Check for updates

An Object-Oriented Approach to Switching Circuit Minimization

James Paul Vita
City College of New York

The determination of a minimal function to realize a specified set of outputs is a basic problem of switching circuit design. Circuit production costs have evidenced a steady decline over the past few years due mainly to technological advances in the areas of silicon compilation and VLSI implementation. In a just a few years, cost factors have taken a backseat to implmentation issues. Nonetheless, basic minimization techniques at the design level, are still a major vehicle for cost containment.

Techniques for the minimization of switching networks by Boolean reduction, Tabular methods, and Karnaugh Maps are well established. Such techniques can be easily implemented in conventional programming languages. Algorithmic implementations of such techniques, however, tend to be rather specialized and lacking in generality. In this paper we shall examine an object oriented approach to the circuit minimization problem. It will be demonstrated that an object representation of a circuit specification, forms the basis for an Expert System with the capability for both simulation and design.

The procedures described in this paper were developed for a Personal Computer using PROLOG-2 (Expert Systems International). PROLOG was chosen for two reasons. First, the goal was to develop a set of procedures which would implement the symbolic minimization of a set of circuit object representations. PROLOG provides intrinsic facilities for the representation of objects as unit clauses. Further, since PROLOG is a rule-based language by nature, Boolean minimization theorems are easily represented by PROLOG rules. Having specified the theorems as rules, the simplification is performed automatically by the PROLOG proof mechanism.

Specifying Circuit Requirements

A minimum specification for a circuit is a set of outputs under all possible input conditions. Such a specification can be represented as a list. A simple combinatorial function of n inputs can have 2^n possible outputs. The output specification is a canonical list, the order of terms being the value of the minterm or maxterm. As an example consider the function f(A,B) = A + B. There are four possible input conditions represented by 00, 01, 10, and 11. The output list corresponding to these conditions (assuming a boolean value of "1" for true and "0" for false) is as follows:

```
[00,01,10,11]
[ 0, 1, 1, 1]
```

The latter of these two lists is the output specification. Each value is the output under the corresponding input in the former list. Another type of possible output value is "Don't Care". This can be repre-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specfic permission.

© ACM 1988 0-89791-271-3/88/0006/0975 $1.50

sented by an "-" in the corresponding list position, as in [1,1,1,-,-,0,0,-]. These values are allowed to be "1" or "0".

Creating Minterm/Maxterm Objects

Analysis of the output function begins with the creation of the minterm and maxterm sets. The dual nature of the minterms and maxterms allows the same procedures to be used with each set. Creation of the minterm objects depends upon the examination of the terms which produce a "1" as output. For the output specification [0,0,0,1], the only term which produces a "1" output is the final one in the list which corresponds to the input condition "11" (A=1,B=1). The form of the minterm and maxterm objects has been chosen to pair the name of the input variable with its value. Thus, the following minterm object is created:

```
min_term([i(A,1),i(B,1)]).
```

The object has "minterm" as its principal functor and has as its only argument a list representing the name-value input pairs. The "i/2" notation is a dummy functor which achieves the required pairing. Representation of the minterm (or maxterms) as lists of inputs allows the same procedures to be used with output specifications requiring a arbitrary number of input variables. The output specification given above generates the following maxterm objects:

```
max_term([i(A,0),i(B,0)]).
max_term([i(A,0),i(B,1)]).
max_term([i(A,1),i(B,0)]).
```

The first step in the minimization is the derivation of the unreduced function expression from the output specification. The output specification list is examined element by element. If the term under consideration produces a "1" output, then a minterm must be generated. If the output is "0" then a maxterm must be generated. If the required output is a "-" ("Don't Care") then both a minterm and maxterm can be generated. Each minterm or maxterm is created by decoding the position of the term in the output specification list. For a function of n inputs, n bit positions must be examined in each of the 2^n terms. Once the minterm and maxterm expressions are generated, the unreduced expression(s) can be determined as a disjunction of the minterms and a conjunction of the maxterms. The first step in the procedure is to solve the goal "solve/1". Consider the following example, using the output specification [0,1,1,1,0,0,0,1].

```
?- solve([0,1,1,1,0,0,0,1]).

Output is a maximal function of 3
variables.

f(a,b,c) = a'b'c + a'bc' + a'bc + abc
f(a,b,c) = Σ (1, 2, 3, 7)

f(a,b,c) = (a+b+c) (a'+b+c)
           (a'+b+c') (a'+b'+c)
f(a,b,c) = Π (0, 4, 5, 6)

yes
?-
```

