IEEE TRANSACTIONS ON COMPUTERS, VOL. 42, NO. 5, MAY 1993

Minimization of AND–EXOR Expressions Using Rewrite Rules

Daniel Brand and Tsutomu Sasao, *Senior Member, IEEE*

Abstract—**This paper considers conditions for generating optimal two-level AND–EXOR representations using rewrite rules. Four results are presented. First, a necessary condition for obtaining minimality is a temporary increase in the size of expressions during minimization. Second, a sufficient condition for obtaining minimality consists of adding certain two rules to rule sets proposed in the literature. Third, we define transformations that allow the minimization of an expression to proceed by minimizing a transformed expression instead. Fourth, we determine experimentally that the above three theoretical results lead to better benchmarks results as well.**

Index Terms— **EXOR, L-transformation, logic minimization, nondeterministic algorithm, Reed–Muller expansion, two level representation.**

I. Introduction

Minimization of AND–OR expressions always played an important role in logic synthesis. The problem has been relatively well understood and there are many good minimization algorithms [14], [10], [13], [3], [19]. The problem of minimizing AND–EXOR expressions, on the other hand, has not been solved to the same degree. Such expressions are of interest because two-level AND–EXOR representations are usually smaller than two-level AND–OR representations [23]. Also many multilevel circuits based on EXOR elements are more advantageous from area, speed and testability point of view [12], [16], [8], [20]. Error detecting circuitry is a typical example, where EXOR's are heavily used. Such multilevel circuits can be obtained from two-level AND–EXOR expressions (which are the subject of this paper) by algebraic factoring [2]. Furthermore, some technologies allow PLA's with optional EXOR elements on their outputs; optimizing such AND–OR–EXOR structures can be done through optimizing AND–EXOR expressions [24].

Therefore a lot of work has been done on the minimization of AND–EXOR expressions [6], [11], [5], [15], [18], [17], [1], [7], [9], [21], [22]. This paper analyzes the method of rewrite rules [6], [17], [7], [21], which has proved to be a successful heuristic approach generating relatively good solutions in reasonable time.

Manuscript received July 18, 1991; revised May 18, 1992. This work was supported in part by Grant in Aid for Scientific Research of the Ministry of Education, Science and Culture of Japan.

D. Brand is with IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.
T. Sasao is with the Department of Computer Science and Electronic Engineering, Kyushu Institute of Technology, Iizuka 820, Japan.

IEEE Log Number 9207302.

Before rewrite rules can be applied, a given function must first be converted to some (nonoptimal) two-level AND–EXOR representation. There are several ways of doing that. For example, a two-level AND–OR representation can be modified to consist of disjoint terms only, and thus can also be considered an AND–EXOR expression.

From now on we will assume that we are given a two-level AND–EXOR representation to be minimized. We will use the symbol $\oplus$ to denote EXOR, concatenation to denote AND, and bar to denote negation. Our measure of minimality will be the number of product terms in the EXOR sum. Some of our results also apply to other measures, as long as the number of product terms remains the most important.

Once we have an AND–EXOR representation, it can be simplified using rewrite rules. One of the earliest rule sets has been proposed by [6] and we paraphrase it below using slightly different notation:

$$x \oplus \bar{x} \rightarrow 1 \tag{S1}$$
$$x \oplus 1 \rightarrow \bar{x} \tag{S2}$$
$$xy \oplus \bar{y} \rightarrow 1 \oplus \bar{x}y \tag{S3}$$
$$xy \oplus \bar{x}\bar{y} \rightarrow \bar{x} \oplus y \tag{S4}$$
$$xy \oplus \bar{x}\bar{y} \rightarrow x \oplus \bar{y}. \tag{S5}$$

While the authors did not state so explicitly, it is safe to assume that their simplifier also contained the rule

$$1 \oplus 1 \rightarrow 0 \tag{S6}$$

In addition, the laws of commutative rings (associative, distributive, $x \oplus 0 = x$, etc.) are needed in the application of any rules. To illustrate the use of rewrite rules, consider the following chain of minimization steps:

$xy\bar{z} \oplus \bar{y}\bar{z} \oplus \bar{x}yz$ (apply (S3) to the first and second term to get)

$\bar{z} \oplus \bar{x}y\bar{z} \oplus \bar{x}yz$ (apply (S1) to the second and third term to get)

$\bar{z} \oplus \bar{x}y.$

The authors of [6] made the following statement about their rules (S1)–(S6):

"The procedure does not guarantee minimality; it contains branching points and our experience shows that all branches lead to fairly economical expressions, but it is unknown whether at least one chain leads to a minimal expression."

The question posed by [6] is the very subject of this paper. In particular we will answer it for the above set of rules. This paper considers the following problems:

1) Given a set of rules, is there always a chain leading to a minimal expression?
2) If the answer to the first question is "no," what rules need to be added?
3) If the answer to the first question is "yes," but not every chain leads to a minimum, how to find a chain that does result in a minimal expression?
4) To what extent are these issues relevant in practice?

A set of rules capable of generating a minimal representation for any given expression will be called "convergent" because by applying such rules in all possible combinations we could eventually converge to a minimal representation.

Definition 1.1: A set of rules is *convergent* iff for any expression E there is a sequence of expressions $E = E_0, \cdots, E_n$ $(n \geq 0)$, where E_{i+1} is obtained from E_i by one of the rules, and E_n has the minimal number of product terms.

Please note that our definition of convergence does not imply termination. Even after finding a minimal expression, rules may continue to be applied because we do not assume a procedure determining whether an expression is minimal.

We do not argue that methods guaranteeing optimality are better than those without such a guarantee. But understanding the requirements for optimality can help in designing better methods. The objective of this paper is not an efficient practical minimizer, but merely an investigation of conditions for optimality. We implemented the approach only to answer some theoretical questions.

