SPFD: A New Method to Express Functional Flexibility

Shigeru Yamashita, Hiroshi Sawada, and Akira Nagoya, *Member, IEEE*

***Abstract*—In this paper, we propose a unique way to express functional flexibility by using sets of pairs of functions called "Sets of Pairs of Functions to be Distinguished"(SPFDs) rather than traditional incompletely specified functions. This method was very naturally derived from a unique concept for distinguishing two logic functions, which we explain in detail in this paper.**

The flexibility represented by an SPFD assumes that the internal logic of a node in a circuit can be freely changed. SPFDs make good use of this assumption, and they can express larger flexibility than incompletely specified functions in some cases.

Although the main subject of this paper is to explain the concept of SPFDs, we also present an efficient method for calculating the functional flexibilities by SPFDs because the concept becomes useful only if there is an efficient calculation method for it. Moreover, we present a method to use SPFDs for circuit transformation along with a proof of the correctness of the method.

We further make a comparison between SPFDs and compatible sets of permissible functions (CSPFs), which express functional flexibility by incompletely specified functions.

As an application of SPFDs, we show a method to optimize LUT (look-up table) networks and experimental results.

***Index Terms*—Circuit optimization, functional flexibility, logic function, look-up table, SPFD.**

I. Introduction

WE CAN sometimes replace the logic function[1] at a certain point (the output or an input connection of a gate) in a combinational logic circuit with another logic function without changing the functionality of the circuit.

In this paper, the condition in which an alternative function can replace a function at a certain point in a circuit is called the "*functional flexibility* (or simply *flexibility*) of the point." Although the concept of functional flexibility is similar to that of "permissible function" in [1], we use the word "flexibility" because we consider changing the internal logic of a node in a circuit, unlike [1].[2]

There are many optimization methods for a combinational multi-level circuit that use functional flexibility, and these are divided into two groups.

- Many optimization methods *explicitly* express functional flexibilities [1]–[4], and then use them in various ways for optimization.
- Automatic test pattern generation (ATPG) based optimization methods [5]–[7] use functional flexibility *implicitly*. One of their advantages is that they can be applied to large circuits due to their implicit usage of functional flexibility.

To express functional flexibilities explicitly, incompletely specified functions or Boolean relations [8]–[10] have been used. Incompletely specified functions can be divided as follows based on the calculation methods used:

- **satisfiability don't cares (SDC)** [11]. These can be calculated from the primary inputs toward the primary outputs of a circuit.
- **observability don't cares (ODC)** [11]. These can be calculated from the primary outputs toward the primary inputs of a circuit. In practical usage, **compatible observability don't cares (CODC)** [3], [4] or **compatible sets of permissible functions (CSPFs)** [1], which are subsets of ODCs, are usually used. This is because they can be relatively easily used to optimize circuits although their representations are less powerful than those of ODCs.

Boolean relations [8]–[10], on the other hand, can express functional flexibilities of multiple points at the same time more efficiently than incompletely specified functions. However, it is difficult to calculate functional flexibilities by Boolean relations and to optimize circuits using them.

A new category for expressing functional flexibility was introduced in [12]. This uses a set of pairs of functions called **Set of Pairs of Functions to be Distinguished (SPFD)**. The concept of SPFDs, where the role of a logic functions is thought to "**distinguish** two functions," differs from those of conventional expressions for the functional flexibility mentioned above. The above concept of SPFDs can efficiently incorporate the possibility that we can modify the internal logic of each node into functional flexibility expressions. There is no straightforward way to achieve this by using the concept of incompletely specified functions, Boolean relations or ATPG based methods. Therefore, we believe the concept of SPFDs provides a new and powerful way to express functional flexibility. Table I summarizes the methods for explicitly expressing functional flexibility. By using the calculation method presented in this paper, representations for functional flexibility by SPFDs have the following properties.

- They are sometimes *greater* than those by CSPFs and CODCs. (If a representation for functional flexibility of a

Manuscript received September 29, 1999; revised February 25, 2000. This paper was recommended by Associate Editor L. Stok.

S. Yamashita and H. Sawada are with NTT Communication Science Laboratories, Kyoto 619-0237, Japan (e-mail: ger@cslab.kecl.ntt.co.jp; sawada@cslab.kecl.ntt.co.jp}).

A. Nagoya is with NTT Network Innovation Laboratories, Kanagawa 239-0847, Japan (e-mail: nagoya@exa.onlab.ntt.co.jp).

Publisher Item Identifier S 0278-0070(00)06428-9.

[1]Logic functions are simply called *functions* in this paper if the context is clear.

[2]Another reason is that some people in the field consider the term "permissible function" in [1] to be unique.

TABLE I
CLASSIFICATION OF FUNCTIONAL FLEXIBILITY REPRESENTATIONS

Representation	Computation		References
Incompletely specified functions	SDC		[11]
	ODC	CODC	[3], [4]
		CSPF	[1]
Boolean relation			[8], [9]
SPFD			[12], **this paper**

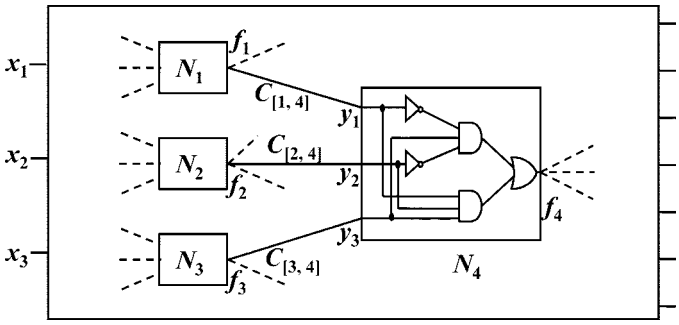

Fig. 1. A Boolean network.

certain point in a circuit expresses more alternative functions at the point than another representation, we say "the representation is *greater* than the other.")

- They are as easy to treat as CSPFs and CODCs.

After the paper [12], many other papers [13]–[16] began referring to the idea expressed in the paper. Originally, paper [12] was presented in the context of field-programmable gate array (FPGA) synthesis; it did not focus on the idea of SPFDs. Therefore, in this paper, we focus much more on our concept of SPFDs and present a thorough explanation of the concept. We also add other experimental results and a comparison of SPFDs and CSPFs.

In the remainder of this paper, we define notation and explain conventional concepts for expressing functional flexibility in Section II. Then, we introduce our concept for expressing functional flexibility, which is useful for understanding SPFDs, and some definitions concerning SPFDs in Section III. A concrete procedure for calculating functional flexibility by SPFDs is presented in Section IV. The use of SPFDs is discussed in Section V. We also present a comparison between SPFDs and CSPFs in Section VI. Section VII presents some experimental results. Finally, we conclude this paper in Section VIII.

II. PRELIMINARIES

We represent combinational logic circuits by *Boolean networks*. A Boolean network (or simply *network*) is defined as a directed acyclic graph (DAG) such that each node has a logic function (not restricted to a simple function such as NAND or NOR) with respect to its inputs. Fig. 1 shows an example of a Boolean network.

We use the following notation.

N_i represents a node in a network.

$C_{[i,j]}$ represents the connection from N_i's output to one of N_j's inputs. If there is $C_{[i,j]}$, N_i is called a *fanin node* of N_j, and N_j is called a *fanout node* of N_i.

f_i represents the logic function at the output of N_i with respect to the primary inputs of the network. This is sometimes called the "global" function of the node.

