing s of C_1 by the right-hand side of the rule, i.e., by g. The formal definition of this transformation is omitted but the following example will be helpful:

Example. Suppose that we wish to apply rule

$$x,((y,z)) \Longrightarrow x,y,z$$

to p-circuit C_1 that is

$$((x_1),((x_2,(g[01],((x_4,x_1))))),x_3).$$

Since we allow x, y and z to match any subcircuit of C_1, there are two different possibilities. The first possibility is to set

$$x = (x_1),\ y = x_2,\ z = (g[01], ((x_4, x_1))),$$

which transforms C_1 into C_2; that is

$$((x_1), x_2, (g[01], ((x_4, x_1))), x_3)$$

The other possibility is to set

$$x = g[01],\ y = x_4,\ z = x_1,$$

and then we obtain a different C_2 such as

$$((x_1), ((x_2, (g[01], x_4, x_1))), x_3).$$

Rule (10) is so-called a substitution and its converse. If label $g[\ell]$ appears in a p-circuit, it can be replaced by the right-hand side of the definition of $g[\ell]$. Conversely, if some subcircuit s_1 of p-circuits coincides with the right-hand side of some definition, say $g[\ell] = s_1$, then the whole s_1 can be replaced by $g[\ell]$. Note that if the definition $g[\ell] = s_1$ does not exists, it can be created by rule (12).

More formally, the transformation from a circuit C_1 is defined by a function $T(C_1, r, k)$. Here, r is a rule and k is an integer. Recall that there may be two or more possibilities for applying rule r to C_1. The integer k is for selecting one of such possibilities, namely, the kth one. (Exactly speaking, we have to define some order for these possibilities.) $T(C_1, r, k)$ returns a (unique) circuit C_2 or *nil* if there are no possibilities for the application of rule r.

Theorem 1. For any circuit C of size n, $T(C, r, k)$ can be computed in time polynomial in n.

Proof. Consider for example rule r: $x, ((y, z)) \Longrightarrow x, y, z$. Then a straightforward way of computing $T(C, r, k)$ is as follows: (i) Regarding circuit C as a string, we decompose C using substrings $s_1, s_2, \cdots, s_5$ such that $C = s_1 s_2, ((s_3, s_4)) s_5$ (each s_i may be the null string). One can see that the number of different such decompositions is polynomial in the size n of C. (ii) Then we check if all of s_2, s_3 and s_4 are subcircuits. If so, we say that the decomposition is proper. Again this can be done in polynomial time. (iii) Now we select the kth one out of the proper decompositions (if any). The argument is similar for other rules. As for rule (12), note that we imposed the restriction that the created p-circuit be of polynomial size. □

We next show that the set $\Re$ of transformation rules is complete. Two circuits C_1 and C_2 are said to be *equivalent* if (i) the number of inputs and outputs is the same in both circuits and the corresponding two outputs, $g[\ell]$ in C_1 and $g[\ell]$ in C_2, realize the same logic function.

Theorem 2. Let C_1 and C_2 be any equivalent circuits. Then there exists a sequence of rules, each of them in $\Re$, which transforms C_1 into C_2.

Remark 1. The theorem only claims the existence of such a sequence. According to the following proof, its length (the number of primary transformations) can easily be exponential. To find an essentially shorter sequence is hard even if there are some.

Proof of Theorem 2. We assume that the circuit has only one output. Extension to the multi-output case is straightforward.

So-called DNF is used as a normal form of circuits: A circuit of n variables is said to be in DNF if it is given as $g[0] = f$ where $f = (p_1, p_2, \cdots, p_m)$. Each p_i must be $(y_1, y_2, \cdots, y_n)$ where $y_i =$ a single variable x_i or its negation (x_i). For any j, p_j must be lexicographically earlier than p_{j+1}, namely, the binary number obtained by replacing each x_i ((x_i), respectively) of p_j by 1 (0, respectively) must be smaller than the similar number for p_{j+1}.