The paper is organized as follows. Section II considers problem 1). Section III considers problem 2). Section IV considers problem 3). Sections V and VI consider the problem 4). Throughout the paper the subject is described informally and for binary valued logic only. For a formal treatment and for extensions to multivalued logic the reader is referred to [4].

II. A Necessary Condition for Convergence

Most of the published rules for minimizing AND–EXOR expressions (e.g., rules (S1)–(S6)) have one important property—their right hand side has no more product terms than their left hand side. This guarantees that no rule can increase the number of product terms and thus guarantees that the final expression has no more product terms than the original one. Unfortunately this also guarantees that the set of rules is nonconvergent, because the minimum of some expressions cannot be obtained without a temporarily increase in the number of product terms.

The proof of this necessary condition for convergence proceeds as follows. Given a set of rules which cannot increase the number of product terms we observe the maximum number m of terms any rule can change. Then we construct an expression that cannot be minimized by any set of rules capable of changing only m terms at a time (without an increase in the number of product terms). The proof for general m [4] is rather long and is not reproduced here, but is available from the authors. For $m = 2$, which is the size of rules used in practice, there is a simple proof given below.

Theorem 2.1: A set of rules is nonconvergent if it satisfies:

1) Each rule has at most two product terms on its left hand side.
2) Each rule has no more product terms on its right than on its left hand side.

Proof: Consider the 4-variable expression:

$$\bar{x}y\bar{z}\bar{w} \oplus x\bar{z} \oplus \bar{x}w \oplus x\bar{y}zw \oplus z\bar{w}.$$

By exhaustive analysis we can determine that no rule satisfying properties 1) and 2) applies to the above expression. (It is relatively easy to see by drawing a Karnaugh map.) Nevertheless the expression is not minimal because there is an equivalent expression with fewer product terms:

$$xyzw \oplus \bar{x}\bar{y}\bar{z}\bar{w} \oplus 1$$

Q.E.D.

III. Sufficient Conditions for Convergence

This section is concerned with making rules convergent, and we already know that it has to be done by term-increasing rules. It is not difficult to design convergent rules; for example, the following set of rules is convergent:

$$x \oplus \bar{x} \rightarrow 1 \tag{S1}$$
$$1 \oplus 1 \rightarrow 0 \tag{S6}$$
$$1 \rightarrow x \oplus \bar{x} \tag{1S}$$
$$0 \rightarrow 1 \oplus 1. \tag{6S}$$

To see why this is so, consider any two-level AND–EXOR expression. It can be converted to a minterm expansion by repeatedly splitting terms into pairs using (1S) and by eliminating duplicates using (S6). Thus {1S, S6} are sufficient to convert any expression to minterms, and therefore {S1, 6S} are sufficient to convert a minterm expansion into any equivalent expression. Therefore {S1, 1S, S6, 6S} are capable of converting any expression to any other.

For practical purposes powerful sets of rules like {S1, 1S, S6, 6S} are not desirable because they generate a large search space. We seek a set of rules that is as weak as possible, yet sufficiently strong to be convergent. The rule (6S) is particularly troublesome because it allows the generation of any pair of identical terms, independently of the given function. Fortunately the rule (6S) can be replaced by

$$\bar{x} \rightarrow x \oplus 1. \tag{2S}$$

The resulting set of rules is incapable of converting any expression to any other, but is capable of converting any expression to a minimal one.

While we talk about a minterm representation of a function, we are not proposing it as practical, and our implementation does not try to generate all minterms. Minterms are used only as a mental tool in our proofs.

Theorem 3.1: The set of rules {S1, 1S, S6, 2S} is convergent.

Outline of Proof: (Formal proof is in [4].)

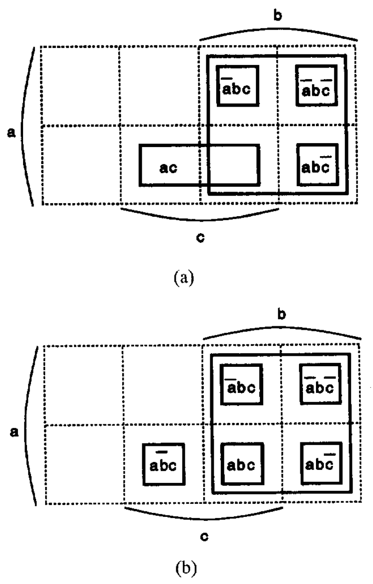

Fig. 1(a). Terms to be split into minterms. (b) Redundancy caused by splitting.

As noted above {1S, S6} are sufficient to convert any expression to its minterm expansion. Therefore we will be done if we can show that {S1, 1S, 2S} are sufficient to convert a minterm expansion to a minimal expression. We will do that by considering the inverse problem, namely we will show that {1S, S1, S2} are sufficient to convert any minimal expression to a minterm expansion.

The rule (1S) alone can generate a minterm expansion, but the expansion will in general contain duplicate minterms. We have to avoid duplicate terms because we have no rule capable of removing them; the rule (S6) cannot be used because that would imply the use of (6S) when going in the opposite direction. In general we avoid any set of terms whose exclusive OR is identically 0; we call such a set a "redundancy" because it can be deleted without changing the function of the original expression. The proof shows that given any expression without any redundancy (for example a minimal expression) we can keep splitting terms without ever generating a redundancy.

To illustrate what needs to be done consider the expression $ac \oplus b \oplus \bar{a}bc \oplus \bar{a}b\bar{c} \oplus ab\bar{c}$ shown by the Karnaugh map of Fig. 1(a), which contains no redundancy. A careless process of generating minterms might split the term ac into two, generating Fig. 1(b), which contains the redundancy $\{b, abc, \bar{a}bc, \bar{a}b\bar{c}, ab\bar{c}\}$.

