IPSJ Transactions on System LSI Design Methodology Vol. 4 70–79 (Feb. 2011)

Regular Paper

Exact Minimum Factoring of Incompletely Specified Logic Functions via Quantified Boolean Satisfiability

Hiroaki Yoshida[†1] and Masahiro Fujita[†1]

This paper presents an exact method which finds the minimum factored form of an incompletely specified Boolean function. The problem is formulated as a Quantified Boolean Formula (QBF) and is solved by general-purpose QBF solver. We also propose a novel graph structure, called an X-B (eXchanger Binary) tree, which compactly and implicitly enumerates binary trees. Leveraged by this graph structure, the factoring problem is transformed into a QBF. Using three sets of benchmark functions: artificially-created, randomly-generated and ISCAS 85 benchmark functions, we empirically demonstrate the quality of the solutions and the runtime complexity of the proposed method.

1. Introduction

Logic factoring is an operation to find a factored form from a Boolean function. The factored form is known as one of the efficient representation styles of Boolean functions and forms a basis of multiple-level logic. Since the factored form corresponds to a static CMOS compound gate, it has also been used to estimate the area of a circuit implementing a Boolean network. For example, the static CMOS compound gate illustrated in **Fig. 1** implements the complementation of a factored form expression: $y = \overline{((ab + c) * d) + ef}$. Although logic factoring is one of the most fundamental problems in multiple-level logic synthesis, the exact solution to this problem is still challenging. Early works [1),2)] provided exact multiple-level logic minimization algorithms, however, the algorithms are very inefficient and apply to very small functions. An improved algorithm [2)] requires a couple of hours to synthesize small circuits with ∼12 2-input NAND gates [3)]. In addition to their computational costs, these algorithms cannot be applied to the minimization of the number of literals in the factored form.

†1 VLSI Design and Education Center, the University of Tokyo

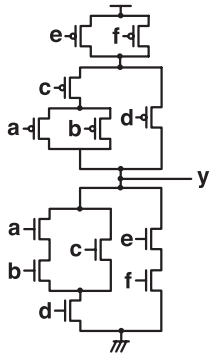


Fig. 1 A static CMOS compound gate.

Recently, Boolean satisfiability solvers have made a dramatic improvement [4),5)] and have been successfully applied to industrial-scale EDA problems such as automatic test pattern generation [6)] and symbolic model checking [7)]. Quantified Boolean formula, which is a generalization of propositional logic, is known as a more natural way to model such problems. As a consequence, a number of efficient QBF decision algorithms have been proposed [8)–11)].

In this paper, we present an exact logic factoring method which first transforms the factoring problem into a QBF and then solving it using general-purpose QBF solver. To transform the problem into a QBF, we propose a novel graph structure, called as an X-B tree, which compactly and implicitly enumerates all possible binary trees with a specific number of leaf nodes. The remainder of this paper is organized as follows. In Section 2, we introduce a novel graph structure, called as an X-B (eXchanger Binary) tree, and then present a method for generating X-B trees. In Section 3, we propose an exact method for minimum logic factoring. After formulating the problem, we explain how to transform the problem into a structural representation using an X-B tree and how to represent the structural representation as a quantified Boolean formula. Section 4 presents experimental results on three sets of benchmark functions to demonstrate the quality and the runtime complexity of the proposed method. Conclusions are drawn in Section 5.

2. X-B Tree and Its Generation

2.1 X-B Tree

A *binary tree* is a graph with *internal nodes* and *leaf nodes* in which every internal node has two children. A *child* is either an internal node or a leaf node. An *X-B (eXchanger Binary) tree* is a rooted unordered binary tree with a new type of nodes, called an *exchanger node*. An exchanger node has the same number of inputs $\{i_1, ..., i_n\}$ and outputs $\{o_1, ..., o_n\}$. Only one of the children is an internal node, and the others are either a leaf node or an exchanger node. An exchanger node has an associated value, called an *exchange index* which determines how the inputs and the outputs are connected. An exchange index c_x is a positive integer $1 \leq c_x \leq n$ where n is the number of the inputs (outputs) of the exchanger. Then, the relation between the inputs and the outputs is given as follows:

$$o_j = i_{((c_x+j-2) \ mod \ n)+1} \quad (1 \leq j \leq n). \tag{1}$$

where i_j is the j-th input and o_j is the j-th output. In other words, an exchanger node can be viewed as a bit shifter. An X-B tree with 7 leaf nodes is shown in **Fig. 2** and a 3-input exchanger node is shown in **Fig. 3**.

Once the assignment of the exchange indices is determined, a binary tree is obtained. By exploring such assignments, all possible binary trees can be obtained. Note that there is not necessarily a one-to-one correspondence from an assignment of exchange indices to a binary tree, i.e., different assignments of exchange indices can result in the same binary tree.

When the logic factoring problem is formulated as a QBF, as explained in the next section, the exchange index of each exchanger node is represented as a vector of binary variables. The *number of total exchange bits* is the number of binary variables required to represent all exchange indices in an X-B tree:

$$n_b = \sum_i \lceil \log_2 n_i \rceil \tag{2}$$

where $\lceil x \rceil$ denotes the smallest integer greater than or equal to x, and n_i is the number of inputs of i-th exchanger node in an X-B tree.