The following clauses are also asserted into the PROLOG database:

```
min_term([i(a,0),i(b,0),i(c,0)]).
min_term([i(a,0),i(b,1),i(c,0)]).
min_term([i(a,0),i(b,1),i(c,1)]).
min_term([i(a,1),i(b,1),i(c,1)]).


max_term([i(a,0),i(b,0),i(c,0)]).
max_term([i(a,1),i(b,0),i(c,0)]).
max_term([i(a,1),i(b,0),i(c,1)]).
max_term([i(a,1),i(b,1),i(c,0)]).
```

Determining Minimal Expressions

For a circuit minimization procedure to be useful and practical, it must satisfy two requirements: com-

pleteness and uniqueness. The results of Boolean reduction are dependent upon the order in which terms are clashed. Thus, it is possible to achieve different solutions. In fact, it is possible for a particular order to make the minimal expression underivable. The completeness requirement insures that we derive all possible reduced forms. The uniqueness requirement insures that we eliminate redundant forms, an important consideration for large circuit expressions.

The advantage of an object-oriented approach to the minimization problem is the ability to symbolically manipulate circuit expressions. The symbolic approach is most closely related to the terms in with design engineers formulate minimization problems. Further, the symbolic approach more naturally captures the functional relationships within the problem.

The first step in the minimization procedure is to state the Boolean reduction theorems as PROLOG rules. Circuit minimization is achieved by repeatedly applying the rules until no further simplification is possible. The result is a minimal expression which is logically equivalent to the original circuit. PROLOG rules were created for the Boolean properties of idempotence, absorption, and consensus.

Logically, the property of idempotence can be stated as follows:

```
X+X = X    1(a)
XX = X     1(b)
```

Expression 1(a) would be represented in PROLOG as:

```
min_term([i(X,1)]).
min_term([i(X,1)]).
```

that is, two clauses for X could be contained in the PROLOG database. The idempotence rule tells us that one of these minterms can be eliminated or "retracted". After application of this rule, the database would consist only of:

```
min_term([i(X,1)]).
```

The absorption property can be described logically as:

```
X+XY = X      2(a)
X(X+Y) = X    2(b).
```

The equivalent PROLOG implementation of 2(a) would be that:

```
min_term([i(X,1)]).
min_term([i(X,1),i(Y,1)]).
```

is equivalent to

```
min_term([i(X,1)]).
```

The larger minterm is completely subsumed by a smaller term, and is therefore redundant. Consequently, the larger term term can be retracted.

A corollary to the absorption property states that:

```
XYZ+X'Y = YZ+X'Y                  3(a)
(X+Y+Z)(X'+Y) = (Y+Z)(X'+Y)      3(b).
```

In 3(a), the X'Y is "nearly" contained within the term XYZ. The only difference is the variable X which is uncomplemented in the first term and complemented in the second term. The corollary states that this cross-signed variable is redundant and can be eliminated. In terms of the PROLOG database:

```
min_term([i(X,1),i(Y,1),i(Z,1)]).
min_term([i(X,0),i(Y,1)]).
```

is equivalent to

```
min_term([i(Y,1),i(Z,1)]).
min_term([i(X,0),i(Y,1)]).
```

or to

```
min_term([i(X,1),i(Y,1),i(Z,1)]).
min_term([i(Y,1)]).
```

The crossed-sign variable can be eliminated from either term when the other conditions are met. The latter variation however, allows a further simplification to simply:

```
min_term([i(Y,1)]).
```

by an additional application of the absorption property.