Therefore we need to split terms carefully. This is how to split the term ac "carefully." First we use the rule (S1) to generate Fig. 2(a). Then we use the rule (S2) to generate Fig. 2(b). Then we use the rule (S2) to generate Fig. 2(c). And finally use (S2) to generate Fig. 2(d).

In general the following systematic procedure can be used on any expression E not containing a redundancy. It uses rules {S1, 1S, S2} to generate a minterm expansion without any redundancy. (In our example $E = ac \oplus b \oplus \bar{a}bc \oplus \bar{a}b\bar{c} \oplus ab\bar{c}$.)

procedure careful_split(E) returns(E)

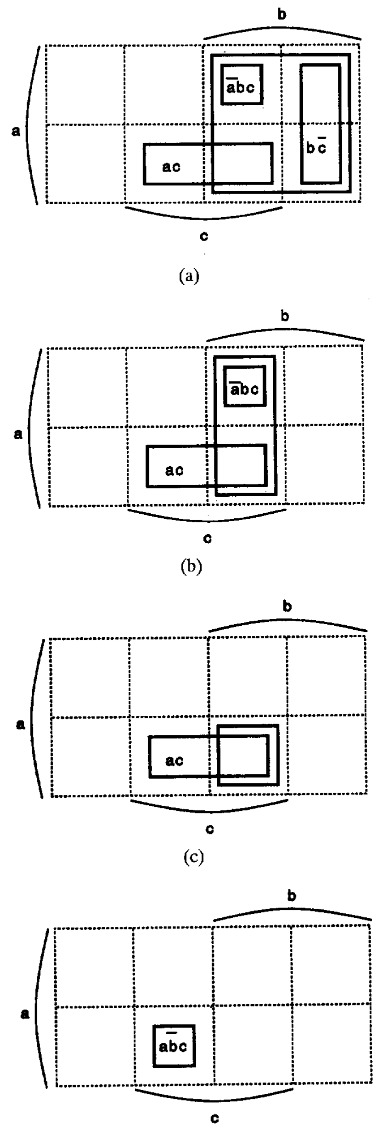

Fig. 2. (a) Obtained from Fig. 1(a) by rule (S1). (b) Obtained from (a) by rule (S2). (c) Obtained from (b) by rule (S2). (d) Obtained from (c) by rule (S2).

1) Choose any term t that is not yet a minterm and any variable x not in t. (In our example $t = ac$ and $x = b$.)
2) If splitting t into $xt \oplus \bar{x}t$ does not create any redundancy then make the split and go to step 1.
3) Do not split t along x because that would cause a redundancy. E must contain a set of terms G such that either $G \oplus xt$ or $G \oplus \bar{x}t$ is a redundancy. Suppose that $G \oplus xt$ is a redundancy. (In our example $G = b \oplus \bar{a}bc \oplus \bar{a}b\bar{c} \oplus ab\bar{c}$.)
4) Apply this procedure recursively to G. This will generate a minterm expansion of G without duplicates. (In our example it will generate the single minterm abc.)
5) The function of G is identical to xt because in step 3 we assumed $G \oplus xt$ to be identically 0. Therefore the

minterm expansion of G covers exactly the cube covered by xt. Use the rule (S1) to merge all those minterms into the one term xt.

6) Use the rule S2: $xt \bigoplus t \rightarrow \bar{x}t$. Go to step 1.

When no term can be selected in step 1 the procedure careful_split has generated a minterm expansion. One detail remains in arguing that the result will have no redundancies. Suppose that the splitting in step 2 caused more than one redundancy. Then the remainder of the algorithm would avoid only one of them, and allow others to exist. The proof in the Appendix shows that this cannot happen by proving that in step 3 there is a unique G.

(end of informal proof)

The set of rules {S1, 1S, S6, 2S} is not minimal because any application of S1 can be replaced by 2S followed by S6. The resulting set {1S, S6, 2S} is minimal, in the sense that none of its subsets is convergent:

Theorem 3.2: The set of rules {1S, S6, 2S} is a minimal convergent set of rules.

Proof: We show that removing any of the three rules would result in a nonconvergent set.

(1S) cannot be removed because that is the only rule in {1S, S6, 2S} capable of adding new literals to terms. This is necessary for convergence as in the example of $\bar{x}\bar{y} \oplus xz \oplus \bar{y}z$, whose minimal representation is $\bar{x}\bar{y}\bar{z} \oplus xyz$. (S6) cannot be removed because that is the only rule in {1S, S6, 2S} capable of reducing the number of terms. (2S) cannot be removed because that is the only rule in {1S, S6, 2S} capable of generating overlapping terms. This is necessary for convergence as in the example of $\bar{x}\bar{y}\bar{z} \oplus x\bar{y}z \oplus xy$, whose minimal representation is $\bar{y}\bar{z} \oplus x$. Q.E.D.

While (2S) cannot be removed from {S1, 1S, S6, 2S} without losing convergence, it is possible that it could be replaced by some term nonincreasing rules. This question is related to the question of a minterm representation of functions: While an increase in the number of product terms is necessary when starting from an arbitrary representation of a function, it may be unnecessary if starting from a minterm representation. We have not been able to answer this question, and will refer to it again when talking about experimental results.

IV. L-Transformations of Expressions

Now suppose that we have a convergent set of rules and we want a chain leading to an optimal solution. Even if the rules are not convergent we may want to find a chain that simply leads to a good solution. It is not our goal here to design a simplifier that explores the search space, but rather to force a given simplifier to do so. For that purpose we use "L-transformations." This section introduces the two-valued version of L-transformation theory, which is special case of the multivalued results [23]. L-transformations are explained only to the extend they are used in Section V.

