

AD-A221 766

WRDC-TR-90-5006



2

PROVING BOOLEAN EQUIVALENCE WITH PROLOG

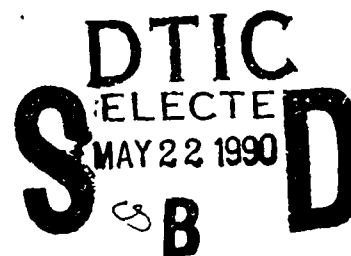
MICHAEL ALAN DUKES, M.S.E.E.
CAPTAIN, U.S. ARMY
AIR FORCE INSTITUTE OF TECHNOLOGY

FRANK MARKHAM BROWN, PhD
PROFESSOR OF ELECTRICAL ENGINEERING
AIR FORCE INSTITUTE OF TECHNOLOGY

February 1990

FINAL REPORT FOR PERIOD JAN 89 TO FEB 90

Approved for public release; distribution unlimited.



ELECTRONIC TECHNOLOGY LABORATORY
WRIGHT RESEARCH AND DEVELOPMENT CENTER
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6543

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patent invention that may in any way be related thereto.

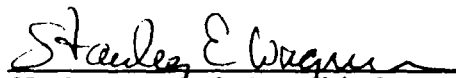
This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.


MICHAEL A. DUKES, Capt, USA
Air Force Institute of Technology


JOHN W. HINES, Chief
Design Branch
Microelectronics Division

FOR THE COMMANDER


STANLEY E. WAGNER, Chief
Microelectronics Division
Electronic Technology Laboratory

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WRDC/ELED, WPAFB, OH 45433-6543 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188		
1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b RESTRICTIVE MARKINGS			
2a SECURITY CLASSIFICATION AUTHORITY			3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.			
2b DECLASSIFICATION/DOWNGRADING SCHEDULE						
4 PERFORMING ORGANIZATION REPORT NUMBER(S)			5 MONITORING ORGANIZATION REPORT NUMBER(S) WRDC-TR-90-5006			
6a NAME OF PERFORMING ORGANIZATION AFIT/ENG		6b OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION WRDC/ELED			
6c ADDRESS (City, State, and ZIP Code) WRIGHT-PATTERSON AFB OH 45433-6543			7b ADDRESS (City, State, and ZIP Code) WRIGHT-PATTERSON AFB OH 45433-6543			
8a NAME OF FUNDING/SPONSORING ORGANIZATION		8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER IN-HOUSE			
8c ADDRESS (City, State, and ZIP Code)			10 SOURCE OF FUNDING NUMBERS			
			PROGRAM ELEMENT NO. 62204F	PROJECT NO. 6096	TASK NO. 40	WORK UNIT ACCESSION NO. 18
11 TITLE (Include Security Classification) PROVING BOOLEAN EQUIVALENCE WITH PROLOG						
12 PERSONAL AUTHOR(S) DUKES, MICHAEL ALAN BROWN, FRANK MARKHAM						
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM 01/89 TO 02/90		14. DATE OF REPORT (Year, Month, Day) Feb 90		
15 PAGE COUNT 22						
16 SUPPLEMENTARY NOTATION The computer software contained herein are "harmless." Already in the public domain.						
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) BOOLEAN METHODS - EQUIVALENCE - LOGIC PROGRAMMING			
FIELD	GROUP	SUB-GROUP				
12	05					
12	05					
19 ABSTRACT (Continue on reverse if necessary and identify by block number) This report explains a Prolog program that performs Boole's Expansion Theorem. The Prolog program proves equivalence of Boolean formulas using the $f = g$ form. The pattern matching feature of Prolog increases the efficiency of the proof process in some cases by avoiding expansion on every term of f and g . A proof of the Prolog program's correctness is also offered. The operators used within the confines of the algorithm are complement, conjunction, disjunction, and exclusive or. Some examples are presented to demonstrate the efficiency of the Prolog program.						
20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS				21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a NAME OF RESPONSIBLE INDIVIDUAL CAPTAIN MICHAEL A. DUKES				22b TELEPHONE (Include Area Code) (512) 255-8626		
				22c OFFICE SYMBOL WRDC/ELED		

Table of Contents

	Page
I. Introduction	1
II. Background: Boole's Expansion Theorem	2
III. Analysis of the Problem	3
IV. Prolog Implementation	5
V. Examples	10
VI. Conclusions	17
VII. References	18



Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-i	

1. Introduction

The purpose of this report is to demonstrate a small Prolog program that verifies the relation $f = g$ where f and g are two Boolean formulas. Portions of the proof process are accomplished using the pattern matching feature of Prolog. The Prolog program was developed to augment the theorem proving process of the Higher Order Logic (HOL) system as described in [1] and [2]. The Prolog program implements Boole's Expansion Theorem within the confines of Prolog's depth-first search. A type of Greedy algorithm is also presented through the generous use of cuts. The operators used within the confines of the algorithm are complement, conjunction, disjunction, and exclusive or. Finally, the Prolog routine is presented with some examples.

The organization of the presentation includes a short explanation of the theory, examination of the structure of the problem, discussion of the solution in Prolog, presentation of some examples, and some conclusions. Boolean formulas of n variables are represented by $f(X)$, $g(X)$, or $h(X)$ where X is an n -variable vector. The operators used are $+$ for disjunction, $*$ for conjunction, $'$ for complement, and $\oplus$ for exclusive or. The $*$ may be dropped when doing so does not lead to ambiguities. Furthermore, f , g , or h may be used in place of $f(X)$, $g(X)$, or $h(X)$, respectively. The elements of X may be enumerated as $X_1, X_2, \dots, X_n$ to show the first element, second element, and up to the n th element.

II. Background: Boole's Expansion Theorem

Boole's Expansion Theorem, which was specialized to switching functions in [3], states for a function of one variable that [4]

$$f(x) = f(1)x + f(0)(1-x).$$

For a function of two variables [4]

$$f(x,y) = f(1,1)xy + f(1,0)x(1-y) + f(0,1)(1-x)y + f(0,0)(1-x)(1-y).$$

The form " $(1-x)$ " was used by Boole to express "the complement of x " and the "+" operator was used as modulo-two sum. Shannon provides an expansion about one variable whose values may only be in $\{0,1\}$ [3]:

$$f(X_1, X_2, \dots, X_n) = X_1 f(1, X_2, \dots, X_n) + X_1' f(0, X_2, \dots, X_n).$$

For two variables the Shannon expansion is

$$\begin{aligned} f(X_1, X_2, \dots, X_n) = & X_1 X_2 f(1, 1, X_3, \dots, X_n) + X_1 X_2' f(1, 0, X_3, \dots, X_n) \\ & + X_1' X_2 f(0, 1, X_3, \dots, X_n) + X_1' X_2' f(0, 0, X_3, \dots, X_n). \end{aligned}$$

The expansion is further generalized to functions of n variables by both Boole [4] and Shannon [3]. A proof that Boole's Expansion Theorem holds for every n -variable Boolean function is provided in [5]. Even though Shannon is often credited with the development of this expansion, it was originally developed by Boole. This expansion process will therefore be referred to in this paper as Boole's Expansion Theorem.

III. Analysis of the Problem

The problem, given two n -variable Boolean formulas, f and g , is to show that they are equivalent using Boole's Expansion Theorem. We may assume that the formulas will be presented separately. Since the problem is to show that $f = g$, the two formulas may be further expanded as follows.

Theorem 1 $f = g$ iff

$$f(1, X_2, \dots, X_n) = g(1, X_2, \dots, X_n) \quad (1)$$

$$f(0, X_2, \dots, X_n) = g(0, X_2, \dots, X_n). \quad (2)$$

Proof 1 Without loss of generality, we will consider the case $\forall x \in \mathbf{B} f(x) = g(x)$ where $\mathbf{B}$ is the carrier for a Boolean algebra. By Boole's Expansion Theorem, it is true that

$$f(x) = xf(1) + x'f(0)$$

and

$$g(x) = xg(1) + x'g(0).$$

We may then perform the appropriate substitution for $f(x) = g(x)$.

$$(f(x) = g(x)) \Leftrightarrow ((xf(1) + x'f(0)) = (xg(1) + x'g(0)))$$

From [6] we have $(u = v) \Leftrightarrow (u \oplus v = 0)$. Thus

$$\begin{aligned} (f(x) = g(x)) &\Leftrightarrow (xf(1) + x'f(0)) \oplus (xg(1) + x'g(0)) = 0 \\ &\Leftrightarrow x'g(0)f'(0) + xg(1)f'(1) + x'f(0)g'(0) + xf(1)g'(1) = 0 \\ &\Leftrightarrow x(g(1)f'(1) + f(1)g'(1)) + x'(g(0)f'(0) + f(0)g'(0)) = 0 \\ &\Leftrightarrow x(g(1)f'(1) + f(1)g'(1)) = 0 \text{ and } x'(g(0)f'(0) + f(0)g'(0)) = 0 \end{aligned}$$