The property of consensus can be demonstrated by:

```
XY+X'Z+YZ = XY+X'Z              4(a)
(X+Y)(X'+Z)(Y+Z)=(X+Y)(X'+Z)   4(b).
```

The PROLOG clauses for 4(a):

```
min_term([i(X,1),i(Y,1)]).
min_term([i(X,0),i(Z,1)]).
min_term([i(Y,1),i(Z,1)]).
```

would minimize to:

```
min_term([i(X,1),i(Y,1)]).
min_term([i(X,0),i(Z,1)]).
```

The above examples have all be conducted by applications of the minimization rules on minterms. However, an exactly dual process can be conducted on the maxterms with identical results.

PROLOG Minimization Procedure

Implementing the minimization procedure in PROLOG is a somewhat tedious process, consisting of four steps. The first step is to produce a copy of each minterm and maxterm and add an extra argument, numbering the terms. Second, at each iteration two terms must be chosen which will be examined for the property of idempotence and absorption. When examination of pairs of terms is exhausted, trios of terms must be examined for the property of consensus. When minimization is completed on any one step, the process must be restarted from the beginning since the new clauses which are generated may reduce other terms. Third, a "matching" process must be performed which examines the components of the pair or trio of the minterm or maxterm. No assumptions can be made about the order of variables within the minterms or maxterms since the minimization procedure will eliminate some but retain other variables (components). Finally, after the components of the minterms and maxterms are "matched", the actual minimization rules may be applied, and the clauses of the database updated. This process is repeated until no further minimization is possible. When this is achieved, the only task remaining is to print the results in a readable format.

Manipulating Terms

As already noted, copying the minterms and maxterms is the first step in the minimization procedure. When PROLOG matches objects in the database it has no insight into the fact that two terms are the same term. Naming the terms prevents PROLOG from inappropriately attempting to apply the idempotence rule to a single clause. Naming the minterms and maxterms by number also allows a degree of control over the order in which terms are compared. Permutations of the clause numbering can be formed. An indirect matching can be performed from these permutations. A "choose" function (implemented in PROLOG) is used to select pairs and trios of clauses from the database. Normally, the PROLOG database is searched from top to bottom when matching clauses. However, the choose function effects an alteration in the clause order without actually physically moving the clauses in the database. This allows a complete set of clause clashings to be examined.

Once the minterms or maxterms are copied and numbered, they are

gathered into a list which is permuted into all possible orderings of clause numbers. Pairs and trios are selected for clashing by a PROLOG choose function. The predicate "match/8" performs the selection of the clauses to be clashed. The clauses "gen_term/2" are the copied and named versions of the minterm or maxterm clauses (depending upon which minimization is being performed). Choose is a resatisfiable predicate which selects permutations of n clauses taken k at a time. Pairs of clauses are passed to the "clash/5" predicate. Trios of clauses (used in the implementation of consensus) are handled by special rules which do not use the clashing procedure. The PROLOG implementation of the matching procedure is given below.

```
match(Term_list,A,AN,B,BN,X,Y,Z) :-
      choose(Term_list,2,[AN|[BN]]),
      gen_term(AN,A),
      gen_term(BN,B),
      AN \= BN,
      clash(A,B,X,Y,Z).

match(Term_list,A,AN,B,BN,C,CN) :-
      choose(Term_list,3,
         [AN|[BN|[CN]]]),
      gen_term(AN,A),
      gen_term(BN,B),
      gen_term(CN,C),
      AN \= BN,
      CN \= AN,
      CN \= BN.
```

Clashing Terms

Clashing terms is a complicated process, in which no assumptions may be made about the ordering of terms or the ordering of variables within terms. The components of the terms must be compared pairwise. Although there are probably several ways in which to effect this comparison, the one selected works efficiently and extracts the information necessary to apply the minimization rules. The clashing process examines two terms and extracts three lists from the terms: a list of matching components, a list of components from the first term unrepresented in the second, and a list of components of the second term unrepresented in the first term. The following example demonstrates how these lists are extracted from clashing of the minterms XY and XZ:

```
?- clash([i(X,1),i(Y,0)],
         [i(X,1),i(Z,1)],
         Match_list,
         ListA,
         ListB).
```