We will use the following notation:

Any time we write an expression $\bar{x}f_0 \oplus xf_1 \oplus f_2$ we mean to imply that

$\bar{x}f_0$ represents all the terms containing $\bar{x}$ with $\bar{x}$ itself factored out,

xf_1 represents all the terms containing x with x itself factored out, and

f_2 represents all the terms containing neither x nor $\bar{x}$.

In order to avoid confusion regarding equality between expressions, we will write $f = g$ to mean that f and g are the same expressions, while $f \equiv g$ will mean that f and g may be different expressions, but representing the same function.

Definition 4.1: For a given expression f and for a literal r (x or $\bar{x}$) the *L-transformation* of f with respect to r, written $[r] \odot f$ for short, is defined as follows.

Express f as $f = \bar{r}f_0 \oplus rf_1 \oplus f_2$.

Then $[r] \odot f = \bar{r}f_0 \oplus f_1 \oplus rf_2$.

In other words, the transformation is obtained by deleting the literal r from any term containing it and adding it to every term containing neither r nor $\bar{r}$.

Example:

$$[\bar{x}] \odot (xy \oplus \bar{x}\bar{z} \oplus \bar{y}) = xy \oplus \bar{z} \oplus \bar{x}\bar{y}$$
$$[z] \odot (xy \oplus \bar{x}\bar{z} \oplus z) = xyz \oplus \bar{x}\bar{z} \oplus 1.$$

If we define composition of two transformations in the usual way as $([r_1][r_2]) \odot f = [r_1] \odot ([r_2] \odot f)$ then we can see from the definition that it is associative, and it is commutative for literals corresponding to different variables. We will abbreviate $[r_1][r_2] \cdots [r_n]$ as $[r_1, r_2, \cdots, r_n]$.

We will use the symbol I to denote the identity transformation, i.e.,

$$I \odot f = f$$

Please note that each transformation is its own inverse, i.e., $[r, r] = I$, and we will write

$$[r]^{-1} = [r]$$

For a sequence R = $[r_1, \cdots r_n]$ of transformations, if we define $R^{-1} = [r_n]^{-1} \cdots [r_1]^{-1}$ then $RR^{-1} = I$. In case that all the transformations in the sequence are pair-wise commutative then $R^{-1} = R$.

Theorem 4.2:

$$\bar{x}f_0 \oplus xf_1 \oplus f_2 \equiv \bar{x}g_0 \oplus xg_1 \oplus g_2 \qquad (1)$$
$$\text{iff } f_0 \oplus g_0 \equiv f_1 \oplus g_1 \equiv f_2 \oplus g_2$$

Proof: Equation (1) is true iff it is true for both $x = 0$ and $x = 1$, i.e.,

$$f_0 \oplus f_2 \equiv g_0 \oplus g_2 \qquad (2)$$

and

$$f_1 \oplus f_2 \equiv g_1 \oplus g_2. \qquad (3)$$

EXORing both sides of (2) with $g_0 \oplus f_2$ and eliminating identical terms

$$f_0 \oplus g_0 \equiv f_2 \oplus g_2. \qquad (4)$$

EXORing both sides of (3) with $g_1 \oplus f_2$ and eliminating identical terms

$$f_1 \oplus g_1 \equiv f_2 \oplus g_2. \qquad (5)$$

The expression (4) is actually equivalent to (2) because we can get (2) our of (4) by EXORing with $g_0 \oplus f_2$ again. Similarly (5) and (3) are equivalent. Q.E.D.

Corollary 4.3: $f \equiv g$ iff $[r] \odot f \equiv [r] \odot g$.

Proof: Follows immediately by applying the theorem to both equalities. Q.E.D.

The theorem and its corollary allow the following procedure to simplify a given expression f:

```
procedure $M(f)$ returns($f'$)
    Choose a sequence $R$ of L-transformations.
    Let $g := R \odot f$.
    Let $g'$ be obtained by simplifying $g$.
    Let $f' := R^{-1} \odot g'$.
```

The resulting f' is a simplification of f; moreover, f' has the minimal number of terms iff g' has. It may be easier to simplify g instead of f directly because g has many properties different from f and therefore a heuristic simplifier may behave quite differently when simplifying g rather than f.

From an expression f containing a variable x we can derive six distinct expressions: $I \odot f, [x] \odot f, [\bar{x}] \odot f, [x, \bar{x}] \odot f, [\bar{x}, x] \odot f$, and $[x, \bar{x}, x] \odot f$. Only the first three are of interest here because the other three can be obtained from the first three by the interchange of x and $\bar{x}$, and hence have the same properties from minimization point of view.

When minimizing an expression of n variables using the procedure M, the sequence R can can be chosen in 3^n different ways. Namely, for each variable x we can chose to put into R the transformation using $I, [x]$, or $[\bar{x}]$. (Please note that in this way we ensure that $R^{-1} = R$.) Many of the 3^n expressions have quite different properties, which will make the given simplifier generate different chains of rules. We run a number of the expressions through the simplifier as a way of partially exploring the search space, for we have found this to be effective in obtaining better solutions.

An L-transformation can be used as a meta-rule for generating new rules. Given a set of rules one can apply L-transformations to both sides of each rule generating new rules. For examples, transforming (S3) with respect to y we obtain

$$x \oplus \bar{y} \rightarrow y \oplus \bar{x}. \qquad \text{(S3*)}$$

We call a set of rules *closed under L-transformations* iff every one of the possible L-transformations generates a rule already present in the original set. Since (S3*) is not present in the original set of rules, (S1)–(S6) form an example of rules that are not closed. For a set of rules that is not closed the procedure M is particularly beneficial because it may generate solutions unobtainable by the rules themselves.

V. Algorithm

Until now we have discussed methods of improving the asymptotic behavior of rewrite rules. In order to evaluate the impact of those methods on short term behavior, we have implemented them in a way described in this section. Our implementation is not a substitute for an exact minimizer, or for a good heuristic minimizer. Its purpose is to evaluate the impact of turning an existing set of rewrite rules into a convergent set by adding term-increasing rules.