Each node N_i corresponds to a completely specified logic function with respect to its inputs. We call this logic function the "*internal logic*" of N_i. Suppose N_i's fanin nodes are $N_{i_1}, N_{i_2}, \ldots,$ and N_{i_n}. The internal logic of N_i can be represented by an expression using *local variables* $y_1, y_2, \ldots,$ and y_n that corresponds to $f_{i_1}, f_{i_2}, \ldots,$ and f_{i_n}, respectively. For example, in Fig. 1, the internal logic of N_4 is expressed as $(y_1 \cdot y_2 \cdot y_3 + \overline{y_1} \cdot \overline{y_2} \cdot y_3)$, where y_1, y_2 and y_3 correspond to f_1, f_2 and f_3, respectively.

Our usage of "functional flexibility" is explained as follows.

- The condition in which an alternative function can replace the global function at the output of N_i is called the "*functional flexibility* of N_i."
- The condition in which an alternative function can replace the global function used for an input to N_j, which corresponds to $C_{[i,j]}$, is called the "*functional flexibility* of $C_{[i,j]}$."

Note that the functional flexibility of $C_{[i,j]}$ generally differs from that of $C_{[i,k]}$ although the global logic functions corresponds to $C_{[i,j]}$ and $C_{[i,k]}$ are the same as f_i.

When we use an incompletely specified function to represent the functional flexibility of N_i, we traditionally interpret the role of the output function of N_i as follows.

Interpretation 1: For some primary input patterns, the function must become one, and for other primary input patterns, it must become zero. This information must be propagated to the fanout nodes of N_i. For the rest of the input patterns, the function can become either one or zero, that is, *DC (Don't Care)*, which means that the logical value at N_i does not affect the primary outputs for the input patterns. This interpretation indeed represents the functional flexibility of N_i.

Interpretation 1 is very natural and efficient when each node in a network can provide only an already fixed logic such as NAND or NOR. Based on the interpretation, we can represent the functional flexibility of N_i by using an incompletely specified function whose ON-Set, OFF-Set and DC (Don't Care)-Set correspond to the input patterns, where f_i must become one, zero, and don't care in *Interpretation 1*. Accordingly, many methods have been proposed to represent functional flexibility based on *Interpretation 1*.

Fig. 2 shows an example for CSPFs [1] for the network shown in Fig. 1. For simplicity, we suppose that the network has only three primary inputs x_1, x_2, and x_3. If f_1, f_2, and f_3 are shown as in Fig. 2 and the internal logic of N_4 is $(y_1 \cdot y_2 \cdot y_3 + \overline{y_1} \cdot \overline{y_2} \cdot y_3)$, f_4 is calculated by $(f_1 \cdot f_2 \cdot f_3 + \overline{f_1} \cdot \overline{f_2} \cdot f_3)$ as shown in Fig. 2. Then, if the functional flexibility of N_4 is represented by CSPF_4 in Fig. 2, the functional flexibilities of $C_{[1,4]}$, $C_{[2,4]}$, and $C_{[3,4]}$ are calculated as $\mathrm{CSPF}_{[1,4]}$, $\mathrm{CSPF}_{[2,4]}$, and $rmCSPF_{[3,4]}$ in Fig. 2, respectively. They are calculated by treating the internal logic of N_4 as a network as shown in Fig. 1 and using the method in [1]. In the example, as long as f_1, f_2, and f_3 change within the flexibilities represented by $\mathrm{CSPF}_{[1,4]}$, $\mathrm{CSPF}_{[2,4]}$, and $\mathrm{CSPF}_{[3,4]}$, it is guaranteed that f_4 changes within the flexibility represented by CSPF_4.

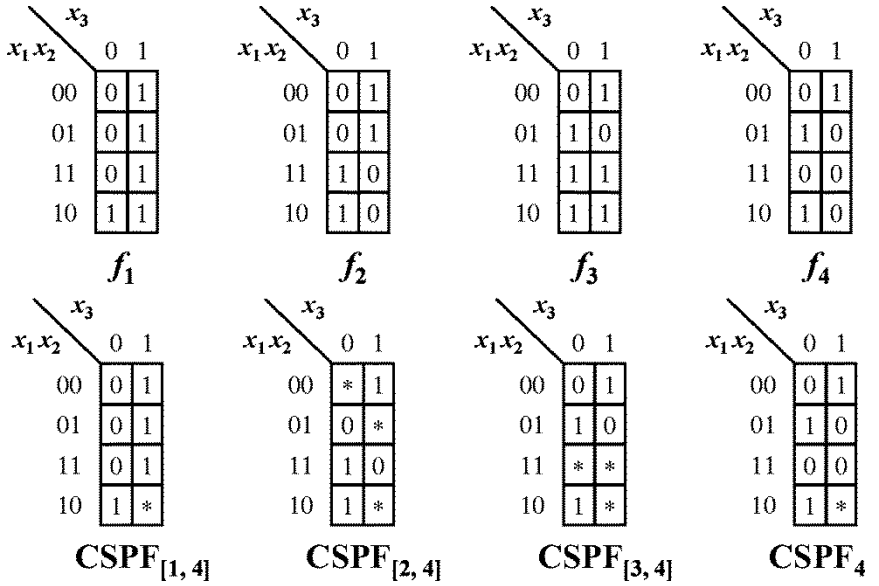

Fig. 2. Functional flexibility by incompletely specified functions.

III. A new Concept for Expressing Functional Flexibility

In this section, we present a new interpretation of the role of logic functions that is different from *Interpretation 1*. This interpretation provides us with a new functional flexibility expression as we see below.

When we design logic circuits at the technology-independent level, we are not restricted to the idea that the internal logic of each node is already fixed. Moreover, if we target LUT (look-up table)-based FPGAs or programmable logic arrays (PLAs), we can assume that the internal logic of each node is not fixed even at the technology-dependent logic synthesis. The new interpretation in this section provides a greater functional flexibility representation than that by incompletely specified functions if we assume that the internal logic of each node can be freely changed.

At first, we define what it means **to distinguish a pair of functions**.

Definition 1: For two logic functions f and g, if $f(X)$ always becomes one for all primary input vectors X where $g(X)$ becomes one, f is said to **include** g, also written $g \leq f$, or $g \Rightarrow f$. Note that "f includes g" $\Leftrightarrow g \cdot \overline{f} = 0$.

Definition 2: If either one of the following two conditions is satisfied, a function f is said to **distinguish** a pair of functions g_1 and g_2.

Condition 1: f includes g_1, and $\overline{f}$ includes g_2. ($g_1 \leq f \leq \overline{g_2}$.)

Condition 2: f includes g_2, and $\overline{f}$ includes g_1. ($g_2 \leq f \leq \overline{g_1}$.)

In other words, f is said to distinguish g_1 and g_2 when one of ON-Set and OFF-Set of f includes g_1 and the other includes g_2.

Now, we formalize an SPFD as follows.

Definition 3: **SPFD (Set of Pairs of Functions to be Distinguished)** is a set of pairs of functions that can be represented as $\{(g_{1a}, g_{1b}), (g_{2a}, g_{2b}), \ldots, (g_{na}, g_{nb})\}$.

We also define the following terminology.

Definition 4: For an SPFD $= \{(g_{1a}, g_{1b}), (g_{2a}, g_{2b}), \ldots, (g_{na}, g_{nb})\}$, a function f is said to **satisfy the SPFDs condition** or simply **satisfy the SPFD** if f distinguishes g_{ia} and g_{ib} for all i.