Then $\forall x \in \mathbf{B}$ we have the system

$$\begin{aligned} x(g(1)f'(1) + f(1)g'(1)) &= 0, \\ x'(g(0)f'(0) + f(0)g'(0)) &= 0. \end{aligned}$$

The above system is true iff $f(1)=g(1)$ and $f(0)=g(0)$. □

The process of Theorem 1 is performed recursively over all variables of the formulas f and g . From the original set of formulas, $f = g$, expansion on the first variable leads to two separate equations. Expansion on the next variable leads to four separate equations. The process continues until 2^n separate equations exist. Boolean formulas that represent the same Boolean function may contain literals such that once a given number of variables have been expanded on, the result might be two Boolean formulas that match in their pattern of literals. Consider the following example for f and g .

$$\begin{aligned} f(x, y, z) &= x(y + z) + y'z. \\ g(x, y, z) &= xy + y'z. \end{aligned}$$

By Theorem 1, $f = g$ holds iff $f(0, y, z) = g(0, y, z)$ and $f(1, y, z) = g(1, y, z)$, i.e., $f = g$ iff the result

$$y'z = y'z$$

and

$$y + z + y'z = y + y'z$$

is verified.

At this point, it is no longer necessary to expand on the formulas produced from $x = 0$, since $y'z = y'z$ simply by pattern matching. Thus, for some $j \leq n$, portions of the expansion process may be performed in $O(j)$ time simply by pattern recognition. The pattern matching feature of Prolog can then supply increased efficiency simply by checking for one Boolean formula to match a second Boolean formula before expanding on its variables.

IV. Prolog Implementation

Based on the considerations in the previous section, a Prolog implementation was generated. In this section, the Prolog code to implement the proof of $f = g$ will be presented. The Prolog code is contained in two files. The first file, called ops, is used to define the operators. The second file, called verify, performs the expansion and verification of the Boolean formulas.

The following is a listing of the file ops. The four operators previously defined for logical disjunction, conjunction, complement, and exclusive or had to be redefined to accommodate Prolog below. The `op(510,yfx,$)` defines the `$` as $\oplus$. The `op(500,yfx,@)` defines the `@` as $+$. The `op(400,yfx,^)` defines the `^` as $*$. The `op(300,fx,~)` defines the `~` as $'$.

```
:- op(510,yfx,$).
:- op(500,yfx,@).
:- op(400,yfx,^).
:- op(300,fx,~).
```

The `eval` clauses provide the proper evaluation for the operators. The format of the `eval` clause is

$$\text{eval}(X \text{ op } Y, Z)$$

or

$$\text{eval}(Y, Z)$$

where X and Y are complemented or uncomplemented terms and Z is the derived term.

```
eval(1 @ _,1):-!.
eval(1 ^ X,X):-!.
eval(1 $ X,~ X):-!.
eval(0 @ X,X):-!.
eval(0 ^ _,0):-!.
eval(0 $ X,X):-!.
eval(_ @ 1,1):-!.
eval(X ^ 1,X):-!.
eval(X $ 1,~ X):-!.
eval(X @ 0,X):-!.
eval(_ ^ 0,0):-!.
eval(X $ 0,X):-!.
eval(~ 0,1):-!.
eval(~ 1,0):-!.
eval(~ X @ X,1):-!.
eval(X @ ~ X,1):-!.
eval(~ X ^ X,0):-!.
eval(X ^ ~ X,0):-!.
eval(X $ X,0):-!.
eval(~ X $ X,1):-!.
eval(X $ ~ X,1):-!.
eval(X @ X,X):-!.
eval(X ^ X,X):-!.
eval(X,X):-!.
```

For the remainder of this section, the clauses of the verify program will be discussed in the order they are called.