2.2 Signatures of Binary Trees

During the construction of X-B trees, it is necessary to check the isomorphism between two binary trees. We use the *bitstring representation* [12),13)] as the signature of binary trees. Since the original bitstring representation is for ordered binary trees, we extend it for unordered binary trees. The bitstring representation is a binary sequence $b_1 b_2 \ldots b_{2n}$ obtained recursively, as described in **Fig. 4**. In the procedure, the last bit is always 0 and hence is omitted. **Figure 5** (a) and (b) illustrate how the signatures of two example binary trees are computed where the bitstring shown next to each node is the signature of the node and the signature of the root node is the signature of the tree.

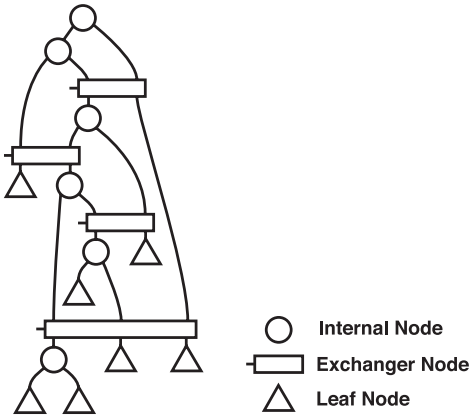
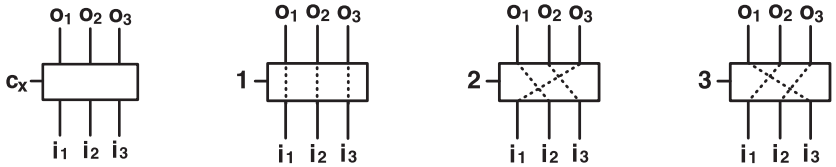


Fig. 2 An X-B tree with 7 leaf nodes.



(a) (b) $c_x = 1$ (c) $c_x = 2$ (d) $c_x = 3$

Fig. 3 An example of 3-input exchanger node.

2.3 Generating X-B Trees

Given the number of leaf nodes, there are many choices of X-B trees depending on how the exchanger nodes are connected. Since we are particularly interested

ComputeSignature

Input: Tree T
Output: Bitstring representation S

1: $V \leftarrow$ the root node of T
2: $S \leftarrow$ ComputeSignatureNode(V)
3: Omit the last bit from S
4: **return** S

(a) Top-level procedure ComputeSignature.

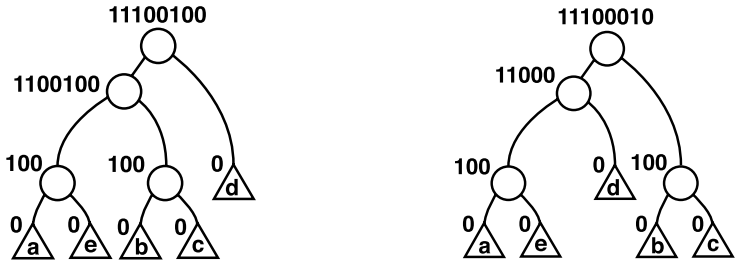
ComputeSignatureNode

Input: Tree node V
Output: Bitstring representation S

1: **if** V is a leaf node
2: **return** "0"
3: **end if**
4: $V_L \leftarrow$ the left child of V
5: $V_R \leftarrow$ the right child of V
6: $S_L \leftarrow$ ComputeSignatureNode(V_L)
7: $S_R \leftarrow$ ComputeSignatureNode(V_R)
8: **if** $S_L > S_R$
9: **return** "1"+S_L+S_R
10: **else**
11: **return** "1"+S_R+S_L
12: **end if**

(b) Procedure ComputeSignatureNode.

Fig. 4 Basic procedure for signature computation.



(a) (b)

Fig. 5 An illustration of computing the signatures of binary trees.

in the minimum X-B tree, all possible X-B trees are first enumerated and the minimum one is chosen.

The proposed generation procedure is constructive in the sense that the X-B trees with L leaf nodes are constructed from the X-B trees with $L-1$ leaf nodes. The procedure starts with a binary tree with one internal node and two child leaf nodes. Given an X-B tree, a set of leaf nodes and their parents is identified as an insertion point. Then, an exchanger node and an internal node are inserted at the point, as illustrated in **Fig. 6**.

