but this can be true if and only if \( f_k \) subsumes \( f_j \) for some \( j \in S \) according to Lemma 2. Therefore, \( f_k \leq f_j = f \) for some \( j \in S \) since \( f_i \) is assumed to be a prime implicant or \( k = j \).

In this note a reduction rule is obtained in order to achieve the simplest representation of a fuzzy function in canonical sum-of-products form. The rule is stated as follows.

**Reduction Rule:** If a term \( f_k \) subsumes a second term \( f_j f_k \) is deleted. This rule is applied repeatedly for all terms of \( F \) until no two terms subsume each other.

A direct consequence of Lemma 1, Theorem 2, and the reduction rule is the following.

**Theorem 3:** The reduction rule generates all the prime implicants of \( F \).

This leads to Theorem 4.

**Theorem 4:** The minimum canonical sum-of-products form of fuzzy function \( F \) is the union of all its prime implicants.

The proof can be obtained in two steps:

1. **Step 1:** \( F \) is the union of all its prime implicants.
2. **Step 2:** The minimum canonical sum-of-products form of \( F \) is the union of all its prime implicants.

**Proof of Step 1:** This is a direct consequence of Lemma 1 and Theorem 3.

**Proof of Step 2:** To prove this, it is only necessary to show that no prime implicant is redundant. Renumbering the prime implicants of \( F \) yields the terms \( f_1, f_2, \ldots, f_m; \) if we next assume that \( f_1 \) is a redundant prime implicant of \( F \) then \( F = f_1 + f_2 + f_3 + \ldots + f_m = f_2 + f_3 + \ldots + f_m \), which implies \( f_1 \leq f_2 + f_3 + \ldots + f_m \). But by Lemma 2, \( f_1 \) subsumes \( f_k \) for some \( k \in \{ 2, 3, \ldots, m \} \), which contradicts the idea that \( f_1 \) is a prime implicant.

In order to simplify a fuzzy function to the minimum canonical sum-of-products form based on Theorems 3 and 4, it is necessary and sufficient to apply the reduction rule. This can be easily carried out by inspection. As an illustration, consider the fuzzy function of three variables used in [8] and [9].

\[
F(X_1, X_2, X_3) = X_1' \cdot X_2' \cdot X_3 + X_1' \cdot X_2 \cdot X_3' + X_1 \cdot X_2' + X_1 \cdot X_2 + X_1 \cdot X_1' \cdot X_3
\]

Applying the reduction rule, the result is

\[
F(X_1, X_2, X_3) = X_1' \cdot X_2' \cdot X_3' + X_1' \cdot X_3' + X_1 \cdot X_2 + X_1 \cdot X_1' \cdot X_3.
\]

**Conclusions**

This note shows that every prime implicant of a fuzzy function is a term of a given fuzzy function and that its minimum canonical sum-of-products form is the union of all its prime implicants. This is in contrast with a Boolean function where not every prime implicant is a term of the given function and its minimum canonical sum-of-products form does not necessarily contain all its prime implicants. The minimum canonical sum-of-products form of a fuzzy function is achieved by deleting its terms that subsume any other term.

**References**


**Functional Transformation in Simplification of Multivalued Switching Functions**

ZVONKO G. VRANESIC, MEMBER, IEEE, AND KHANDKER M. WALIUZZAMAN

Abstract—A concept of functional transformation of multivalued switching functions, used to obtain simplified implementations is presented. Given some function \( f \), a transformed function \( g \) is generated from \( f \) by performing some or all of the truth values. Then, implementing \( g \) and performing a reverse permutation, a simpler implementation for \( f \) is obtained. A \( P \) matrix technique is given to facilitate determination of the required permutation.

Index Terms—Combinational circuits, functional transformation, many-valued logic, simplification, switching algebra.

I. FUNCTIONAL TRANSFORMATION

Practical design of logic networks is subject to constraints such as cost, speed, fan-in, fan-out, tolerances, etc. Clearly, the foremost of these is cost, which has given impetus to development of numerous simplification techniques aimed at reducing the cost by minimizing the number of basic elements in the network.