As shown in Section II, adding the two term increasing rules (1S) and (2S) is sufficient for convergence. A problem is when to apply them. The most liberal solution would allow their application at arbitrary times; this would guarantee a branch to an optimum, but also a branch with ever increasing number of terms. Our solution tries to be as restrictive as possible without jeopardizing convergence. Since we do not know a deterministic procedure for obtaining an optimum, our implementation is nondeterministic. As will be explained later in this section, our algorithm may fail to find an optimum, but the probability of failure approaches 0 with increasing computation time.

For our experiments we have used the simplifier EXMIN [21], which is known to produce in general very good results, but not always optimal. It does not increase the number of terms and therefore is not convergent. It is known to be closed under L-transformations. Our implementation did not change EXMIN, but rather treated it as a "black box" and invoked it repeatedly. (Recently, EXMIN2 [24], an improved version of EXMIN, has been developed. It uses a term-increasing rule as well as term-nonincreasing rules to improve the quality of the solutions. Thus, the solutions are usually better than those of EXMIN, but the computation time is longer.)

The implementation is based on the following procedure $P(f)$, which takes an expression f as an argument and returns a modified expression g. It divides f into two, calls itself recursively on the two halves and then combines the results together. When called on the top-most level it is supposed to return a minimum. However, the recursive calls are not supposed to return a minimum, not even necessarily a good solution; they are supposed to return expressions which can be eventually combined into a minimum. We do not know how to generate such expressions; therefore the procedure is nondeterministic.

```
procedure $P(f)$ returns($g$)
1) Possibly let $g := f$ and go to step 6
2) Chose a variable $x$
3) Let $g_1 := P(xf), g_2 := P(\bar{x}f), g := g_1 \bigoplus g_2$
4) Operating on $g$, apply rule (S1) to some pairs of terms,
   one from $g_1$, the other from $g_2$.
5) Apply the rule (2S) to some terms of $g$
6) Possibly apply EXMIN to $g$
7) Return $g$
```

The procedure P contains words like "possibly" and "some" because it is nondeterministic. All the nondeterministic decisions are done randomly, but the probability distributions are biased toward values that we found experimentally to speed up convergence.

Now we will explain each step in turn:

1) The procedure calls itself recursively in step 3, and step 1 controls the depth of recursion. For example, we would go to step 6 if f consisted of minterms only. We also have to limit the depth of recursion for practicality. However, to guarantee convergence, we allow larger and larger depth every once a while.

2) We chose a variable x randomly, but making sure that neither xf nor $\bar{x}\ f$ is empty. That is the only requirement for convergence. To speed up convergence we give preference to partitions that do split some terms into two, but do not split too many terms.
3) The expression xf implies that each term of f is ANDed with x, which may result in the elimination of some terms and and reduction of others. Step 3 is where the rule (1S) takes place.
4) Apply the rule (S1) only along the chosen variable x, but not whenever possible. To guarantee convergence we must disallow some possible merges because a merge might prevent something better later.
5) The rule (2S) is applied to some terms of g along the variables x, possibly followed by more applications of (2S), (1S) and (S1) to newly generated terms. As mentioned in Section III we do not know whether this steps is necessary for convergence in the presence of step 6.
6) EXMIN always returns an expression no larger than its input. Therefore there is no harm in using it at the highest level, where we would like P to generate a minimum. However, in the recursive calls EXMIN is not always executed.

The procedure P is applied in the following procedure Q to minimize a given expression e.
($\tau(f)$ denotes the number of terms in an expression f.)

```
procedure $Q(e)$ returns($e$)
    do forever
        $f := e$ /* $e$ is always the best expression so far */
        Chose a random sequence $R$ of L-transformations
        $f := R \odot f$
        $f := P(f)$
        if $\tau(f) < \tau(e)$ then $e := R^{-1} \odot f$
    end
```

The last step of the procedure Q guarantees that the number of product terms of e cannot increase. The body of the loop allows a possibility of decrease, but not a guarantee. Therefore there exists a possibility of e remaining nonoptimal forever; however the probability of this happening goes to 0 with an increasing number of iterations. (The reason is that there is only a finite number of expressions with any given number of terms, and for each of the expressions there is a nonzero probability of being reduced during one iteration.) The result is the probability of 1 for reaching an optimum, but no bound on how many iterations it may require. While that is not very satisfying from a practical point of view it is sufficient for our experiments. The absence of any bound is expressed by by the phrase "do forever," but in our experiments we naturally stopped the loop after several iterations, as reported in the next section.

As will be reported in the next section we ran experiments with various restrictions on term increasing rules. Disallowing the rule (2S) is achieved by never executing step 5 of the procedure P. Disallowing both term-increasing rules is done by always executing step 1 of the procedure P; in addition, in this case we never update e in the last step of the procedure Q so that EXMIN always works on a transformation of the very original input.

VI. Experimental Results

We have run experiments in an effort to answer the following two questions:

1) While an increase in the number of terms during simplification is necessary to obtain the optimum, it may not be needed to obtain merely good solutions. It is possible that enlarging the search space by term increasing rules does not add many good solutions on top of those already present.
2) We have mentioned that we do not know whether the rule (2S) is necessary for convergence. While we cannot answer that question, we can see experimentally whether it tends to improve the result.

We performed experiments on three classes of functions because different functions have different simplification behavior. The results are reported in Tables I, II, and III.

For each function the number under SIZE gives the smallest number of terms found by any of the methods described in this paper. The next three columns indicate percentage improvement achieved by many iterations of the procedure P in comparison with one run of EXMIN.

The columns labeled "2" refer to runs where both term-increasing rules (1S) and (2S) are used.