The insertion points are computed as follows. First, a leaf node is replaced

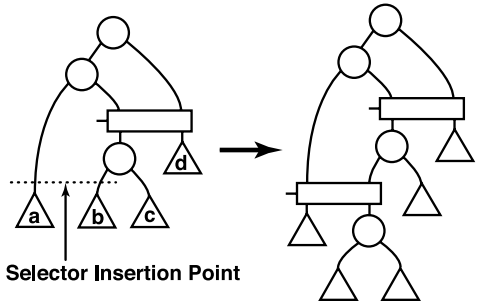
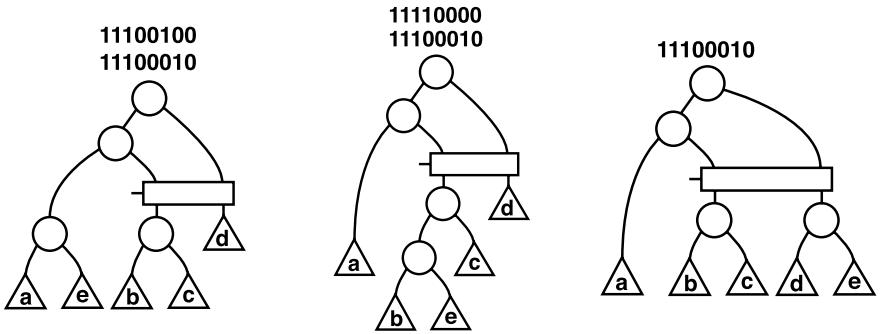


Fig. 6 Inserting an exchanger node.



(a) The insertion point is a. (b) The insertion point is b. (c) The insertion point is d.

Fig. 7 An illustration of computing the signatures of X-B trees.

with an internal node and two child leaf nodes. Then, all the signatures are computed by exploring all possible assignments of the exchange indices. After computing the signatures for all leaf nodes, a covering table is constructed where the rows correspond to the signatures and the columns to the leaf nodes. By solving the covering problem, the minimum set of leaf nodes is found and an exchanger node is inserted between the leaf nodes and their parents. We explain this procedure using the X-B tree with 4 leaf nodes in Fig. 6 (left). **Figure 7** (a)-(c) show three X-B trees after replacing a leaf node with an internal node and two child leaf nodes. The signatures of each X-B tree is shown at the top of the tree. For example, the X-B tree in Fig. 7 (a) includes two binary trees shown in

Table 1 A covering table.

Signature	a	b	d
11100100	1	0	0
11100010	1	1	1
11110000	0	1	0

Table 2 Characteristics of minimum X-B trees.

#leaf nodes	*#internal nodes*	*#exchangers*	*#total exchange bits*	*#encoded binary trees*
2	1	0	0	1
3	2	0	0	1
4	3	1	1	2
5	4	2	2	3
6	5	3	3	6
7	6	4	5	11
8	7	5	7	23
9	8	6	9	46
10	9	7	11	98
11	10	8	13	207
12	11	9	15	451
13	12	10	17	983
14	13	11	20	2,179
15	14	12	22	4,850
16	15	13	25	10,905

Fig. 5 and hence the signatures of this X-B tree are 11100100 and 11100010. As a result, a covering table is constructed as shown in **Table 1** where each column corresponds to one of the three X-B trees in Fig. 7 and each row to one of the signatures. Since the minimum covering is $\{a, b\}$, the selector insertion point is at the leaf nodes a and b as shown in Fig. 6 (right).

2.4 Complexity of X-B Trees

Table 2 shows the characteristics of the minimum X-B trees obtained by the method in the previous section. The numbers of leaf nodes L, internal nodes N and exchanger nodes X hold the following relation:

$$L = N + 1 = X + 3. \tag{3}$$

An upper bound on the number of total exchange bits is derived as follows. Suppose that an X-B tree with L leaf nodes is being constructed by inserting an exchanger node into an X-B tree with $L-1$ leaf nodes. In the worst case, the number of inputs of the exchanger node is $L-2$. Hence, an upper bound on the number of total exchange bits n_b is calculated from Eq. (2):

$$n_b = \sum_{l=4}^{L} \lceil \log_2(l-2) \rceil < \sum_{l=4}^{L} \lceil \log_2 L \rceil = O(L \log_2 L). \tag{4}$$

Thus, X-B trees can efficiently encode exponential number of binary trees in a single graph.

3. Exact Minimum Factoring

3.1 Problem Formulation

A *literal* is a variable or its negation. A *factored form* is a representation of a Boolean function and defined recursively as follows: 1) a literal is a factored form; 2) a sum of factored forms is a factored form; 3) a product of factored forms is a factored form. In general, the factored form of a Boolean function is not unique. For example, the following expressions; $abc + abd + cd$, $ab(c + d) + cd$ and $abc + (ab + c)d$ are all the factored forms of a Boolean function. A factored form is *minimum* if and only if the number of literals is the least among all possible factored forms.

An AND/OR binary tree is a rooted binary tree where the type of each internal node is either a 2-input AND operator or a 2-input OR operator. By regarding leaf nodes as literals, an arbitrary factored form can be represented as an AND/OR binary tree.