Until the last decade only 2-valued logic networks found use in practice. However, in recent years the possibility of utilizing multivalued networks has been investigated, resulting in several multivalued switching algebras [1]–[4], at least two of which [2], [3] hold promise for successful practi-
rical application. Workable circuits for the algebra in [3] have also been developed, using conventional components [6].

In general the cost of multivalued circuits taken on a per gate basis is greater than that of 2-valued gates. Thus it is particularly important to have effective means of reducing the total number of gates in such networks.

This note presents a powerful concept of functional transformation through permutation of truth values resulting in reduced cost implementation of multivalued functions. The idea is not to implement a given switching function \( f \) in terms of its truth table, but instead to implement a different function \( g \), where \( g \) is obtained from \( f \) through a simple permutation of truth values. Finally, when an implementation for \( g \) is obtained, the truth values in the output of \( g \) must be permuted back to the original values so as to result in the required function \( f \).

For 2-valued functions this corresponds to implementation of a complemented function and inversion of its output, which in most cases does not reduce the cost. However, for \( R \)-valued functions where \( R \geq 2 \) such a technique often offers significant reduction in cost, as illustrated by the following example.

Consider the 4-valued function \( f(x, y) \) defined by the truth table.

\[
\begin{array}{c|cccc}
 y & 0 & 1 & 2 & 3 \\
 \hline
 x & 0 & 2 & 0 & 1 \\
 1 & 0 & 0 & 1 & 3 \\
 2 & 1 & 1 & 1 & 3 \\
 3 & 3 & 3 & 3 & 3 \\
\end{array}
\]

Its straightforward implementation in terms of the basic set in [3] (basic set is defined in the Appendix) is

\[
f(x, y) = (x^3 - y^2)^p + (x + y)^q + (x^2 - y^2)^r.
\]

The resulting network, shown in Fig. 1, requires 11 gates.

However, let \( g(x, y) = \Pi \{ f(x, y) \} \) be a function obtained from \( f(x, y) \) through a simple permutation \( \Pi \) of the truth values, where \( \Pi \):

\[
0 \mapsto 1 \\
1 \mapsto 2 \\
2 \mapsto 0 \\
3 \mapsto 3.
\]

The truth table for \( g(x, y) \) follows.

\[
\begin{array}{c|cccc}
 y & 0 & 1 & 2 & 3 \\
 \hline
 x & 0 & 2 & 0 & 1 \\
 1 & 1 & 1 & 1 & 3 \\
 2 & 3 & 3 & 3 & 3 \\
 3 & 3 & 3 & 3 & 3 \\
\end{array}
\]

Clearly, it can be implemented simply as

\[ g = x + y. \]

Next it is necessary to permute the truth values in \( g \) back to the original values in order to obtain a valid realization for \( f \), so that \( f(x, y) = \tau \{ g(x, y) \} \). Thus the required reverse permutation is \( \tau \):

\[
0 \mapsto 2 \\
1 \mapsto 0 \\
2 \mapsto 1 \\
3 \mapsto 3.
\]

Implementation of such reverse permutations clearly becomes a simple task of implementing unary functions [3]. Any mapping of truth values \( S \mapsto T \) in the function \( g \) is readily achieved as \( g^S_T \) terms. Thus, in general, for a reverse permutation

\[
A \mapsto J \\
B \mapsto K \\
\vdots \\
C \mapsto L
\]

have \( f = g^A_J + g^B_K + \cdots + g^C_L \). Furthermore, note that: 1) a mapping to zero, i.e., \( S \mapsto 0 \), gives \( g^S_0 \), which is equal to a constant 0; hence it need not be included explicitly; 2) a mapping from zero, i.e., \( 0 \mapsto T \), gives \( g^T \) which does not require a cycling operation.