Execution of the above goal produces the following bindings:

```
?- clash([i(X,1),i(Y,0)],
         [i(X,1),i(Z,1)],
         [i(X,1)],
         [i(Y,1)],
         [i(Z,1)]).
```

The code which effects these bindings is as follows:

```
clash([],[],[],[],[]) :- !.
clash([],X,Match_list,ListA,ListB) :-

append([],Match_new,Match_list),
      append([],NewA,ListA),
      append(X,NewB,ListB),
      clash([],[],Match_new,
         NewA,NewB).
clash(X,[],Match_list,ListA,ListB) :-

append([],Match_new,Match_list),
      append(X,NewA,ListA),
      append([],NewB,ListB),
      clash([],[],Match_new,
         NewA,NewB).
clash([HeadA|RestA],X,Match_list,
   ListA,ListB) :-
      member(HeadA,X),
      delete(HeadA,X,RestB),
      append([HeadA],Match_new,
         Match_list),
      append([],NewA,ListA),
      append([],NewB,ListB),
      clash(RestA,RestB,Match_new,
         NewA,NewB).
```

```
clash(X,[HeadB|RestB],Match_list,
   ListA,ListB) :-
      member(HeadB,X),
      delete(HeadA,X,RestA),
      append([HeadA],Match_new,
         Match_list),
      append([],NewA,ListA),
      append([],NewB,ListB),
      clash(RestA,RestB,Match_new,
         NewA,NewB).
clash([HeadA|RestA],[HeadB|RestB],
   Match_list,ListA,ListB) :-
      HeadA \= HeadB,
      append([HeadA],NewA,ListA),
      append([HeadB],NewB,ListB),

append([],Match_new,Match_list),
      clash(RestA,RestB,Match_new,
         NewA,NewB).
```

The clash procedure examines the elements of the two lists and recursively builds the matching, and nonmatching lists by determining the set membership of each list element. The "clashed" lists are in a form that can be minimized by PROLOG rules.

Minimization Rules

The minimization is performed by simple PROLOG rules. For all of the Boolean theorems except consensus, the results of the "clash/5" procedure are passed to the "reduce/7" predicate. The calls to the reduce predicate have the form:

```
reduce(TermA,NumberA,TermB,NumberB,
   Matching,RestA,RestB).
```

Consensus is handled as a exception by special rules.

The rule which identifies and reduces the expressions by the idempotence property is as follows:

```
/*  Exactly Matching Terms  */
/*                          */
/*           AA = A         */
/*       (A+A) = A          */
/*                          */
reduce(A,AN,B,BN,X,[],[]) :-
      A = B,
      retract(gen_term(AN,A)).
```

The arguments to "reduce/7" are the two clauses being compared (A, B), the ordering number (AN, BN), the list of matching subterms from the two clauses, the nonmatching subterms from the first clause, and the nonmatching subterms from the second clause. As can be seen above, this definition of the reduce predicate recognizes that A and B are the same clause. There are no nonmatching terms as indicated by the null lists ([]). When these conditions are met one of the clauses can be eliminated. As can be seen above, one of the clauses is retracted from the PROLOG database.

The absorption rule is somewhat more complicated and is given by:

```
/*  One Term is Completely Subsumed
         by a Larger Term          */
/*                                */
/*          AB+ABC = AB           */
/*     (A+B)(A+B+C) = (A+B)       */
/*                                */
reduce(A,AN,B,BN,X,[],_) :-
      retract(gen_term(BN,B)).
reduce(A,AN,B,BN,X,_,[]) :-
      retract(gen_term(AN,A)).
```

In this case, it can be seen that only one nonmatching list is null ([]). This indicates that one of the terms is contained within the other but the two do not match exactly. This is a more general case for the exactly matching case above. When these conditions are met, the larger of the two terms may be retracted from the PROLOG database. There are two instances of this predicate because the larger term may be the first or second of those "clashed". The reduction rule must be able to recognize both variations in order to effect a complete minimization.