The problem addressed in this chapter can be formulated as follows: *Given an incompletely specified Boolean function* (f, d, r) *of variables* $V = \{v_1, \ldots, v_{|V|}\}$*, find a factored form with the minimum number of literals.* Alternatively, we can formulate it as follows: *Given an incompletely specified Boolean function* (f, d, r)*, find an AND/OR binary tree with the minimum number of leaf nodes which implements the Boolean function.*

3.2 Constructing a QBF

The problem is modeled as a miter structure [14] illustrated in **Fig. 8**. It checks the equivalence between the given Boolean function and an AND/OR X-B tree. An AND/OR X-B tree is an X-B tree with the following modifications. An internal node, called as an *operator node*, has its associated variable $c_o \in \{0, 1\}$

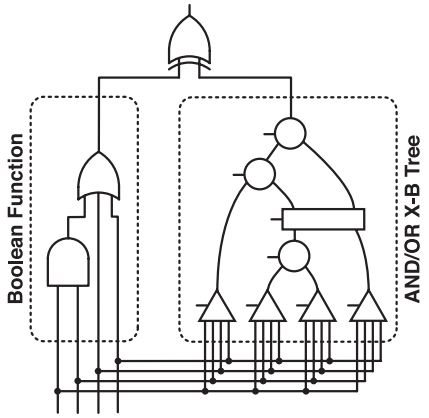
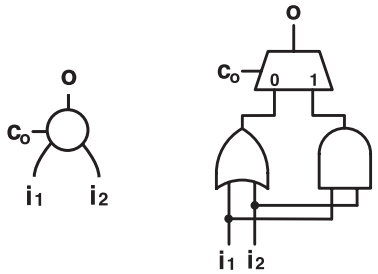
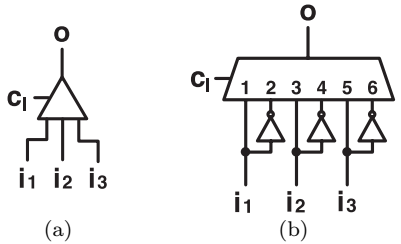


Fig. 8 A miter structure.



(a) (b)

Fig. 9 (a) operator node and (b) its equivalent logic circuit.



(a) (b)

Fig. 10 (a) literal node and (b) its equivalent logic circuit.

to specify whether the node type is AND or OR. **Figure 9** shows an operator node and its equivalent logic circuit. A leaf node, called as a *literal node*, has its associated variable $c_l \in \{1, \ldots, 2|V|\}$ to specify a literal $l \in \{v_1, \overline{v_1}, \ldots, v_{|V|}, \overline{v_{|V|}}\}$. **Figure 10** shows a literal node and its equivalent logic circuit. Recall that c_x is an exchange index explained in Section 2.1. The three types of variables, c_x, c_o and c_l, are called as *configuration variables* $C = \{c_1, \ldots, c_{|C|}\}$. An arbitrary AND/OR binary tree with a specific number of leaf nodes can be represented by an AND/OR X-B tree with an assignment of the configuration variables.

A quantified Boolean formula is constructed based on this model. The clauses of the quantified Boolean formula consist of four categories: *function constraints*, *operator node constraints*, *exchanger node constraints* and *literal node constraints* where each constraint corresponds to a node in the miter structure. Also, *tree constraints* are added to the clauses for reducing the solution space.

3.2.1 Function Constraints

Let o_{root} be the variable corresponding to the output of the root operator node in the AND/OR X-B tree. The function constraints check the equivalence of the Boolean function and the AND/OR X-B tree:

$$\xi_f = (f \equiv o_{root}) + d \tag{5}$$

where f and d are the on set and the *don't care* set of the given Boolean function, respectively. If the assignment of the input variables is *don't care*, ξ_f is true regardless of the values of f and o_{root}. Thus, the *don't care* condition is taken into account.

3.2.2 Operator Node Constraints

The operator node constraints represent the operator nodes in the AND/OR X-B tree. For each operator node, the following formula is constructed:

$$\xi_o = \overline{c_o}\big(o \equiv (i_1 + i_2)\big) + c_o\big(o \equiv (i_1 \cdot i_2)\big) \tag{6}$$

where i_1 and i_2 are the variables corresponding to the outputs of the child nodes of the operator node.

3.2.3 Exchanger Node Constraints

Let n be a positive integer $1 \leq n \leq m$. Then, a cube representation of n is defined as follows:

$$\mathcal{CUBE}(x, m, n) = \prod_{i=1}^{\lceil \log_2 m \rceil} (x_i \equiv b_i) \tag{7}$$

where $b_1 b_2 \ldots b_{\lceil \log_2 m \rceil}$ is a binary bit-vector representation of a decimal integer $n - 1$. For example, $\mathcal{CUBE}(x, 4, 1) = \overline{x_1} \cdot \overline{x_2}$, $\mathcal{CUBE}(x, 4, 2) = x_1 \cdot \overline{x_2}$, $\mathcal{CUBE}(x, 4, 3) = \overline{x_1} \cdot x_2$ and $\mathcal{CUBE}(x, 4, 4) = x_1 \cdot x_2$.