The first clause called in verify is `go`. The success of this clause is based upon the existence of two Boolean formulas expressed within a fact of arity two called `eqtn`. Further, the `eq` clause must be satisfied with regard to the two Boolean formulas from the `eqtn` fact. Prior to execution of the `eq` clause, the `evaluate` clause is called to reduce expressions that meet the criteria of the `eval` clauses. A Boolean formula, $f(u, v, w, x, y, z)$, where u and v have been set to some value in $\{0,1\}$, could then be reduced to a formula of $f(w, x, y, z)$ before executing the `eq` clause.

```
go :-
    eqtn(X,Y),
    evaluate(X,XNew),
    evaluate(Y,YNew),
    eq(XNew,YNew).
```

The `evaluate` clause is also called from a later clause called `divide`. The `evaluate` clause calls upon the `eval` clauses loaded from the ops file containing the operator definitions. At this point, the Boolean formula is reduced based upon eliminations of terms under the `eval` rules.

```
evaluate(X,X):-atomic(X),!.
evaluate(~F,FReduced) :-
    evaluate(F,FTemp),
    eval(~FTemp,FReduced),!.
evaluate(L@R,Resolved) :-
    evaluate(L,LNew),
    evaluate(R,RNew),
    eval(LNew@RNew,Resolved).
evaluate(L^R,Resolved) :-
    evaluate(L,LNew),
    evaluate(R,RNew),
    eval(LNew^RNew,Resolved).
evaluate(L $ R,Resolved) :-
    evaluate(L,LNew),
    evaluate(R,RNew),
    eval(LNew $ RNew,Resolved).
```

The `eq` clause calls other clauses in order to perform Boole's Expansion Theorem. The clause first checks to see if there exists a straight pattern match between both formulas. If so, then success is achieved upon this branch of the depth-first search tree. However, should immediate success not be achieved, a variable is first extracted from the f Boolean formula through the `extract` clause. The next step is to generate the $f(0)$, $f(1)$, $g(0)$, and $g(1)$ Boolean formulas from the f and g Boolean formulas using the variable chosen from the `extract` clause. Then the `eq` clause is called recursively to see if $f(0) = g(0)$ and $f(1) = g(1)$.

```
eq(X,X):-!.
eq(F,G) :-
    extract(X,F),
    divide(F,X,F0,F1),
    divide(G,X,G0,G1),!,
    eq(F0,G0),!,
    eq(F1,G1),!.
```

The **extract** clause finds the first available variable in the operator tree for f . The first **extract** clause checks to see if a leaf node has been reached. Should it be the case that a leaf node is reached, then the leaf node is checked to be either a 1 or 0. Otherwise, the leaf node is a variable. The other **extract** clauses allow for search down the tree on the four operators complement, disjunction, conjunction, or exclusive or.

```
extract(X,X) :-
    atom(X),!.

extract(X,~Y) :-
    extract(X,Y).

extract(X,L@_) :-
    extract(X,L).

extract(X,@R) :-
    extract(X,R).

extract(X,L^_) :-
    extract(X,L).

extract(X,~R) :-
    extract(X,R).

extract(X,L $ _) :-
    extract(X,L).

extract(X,_ $ R) :-
    extract(X,R).
```

The next clause called upon by the **eq** clause is **divide**. The **divide** clause performs two functions. First, the $f(0)$ and $f(1)$ Boolean formulas are generated strictly by replacing each occurrence of the variable of interest with the appropriate 0 or 1 value in the operator tree for f . The second part of the **divide** clause involves evaluation of the $f(0)$ and $f(1)$ Boolean formulas to eliminate the occurrence of 0 and 1 where possible, and occurrences of terms that are eliminated due to the assignment of 0 or 1.

```
divide(F,X,F0,F1) :-
    remove_x_0(F,X,F0Temp),
    remove_x_1(F,X,F1Temp),
    evaluate(F0Temp,F0),
    evaluate(F1Temp,F1).
```

The next clause considered is the **remove_x_0** clause called by the **divide** clause. The purpose of this clause is to search the operator tree of a Boolean formula and replace every occurrence of the given variable with a 0. The **remove_x_0** clause returns the new operator tree when all leaves have been visited.

```
remove_x_0(Y,X,Y) :-
    atom(Y),
```

```

Y \== X,!.
remove_x_0(~Y,X,~Y) :-
    atom(Y),
    Y \== X,!.
remove_x_0(X,X,0):- !.
remove_x_0(~X,X,1):- !.
remove_x_0(~Y,X,~NewY) :-
    !,
    remove_x_0(Y,X,NewY).
remove_x_0(L @ R,X,LNew @ RNew) :-
    !,
    remove_x_0(L,X,LNew),
    remove_x_0(R,X,RNew).
remove_x_0(L ~ R,X,LNew ~ RNew) :-
    !,
    remove_x_0(L,X,LNew),
    remove_x_0(R,X,RNew).
remove_x_0(L $ R,X,LNew $ RNew) :-
    !,
    remove_x_0(L,X,LNew),
    remove_x_0(R,X,RNew).