The columns labeled "1" refer to runs where only the one term-increasing rule (1S) is used.

The columns labeled "0" refer to runs where no term increasing rules are used, but only L-transformations and EXMIN are used.

In order to answer question 1) posed at the beginning of this section, we compare the columns labeled "1" and "0" for many iterations of P. While a statistical analysis would be difficult, we can see to what extent the results are consistent with the hypothesis that any differences are due to random variations only. We see that for only one function did column "0" produce better improvement than column "1," while column "1" produced better improvement more than half of the time. This is not likely to be due to random variations only and indicates that a temporary increase in the number of terms does improve the chances of finding better solutions.

To answer question 2) we compare the columns labeled "2" and "1" for many iterations of P. Here the results are much less conclusive. While column "2" produced smaller expressions more often than column "1," most of the time they were equal. Based on this data we cannot reject the hypothesis that any variations are purely statistical.

The purpose of this research is not to implement a good minimization algorithm for AND-EXOR expression, but to evaluate some theoretical consideration. However, the nondeterministic minimizer obtained better solutions than any other deterministic "minimizer" published to date, although it required more computation time. In the following table, data for EXORCISM, HERMES and EXMIN are obtained from [9], [25], [21].

TABLE I
FUNCTIONS $E(n,k)$ DEFINED IN [20]

FUNCTION	SIZE	30 ITERATIONS		
		2	1	0
E(6,3)	12	14	14	14
E(6,4)	11	8	8	8
E(7,2)	15	0	0	0
E(7,3)	21	19	19	11
E(7,4)	22	4	4	4
E(7,5)	15	6	6	6
E(8,2)	20	5	5	5
E(8,3)	32	15	13	13
E(8,4)	32	32	32	11
E(8,5)	34	10	8	8
E(8,6)	20	0	0	0
E(9,2)	24	0	11	7
E(9,3)	48	12	12	14
E(9,4)	58	22	19	8
E(9,5)	60	26	22	14
E(9,6)	49	11	9	5
E(9,7)	24	11	11	7
E(10,2)	32	9	9	9
E(10,3)	70	11	10	5
E(10,4)	105	10	15	3
E(10,5)	91	41	41	10
E(10,6)	101	21	25	12
E(10,7)	70	0	4	1
E(10,8)	32	11	11	11

TABLE II
ARITHMETIC FUNCTIONS

FUNCTION	SIZE	50 ITERATIONS		
		2	1	0
ADR4	31	6	6	6
LOG8	96	2	7	4
MLP4	61	8	6	5
NRM4	73	3	3	3
RDM8	31	0	0	0
ROT8	35	5	3	0
SQR8	114		6	3
WGT8	54	18	11	0

TABLE III
RANDOM FUNCTIONS

FUNCTION	SIZE	40 ITERATIONS		
		2	1	0
R(6,8)	6.1	0	0	0
R(6,16)	8.6	3	3	3
R(6,24)	9.6	11	8	6
R(6,32)	10.3	10	11	8
R(6,40)	10.6	12	12	7
R(6,48)	9.7	20	20	18
R(6,56)	7.0	17	17	13
R(7,16)	11.2	3	3	3
R(7,32)	15.6	7	7	5
R(7,48)	17.3	11	11	4
R(7,64)	18.7	15	15	8
R(7,80)	19.3	14	15	7
R(7,96)	16.7	27	26	18
R(7,112)	12.3	17	17	16
R(8,32)	20.6	6	5	5
R(8,64)	29.4	7	7	4
R(8,96)	34.3	11	9	6
R(8,128)	35.6	14	15	6
R(8,160)	35.4	17	16	7
R(8,192)	29.8	28	28	7
R(8,224)	21.2	34	34	32

Table IV is not meant for comparison, because computation times are not comparable and initial representations are not identical. Instead Table IV indicates how much room for improvement exists in the current state of the art, which is still very immature compared with inclusive-OR sum of products minimization.

TABLE IV
RESULTS FOR DIFFERENT MINIMIZATION ALGORITHMS

FUNCTION	EXORCISM	HERMES	EXMIN	non-Deterministic
ADR4	34	34	32	31
MLP4	212	92	66	61
SQR6	40	39	39	35

VII. CONCLUSIONS

We have investigated conditions that make rewrite rules convergent, i.e., capable of generating optimal AND–EXOR expressions. We have given one necessary condition for convergence, namely, a temporary increase in the number of terms during simplification must be allowed.

We have also given a sufficient condition for convergence. Assuming that a given set of rules already contains the rules (S1), (S6), which is normally the case, the set of rules can be made convergent by adding (1S) and (2S). We have also considered the possibility of replacing (2S) by the term nonincreasing rules of EXMIN, but were not able to prove or disprove whether this would result in a convergent set of rules.

We have shown how a given expression can be transformed in various ways so as to explore the search space of a given simplifier.

We have performed experiments in order to answer the two questions stated at the beginning of Section VI. From the experiments we drew these conclusions:

1) Adding the rule (1S) to EXMIN does give better results in comparison with a more thorough exploration of the search space.
2) Adding the rule (2S), in addition to (1S) does not improve the results significantly.

Given the above conditions for convergence of AND–EXOR expressions, a natural question arises as to what are the convergence conditions for AND–OR expressions. It is easy to see that a convergent set of rules can be obtained from the following two equations by allowing bidirectional application:

$$x + \bar{x} = 1$$
$$1 + 1 = 1.$$

The argument is based on the fact that these rules are capable of turning any AND–OR expression into a minterm expansion. However, reverse applications of the two equations increase the number of product terms, and it is unknown whether such an increase is necessary for the minimization of AND–OR expressions.

APPENDIX

Section III contains an informal proof that (S1, S6, 1S, 2S) is a convergent set of rules. The informal proof is not sufficient for one detail (Lemma A6), for which we need some formalism.