Note that it is implicitly assumed that ($g_{ia} \cdot g_{ib} = 0$) in the above SPFD; otherwise, no function can distinguish g_{ia} and g_{ib}.

Suppose N_j is a fanin node of N_i. When the internal logic of N_i can be freely changed, it can always be changed so that f_i does not change even if f_j is negated. This means that whether f_j is one or zero is not important to N_i. A truly important piece of information that must be propagated to N_i from N_j can be thought of as follows.

Interpretation 2: The values of a function f must be different between certain primary input patterns and other input patterns. (Here, "different" means that one value is one and the other value is zero.) Only this information is really important to the nodes that use f as their inputs.

From *Interpretation 2*, we can conclude that the truly useful information that a function f must propagate is that it distinguishes a certain set of functions from another set of functions. Finally, we reach the following proposal.

Proposal 1: It can always be assumed that the alternative functions for f_i must satisfy a certain SPFD. Therefore, we propose using an SPFD to represent the functional flexibility of N_i.

We define other notation here.

- SPFD_i is an SPFD that represents the functional flexibility of N_i.
- $\mathrm{SPFD}_{[i,j]}$ is an SPFD that represents the functional flexibility of $C_{[i,j]}$.

SPFD_i and $\mathrm{SPFD}_{[i,j]}$ are sometimes referred to simply as "the SPFD of the node" and "the SPFD of the connection," respectively.

If an SPFD has only one pair, the functional flexibility represented by the SPFD can also be represented by an incompletely specified function.[3] If an SPFD has more than one pair, how-

[3]To be exact, we need an incompletely specified function and its simple negation to represent the same flexibility as the SPFD.

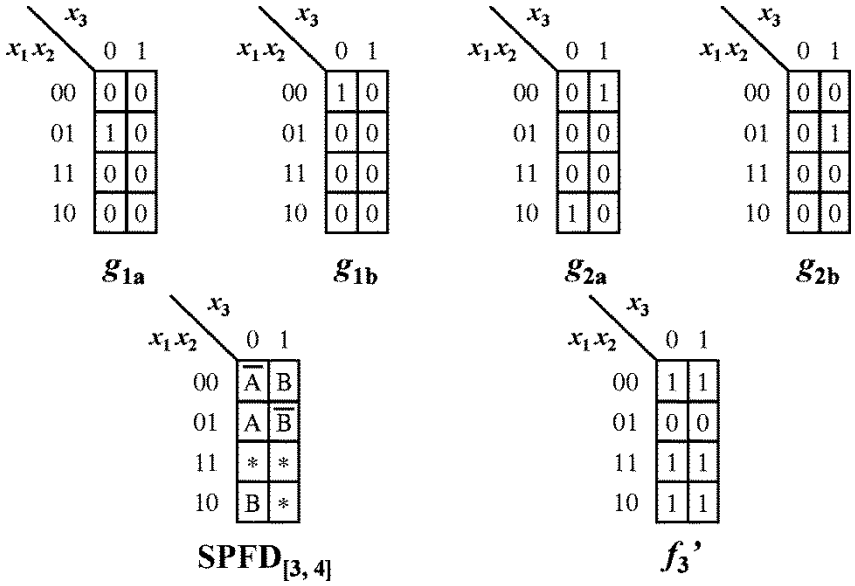

Fig. 3. Functional flexibility by an SPFD.

ever, the functional flexibility represented by the SPFD cannot be represented by using an incompletely specified function, as will be described later.

In the example shown in Figs. 1 and 2, the method described in the next section calculates $\mathrm{SPFD}_{[3,4]}$ as $\{(g_{1a}, g_{1b}), (g_{2a}, g_{2b})\}$, where g_{1a}, g_{1b}, g_{2a} and g_{2b} are as shown in Fig. 3.

The functions that satisfy $\mathrm{SPFD}_{[3,4]}$ are functions whose intuitive truth tables are as follows:

- the values corresponding to A and $\overline{A}$ in the truth table shown as "$\mathrm{SPFD}_{[3,4]}$" in Fig. 3 must be different (one is one and the other is zero);
- the values corresponding to B and $\overline{B}$ in the truth table shown as "$\mathrm{SPFD}_{[3,4]}$" in Fig. 3 must be different (one is one and the other is zero).

Therefore, we can see that f_3' in Fig. 3 satisfies $\mathrm{SPFD}_{[3,4]}$. Since f_3' is not included in either $\mathrm{CSPF}_{[3,4]}$ in Fig. 2 or the simple negation of $\mathrm{CSPF}_{[3,4]}$, we cannot replace f_3 with f_3' if we use $\mathrm{CSPF}_{[3,4]}$ to represent the functional flexibility.

The above example indicates that SPFDs are greater than incompletely specified functions. However, the concept of SPFDs are really useful only if we have a method to calculate SPFDs and to utilize them efficiently, which is discussed in the rest of this paper.

IV. Calculating SPFDs

Here, we propose a method to calculate the functional flexibility of a function using an SPFD. Note that the method is not unique. In our method, SPFD_i has only one pair for efficiency, although $\mathrm{SPFD}_{[i,j]}$ generally has more than one pair.

A. The Outline of Our Calculation Method

Our method calculates the functional flexibility of each node from the primary outputs like CSPF [1] and CODC [3]. The procedure of calculating SPFD_i for each node N_i is as follows.

1) For all nodes N_i, calculate f_i.
2) For all nodes N_i, set $\mathrm{SPFD}_i = null$ (initial value).
3) For all primary input nodes N_i, call the following recursive procedure $Cal\_\mathrm{SPFD}(N_i)$.

Procedure $Cal\_\mathrm{SPFD}(N_i)$:

Step 1) If $\mathrm{SPFD}_i \neq null$ (this means SPFD_i has already been calculated), return.

Step 2) If N_i is a primary output node of the network, let SPFD_i be $\{(f_i, \overline{f_i})\}$.

Step 3) In the other two cases, for each node N_j that is a fanout node of N_i, call $Cal\_\mathrm{SPFD}(N_j)$.

Step 4) Calculate SPFD_i in the following way. We can assume that $\mathrm{SPFD}_{[i,j]}$ has already been calculated as $\{(g_{[i,j]1a}, g_{[i,j]1b}), (g_{[i,j]2a}, g_{[i,j]2b}), \ldots, (g_{[i,j]m_{[i,j]}a}\, g_{[i,j]m_{[i,j]}b})\}$, where N_j is a fanout node of N_i. f_i always satisfies $\mathrm{SPFD}_{[i,j]}$ calculated by our method as described later, and thus we assume that f_i includes $g_{[i,j]ka}$ and $\overline{f_i}$ includes $g_{[i,j]kb}$ for all j, k. In the above assumption, calculate SPFD_i as $\{(\sum_{j,k} g_{[i,j]ka}, \sum_{j,k} g_{[i,j]kb})\} = \{(f_{ON}, f_{OFF})\}$.

Step 5) If N_i is not a primary input node of the network, call $Propagate\_\mathrm{SPFD}(N_i)$, which will be described later.

B. Calculating the SPFDs of the Input Connections of a Node

Here, we explain how our method calculates the SPFDs of the input connections of N_i. Suppose N_i has n fanin nodes that are $N_{i_1}, N_{i_2}, \ldots$, and N_{i_n}. In the rest of paper, in order to avoid subscripted subscripts, we refer to $N_{i_1}, N_{i_2}, \ldots$, and N_{i_n} as simply $N_1, N_2, \ldots$, and N_n, respectively, when we focus on only one node N_i. In other words, we assume N_i's fanin nodes are $N_1, N_2, \ldots$, and N_n, whose output functions are referred to as $f_1, f_2, \ldots$, and f_n, respectively.