The exchanger node constraints represent the exchanger nodes in the AND/OR X-B tree. For each exchanger node, the following formula is constructed:

$$\xi_x = \sum_{i=1}^{n}\Big(\mathcal{CUBE}(c_x, n, i)\cdot\prod_{j=1}^{n}\big(o_j \equiv i_{((i+j-2)\ mod\ n)+1}\big)\Big) \cdot \overline{\sum_{i=n+1}^{2^{\lceil \log_2 n\rceil}} \mathcal{CUBE}(c_x, n, i)} \tag{8}$$

where n is the number of the inputs (outputs) of the exchanger node, o_j is the variable corresponding to the j-th output of the operator node, and i_j is the variable corresponding to the output of the j-th child node of the operator node.

3.2.4 Literal Node Constraints

The literal node constraints represent the literal nodes in the AND/OR X-B tree. For each literal node, the following formula is constructed:

$$\xi_l = \sum_{i=1}^{|V|}\Big(\mathcal{CUBE}(c_l, 2|V|, 2i-1)\big(o \equiv v_i\big) + \mathcal{CUBE}(c_l, 2|V|, 2i)\big(o \equiv \overline{v_i}\big)\Big) \cdot \overline{\sum_{i=2|V|+1}^{2^{\lceil \log_2 2|V|\rceil}} \mathcal{CUBE}(c_l, 2|V|, i)}. \tag{9}$$

3.2.5 Tree Constraints

As mentioned in Section 2, the encoding of X-B trees is redundant, i.e., different assignments of the exchange indices can correspond to the same binary tree structure. The tree constraints restrict the possible values for the exchange indices so that every valid assignment of the exchange indices corresponds to a different binary tree structure. Let $A = \{a_1, ..., a_X\}$ be an X-tuple of assignments of the exchanger indices where X is the number of the exchanger nodes in the AND/OR X-B tree. In this way, A specifies a binary tree structure. Let $\mathfrak{A} = \{A_1, ...A_{|\mathfrak{A}|}\}$ be a family of A such that no two elements in $\mathfrak{A}$ correspond to the same binary tree structure. Then, the tree constraints are formulated as follows:

$$\xi_t = \sum_{A\in\mathfrak{A}}\prod_{a_i\in A}\mathcal{CUBE}(c_{x_i}, n_i, a_i) \tag{10}$$

where c_{x_i} is the exchanger index of the i-th exchanger node and n_i is the number of the inputs (outputs) of the i-th exchanger node.

3.2.6 Constructing a Final QBF

Recall that $C = \{c_1, \ldots, c_{|C|}\}$ is a set of configuration variables and $V = \{v_1, \ldots, v_{|V|}\}$ is a set of function variables. Let $O = \{o_1, \ldots, o_{|O|}\}$ be the variables corresponding to the outputs of the exchanger, operator and literal nodes. Then, a quantified Boolean formula ξ is constructed by combining all the constraints Eqs. (5), (6), (8), (9) and (10) and introducing existential and universal quantifiers:

$$\xi = \exists C \forall V \exists O\ \xi_f\left(\prod_{i=1}^{L-1}\xi_{o_i}\right)\left(\prod_{i=1}^{L-3}\xi_{x_i}\right)\left(\prod_{i=1}^{L}\xi_{l_i}\right)\xi_t \tag{11}$$

where ξ_{o_i}, ξ_{s_i} and ξ_{l_i} are the constraints corresponding to the i-th node of each type, and L is the numbers of the leaf nodes in the AND/OR X-B tree. Note that the number of the exchanger nodes X is given as $L-3$ from Eq. (3).

3.3 Finding the Minimum Factored Form

Given the number L of the leaf nodes in the AND/OR X-B tree, a QBF ξ is constructed as described in the previous section. If ξ is satisfiable, it implies that there is a factored form with L or less literals. To find the minimum factored form, we start with $L = |V|$ literals. Note that there does not exist any factored form with less than $|V|$ literals because every variable must appear in the factored form. If the QBF is satisfiable, the assignment of the configuration variables is computed and the minimum factored form is obtained. Otherwise, L is incremented by one and the procedure is repeated until the minimum factored form is obtained. **Figure 11** describes a basic procedure for finding the minimum factored form.

3.4 Complexity of QBFs

First, we derive an upper bound on the number of variables used in the QBFs constructed by the proposed method. Upper bounds on the numbers of configuration variables C, function variables V and output variables O are given as follows:

$$|C| = (L-1) + L\lceil log_2 2V\rceil + O(Llog_2 L) = O(Llog_2 L) \tag{12}$$

$$|V| = O(|V|) \tag{13}$$

ExactFactor

Input: Incompletely specified Boolean function (f, d, r)
Output: Minimum factored form expression F

```
1:  L ← |V|
2:  loop
3:    X ← an AND/OR X-B tree with L leaf nodes
4:    Construct a QBF ξ from X and (f, d, r)
5:    Solve the QBF ξ
6:    if ξ is satisfiable
7:      A ← a satisfiable assignment of ξ
8:      T ← an AND/OR tree by assigning A to X
9:      Transform T to a factored form F
10:     return F
11:   end if
12:   L ← L + 1
13: end loop
```

Fig. 11 Basic procedure for finding minimum factored form.

Table 3 Upper bounds on QBF sizes.