We assume the usual definitions of variable, literal, product term, minterm.

Definition A1: An *ESOP* is a list of product terms, written as $t_1 \oplus \cdots \oplus t_n$ $(n \geq 0)$.

We will write *copies*(t, E) to denote the number of appearances of a term t in an ESOP E.

Example: Consider the ESOP $E = xy \oplus \bar{z} \oplus xy \oplus 1$. Then
copies$(xy, E) = 2$,
copies$(\bar{z}, E) = 1$,
copies$(1, E) = 1$,
copies$(t, E) = 0$ for all other product terms t.

Definition A2: For two ESOP's E and F,
"$E = F$" if copies(t, E) = copies(t, F) for every term t.
"$E \equiv F$" iff E and F evaluate to the same result for any values substituted for their variables.

Examples:

$$t \oplus s \oplus t = s \oplus t \oplus t \neq s$$
$$t \oplus s \oplus t \equiv s \oplus t \oplus t \equiv s.$$

Note: As a result of Definition A2 an ESOP E is uniquely defined by specifying copies (t, E) for every term t.

Definition A3: If t is a term and E, F are ESOP's then we define
"$t \in E$" if copies$(t, E) > 0$
"$E \subseteq F$" iff copies$(t, E) \leq$ copies(t, F), for every term t.
"$E \oplus F$" is defined by copies$(t, E \oplus F)$ = copies(t, E)+ copies(t, F), for every term t.
If $E \subseteq F$ then "$F - E$" is defined by copies$(t, F - E)$ = copies(t, F) − copies(t, E), for any term t.
"$E \cap F$" is delined by copies$(t, E \cap F)$ = min(copies(t, E), copies(t, F)) for every term t.
The ESOP "0" is defined by copies$(t, 0) = 0$ for every term t.

Note: Before we can write an expression $E - F$ we have to verify that $E \subseteq F$ because otherwise $E - F$ is undefined.

We use the symbol "0" to denote the number 0 as well as the empty ESOP, as no confusion can arise.

Definition A4: If t is a product term not containing the variable x then the AND of x and t, written xt, is obtained by including x in t. Similarly for $\bar{x}t$.

Definition A5: An ESOP E *has a redundancy* K if

$$K \subseteq E, K \equiv 0, K \neq 0.$$

Note: A redundancy K consists of terms that can be removed without changing the function of E.

Lemma A6: Let F be an ESOP, t be a term and x be a variable satisfying

$$\begin{aligned} &x \text{ does not appear in the term } t, \\ &F \oplus t \text{ has no redundancy, and} \\ &F \oplus xt \oplus \bar{x}t \text{ has a redundancy.} \end{aligned} \tag{100}$$

Then $F \oplus xt \oplus \bar{x}t$ has a unique redundancy K. Moreover,

$$K \text{ contains either } xt \text{ or } \bar{x}t, \text{ but not both.} \tag{101}$$

Proof: We first prove (101) for any redundancy K of $F \oplus xt \oplus \bar{x}t$. If K contained neither xt nor $\bar{x}t$ then $K \subseteq F$, violating (100). If K contained both then $K - (xt \oplus \bar{x}t) \oplus t \equiv 0$, violating (100). This proves (101).

To prove that K is unique assume two redundancies K, L of $F \oplus xt \oplus \bar{x}t$.

$$\text{Let } I = K \cap L, K' = K - I, L' = L - I. \tag{102}$$

Since $K \equiv L \equiv 0$

$$I \oplus K' \equiv I \oplus L'. \tag{103}$$

EXORing both sides of (103) with $I \oplus L'$ and simplifying we get

$$K' \oplus L' \equiv 0. \tag{104}$$

From (102), $K' \cap L' = 0$ and therefore

$$K' \oplus L' \subseteq F \oplus xt \oplus \bar{x}t. \tag{105}$$

Suppose that

$$\text{one of } K, L \text{ contains } xt \text{ and the other contains } \bar{x}t. \tag{106}$$

Then

$$\begin{aligned} &K' \oplus L' - (xt \oplus \bar{x}t) \oplus t \subseteq F \oplus t && \text{(by (105))} \\ &K' \oplus L' - (xt \oplus \bar{x}t) \oplus t \equiv 0 && \text{(by (104))}. \end{aligned}$$

Thus $K' \oplus L' - (xt \oplus \bar{x}t) \oplus t$ is a violation of (100). Therefore (106) cannot be true.

If both K and L contain xt then K' and L' contain neither xt nor $\bar{x}t$ (by (101)),

$$\text{and therefore } K' \oplus L' \subseteq F. \tag{107}$$

This implies $K' \oplus L' = 0$ (by (100), (104), (107) and Definition A5 of redundancy). Therefore $K' = 0$ and $L' = 0$ and hence $K = L$. The same argument applies if both K and L contain $\bar{x}t$. Q.E.D.

ACKNOWLEDGMENT

We thank Mr. Kouda for the use of his programs as part of our implementation.