Therefore, for the preceding reverse permutation \( \tau \), the required implementation of \( f \) is

\[ f = g^1 + g^2 + g^3 \]

This expression is realized with only 7 gates, as shown in Fig. 2, reducing the cost significantly.
II. P-MATRIX PROCEEDURE

The previous example indicates the usefulness of the concept of functional transformation. However, it would be of little value if no workable technique could be defined to determine permutations leading to the simplest functional expressions. The following P matrix procedure provides means for finding the desired permutation when implementation in terms of basic sets based on sum and product functions and unary functions of the Postian cycling type [5] are used, as for example in the algebra of [3].

An R-valued function \( f(x_1, \ldots, x_n) \) is said to be of "uniform ascending monotonicity" over one of its variables \( x_i \) if for each of its remaining variables \( (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \) the function value remains constant or increases by a single step\(^1\) when \( x_i \) is increased by a single step. If the function is of uniform ascending monotonicity over all of its variables then the function is said to have no "breaks." Any failure to satisfy the condition of uniform ascending monotonicity is called a break. The total number of breaks over all variables is called the "breakcount" of the function.

Functions obtained through permutation of truth values exhibit a useful property; namely, the functions with lower breakcounts tend to have simpler implementations than those with higher breakcounts. However, for purposes of cost comparison it is necessary to include the cost of the reverse permutation \( \tau \) which returns the original truth values. In view of the overall cost it may occur that a permuted function with a lower breakcount has a more expensive implementation than the original function (in such cases breakcounts are typically very close in value). Nevertheless, the use of the breakcount as the indicator of the ease of implementation of the function provides excellent heuristic means for finding the best implementation. Testing a considerable sample of randomly chosen functions it was found that for more than 90 percent of the functions the lowest breakcount corresponded to the simplest implementation [7].

In order to facilitate determination of breakcount of functions obtained through permutation of variables, a P matrix is defined: the P matrix of an R-valued function \( f(x_1, \ldots, x_n) \) is an \( R \times R \) matrix with the elements \( p_{ij}, 0 \leq i, j < R \) in the \( i \)th row and \( j \)th column, \( p_{ij} \) is the number of times the truth value \( i \) is immediately followed by the truth value \( j \) when the function \( f \) is scanned in ascending order of truth values of its variables \( x_q, q = 1, 2, \ldots, n \) for a constant valuation \( x_q = x_{1}x_{2} \ldots x_{q-1}x_{q+1} \ldots x_{n} \) and repeated over all possible valuations.

Then the breakcount is the sum of the elements of the P matrix, except those on the main diagonal or immediately above it; i.e.,

\[
\text{breakcount} = \sum_{i=0}^{R-1} \sum_{j=0}^{R-1} p_{ij}, \quad j \neq i, \quad j \neq i + 1.
\]

\(^1\) A step is a unit change in truth value.

The \( P \) matrix for the function \( f(x, y) \) defined by the truth table in Section I indicates the breakcount of 8.

<table>
<thead>
<tr>
<th>( p )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

Breakcount = 8.

A \( P \) matrix for any function \( h \) obtained from \( f \) through a permutation of truth values is readily obtained from the \( P \) matrix for \( f \) by performing the corresponding permutation of rows and columns.

Since it is much easier to compute breakcounts of permuted functions by manipulating the \( P \) matrix (instead of direct computation from the truth table), we will use the \( P \) matrix to determine the permutation that minimizes the breakcount. It is readily found that if in the preceding \( P \) matrix we interchange rows and columns 0 and 1 and then interchange 0 and 2 a new \( P \) matrix (\( P^* \)) is generated which has a zero breakcount.

<table>
<thead>
<tr>
<th>( p^* )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

Breakcount = 0.

Evidently, this corresponds to the permutation II applied to the function \( f \) in Section I.

The \( P \) matrix and breakcount concept is used in this note as a specific tool helpful in obtaining the best functional transformation. However, its usefulness is quite general when Postian multivalued systems are considered; e.g., it provides simple means for finding optimum decompositions of such multivalued functions [8].

III. CONCLUSIONS

Functional transformation provides powerful means for simplification of multivalued switching functions resulting in reduced cost implementations.

Practical synthesis methods can be developed to utilize such transformations in particular algebras as demonstrated by the \( P \)-matrix procedure for the algebra in [3].