```

The `remove_x_1` clause is included below for completeness. Everything mentioned for the `remove_x_0` clause is also valid here.

```

remove_x_1(Y,X,Y) :-
    atom(Y),
    Y \== X,!.
remove_x_1(~Y,X,~Y) :-
    atom(Y),
    Y \== X,!.
remove_x_1(X,X,1):- !.
remove_x_1(~X,X,0):- !.
remove_x_1(~Y,X,~NewY) :-
    !,
    remove_x_1(Y,X,NewY).
remove_x_1(L @ R,X,LNew @ RNew) :-
    !,
    remove_x_1(L,X,LNew),
    remove_x_1(R,X,RNew).
remove_x_1(L ~ R,X,LNew ~ RNew) :-
    !,
    remove_x_1(L,X,LNew),
    remove_x_1(R,X,RNew).
remove_x_1(L $ R,X,LNew $ RNew) :-
    !,
    remove_x_1(L,X,LNew),
    remove_x_1(R,X,RNew).

```

The `evaluate` clause is explained after the discussion of the `go` clause.

All of the clauses described above may be placed in one file to be loaded at once; however, there is a specific order of declaration that must be followed. The four operation declarations using the `op` directive must be read first by Prolog. Afterwards, the remaining clauses may appear in any order. If Quintus Prolog is being used, the clauses with the same clause head should be grouped together.

Caution in writing the code was used to ensure conformance to Clocksin and Mellish standard Prolog [7]. To date, the system runs under Quintus Prolog, CProlog, and Prolog86. The code has been run on an IBM PC-AT, SUN 4, VAX 11/785, MicroVAX 3600, and VAX 8800. For reading in Boolean formulas greater than one or two pages in length, Quintus Prolog appears to be the only implementation of the three that succeeds.

V. Examples

This Boolean formula verification system has largely been used to verify hardware specifications and implementations. Some of the more interesting examples of the use of this routine have been in comparing large Boolean formulas that make extensive use of the exclusive or operator. Most of the examples presented below compare Boolean formulas containing exclusive or operations.

An example run in Quintus Prolog is provided. In this case, we wish to prove De Morgan's Law between f and g . The formulas are declared in a clause called `eqtn` within a file called `equation`.

```
eqtn((~( x ~ y)),(~ x @ ~ y)).
```

The following is a log of the session verifying the equivalence of both formulas.

```
Quintus Prolog Release 2.4.2 (Sun-4, SunOS 4.0)
Copyright (C) 1988, Quintus Computer Systems, Inc. All rights reserved.
1310 Villa Street, Mountain View, California (415) 965-7700
```

```
| ?- compile(['ops']).
[compiling /tmp_mnt/auto/quintus/mdukes/wrdc2/ops...]
[ops compiled 0.400 sec 1,656 bytes]

yes
| ?- compile(['verify']).
[compiling /tmp_mnt/auto/quintus/mdukes/wrdc2/verify...]
[verify compiled 1.417 sec 3,428 bytes]

yes
| ?- ['equation'].
[consulting /tmp_mnt/auto/quintus/mdukes/wrdc2/equation...]
[equation consulted 0.034 sec 280 bytes]

yes
| ?- go.

yes
| ?- halt.
```

If we change the Boolean formulas of the equation file from

```
eqtn((~( x ~ y)),(~ x @ ~ y)).
```

to

```
eqtn((~( x ~ y)),(~ x ~ ~ y)).
```

the following result will be obtained.

```
CProlog version 1.2a
| ?- ['ops'].
ops consulted 1352 bytes 0.150000 sec.
```

```
yes
| ?- ['verify'].
verify consulted 4980 bytes 0.466666 sec.
```

```
yes
| ?- ['equation'].
equation consulted 84 bytes 0 sec.
```

```
yes
| ?- go.
```

```
no
| ?- halt.
```