When SPFD_i has already been calculated as $\{(f_{ON}, f_{OFF})\}$, $\mathrm{SPFD}_{[k,i]}$ can be calculated by the following procedure $Propagate\_\mathrm{SPFD}$, which is called from Cal_SPFD.

Procedure $Propagate\_\mathrm{SPFD}(N_i)$:

Step 1) For each N_i's fanin node N_k, set $\mathrm{SPFD}_{[k,i]} = \emptyset$ (initial value).

Step 2) Make a new SPFD F, which has the same information as SPFD_i but has more than one pair of functions, by the procedure $Divide\_\mathrm{SPFD}(N_i)$, which will be described later.

Step 3) Assign each pair of functions in F to $\mathrm{SPFD}_{[k,i]}$, where N_k is one of the fanin nodes of N_i, by the procedure $Assign\_\mathrm{SPFD}(F, N_i)$, which will be described later.

Procedure $Divide\_\mathrm{SPFD}(N_i)$:

Step 1) Calculate 2^n logic functions from all the possible logical products of $f_1, f_2, \ldots$, and f_n, where each f_j is negated or not (e.g., $\overline{f_1} \cdot \overline{f_2} \cdots \overline{f_n}, \ldots, f_1 \cdot f_2 \cdots f_n$). Let these functions be $b_{0\ldots0}, \ldots, b_{1\ldots1}$, where the index of b_v is an n bits binary number that satisfies the following condition.

- The kth bit (from the left) of v is zero or one, depending on whether f_k is negated or not in b_v.

For example, $b_{011} = \overline{f_1} \cdot f_2 \cdot f_3$ when $n = 3$. This rule for v will be used in $Assign\_\mathrm{SPFD}$ and $Modify\_Logic$, which will be described later.

Step 2) For all b_v, calculate $a_v = b_v \cdot (f_{ON} + f_{OFF})$. This is because $(f_{ON} + f_{OFF})$ is the Care-Set for f_i.

Step 3) Make the following two sets from a_v calculated in Step 2.

$F1$
$= \{a_v | a_v$ is not constantly 0, and is included in $f_{ON}.\}$
$F0$
$= \{a_v | a_v$ is not constantly 0, and is included in $f_{OFF}.\}$

Note that each a_v sometimes becomes the constant zero function, or is always included in either one of f_{ON} and f_{OFF}.

Step 4) Calculate a Cartesian product $F = F1 \times F0$.

Next, we assign each pair of functions (a_l, a_m) in F to one of $\mathrm{SPFD}_{[j,i]}$ by $Assign\_\mathrm{SPFD}(F, N_i)$. If the indexes of a_l and a_m are based on the rule in Step 1 of $Divide\_\mathrm{SPFD}(N_i)$, one of the expressions as logical products for a_l and a_m by using $f_1, f_2, \ldots$ and f_n always has f_k, and the other always has $\overline{f_k}$, where k is one of the different bits between l and m compared as binary numbers from the left. In such cases, therefore, we know that f_k can distinguish a_l and a_m. For example, (a_{010}, a_{001}) can be distinguished by both f_2 and f_3. Accordingly, we can assign (a_{010}, a_{001}) to either one of the roles of f_2 and f_3. In such cases, we can consider various heuristics to determine which role the pair is assigned to. For example, we can consider a heuristic that does not assign pairs to an input connection that we want to remove. For simplicity, in this paper we assign such pairs to $\mathrm{SPFD}_{[k,j]}$ with the smallest k among all candidates.

Here we introduce the following operator to explain $Assign\_\mathrm{SPFD}(F, N_i)$.

Definition 5: $diff(l, m)$ is an operation to find the left most bit where the corresponding bits of l and m are different.

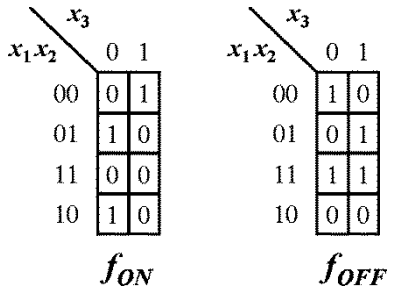

Fig. 4. An example of SPFD calculation (1).

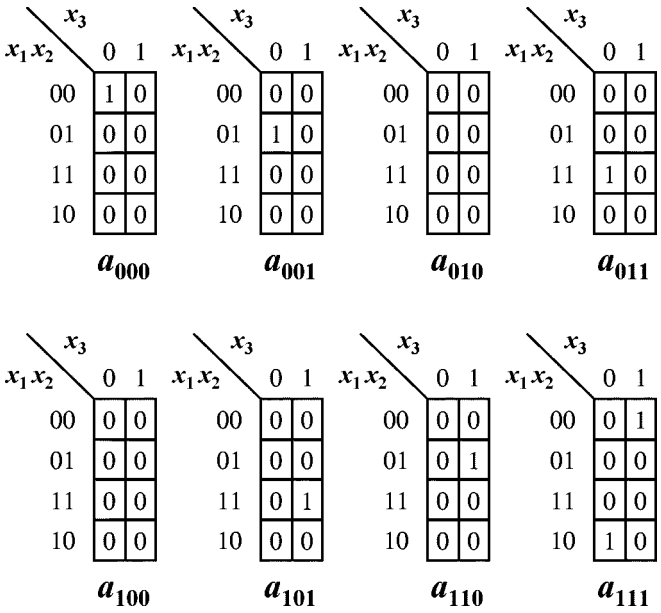

Fig. 5. An example of SPFD calculation (2).

For example, $diff((010), (001)) = 2$.

Procedure $Assign\_\mathrm{SPFD}(F, N_i)$:

Step 1) Select one pair of functions in F as (a_l, a_m) one by one, and go to Step 2. If there is no pair to select, return.

Step 2) Update $\mathrm{SPFD}_{[k,i]}$ as $\mathrm{SPFD}_{[k,i]} \cup \{(a_l, a_m)\}$, where $k = diff(l, m)$. Go to Step 1.

Note that f_k can distinguish a_l and a_m, which guarantees that f_k currently satisfies $\mathrm{SPFD}_{[k,i]}$.

C. An Example of the Procedure $Propagate\_\mathrm{SPFD}(N_i)$

The procedure is explained using the following example. Fig. 1 shows a Boolean network where N_4 has three fanin nodes N_1, N_2, and N_3, and f_1, f_2, and f_3 are as shown in Fig. 2. SPFD_4 has already been calculated as $\{(f_{ON}, f_{OFF})\}$, where f_{ON} and f_{OFF} (shown in Fig. 4) are the ON-Set and the OFF-Set of "CSPF_4" shown in Fig. 2. In the above situation, $\mathrm{SPFD}_{[1,4]}$, $\mathrm{SPFD}_{[2,4]}$, and $\mathrm{SPFD}_{[3,4]}$ are calculated as follows. Note that the conditions are the same as the case where $\mathrm{CSPF}_{[1,4]}$, $\mathrm{CSPF}_{[2,4]}$, and $\mathrm{CSPF}_{[3,4]}$ in Fig. 2 are calculated in Section II.