Type	*#clauses*	*#literals*
Function Constraints	$O(Prod(f) + Prod(r))$	$O(Lit(f) + Lit(r))$
Operator Constraints	$O(L)$	$O(L)$
Exchanger Constraints	$O(L^3)$	$O(L^3 log_2 L)$
Literal Constraints	$O(L\|V\|log_2\|V\|)$	$O(L\|V\|^2 log_2\|V\|)$
Tree Constraints	$O(2^L L)$	$O(2^L L^2 log_2 L)$

$$|O| = L + (L-1) + \sum_{i=1}^{L}(i-1) = O(L^2). \tag{14}$$

where L is the number of literals, i.e., the number of leaf nodes in the corresponding AND/OR X-B tree. Hence, an upper bound on the QBF variables is $O(L^2)$ since $L \geq |V|$.

Table 3 presents upper bounds on the numbers of clauses and literals of each constraint type in conjunctive normal form. In the table, $Prod(g)$ and $Lit(g)$ are the numbers of products and literals in disjunctive normal form (sum-of-product form) of Boolean function g. Also, f and r are the on set and off set of the incompletely specified Boolean function given as an input to the factoring problem.

4. Experimental Results

We have implemented the proposed method called *Exact Factor* in C++ on top of the logic manipulation class library *Logica* which we have developed. As a QBF solver, we have examined a number of state-of-the-art QBF solvers: ssolve [8], SEMPROP [9], sKizzo [10], and Quantor [11]. Among these solvers, we chose sKizzo which solved our QBF problem instances in the shortest runtime.

4.1 Artificially-created Examples

First, we conducted an experiment on a set of artificially-created benchmark functions. The artificial benchmark functions are categorized into two groups: *algebraic group* and *Boolean group*. The functions in the algebraic group are the functions of which the minimum factored form can be obtained without the specific features of Boolean algebra. In contrast, the functions in the Boolean group are the functions of which the minimum factored form can be obtained only if the specific features of Boolean algebra are used.

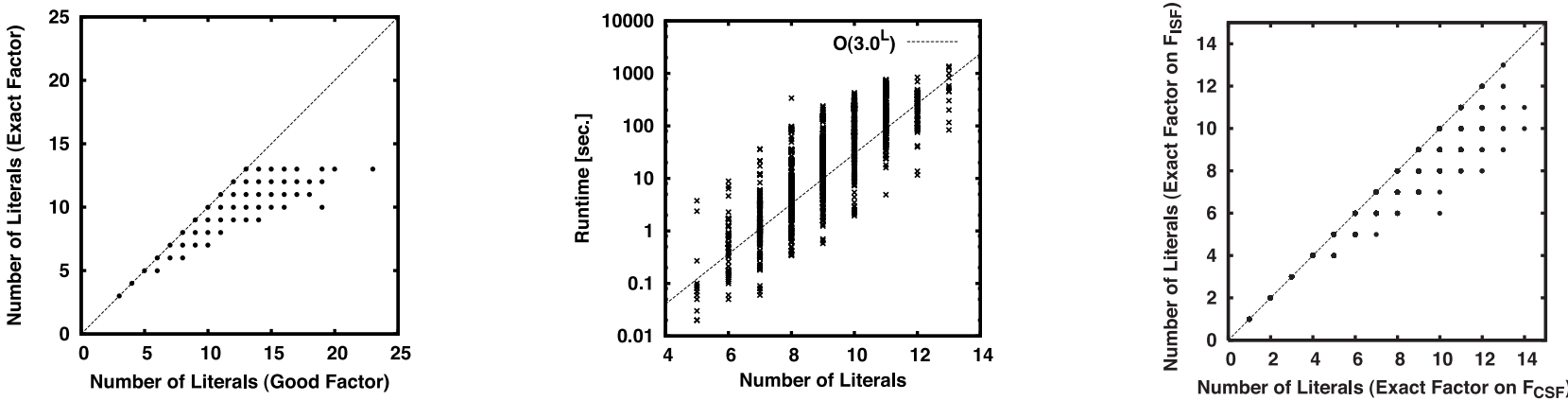
The results are shown in **Table 4**. As a reference, we present the results of ESPRESSO two-level minimizer [15] and Good Factor factoring algorithm [16]. In the table, the first two columns give the name of the function and the number of the input variables, respectively. Columns 3, 5 and 7 show the numbers of literals of the sum-of-products form generated by ESPRESSO, the factored form generated by Good Factor, and the factored form generated by the proposed method. Columns 4, 6 and 11 present the CPU times in seconds of the corresponding methods. "< 0.1" denotes that the CPU time is less than 0.1 seconds. Columns 8, 9 and 10 give the numbers of variables, clauses and literals of the satisfiable QBF when the minimum solution is found.

4.2 Randomly-generated Examples

Next, we conducted another experiment on benchmark functions which are randomly generated as follows. We first build an AND/OR X-B tree with a random number of literals. Then, a factored form expression and its corresponding Boolean function are obtained by randomly assigning the values of all configuration variables. Note that the generated factored form may be redundant and hence is not necessarily minimum. In this experiment, we generated 10,000 problem instances such that the number of literals ranges from 4 to 13. Just as the

Table 4 Experimental results on artificially-created examples.