[Prolog execution halted]

The next example involves the consideration of parity generation for an eight-input odd parity generation circuit. For this example, a, b, c, d, e, f, g , and h will be used to designate the input variables. Consider the following specification for odd parity generation:

$$j = (a(b(c(d(e(f(g(h)))))))).$$

Even though the expression for j is fairly straightforward the problem is in the implementation. If the expression for j were implemented directly, a delay of seven exclusive or gates would be incurred. Upon rearranging the variables using the associative and commutative properties of exclusive or, an equivalent Boolean formula is obtained:

$$k = (((h(g)(f(e)))(d(c)(b(a)))).$$

A new delay of three exclusive or gates would result for the implementation. Figure 1 shows both the specification and implementation. Using Boole's Expansion Theorem to verify $j = k$ we obtain

Quintus Prolog Release 2.4.2 (Sun-4, SunOS 4.0)
 Copyright (C) 1988, Quintus Computer Systems, Inc. All rights reserved.
 1310 Villa Street, Mountain View, California (415) 965-7700

```
| ?- compile(['ops']).
[compiling /tmp_mnt/auto/quintus/mdukes/wrdc2/ops...]
[ops compiled 0.350 sec 1,656 bytes]
```

```
yes
| ?- compile(['verify']).
[compiling /tmp_mnt/auto/quintus/mdukes/wrdc2/verify...]
[verify compiled 1.417 sec 3,428 bytes]
```

```
yes
| ?- ['equation'].
[consulting /tmp_mnt/auto/quintus/mdukes/wrdc2/equation...]
[equation consulted 0.033 sec 372 bytes]
```

```
yes
```