At first, $\mathrm{SPFD}_{[1,4]}$, $\mathrm{SPFD}_{[2,4]}$, and $\mathrm{SPFD}_{[3,4]}$ are set to $\emptyset$ in Step 1 in $Propagate\_\mathrm{SPFD}(N_4)$. Next, $Divide\_\mathrm{SPFD}(N_4)$ proceeds as follows.

$$F1=\{a_{001}, a_{111}\}$$
$$F0=\{a_{000}, a_{011}, a_{101}, a_{110}\}$$
$$F=\begin{Bmatrix}(a_{001},a_{000}),(a_{001},a_{011}),(a_{001},a_{101}),(a_{001},a_{110}),\\(a_{111},a_{000}),(a_{111},a_{011}),(a_{111},a_{101}),(a_{111},a_{110})\end{Bmatrix}$$

Fig. 6. An example of SPFD calculation (3).

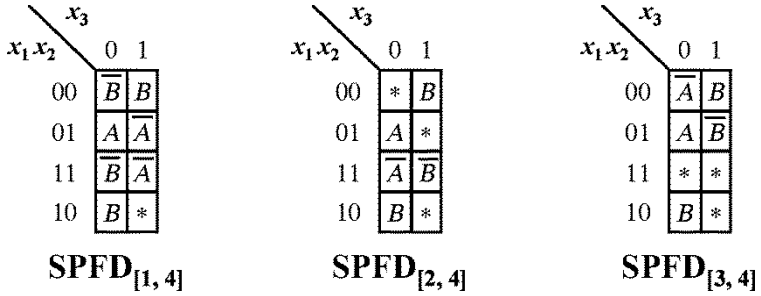

Fig. 7. Conditions by SPFDs.

- At Step 2, $a_{000}, \ldots, a_{111}$ are calculated as shown in Fig. 5. For example, a_{011} is calculated from $\overline{f_1} \cdot f_2 \cdot f_3 \cdot (f_{ON} + f_{OFF})$.
- At Step 3, $F0$ and $F1$ are calculated as shown in Fig. 6.
- At Step 4, F is calculated as shown in Fig. 6.

The above F represents the same flexibility as SPFD_4, but F has more pairs than SPFD_4, which means that the former has potentially more information than the latter. From another point of view, the information of SPFD_4 ($=\{(f_{ON}, f_{OFF})\}$) is divided by $Divide\_\mathrm{SPFD}(N_4)$ into many pieces of information in F, but the total information is the same between F and SPFD_4. Therefore, the condition represented by SPFD_4 can be thought of as stating that f_4 must distinguish a_l and a_m for all pairs (a_l, a_m) in F. We can always modify the internal logic of N_4 such that f_4 satisfies the condition by F if for each pair (a_l, a_m) at least one of f_1, f_2, and f_3 distinguishes a_l and a_m. We can actually replace f_1, f_2, and f_3 with another group of functions able in total to satisfy the above condition by F, even if the number of functions in the group is not three. From the above interpretation of F, which can be thought of as the heart of the concept of SPFDs, we can consider that F represents the functional flexibilities of all N_4's fanin nodes together.

It is possible to resynthesize N_4 using the functional flexibility by F, but we only consider how to propagate the functional flexibility to the fanin nodes of N_4 here. To do so, $Assign\_\mathrm{SPFD}(F, N_4)$ assigns each pair of functions in F to one of $\mathrm{SPFD}_{[1,4]}$, $\mathrm{SPFD}_{[2,4]}$, and $\mathrm{SPFD}_{[3,4]}$. As we will see in the following, the procedure assigns pairs so that the current f_1, f_2, and f_3 satisfy $\mathrm{SPFD}_{[1,4]}$, $\mathrm{SPFD}_{[2,4]}$, and $\mathrm{SPFD}_{[3,4]}$, respectively.

- At Step 1, $Assign\_\mathrm{SPFD}(F, N_4)$ selects the first pair (a_{001}, a_{000}); go to Step 2.
- It then adds (a_{001}, a_{000}) to $\mathrm{SPFD}_{[3,4]}$ since $diff((001), (000)) = 3$.

a_{001} and a_{000} correspond to $(\overline{f_1} \cdot \overline{f_2} \cdot f_3)$ and $(\overline{f_1} \cdot \overline{f_2} \cdot \overline{f_3})$, respectively. The difference between the two logical products is whether f_3 is negated or not, which means that only f_3 can distinguish the pair. We can easily find functions able to distinguish the pair by only checking the indexes of a_l and a_m, if the indexes of a_l and a_m are based on the rule in Step 1 of $Divide\_\mathrm{SPFD}(N_i)$. We assign each pair in F to one of $\mathrm{SPFD}_{[1,4]}$, $\mathrm{SPFD}_{[2,4]}$, and $\mathrm{SPFD}_{[3,4]}$ in the same way.

All bits of the indexes of (111) and (000) are different. As mentioned before, although we have freedom in assigning (a_{111}, a_{000}) to either one of the roles of f_1, f_2, and f_3, $Assign\_\mathrm{SPFD}$ assigns it to $\mathrm{SPFD}_{[1,4]}$, and the SPFDs are calculated as follows:

$$\begin{aligned}\mathrm{SPFD}_{[1,4]} &= \{(a_{001}, a_{101}), (a_{001}, a_{110})\\ &\qquad (a_{111}, a_{000}), (a_{111}, a_{011})\}\\ \mathrm{SPFD}_{[2,4]} &= \{(a_{001}, a_{011}), (a_{111}, a_{101})\}\\ \mathrm{SPFD}_{[3,4]} &= \{(a_{001}, a_{000}), (a_{111}, a_{110})\}.\end{aligned}$$

The functions satisfying these SPFDs are functions whose intuitive truth tables can be represented as in Fig. 7.

The truth tables in the figure have the meaning that the values of A and B must be different from those of $\overline{A}$ and $\overline{B}$, respectively.[4] In other words, we can assign either one or zero to A, B, and $*$ in the truth tables. Therefore, $\mathrm{SPFD}_{[1,4]}$ represents $2^3 = 8$ alternative functions for f_1. $\mathrm{SPFD}_{[2,4]}$ and $\mathrm{SPFD}_{[3,4]}$ represent 32 alternative functions. From Fig. 2, we see that CSPFs can represent only two, eight, and eight alternative functions for f_1, f_2, and f_3, respectively, for the same situation.

Suppose N_j and N_k are fanin nodes of N_i. If f_k changes after calculating the flexibility of $C_{[j,i]}$, the flexibility of $C_{[j,i]}$ may change. However, as long as f_k changes within the flexibility by $\mathrm{SPFD}_{[k,i]}$, at the same time f_j can also change within the flexibility by $\mathrm{SPFD}_{[j,i]}$ if the internal logic of N_i is appropriately modified. Therefore, we can transform multiple points in a network at the same time by using the functional flexibilities by SPFDs. This property of SPFDs is the same as that of CSPFs and CODCs ("C" means compatible.), and very useful for efficient network transformations. How to modify the internal logic of a node at transformation and its correctness will be discussed in Section V.

[4]Note that there is no correlation between the same letters (A and B) in the different truth tables, such as $\mathrm{SPFD}_{[1,4]}$ and $\mathrm{SPFD}_{[2,4]}$.

```
Procedure Modify_Logic(N_i)
{
    for ( r = 1; r ≤ l_1; r++) {
        a_l = the r-th function in F1.
        for ( j = 1; j ≤ l_0; j++) {
            a_m = the j-th function in F0.
            k = diff(l, m).
            if (f_k includes a_l) {
                m_r = m_r · y_k.
            }
            else{ /* f_k' includes a_l in this case. */
                m_r = m_r · y_k'.
            }
        }
    }
}
```

Fig. 8. **Procedure** $Modify\_Logic(N_i)$.