Function	*#inputs*	*ESPRESSO* [15]		*Good Factor* [16]		*Exact Factor*				
							QBF			*CPU time*
		#literals	*CPU time [sec]*	*#literals*	*CPU time [sec]*	*#literals*	*#variables*	*#clauses*	*#literals*	*[sec]*
algebraic1	8	14	< 0.1	8	< 0.1	8	81	262	1034	14.3
algebraic2	6	36	< 0.1	11	< 0.1	11	115	584	2492	70.4
algebraic3	6	15	< 0.1	11	< 0.1	**10**	103	370	1301	8.9
algebraic4	9	19	< 0.1	9	< 0.1	9	103	369	1394	90.7
boolean1	5	11	< 0.1	8	< 0.1	**6**	54	142	465	0.2
boolean2	6	26	< 0.1	16	< 0.1	**10**	103	374	1322	10.1
boolean3	5	13	< 0.1	10	< 0.1	**8**	78	232	747	1.15
boolean4	6	22	< 0.1	18	< 0.1	**11**	115	584	2492	68.8
boolean5	6	30	< 0.1	20	< 0.1	**12**	128	718	3131	69.2



(a) Literal count comparison between Good Factor and Exact Factor. (b) Literal count vs. Exact Factor runtime. (c) Literal count comparison between Exact Factor on F_{CSF} and F_{ISF}.

Fig. 12 Experimental results on randomly-generated examples.

previous experiment, we performed both Good Factor and Exact Factor on the randomly-generated benchmark functions. **Figure 12** (a) compares the numbers of literals of the factored forms obtained by Good Factor and Exact Factor. In this figure, Exact Factor could improve the number of literals by up to about 39% compared to Good Factor. Figure 12 (b) plots the runtime of Exact Factor. As expected, due to the exponential nature of the satisfiability problems, the figure shows an exponential runtime complexity. From this plot, we can see that our method can solve problem instances with up to 14 literals within a few hours.

To demonstrate the impact of *don't cares* on logic factoring, we generated incompletely specified benchmark functions by adding *don't cares* to the same set of the randomly-generated benchmark functions. Each *don't care* consists of four cubes where each cube consists of randomly-selected four literals. Thus, up to 12.5% of all minterms become *don't care*. Let $F_{ISF} = (f, d, r)$ be an incompletely specified function. We first obtain a completely specified function

Table 5 Experimental results on ISCAS 85 benchmark suite.

Circuit	*All nodes*				*Improved nodes*			
	#nodes	*Total #literals*			*#nodes*	*Total #literals*		
		Good Factor [16]	*Exact Factor*	*Improvement [%]*		*Good Factor* [16]	*Exact Factor*	*Improvement [%]*
C432	64	406	406	0.0	0	—	—	—
C499	96	1,312	1,312	0.0	0	—	—	—
C880	100	826	818	1.0	7	75	67	10.7
C1355	96	1,312	1,312	0.0	0	—	—	—
C1908	97	1,521	1,519	0.1	2	16	14	12.5
C2670	149	1,730	1,718	0.7	8	79	67	15.2
C3540	337	2,897	2,832	2.2	44	460	395	14.1
C5315	433	5,212	4,999	4.1	102	1,084	871	19.6
C6288	449	10,713	10,689	0.2	11	163	139	14.7
C7522	507	6,026	5,716	5.1	173	1,605	1,295	19.3

F_{CSF} by performing ESPRESSO on F_{ISF}. Then, we compare two factored forms obtained by: (a) Exact Factor on F_{CSF} and (b) Exact Factor on F_{ISF}. The results in Fig. 12 (c) demonstrate the effectiveness of Exact Factor with taking account of *don't care*.

4.3 ISCAS 85 Benchmark Suite

Finally, we perform a literal count minimization on 10 multiple-level logic circuits from ISCAS 85 benchmark suite using Good Factor and Exact Factor. Using SIS [17] with RASP FPGA/CPLD Technology Mapping and Synthesis Package [18], each circuit is synthesized by 8-input LUT mapping (`dmig -k 8; flowmap -k 8`) to obtain a Boolean network with reasonably large nodes. We count the total number of literals where each node in the Boolean network is represented as a factored form. For each node n, $EF(n)$ and $GF(n)$ are defined as the numbers of literals of the factored form obtained by Exact Factor and Good Factor, respectively. In **Table 5**, *All nodes* compares the literal counts of N and *Improved nodes* compares the literal counts of N_{imp} where N is a set of all nodes and $N_{imp} = \{n | EF(n) < GF(n), \forall n \in N\}$. Although the overall improvement is not significant, we observe that there is still some room for improvement.

5. Conclusions

Logic factoring is a fundamental but still challenging problem in multiple-level logic synthesis. In this paper, we presented an exact method which finds the minimum factored form of an incompletely specified Boolean function. The problem is formulated as a quantified Boolean formula and is solved by general-purpose QBF solver. We also proposed a novel graph structure, called an X-B tree, which implicitly enumerates binary trees. Using this graph structure, the factoring problem is compactly transformed into a QBF. Using three sets of benchmark functions, we empirically demonstrated the quality of the solutions and the runtime complexity of the proposed method.