Recognizing partially matching terms differing in sign by one subterm variable is somewhat more com-

plicated. The PROLOG rules which accomplishes this reduction is as follows:

```
/*      One Subterm Subsumed by
        Larger Term Except for a
         Single Variable which
         Differs in Sign              */
/*                                    */
/*        ABC+A'B = BC+A'B            */
/*    (A+B+C)(A'+B) = (B+C)(A'+B)     */
/*                                    */
reduce(A,AN,B,BN,X,[i(N,0) | [] ],
   ListB) :-
      member(i(N,1),ListB),
      delete(i(N,1),B,NewB),
      retract(gen_term(BN,B)),
      assert(gen_term(BN,NewB)).
reduce(A,AN,B,BN,X,[i(N,1) | [] ],
   ListB) :-
      member(i(N,0),ListB),
      delete(i(N,0),B,NewB),
      retract(gen_term(BN,B)),
      assert(gen_term(BN,NewB)).
reduce(A,AN,B,BN,X,ListA,
    [i(N,0) | [] ]) :-
      member(i(N,1),ListA),
      delete(i(N,1),A,NewA),
      retract(gen_term(AN,A)),
      assert(gen_term(AN,NewA)).
reduce(A,AN,B,BN,X,ListA,
   [i(N,1) | [] ]) :-
      member(i(N,0),ListA),
      delete(i(N,0),A,NewA),
      retract(gen_term(AN,A)),
      assert(gen_term(AN,NewA)).
```

Here we assume that the two terms will match up to a single subterm. A check is made to see if the single subterm is a complemented version of some member of what is left over in the other term. If this condition is met, the variable from be deleted from either term. By convention, we choose to eliminate the variable from the larger term. Four instances of the predicate are require because we cannot make any assumptions about the position that the crossed-sign variable will be found. It can be found in either term and its value in the first may be "0" or "1". This creates four specific cases. The predicate relies on a destructive list function (implemented in PROLOG) which allows the deletion of a member from a list.

Recognizing the property of concensus is the most difficult of all of the Boolean theorems to implement. It involves the simultaneous comparison of three terms. As in the other cases, no assumption may be made about the positions in which the matching subterms will be found.

```
/* One Term is a Cross Product of
          Two Other Terms           */
/*                                  */
/*       A'B+BC+AC = A'B+AC         */
/* (A'+B)(B+C)(A+C) = (A'+ B)(A+C)  */
/*                                  */
comp(1,0).
comp(0,1).

reduce([i(L,LV0),i(M,MV)],AN,
   [i(L,LV1),i(O,OV)],BN,
   [i(M1,MV1),i(O1,OV1)],CN) :-
       comp(LV0,LV1),
       permutation([i(M,MV),i(O,OV)],
          [i(M1,MV1),i(O1,OV1)]),
       retract(gen_term(CN,_)).

reduce([i(L,LV0),i(M,MV)],AN,
   [i(M1,MV1),i(O1,OV1)],BN,
   [i(L,LV1),i(O,OV)],CN) :-
       comp(LV0,LV1),
       permutation([i(M,MV),i(O,OV)],
          [i(M1,MV1),i(O1,OV1)]),
       retract(gen_term(BN,_)).

reduce([i(M1,MV1),i(O1,OV1)],AN,
   [i(L,LV0),i(M,MV)],BN,
   [i(L,LV1),i(O,OV)],CN) :-
       comp(LV0,LV1),
       permutation([i(M,MV),i(O,OV)],
          [i(M1,MV1),i(O1,OV1)]),
       retract(gen_term(AN,_)).
```