We note that the notion of permutation of truth values has no independent significance when applied to the inputs since it merely becomes a special case of unary functions of input variables that are commonly used in most simplification techniques [3].

APPENDIX

BASIC FUNCTIONS USED IN THE \( R \)-VALUED ALGEBRA OF [3]

1) Sum \[ x + y = \max (x, y). \]
2) Product \[ x \cdot y = \min (x, y). \]
Contributors

Thomas F. Arnold was born in New York, N. Y., on September 23, 1943. He received the S.B. and S.M. degrees in electrical engineering from the Massachusetts Institute of Technology, Cambridge, in 1964 and 1965, respectively, and the D. Eng. Sc. degree in electrical engineering from Columbia University, New York, N. Y., in 1969.

In 1965 he joined the technical staff at Bell Telephone Laboratories, Holmdel, N. J., where he has participated in the development of electronic switching systems. He is presently engaged in hardware and software development for peripheral units and is especially concerned with the problems of highly reliable systems.

Dr. Arnold is a member of Tau Beta Pi,Eta Kappa Nu, and Sigma Xi.

Yun-Chung Cho (S'71) was born in Canton, China, on August 15, 1947. He received the B.S. degree from National Chiao Tung University, Taiwan, China, in 1968 and the M.S. degree from the University of California, Berkeley, in 1970.

He is now a doctoral student and Research Assistant in the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. His current research interests include picture processing, compiling techniques, and switching theory.

Mr. Cho is a member of Eta Kappa Nu and the Association for Computing Machinery.

Edward S. Davidson (S'67–M'68) was born in Boston, Mass., on December 27, 1939. He received the A.B. degree in mathematics from Harvard University, Cambridge, Mass., in 1961, the M.S. degree in communications science from the University of Michigan, Ann Arbor, in 1962, and the Ph.D. degree in electrical engineering from the University of Illinois, Urbana, in 1968.

From 1962 to 1965 he was employed by Honeywell, EDP, where he was responsible for the logic design of several H-200 family components and the H-300 I/O control system. He also participated in algorithm development and exploratory studies for logic design automation. While completing his doctoral studies at the University of Illinois, he held research assistantship in the Digital Computer Laboratory, working on a block-structured logic specification language and the traveling salesman problem. He later served as Research Associate in the Coordinated Science Laboratory, where he developed an algorithm for NAND network design. Since 1968 he has been Assistant Professor of Electrical Engineering, Stanford University, where he has been teaching courses in switching theory, logic design, computer organization, and discrete mathematics. Present research interests include NAND network design, efficient memory architectures, and design and control of pipelined architecture computers. He has been active in exploring causes and solutions to unemployment and defense dependency in the engineering profession.

Dr. Davidson has served as Local Arrangements Chairman, Vice Chairman, and Chairman of the San Francisco Bay Area Chapter of the Computer Society and is a member of the Association for Computing Machinery.

Keinosuke Fukunaga (M'66) was born in Himeji, Japan, on July 23, 1930. He received the B.S. degree in electrical engineering from Kyoto University, Japan, in 1953, the M.S.E.E. degree from the University of Pennsylvania, Philadelphia, in 1959, and the Ph.D. degree from Kyoto University, Japan, in 1962.

From 1953 to 1957 he was with the Central Research Laboratories of Mitsubishi Electric Company, Japan, where he performed system analysis on automatic control systems. From 1957 to 1959 he was with the Moore School, University of Pennsylvania, working in the field of switching theory. From 1959 to 1966 he was with the Mitsubishi Electric Company, first with the Central Research Laboratories, working on computer applications in control systems, and then with the Computer Division, where he was in charge of hardware development. Since 1966 he has been an Associate Professor of Electrical Engineering at Purdue University, Lafayette, Ind. During the summers of 1967, 1968, and 1969 he worked with IBM, Endicott, N. Y., and Rochester, Minn. His interests include pattern recognition and pattern processing.

Dr. Fukunaga is a member of Eta Kappa Nu.