Then it turns out that for any circuit C, there is a sequence of rules, each in $\Re$, that transforms C into the normal form. Note that this is enough to claim the theorem by the following reason: Let C_1 and C_2 be equivalent circuits. Then for each C_i, there is a sequence $r_{i1} r_{i2} \cdots r_{it_i}$ of transformations. Note that these two sequences must transform both C_1 and C_2 into exactly the same string, since the normal form is unique for any logic function. Now consider the sequence $S = r_{11} r_{12} \cdots r_{1t_1} \overline{r_{2t_2}} \cdots \overline{r_{22}}\ \overline{r_{21}}$, where $\overline{r}$ is the opposite of r, namely, if r is $f \Longrightarrow g$ then $\overline{r}$ is $g \Longrightarrow f$ for rules (1)$\sim$(10) and if r is (11) then $\overline{r}$ is (12).

The algorithm for getting the sequence $r_{i1} r_{i2} \cdots r_{it_i}$ is fairly complicated. We have to omit it, but the following example for reducing the level of circuits (from four to three) would be helpful.

$$\begin{array}{l} (x_1, (x_2, (x_3, (x4)))) \\ \qquad \downarrow \quad (8)\ (x, (y, z)) \Longrightarrow (((x, (y)), (x, (z)))) \\ (x_1, (((x_2, (x_3)), (x_2, ((x4)))))) \\ \qquad \downarrow \quad (4)\ x, ((y, z)) \Longrightarrow x, y, z \\ \qquad \downarrow \quad (7)\ ((x)) \Longrightarrow x \\ (x_1, (x_2, (x_3)), (x_2, x_4)) \end{array}$$ □

4. Random Circuit Generator

Application of the complete set $\Re$ will be wide. A natural possibility is to use it for simplifying circuits, since it is guaranteed that there is a path from a given circuit to any simpler one. Of course, however, very careful choice of rules should be needed for this purpose, which is far from easy. Then what happens if we chose rules *carelessly*? The circuit is probably not simplified but is complicated. That meets our present goal!

Generator RC-GEN.

Input: A circuit C_1

Output: A circuit C_2 that is equivalent to C_1 and is probably more complicated than C_1.

Step1: $C \leftarrow C_1$

Step2: Select a rule r at random from $\Re$.

Step3: Apply r to C to get C'. If there are two or more possibilities, select one of them at random.

Step4: $C \leftarrow C'$ and repeat Step2-Step4 some specified times.

Step5: $C_2 \leftarrow C$

This basic structure of RC-GEN needs appropriate modification for actual implementation. Suppose, for example, that

$$((x_2, x_3), ((x_3,), x_4), ((x_2), x3))$$

is chosen as the initial circuit C. Now our experiment shows that if all rules are applied with the same probability then such a circuit as follows is generated after 50-time execution of the main loop.

```
(((((((((x1,x3))),((((x3,(0)))),((((x4,x4)))))))),((((
x2,((((((((((x1,x3))),((x3,1))),((((x1,x3))))),1,((((
(x1,x3))),((((x3,1)))))),((((x4,x4,x4))))))),((((x1,x3)
)),(((((x4,x4)))))))),(((((((x1,1,x3))),((((x3,(0)))),(
(((x4,x4)))),((((x4))))))))))))),x3))))
```

One can easily feel that the circuit does not make much sense as a usual logic circuit. Our consideration for better implementation is as follows:

(1) Clearly there are too many parentheses, which can be removed by applying the $\Longrightarrow$ direction of rule (5) more frequently. The current setting is 30 times as high as the normal probability.

(2) The $\Longleftarrow$ direction of (3) and (4) makes the string (circuit) longer. However, it is simply a repetition of exactly the same thing. What is desirable is that after the application of these rules, a lot of other rules are applied to the same subcircuits and these are changed into different ones. We decided to restrict the number of applications of these rules into only twice during the whole course of the generation.