A Complete Example

The reduction procedure described above are carried out until no further minimization can be performed. The procedures are carried out on both minterms and maxterms,

yielding minimal and equivalent, though not necessarily equal expressions. The minimization procedure is concluded by printing the reduced solution(s) found and tallying up the number of gates required by each simplified form. The following is a complete example of the techniques using the output specification [0,1,1,1,0,0,0,1].

```
?- solve([0,1,1,1,0,0,0,1]).

Output is a maximal function of 3
 variables.

f(a,b,c) = a'b'c + a'bc' + a'bc + abc
f(a,b,c) = Σ (1, 2, 3, 7)

f(a,b,c) = (a + b + c) (a' + b + c)
        (a' + b + c') (a' + b' + c)
f(a,b,c) = Π (0, 4, 5, 6)

yes
?- minimize.

Minimal Expression Derived from
Minterms

bc + a'b + a'c

Gate Summary
7 Total Gates (3 And, 2 Or, 2 Not)


Minimal Expression Derived from
Maxterms

(a' + c) (a' + b) (b + c)

Gate Summary
7 Total Gates (2 And, 3 Or, 2 Not)


yes
?-
```

The expression derived from minterms is kept in sum of products form, while the expression derived from maxterms is kept in product of sums form. Expanding the product of sums form derived from the maxterms, and performing a little extra minimization shows that the two expressions are indeed equivalent. There is no obvious simplification that will reduce bc + a'b + a'c. The expression is thus considered minimal.

Completeness and Uniqueness

Early in this paper, it was stated that a useful minimization algorithm would have to have the properties of completeness and uniqueness. Ideally, it would desired for the algorithm to derive a single absolutely minimal form. However, we know from applications of Boolean reduction methods that several logically equivalent though precisely different forms are often attainable. Even among forms which require an equal number of logical gates, a particular form may be more desirable than another because it is composed of a preponderance of a particular type of gate which is easier to fabricate, less expensive, or more commonly available. Thus, we would really like to have the opportunity to examine all possible minimal forms.

Since minimization by Boolean reduction requires clashing of terms, different results are possible when the clashing order is varied. To comply with the completeness requirement, it is necessary to employ a procedure which produces a complete set of the possible clashes among terms. It is here that the use of PROLOG as a development tool finds its greatest advantage. The minimization procedure was initiated by producing a numbered copies of the original minterms and maxterms. Clashes were selected by means of a "choose" procedure applied to permutations of the original minterm and maxterm order. The natural backtracking mechanisms inherent in PROLOG carry the procedure through all possible permutations of the original minterm and maxterm sets.

The effect is to attempt all possible clashings of the terms.

There are enormous costs in terms of computation time to a procedure employing such a strategy. The worst-case behavior of a simple algorithm to find a single minimal solution by clashing is $\binom{n}{2}$. However, when all possible clashings are examined the order of complexity rises to $\binom{n}{2} \cdot n!$ This can be demonstrated by observing that the $\binom{n}{2}$ clashes must be performed against the n! possible orderings of the original minterms and maxterms.

The uniqueness requirement can be stated as a desire to examine only unique solutions. Many of the clashings will yield identical solutions. As each solution is obtained, we can determine whether it is a permutation of a solution already derived. Redundant solutions can thus be eliminated. This adds a level of complexity which is trivial when compared to the complexity added by the completeness requirement.

Observations

The object of this paper has been to describe a set of procedures to perform a symbolic minimization of switching circuit expressions by a PROLOG implementation of Boolean simplification rules. The application of such rules with simple circuits at least, produces accurate results. Artificial constraints on completeness and uniqueness were placed on the results to be derived. The results are undoubtedly of theoretical significance. However, as in many applications of Artificial Intelligence to real life problems, the practical significance given the computational overheads involved can be questioned at many levels. The questions raised are certainly valid given the level of computational "horsepower" presently attainable. However, on a daily basis we are presented with the evidence that computing machines are advancing to a level of unprecedented computational power. Given these technological advantages, it is not heresy to predict, that algorithms which approach problems in the same manner as the human user, will be technically, as well as, theoretically practical.

The author welcomes any feedback on the material presented here. Correspondence should be addressed to the author at City College of New York. A text of the code will be made available to interested conference participants.