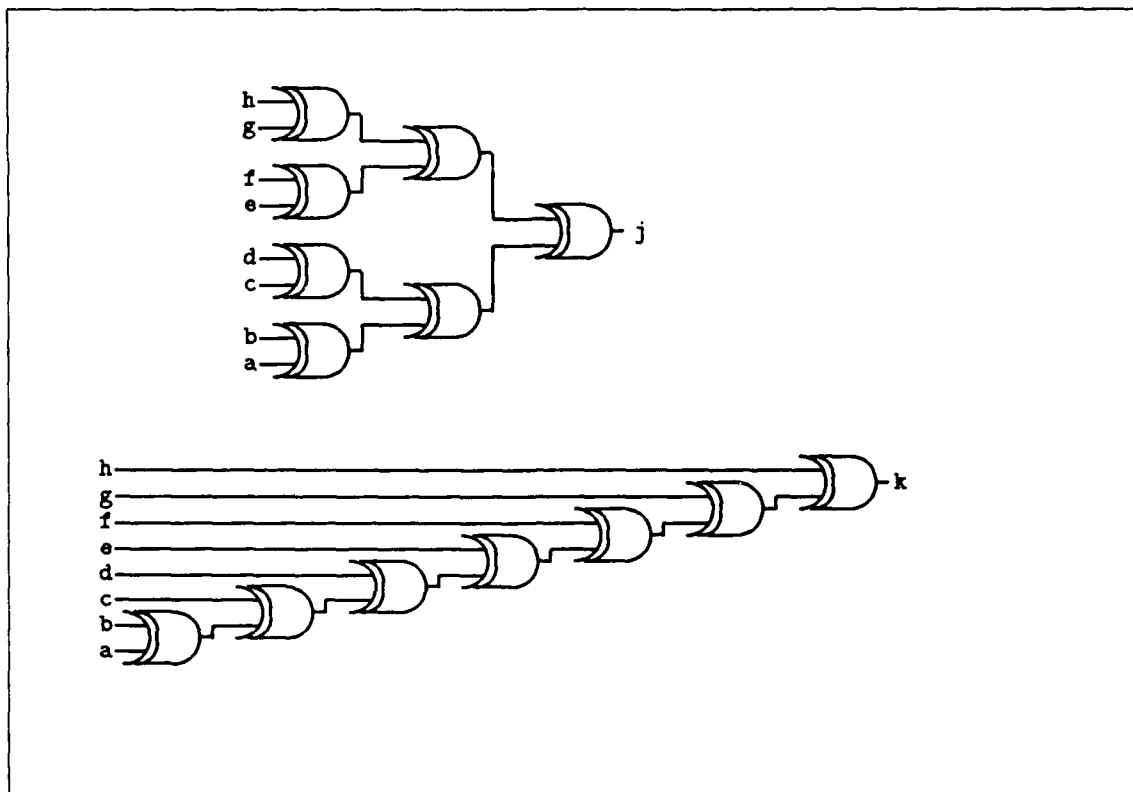


Figure 1. Specification and Implementation of Parity Generator.

```
| ?- go.
```

```
yes
```

```
| ?-
```

```
Stopped
```

```
[33]ares ps -ug
```

```
USER      PID %CPU %MEM    SZ   RSS TT  STAT  START    TIME  COMMAND
mdukes    9421  0.0  2.7  248   840 p1  T    11:39    0:02 Prolog /usr2/eng/mdukes/
```

```
[34]ares fg
```

```
prolog
```

```
halt
```

From the statistics gathered by the system for the Quintus Prolog session on a SUN 4, only two seconds of CPU time were spent in the evaluation. The HOL system was used to expand the formulas j and k using the identity

$$a\$b = (\sim a \wedge b) \vee (a \wedge \sim b).$$

The following new formulas for j and k , called j_alt and k_alt respectively, were obtained.

$$j_alt = ((\sim(\sim(h \wedge g \vee h \wedge \sim g) \wedge (\sim f \wedge e \vee f \wedge \sim e)) \vee$$


```

      (~h ~ g @ h ~ ^g) ~ ^ (~f ~ e @ f ~ ^e)) ~
      (~ (~d ~ c @ d ~ ^c) ~ ^ (~b ~ a @ b ~ ^a) @
      (~d ~ c @ d ~ ^c) ~ ^ (~b ~ a @ b ~ ^a)) @
      (~ (~h ~ g @ h ~ ^g) ~ ^ (~f ~ e @ f ~ ^e) @
      (~h ~ g @ h ~ ^g) ~ ^ (~f ~ e @ f ~ ^e)) ~
      (~ (~d ~ c @ d ~ ^c) ~ ^ (~b ~ a @ b ~ ^a) @
      (~d ~ c @ d ~ ^c) ~ ^ (~b ~ a @ b ~ ^a)))

```

and

```

k_alt = ( ^a ~
  (~b ~
    (~c ~
      (~d ~
        (~e ~ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)) @
          e ~ ^ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h))) @
        d ~
          (~e ~ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)) @
            e ~ ^ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)))) @
      c ~
        (~d ~
          (~e ~ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)) @
            e ~ ^ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h))) @
          d ~
            (~e ~ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)) @
              e ~ ^ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)))) @
        b ~
          (~c ~
            (~d ~
              (~e ~ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)) @
                e ~ ^ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h))) @
              d ~
                (~e ~ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)) @
                  e ~ ^ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)))) @
            c ~
              (~d ~
                (~e ~ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)) @
                  e ~ ^ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h))) @
                d ~
                  (~e ~ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)) @
                    e ~ ^ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)))) @
              a ~
                (~b ~
                  (~c ~
                    (~d ~
                      (~e ~ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)) @
                        e ~ ^ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h))) @
                      d ~
                        (~e ~ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)) @
                          e ~ ^ (~f ~ (~g ~ h @ g ~ ^h) @ f ~ ^ (~g ~ h @ g ~ ^h)))) @
                    c ~
                      (~d ~

```

```

      (~e ~ (~f ~ (~g ~ h @ g ~ ~h) @ f ~ (~g ~ h @ g ~ ~h)) @
      e ~ (~f ~ (~g ~ h @ g ~ ~h) @ f ~ (~g ~ h @ g ~ ~h))) @
d ~
      (~e ~ (~f ~ (~g ~ h @ g ~ ~h) @ f ~ (~g ~ h @ g ~ ~h)) @
      e ~ (~f ~ (~g ~ h @ g ~ ~h) @ f ~ (~g ~ h @ g ~ ~h)))) @
b ~
      (~c ~
      (~d ~
      (~e ~ (~f ~ (~g ~ h @ g ~ ~h) @ f ~ (~g ~ h @ g ~ ~h)) @
      e ~ (~f ~ (~g ~ h @ g ~ ~h) @ f ~ (~g ~ h @ g ~ ~h))) @
      d ~
      (~e ~ (~f ~ (~g ~ h @ g ~ ~h) @ f ~ (~g ~ h @ g ~ ~h)) @
      e ~ (~f ~ (~g ~ h @ g ~ ~h) @ f ~ (~g ~ h @ g ~ ~h)))) @
      c ~
      (~d ~
      (~e ~ (~f ~ (~g ~ h @ g ~ ~h) @ f ~ (~g ~ h @ g ~ ~h)) @
      e ~ (~f ~ (~g ~ h @ g ~ ~h) @ f ~ (~g ~ h @ g ~ ~h))) @
      d ~
      (~e ~
      (~f ~ (~g ~ h @ g ~ ~h) @ f ~ (~g ~ h @ g ~ ~h)) @
      e ~
      (~f ~ (~g ~ h @ g ~ ~h) @ f ~ (~g ~ h @ g ~ ~h)))))) @