(3) The most important rule for modifying circuits is probably (8). We set the probability of both directions of (8) three times as high as the normal probability.

(4) As for rule (10), there are less problems for the substitution (the $\Longleftarrow$ direction) but are several difficulties for the opposite direction. First of all, there is almost no possibility of the rule's being actually applied, since even if we wish to replace a subcircuit s by $g[\ell]$, the definition $g[\ell] = s$ probably does not exist. Hence, we need to combine this rule with the creation of $g[\ell] = s$, i.e., rule (12). To emphasize its objective (management of multiple fan-out), it may be more appropriate that the rule can be applied only if we can find two or more subcircuits s_1 and s_2 such that $s_1 = s_2$. Then both subcircuits can be replaced by the same label, say $g[\ell]$. At this moment, the substitution is excluded from the rule set and the creation is allowed only if the two (or more) equivalent subcircuits above mentioned are found.

After these considerations, the circuit given at the beginning was transformed into the following circuit (by 500-time repetition) that appears much "better" than before:

```
((((((((((x2,(x3,x1),((x3),x4)),((x4,(x3),1),(x1,x3),1),
(0)),(x1)),((((x3),0),((x3,x1),((((x3),x4),x2),(x3),x4)
),(0,(x1))),(x3))),x3),(((((((x2,(0),(x1,x3),((x3),x4,(0
))),x4,(x3)),(x1),(((x1,(x2)),(x1,((x1,x3))),(x1,(((x3),
x4,1)))),x3,1)),((1),(x1,x3)),(((x3),(x2,((x3),x4)),x4),
(x1,x3))),((((x2,(x1,x3),((x3),1,x4,(0))),x4,(x3)),(x1),
(x1,(x2,(x1,x3),((x3),x4)),1,x3)),x3)),(0)),(x4))),x2),(
(x1,(0),x3),((((x3),x4),x2),(x3),x4),(x3,1))))
```

5. Experiments

For the first experiment, we selected the majority function of six variables, namely

```
g0=((x1,x2,x3),(x1,x2,x4),(x1,x2,x5),(x1,x2,x6),(x1,x3,x4),
   (x1,x3,x5),(x1,x3,x6),(x1,x4,x5),(x1,x4,x6),(x1,x5,x6),
   (x2,x3,x4),(x2,x3,x5),(x2,x3,x6),(x2,x4,x5),(x2,x4,x6),
   (x2,x5,x6),(x3,x4,x5),(x3,x4,x6),(x3,x5,x6),(x4,x5,x6))
```

For this experiment, we excluded the rules from (10) to (12) so that only tree-like (single fan-out) circuits would be involved. From this initial circuit, we generated a number of circuits by applying the rules 2000 times. (Since there are cases when the selected rule cannot be applied, the actual number is one half or one third of 2000.) Those circuits are then given to SIS where we selected, as the optimizing option, the algebraic factorization (i.e.,"script.algebraic") and the area oriented technology mapping using the library including only NAND and NOR gates of up to four inputs.

The result is shown in Table 1. In each row, the left half gives data for the test circuit and the right half for the SIS's output. For example, No.1 test circuit contains 1894 gates and 3650 connections and its network level is 24, which is simplified by SIS into 23 gates, 46 connections and 8 levels in 17.9 sec. (SUN SPARC 2). No.0 row shows the result when the initial circuit itself is given to SIS. (This is the same for Tables 2 and 3.)

Generally, SIS showed a remarkable performance. However, there are subtle differences among the five test circuits illustrated in the table. For example, No.2 is worth than No.1 even if the test circuit is smaller. No.3 is the worst that is about four times as bad as the best in each

of the number of gates, connections and levels.