From the above example, we see that there are some cases when SPFDs by $Propagate\_\mathrm{SPFD}$ are greater than CSPFs and CODCs, although they are calculated with almost the same efficiency.

V. How to Use SPFDs

A. Modifying the Internal Logic of a Node

If f_l satisfies $\mathrm{SPFD}_{[k,i]}$, $C_{[k,i]}$ can be replaced with N_l's output. We may need to modify the internal logic of N_i of which $C_{[k,i]}$ is a current input connection. The reason is that the functional flexibility by $\mathrm{SPFD}_{[k,i]}$ inherently assumes that the internal logic of N_i can be freely changed.

Suppose N_i has n fanin nodes, which are referred to as $N_1, N_2, \ldots$. and N_n. When the functions $f_1, f_2, \ldots$, and f_n change due to a transformation of a network, the procedure $Modify\_Logic(N_i)$, whose algorithm is shown in Fig. 8, modifies the internal logic of N_i so that f_i satisfies SPFD_i. Note that we replace f_k with another function, which satisfies the corresponding SPFD; accordingly, we can expect $f_1, f_2, \ldots$, and f_n after the transformation to always satisfy $\mathrm{SPFD}_{[1,i]}$, $\mathrm{SPFD}_{[2,i]}$, $\ldots$, and $\mathrm{SPFD}_{[n,i]}$, respectively.

In Fig. 8, let $F1$, $F0$ be sets of functions calculated at Step 3 in $Divide\_\mathrm{SPFD}(N_i)$ when $\mathrm{SPFD}_{[1,i]}$, $\mathrm{SPFD}_{[2,i]}$, $\ldots$, and $\mathrm{SPFD}_{[n,i]}$ are calculated. Let the numbers of functions in $F1$ and $F0$ be l_1 and l_0, respectively. Suppose *local variables* for the internal logic of N_i are $y_1, y_2, \ldots$, and y_n that correspond to $f_1, f_2, \ldots$, and f_n, respectively. The modified logic is calculated as $\sum_{r=1}^{l_1} m_r$, where each m_r is a minterm in the space of the local variables $y_1, y_2, \ldots$, and y_n. Let the initial value of each m_r be constantly one.

For the situation specified by Figs. 1 and 2, the functional flexibility of $C_{[3,4]}$ can be represented by $\mathrm{SPFD}_{[3,4]}$ in Fig. 7, as explained in Section IV-C. Accordingly, the input function to N_4, which corresponds to $C_{[3,4]}$, can be changed to f_3' in Fig. 3. If we replace the function with f_3', we need to modify the internal logic of N_4. The required modification can be done by $Modify\_Logic(N_4)$ as follows. $F1$ and $F0$ are as shown in Fig. 6. For the first function in $F1$ (a_{001}), m_1 is calculated as $\overline{y_1} \cdot \overline{y_2} \cdot \overline{y_3}$ from the following facts, where $y_1, y_2, \ldots$, and y_3 correspond to $f_1, f_2, \ldots$, and f_3', respectively.

- $diff((001), (000)) = 3$;
- $diff((001), (011)) = 2$;
- $diff((001), (101)) = 1$;
- $diff((001), (110)) = 1$;
- $\overline{f_1}$ includes a_{001};
- $\overline{f_2}$ includes a_{001};
- f_3' includes a_{001}.

For the second function in $F1(a_{111})$, m_2 is calculated as $y_1 \cdot y_2 \cdot y_3$ in the same way. In the example, the global function calculated from $(m_1 + m_2)$ actually satisfies SPFD_4, and therefore we know that the internal logic should be modified to $(m_1 + m_2)$. We explain the correctness of $Modify\_Logic$ in the next section.

B. Correctness of $Propagate\_\mathrm{SPFD}$ and $Modify\_Logic$

$Propagate\_\mathrm{SPFD}$ and $Modify\_Logic$ are applied together to transform a network. In this section, we show that we can correctly transform a network by using the two procedures. It is sufficient to prove the following theorem to do so.

Theorem 1: Suppose N_i has n fanin nodes, which are referred to as $N_1, N_2, \ldots$, and N_n, after a transformation of a network. If, for all $k \in \{1, \ldots, n\}$, f_k satisfies $\mathrm{SPFD}_{[k,i]}$, which is calculated by $Propagate\_\mathrm{SPFD}$ for the kth input connection before the transformation, $Modify\_Logic(N_i)$ can always modify the internal logic of N_i so that f_i satisfies SPFD_i.

Proof: The modified logic by $Modify\_Logic(N_i)$ can be represented as follows:

$$\text{the modified logic of } N_i = \sum_{r=1}^{l_1} \prod_{h=1}^{l_0} P_{rh} \tag{1}$$

where each variable in the equation has the following meaning:

- l_1 and l_0 are the numbers of functions in $F1$ and $F0$, respectively, where $F1$ and $F0$ are sets of functions calculated at Step 3 in $Divide\_\mathrm{SPFD}(N_i)$ when the SPFDs of the input connections of N_i are calculated.
- P_{rh} is defined as follows, where y_k is a local variable for the internal logic of N_i, which corresponds to f_k. Let the rth function in $F1$ be a_l, the hth function in $F0$ be a_m, and $k = diff(l, m)$ (l and m are n bits binary numbers).
  (Case 1) if $\underline{f_k}$ includes a_l, $P_{rh} = y_k$.
  (Case 2) if $\overline{f_k}$ includes a_l, $P_{rh} = \overline{y_k}$.

In Step 2 of $Assign\_\mathrm{SPFD}(N_i)$, (a_l, a_m) is added to $\mathrm{SPFD}_{[k,i]}$ if $k = diff(l, m)$. Accordingly, it is guaranteed that the current f_k can distinguish a_l and a_m. Therefore, the following can be obtained:

- the global function represented by P_{rh} becomes one for all of the primary input combinations where $a_l = 1$;
- the global function represented by P_{rh} becomes zero for all of the primary input combinations where $a_m = 1$.

The above leads us to the following:

- for all h, the global function represented by P_{rh} becomes one for all of the primary input combinations where a_l becomes one;
- for all of the primary input combinations where any a_m in $F0$ becomes one, there is at least one h such that the global function represented by P_{rh} becomes zero for all r.

Consequently, the modified f_i becomes one when any a_l in $F1$ becomes one, and zero when any a_m in $F0$ becomes one.

Therefore, f_i can distinguish any function in $F1$ and that in $F0$, that is, it satisfies SPFD_i. ■

C. Optimizing an LUT Network by Using SPFDs

Brayton [13] presents an example where SPFDs can be used for optimizing technology-independent circuits more efficiently than traditional methods. Moreover, the paper suggests that SPFDs may be used for finding Boolean divisors, which cannot be found by traditional methods. However, we present a method to optimize an LUT network in this paper. The reason is that SPFDs can be applied very naturally to LUT networks because the internal logic of each node can be changed without affecting the network cost, which is an essential assumption for the concept of SPFDs.

Procedure $Optimizing\_an\_LUT\_network$:

Step 1) For each node N_i and each connection $C_{[j,k]}$ in the network, calculate f_i, SPFD_i and $\mathrm{SPFD}_{[j,k]}$.