REFERENCES

[1] Ph. W. Besslich, "Efficient computer method for ExOR logic design," *IEE Proc.*, vol. 130, pt. E, pp. 203–206, 1983.
[2] R. K. Brayton and C. T. McMullen, "The decomposition and factorization of Boolean expressions," in *Proc. ISCAS,* 1982, pp. 49–54.
[3] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis.* Boston, MA: Kluwer, 1984.
[4] D. Brand and T. Sasao, "On the minimization of AND-EXOR expressions," Res. Rep. RC 16739, IBM T. J. Watson Research Center, Apr. 1991.
[5] M. Davio, J-P. Deschamps, and A. Thayse, *Discrete and Switching Functions.* New York: McGraw-Hill Int., 1978.
[6] S. Even, I. Kohavi, and A. Paz, "On minimal modulo-2 sum of products for switching functions," *IEEE Trans. Electron. Comput.,* vol. EC-16, pp. 671–674, Oct. 1967.
[7] H. Fleisher, M. Tavel, and J. Yeager, "A computer algorithm for minimizing Reed–Muller canonical forms," *IEEE Trans. Comput.,* vol. C-36, no. 2, pp. 247–250, Feb. 1987.
[8] H. Fujiwara, *Logic Testing and Design for Testability.* Cambridge, MA: Computer Systems Series, M.I.T. Press, 1986.
[9] M. Helliwel and M. Perkowski, "A fast algorithm to minimize multi-output mixed polarity generalized Reed-Muller forms," in *Proc. 25th Design Automat. Conf.,* June 1988, pp. 427–432.
[10] S. J. Hong, R. G. Cain, and D. L. Ostapko, "MINI: A heuristic approach for logic minimization," *IBM J. Res. Develop.,* vol. 18, pp. 443–458, Sept. 1974.
[11] A. Mukhophadhyay and G. Schmitz, "Minimization of EXCLUSIVE-OR and LOGICAL EQUIVALENCE switching circuits," *IEEE Trans. Comput.,* vol. C-19, no. 2, pp. 132–140, Feb. 1970.

[12] D. E. Muller, "Application of Boolean algebra to switching circuit design and to error detection," *IRE Trans. Electron. Comput.*, vol. EC-3, pp. 6–12, Sept. 1954.

[13] S. Muroga, *Logic Design and Switching Theory.* New York: Wiley, 1979.

[14] W. V. Quine, "A way to simplify truth functions," *Amer. Math. Monthly*, vol. 62, pp. 627–631, Nov. 1955.

[15] G. Papakonstantinou, "Minimization of modulo-2 sum of products," *IEEE Trans. Comput.*, vol. C-28, pp. 163–167, Feb. 1979.

[16] S. M. Reddy, "Easily testable realization for logic functions," *IEEE Trans. Comput.*, vol. C-21, pp. 1183–1188, Nov. 1972.

[17] J. P. Robinson and C. L. Yeh, "A method for modulo-2 minimization," *IEEE Trans. Comput.*, vol. C-31, pp. 800–801, Aug. 1982.

[18] K. K. Saluja and E. H. Ong, "Minimization of Reed–Muller canonical expansion," *IEEE Trans. Comput.*, vol. C-28, pp. 535–537, Feb. 1979.

[19] T. Sasao, "Input variable assignment and output phase optimization of PLA's," *IEEE Trans. Comput.*, vol. C-33, no. 10, pp. 879–894, Oct. 1984.

[20] T. Sasao and P. Besslich, "On the complexity of MOD-2 sum PLA's," *IEEE Trans. Comput.*, vol. 39, no. 2, pp. 262–266, Feb. 1990.

[21] T. Sasao, "EXMIN: A simplification algorithm for Exclusive-OR-Sum-of-Products expressions for multiple-valued input two-valued output functions," ISMVL-90, Charlotte, NC, May 1990.

[22] ——, "Exclusive-or Sum-of-Products expressions: Their properties and minimization algorithm," IEICE Tech. Rep., VLD90-87, Dec. 1990.

[23] ——, "A transformation of multiple-valued input two-valued output functions and its application to simplification of exclusive-or sum-of-products expressions," in *Proc. ISMVL-91*, Victoria, BC, Canada, May 1991, pp. 270–279.

[24] T. Sasao, "Logic Synthesis with EXOR Gates," in *Logic Synthesis and Optimization*, T. Sasao, Ed. Boston: Kluwer, 1993, pp. 259–285.

[25] J. M. Saul, "An improved algorithm for the minimization of mixed polarity Reed-Muller representation," in *Proc. ICCD-90*, Cambridge, MA, pp. 372–375.

Daniel Brand was born in Prague, Czechoslovakia, on April 8, 1949. He received the Ph.D. degree in computer science from the University of Toronto in 1976.

Since then he has been working at the IBM T. J. Watson Research Center with temporary stays at the IBM Zurich Research Laboratory, Beijing Institute of Aeronautics and Astronautics and the Kyushu Institute of Technology. He has worked in the areas of verification, communication protocols and logic synthesis.

Tsutomu Sasao (S'72–M'77–SM'90) was born in Osaka, Japan, on January 26, 1950. He received the B.E., M.E., and Ph.D. degrees in electronic engineering from Osaka University, Osaka, Japan, in 1972, 1974, and 1977, respectively.

Since 1988, he has been with the Department of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka, Japan. From 1977 to 1988, he was on the Faculty of Osaka University, Osaka, Japan. In 1982, he spent a year at the IBM T. J. Watson Research Center, where he developed an AND–OR PLA minimization system, and a multilevel logic synthesis system. In 1990, he spent three months at the Naval Postgraduate School, Monterey, CA, where he improved the AND–EXOR minimization program. His research interests include logic design and switching theory, especially, the complexity analysis of logic circuits, design methods for PLA's, and application of multiple-valued logic.

Dr. Sasao was the Asia Area Program Chairman in 1984, and the Program Chairman in 1992 for the International Symposium on Multiple-Valued Logic (ISMVL). Also he was the Organizer and the Chairman of the International Symposium on Logic Synthesis and Microprocessor Architecture, Iizuka, Japan, in July 1992. He has published four books on switching theory and logical design in Japanese including a book on PLA design and test. He is a member of the Institute of Electronics, Information and Communication Engineers (IEICE), and is an Associate Editor of *IEICE Transactions on Information and Systems*. He is also a member of the Information Processing Society of Japan. He was the Chairman of the Japanese Research Group on Multiple-Valued Logic during 1989–1991. He received the NIWA Memorial Award in 1979, and a Distinctive Contribution Award from the IEEE Computer Society MVL-TC for the paper presented at ISMVL 1986.