The second experiment is for the 4-bit adder, where the full set of rules are included. Although it has five outputs (including the final carry), we put them together into a single output in two different ways. The one is that the final output becomes one iff exactly two of the four outputs of the adder (excluding the final carry) are one. Namely the initial circuit is given as follows:

```
g0=((g1,g2),(g1,g3),(g1,g4),(g2,g3),(g2,g4),(g3,g4))
g1=(((x1),x2,(g5)),(x1,(x2),(g5)),((x1),(x2),g5),(x1,x2,g5))
g2=(((x3),x4,(g6)),(x3,(x4),(g6)),((x3),(x4),g6),(x3,x4,g6))
g3=(((x5),x6,(g7)),(x5,(x6),(g7)),((x5),(x6),g7),(x5,x6,g7))
g4=((x7,(x8)),((x7),x8))
g5=((x1,x2),(x2,g6),(g6,x1))
g6=((x3,x4),(x4,g7),(g7,x3))
g7=((x5,x6),(x6,g8),(g8,x5))
g8=((x7,x8))
```

The other is that the output becomes zero if the five outputs of the adder are 01010. Results are shown in Tables 2 and 3, where we can also see differences of up to twice.

Table 1.

No.	source			output			time
	gate	conn.	level	gate	conn.	level	
0	21	80	2	20	39	8	1.8
1	1894	3650	24	23	46	8	17.9
2	732	1375	16	39	81	9	8.3
3	1738	3132	20	103	212	22	28.6
4	882	1657	15	41	83	10	9.0
5	146	283	13	26	49	6	2.4

Table 2.

No.	source			output			time
	gate	conn.	level	gate	conn.	level	
0	55	113	11	48	86	18	4.7
1	236	463	22	52	95	16	8.2
2	180	354	27	56	100	15	7.1
3	454	942	26	104	202	19	24.2
4	340	746	37	93	172	33	16.6
5	359	763	39	48	86	20	11.7

Table 3.

No.	source			output			time
	gate	conn.	level	gate	conn.	level	
0	56	112	12	46	84	19	3.9
1	319	618	27	79	153	21	14.6
2	176	358	24	39	77	12	6.5
3	715	1627	34	67	132	26	26.6
4	926	1861	35	52	98	20	23.0
5	159	335	23	37	73	12	6.7

6. Concluding Remarks

The reason why random circuits have been seldom used for testing the performance of logic optimizers is that we did not know how to generate the random circuits appropriately. The main contribution of this paper is to propose the random transformation for that purpose, to establish its theoretical foundations and to claim that the method is useful or at least is worth doing more research and development. Testing other optimization systems and trying to use the result to improve the optimizers will be obvious future work.

References

[1] R. K. Brayton, R. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang, "Mis: A multiple-level logic optimization system," *IEEE Trans.* CAD, 6, pp. 1062-1081, 1987.

[2] K. Hino and K. Iwama,"On a complete set of basic operations to transform between equivalent switching circuit," Technical Report of the Institute of Electronics, Information and Communication Engineers, COMP92-67 (1992-11) (in Japanese).

[3] K. Iwama, H. Abeta, and E. Miyano, "Random generation of satisfiable and unsatisfiable CNF predicates," *in* Proc. 12th IFIP World Computer Congress, pp. 322-328, 1992.

[4] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, "The Transduction method – Design of logic networks based on permissible functions," *IEEE Trans. Comput.* 38, 10, 1989.

[5] D. Mitchell, B. Selman, and H. Levesque, "Hard and easy distributions of SAT problems," *in* Proc. 10th National Conference on Artificial Intelligence, pp. 459-465, 1992.

[6] E. M. Sentovich, K. J. Singh, *et al.*, "SIS: A system for sequential circuit synthesis," Memorandum No. UCB/ERL M92/41, 1992.

[7] G. Tinhofer, "Generating graphs uniformly at random," *in* Computational graph theory, pp. 235-255, Springer, 1990.

[8] S. Yang, "Logic synthesis and optimization Benchmarks user guide version 3.0," *in* 1991 MCNC International Workshop on Logic Synthesis.