Step 2) Select a connection $C_{[j,k]}$ in the network one by one, and go to Step 3. If all connections in the network have been selected, halt.

Step 3) If $\mathrm{SPFD}_{[j,k]} = \emptyset$, remove $C_{[j,k]}$ and go to Step 5. Otherwise, go to Step 4.

Step 4) If there is a node N_i such that f_i satisfies $\mathrm{SPFD}_{[j,k]}$, replace $C_{[j,k]}$ with N_i's output and go to Step 5. If there is no such node, go to Step 2.

Step 5) If f_k no longer satisfies SPFD_k due to a network transformation in Step 3 or Step 4, apply $Modify\_Logic(N_k)$. Go to Step 2.

We can consider using heuristics for the order of selecting $C_{[j,k]}$ and N_i in Steps 3 and 4. However, in the experiment described in Section VII, we did not use any heuristic for the order of selecting connections and nodes, and we always replaced a connection with another if possible without confirming whether the replacement was really good for improving the network.

VI. A Comparison Between SPFDs and CSPFs

In this section, we compare SPFDs and CSPFs. Our comparison demonstrates why SPFDs are a powerful method for representing functional flexibility.

Here, we consider the functional flexibility of $C_{[j,i]}$ in Fig. 9 by using both an SPFD and a CSPF. Suppose the internal logic of node N_i is expressed by an irredundant sum-of-products form, and f_j is used for both products q_l and q_m, where f_j is negated for q_l as shown in Fig. 9. Let F_{q_l} and F_{q_m} be the global functions that q_l and q_m represent, respectively. Note that all functions such as f_j and F_{q_l} are logic functions of the primary inputs of the network.

The functional flexibility of $C_{[j,i]}$ by a CSPF is calculated as follows. First, the flexibilities of the AND gates corresponding to q_l and q_m are calculated as follows:

- the flexibility of the AND gate corresponding to q_l can be represented such that ON-Set is F_{q_l} and OFF-Set is $\overline{f_i}$;
- the flexibility of the AND gate corresponding to q_m can be represented such that ON-Set is F_{q_m} and OFF-Set is $\overline{f_i}$. (Note that $q_l \cdot q_m = 0$.)

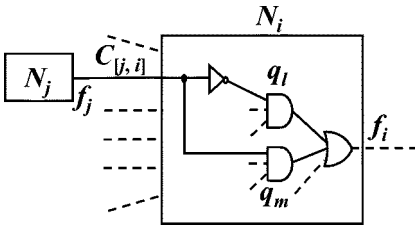

Fig. 9. The internal logic of a node.

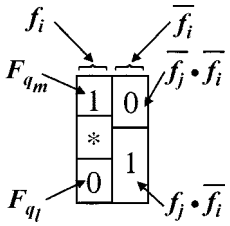

Fig. 10. Flexibility by a CSPF.

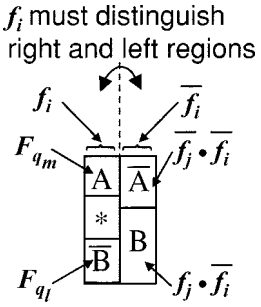

Fig. 11. Flexibility by an SPFD

Next, the flexibility of $C_{[j,i]}$ by a CSPF is calculated:

- from the flexibility of the AND gate corresponding to q_l, f_j must be a function such that OFF-Set is F_{q_l} and ON-Set is $(f_j \cdot \overline{f_i})$;
- from the flexibility of the AND gate corresponding to q_m, f_j must be a function such that ON-Set is F_{q_m} and OFF-Set is $(\overline{f_j} \cdot \overline{f_i})$.

From the above, the flexibility of $C_{[j,i]}$ can be represented by an incompletely specified function whose OFF-Set is $(F_{q_l} + \overline{f_j} \cdot \overline{f_i})$ and whose ON-Set is $(F_{q_m} + f_j \cdot \overline{f_i})$.

Functions that satisfy this flexibility are functions whose intuitive truth tables can be represented as shown in Fig. 10. In the table, the region indicated as F_{q_m}, as an example, corresponds to the region where the values of the truth table for F_{q_m} become one.

Using the concept of SPFDs, we know that f_j must distinguish the following two pairs of functions:

- F_{q_l} and $f_j \cdot \overline{f_i}$;
- F_{q_m} and $\overline{f_j} \cdot \overline{f_i}$.

Functions that satisfy this flexibility are functions whose intuitive truth tables can be represented as shown in Fig. 11. (The interpretations of the figure are the same as those of Figs. 3 and 10.) The CSPFs and the SPFDs of the input connections of N_i vary depending on the order of priority for the input connections. In this example, the priority for $C_{[j,i]}$ is thought to be the same for calculating both the CSPF and the SPFD because in

TABLE II
EXPERIMENTAL RESULTS

Circuit	Initial			CSPF				SPFD			
Name	LUT	conn	lev	LUT	conn	lev	CPU	LUT	conn	lev	CPU
C1908	103	429	13	101	419	13	19.57	98	393	11	13.81
C432	66	275	17	65	270	16	30.2	63	257	16	4.28
alu2	109	482	19	100	434	17	1.05	97	378	17	0.98
alu4	208	862	24	200	723	23	9.12	198	745	22	2.55
apex6	194	894	10	185	811	6	1.29	181	803	10	3.61
apex7	73	292	6	70	257	6	0.29	68	255	6	0.41
cordic	17	76	8	12	52	8	0.52	12	52	8	0.07
dalu	331	1393	16	292	1148	11	7.83	287	1125	9	6.66
des	1118	4663	11	1113	4513	11	114.53	1111	4508	11	292.56
example2	105	451	5	103	441	5	13.78	101	415	5	1.13
frg2	339	1307	8	307	1181	8	0.07	278	1039	8	7.17
i9	138	679	5	137	675	5	9.9	137	675	5	3.14
k2	536	2325	9	533	2267	9	1.4	533	2267	9	11.53
lal	36	142	4	35	141	4	0.47	31	121	3	0.13
rot	192	753	14	188	712	13	50.13	187	707	13	142.66
t481	404	1738	21	388	1568	21	1.28	379	1505	21	8.85
term1	69	303	7	50	214	7	1.25	45	186	6	0.4
too_large	188	882	12	180	822	12	0.02	179	805	12	14.27
ttt2	53	237	4	50	217	4	1.53	47	196	4	0.24
vda	246	1043	8	244	990	8	1.28	246	992	8	1.95
x1	111	455	6	104	418	6	1.25	99	393	6	1.47
x2	13	56	3	13	52	3	0.02	12	48	3	0.03
x3	205	938	6	203	929	6	1.53	189	830	5	2.99
x4	140	598	4	113	456	4	0.17	110	441	4	1.18
total	4994	21273	240	4786	19710	226	268.48	4688	19136	222	522.07
ratio	1.00	1.00	1.00	0.96	0.93	0.94	-	0.94	0.90	0.93	-

both cases we consider identical regions as Don't Cares in the intuitive truth tables.