References

1) Lawler, E.L.: An Approach to Multilevel Boolean Minimization, *J. ACM*, pp.283–295 (1964).
2) Davidson, E.: An Algorithm for NAND Decomposition under Network Constraints, *IEEE Trans. Comput.*, Vol.18, pp.1098–1109 (1969).
3) Drechsler, R. and Gunther, W.: Exact Circuit Synthesis, *IEEE Int. Workshop on Logic Synthesis* (1998).
4) Silva, J.P.M. and Sakallah, K.A.: GRASP—A New Search Algorithm for Satisfiability, *Proc. IEEE Int. Conf. Comput. Aided Des.*, pp.220–227 (1997).
5) Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L. and Malik, S.: Chaff: Engineering an Efficient SAT Solver, *Proc. ACM/IEEE Design Automation Conf.*, pp.530–535 (2001).
6) Larrabee, T.: Test Pattern Generation Using Boolean Satisfiability, *IEEE Trans. Comput. Aided Des.*, Vol.11, No.1, pp.4–15 (1992).
7) Biere, A., Cimatti, A., Clarke, E. and Zhu, Y.: Symbolic Model Checking without BDDs, *Proc. ACM/IEEE Design Automation Conf.*, pp.193–207 (1999).

8) Feldmann, R., Monien, B. and Schamberger, S.: A Distributed Algorithm to Evaluate Quantified Boolean Formulas, *Proc. National Conf. Artificial Intelligence*, pp.285–290 (2000).
9) Letz, R.: Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas, *Proc. TABLEAUX2002, Vol.2381 of LNAI*, pp.160–175 (2002).
10) Benedetti, M.: sKizzo: a QBF decision procedure based on Propositional Skolemization and Symbolic Reasoning, *ITC-Irst Tech. Rep. TR04-11-03* (2004).
11) Biere, A.: Resolve and Expand, *Proc. Intl. Conf. Theory and Applications of Satisfiability Testing, LNCS, Springer* (2005).
12) Proskurowski, A.: On the Generation of Binary Trees, *J. ACM*, Vol.27, No.1, pp.1–2 (1980).
13) Zaks, S.: Lexicographic Generation of Ordered Trees, *Theoretical Computer Science*, Vol.10, pp.63–82 (1980).
14) Brand, D.: Verification of Large Synthesized Designs, *Proc. IEEE Int. Conf. Comput. Aided Des.*, pp.534–537 (1993).
15) Brayton, R., Sangiovanni-Vincentelli, A., Hachtel, G. and McMullin, C.: *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston (1984).
16) Brayton, R.K., Rudell, R., Sangiovanni-Vincentelli, A. and Wang, A.R.: MIS: A Multiple-level Logic Optimization System, *IEEE Trans. Comput. Aided Des.*, Vol.6, No.6, pp.1062–1081 (1987).
17) Sentovich, E.M., Singh, K.J., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P.R., Brayton, R.K. and Sangiovanni-vincentelli, A.: SIS: A System for Sequential Circuit Synthesis, Technical Report UCB/ERL M92/41, University of California, Berkeley (1992).
18) Cong, J., Peck, J. and Ding, Y.: RASP: A General Logic Synthesis System for SRAM-based FPGAs, *Proc. ACM/SIGDA Int. Symp. FPGAs*, pp.137–143 (1996).

(Received May 29, 2010)
(Revised September 3, 2010)
(Accepted October 22, 2010)
(Released February 8, 2011)

(Recommended by Associate Editor: *Yusuke Matsunaga*)

Hiroaki Yoshida received his B.S., M.S. and Ph.D. degrees in electronic engineering from the University of Tokyo, Tokyo, Japan, in 2000, 2002, and 2007, respectively. From 2002 to 2006, he was a Senior Software Engineer at Zenasis Technologies, Inc., in San Jose, CA., where he was working on the development of a leading-edge logic/physical/transistor-level timing optimization tool. He is currently a Project Assistant Professor with VLSI Design and Education Center (VDEC), the University of Tokyo. His research interests include high-level, logic-level and transistor-level optimization of high-performance digital circuits.

Masahiro Fujita received his B.S. degree in electrical engineering in 1980, and M.S. and Ph.D. degrees in information engineering from the University of Tokyo, Tokyo, Japan in 1982 and 1985, respectively. From 1985 to 1993, he was a Research Scientist with Fujitsu Laboratories, Kawasaki, Japan. From 1994 to 1999, he was the Director of the Advanced Computer-Aided Design Research Group, Fujitsu Laboratories of America, Sunnyvale, CA. He is currently a Professor in the Department of Electrical Engineering, the University of Tokyo, Tokyo, Japan. He has been on program committees for many conferences dealing with digital design and is an Associate Editor of Formal Methods on Systems Design. His primary research interest is in the computer-aided design of digital systems. Dr. Fujita received the Sakai Award from the Information Processing Society of Japan in 1984.