We can observe that SPFDs are greater than CSPFs from a comparison between Figs. 10 and 11. In general, the essential difference between SPFDs and CSPFs is as follows. If f_j is used as positive for one product q_m and as negative for another product q_l as in the above example, $\mathrm{SPFD}_{[j,i]}$ is greater than the CSPF of $C_{[j,i]}$. This is because we consider all combinations, whether f_j is negated or not, for each product in the context of SPFDs, unlike in that of CSPFs. In the above example, $\mathrm{SPFD}_{[j,i]}$ takes account of all the cases, whether f_j is negated or not, for each q_l and q_m. Accordingly, if we replace f_j with another function, we usually need the modification of the internal logic of N_i, which can be done by $Modify\_Logic$.

VII. EXPERIMENTAL RESULTS

We know that the functional flexibility by an SPFD is usually greater than that by an incompletely specified function. However, we cannot know about any comparison between the computation time and final results of optimization methods using SPFDs or incompletely specified functions. Therefore, we carried out experiments of the optimization methods using SPFDs and CSPFs as follows. In the experiments, a 5-input LUT architecture was assumed. The SIS (a system for sequential circuit synthesis of UC Berkeley) technology mapper commands were used to generate initial networks, i.e., eliminate 2, gkx−ac, simplify −d, xl_part_coll −m−g 2, xl_coll_ck, xl_partition −m, simplify, xl_imp, xl_partition −t, xl_cover −e 30 −u 200, xl_coll_ck −k. These commands are recommended by the SIS package document. The results of the optimization method described in Section V-C for the initial circuits are shown in the columns "SPFD" in Table II. In the table, "LUT," "conn," and "lev" mean the number of LUTs, the number of connections, and the number of network levels. Each "CPU" column gives the CPU time in seconds on a Sun Ultra 2 2200.

When we calculate an SPFD of an input connection of a node N_i, which has n fanin nodes referred to as N_1, N_2, ..., and N_n, we must calculate 2^n logical products at Step 1 in $Divide\_\mathrm{SPFD}$. By doing so, we can consider all of the functions realized by any combination of f_1, f_2, ..., and f_n. Unfortunately, we think this may become a drawback of SPFDs because the calculation time of SPFDs grows exponentially with an increasing number of n (number of inputs of a node). Fortunately, we know from the experiments that the calculation time is not so large for optimizing LUT networks because the number of inputs of a node (LUT) is only five. In fact, the calculation time was not so large except for networks where the binary decision diagram (BDD) used to represent functions was extremely large.[5]

We employed the same optimization method using CSPFs instead of SPFDs for the same initial circuits. The results are shown in the columns "CSPF" in Table II. In the optimization method, we expressed the internal logic of an LUT as an irredundant two-level AND-OR network and calculated CSPFs in the network by the method in [1]. In the table, the row "total" shows the total numbers of "LUT," "conn," "lev," and "CPU." The row

[5]For example, a large part of the execution time for "des" was spent doing *garbage collection* of the BDD nodes.

"ratio" shows the ratios of both "SPFD" and "CSPF" to "Initial." Comparing the columns "SPFD" and "CSPF," we see that, on average, the execution time for "SPFD" is 1.9 times larger than that for "CSPF," while the number of LUTs for "SPFD" is almost 2% better than that for "CSPF."

The heuristic in the implemented method was very simple, so we expect that better results can be obtained with further experiments on the heuristic.

VIII. Conclusion

We presented a new concept "**to distinguish a pair of functions**" for an interpretation of the role of logic functions. This concept leads us in a very natural way to a powerful functional flexibility expression for networks where the internal logic of each node is not fixed. The expressions use SPFDs. The concept of SPFDs is a more powerful way to express functional flexibility than incompletely specified functions. We also presented how to calculate and use SPFDs in this paper. From experimental results of optimizing LUT networks by SPFDs, we saw that SPFDs can be calculated in a reasonable time.

We plan to improve the optimization method presented in the paper and to study other applications of SPFDs.

Acknowledgment

The authors would like to thank the anonymous reviewers for their useful and kind suggestions that contributed to this paper.

References

[1] 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.*, vol. 38, pp. 1404–1424, Oct. 1989.

[2] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 1062–1081, Nov. 1987.

[3] H. Savoj and R. K. Brayton, "The use of observability and external don't cares for simplification of multi-level networks," in *Proc. Design Automation Conf.*, June 1990, pp. 297–301.

[4] H. Savoj, R. K. Brayton, and H. J. Touati, "Extracting local don't cares for network optimization," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1991, pp. 514–517.

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

[6] W. Kunz and D. K. Pradhan, "Multi-level logic optimization by implication analysis," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1994, pp. 6–13.

[7] L. A. Entrena and K. T. Cheng, "Combinational and sequential logic optimization by redundancy addition and removal," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 909–916, July 1995.

[8] R. K. Brayton and F. Somenzi, "Boolean relations and the incomplete specification of logic networks," in *Proc. VLSI'89*, Aug. 1989, pp. 231–240.

[9] F. Somenzi and R. K. Brayton, "An exact minimizer for Boolean relations," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1989, pp. 316–319.

[10] Y. Watanabe and R. K. Brayton, "Heuristic minimization of multiple-valued relations," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1991, pp. 126–129.

[11] 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, "Multilevel logic minimization using implicit don't cares," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 723–740, June 1988.

[12] S. Yamashita, H. Sawada, and A. Nagoya, "A new method to express functional permissibilities for LUT based FPGAs and its applications," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1996, pp. 254–261.

[13] R. K. Brayton, "Understanding SPFDs: A new method for specifying flexibility," in *Notes Int. Workshop Logic Synthesis (IWLS'97)*, May 1997.

[14] R. K. Brayton and S. Sinha, "Implementation and use of SPFDs," in *Notes Int. Workshop Logic Synthesis (IWLS'98)*, June 1998.

[15] J. M. Hwang, F. Y. Chiang, and T. T. Hwang, "Re-engineering approach to low power FPGA design using SPFD," in *Proc. Design Automation Conf.*, June 1998, pp. 722–725.

[16] R. K. Brayton and S. Sinha, "Implementation and use of SPFDs in optimizing Boolean networks," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1998, pp. 103–110.

Shigeru Yamashita received the B.E. and M.E. degrees in information science from Kyoto University, Kyoto, Japan, in 1993 and 1995, respectively.

In 1995, he joined NTT Communication Science Laboratories, Kyoto, Japan, where he has been engaged in research of computer aided design of digital systems and computer architecture.

Mr. Yamashita is a member of IEICE and IPSJ.

Hiroshi Sawada was born in Osaka, Japan, on October 31, 1968. He received the B.E. and M.E. degrees in information science from Kyoto University, Kyoto, Japan, in 1991 and 1993, respectively.

In 1993, he joined NTT Communication Science Laboratories, Kyoto, Japan, where he has been engaged in research of computer aided design of digital systems and computer architecture.

Mr. Sawada is a member of IEICE and IPSJ.

Akira Nagoya (M'98) received the B.E. and M.E. degrees in electronic engineering from Kyoto University, Kyoto, Japan, in 1978 and 1980, respectively.

He joined the NTT Electrical Communication Laboratories, Kanagawa, Japan, in 1980 where he is now a Research Group Leader at the Network Innovation Laboratories. From 1980–1987, he was engaged in research and development of mainframe CPU architecture. Since 1989, he has been engaged in research of CAD system for ASIC design. From 1990–1991, he was with the Department of Computer Science, University of Illinois at Urbana-Champaign, as a Visiting Scholar. His areas of research interest are reconfigurable computing, computer architecture, VLSI design, and electronic design automation.

Mr. Nagoya is a member of IEICE and IPSJ. He received the Okochi Memorial Technology Prize in 1992.