```

Attempting to use HOL to rearrange j or k through the laws of commutativity, associativity, distributivity, or De Morgan would have been tedious. Using an HOL tactic called `BOOL_CASES_TAC` [2] for this small example would have required a relatively large amount of CPU time and memory; however, the Prolog program accomplishes the task more efficiently as shown in the expansion that follows.

Quintus Prolog Release 2.4.2 (Sun-4, SunOS 4.0)
 Copyright (C) 1988, Quintus Computer Systems, Inc. All rights reserved.
 1310 Villa Street, Mountain View, California (415) 965-7700

```

| ?- compile(['ops']).
[compiling /tmp_mnt/auto/quintus/mdukes/wrdc2/ops...]
[ops compiled 0.400 sec 1,656 bytes]

yes
| ?- compile(['verify']).
[compiling /tmp_mnt/auto/quintus/mdukes/wrdc2/verify...]
[verify compiled 1.450 sec 3,428 bytes]

yes
| ?- ['equation'].
[consulting /tmp_mnt/auto/quintus/mdukes/wrdc2/equation...]
[equation consulted 0.567 sec 4,996 bytes]

yes
| ?- go.

yes
| ?-

```

Stopped

[32]ares ps -ug

USER	PID	%CPU	%MEM	SZ	RSS	TT	STAT	START	TIME	COMMAND
mdukes	9457	8.6	2.7	248	864	p1	T	12:06	0:04	Prolog /usr2/eng/mdukes/

[33]ares fg

prolog

halt.

Only four seconds of CPU time were expended to verify $j_alt = k_alt$. If k_alt is altered such that the sixth line up in the formula of the equation file is changed from

$$e^{-(f^{(g^h@g^h)}@f^{(g^h@g^h)))@}$$

to

$$e^{-(f^{(g^h@g^h)}@f^{(g^h@g)})@}$$

the following result is obtained, indicating a failure of equivalence.

Quintus Prolog Release 2.4.2 (Sun-4, SunOS 4.0)

Copyright (C) 1988, Quintus Computer Systems, Inc. All rights reserved.

1310 Villa Street, Mountain View, California (415) 965-7700

| ?- compile(['ops']).

[compiling /tmp_mnt/auto/quintus/mdukes/wrdc2/ops...]

[ops compiled 0.383 sec 1,656 bytes]

yes

| ?- compile(['verify']).

[compiling /tmp_mnt/auto/quintus/mdukes/wrdc2/verify...]

[verify compiled 1.433 sec 3,428 bytes]

yes

| ?- ['equation'].

[consulting /tmp_mnt/auto/quintus/mdukes/wrdc2/equation...]

[equation consulted 0.533 sec 4,984 bytes]

yes

| ?- go.

no

| ?-

Stopped

[32]ares ps -ug

USER	PID	%CPU	%MEM	SZ	RSS	TT	STAT	START	TIME	COMMAND
mdukes	9476	11.9	2.7	248	864	p1	T	12:14	0:04	Prolog /usr2/eng/mdukes/

[33]ares fg

prolog

halt.

Again only four seconds of CPU time were expended. Because of the extensive use of cuts, failure usually occurs much sooner than success for larger formulas. Some larger formulas of greater than eight variables have been run through the expansion routine. One expansion involving two

formulas of sixteen variables and greater than 470 pages was run through the expansion routine in less than 15 minutes.

VI. Conclusions

The Prolog implementation of Boole's Expansion Theorem using the $f = g$ form appears to be very simple and efficient. Part of the success of the routine is in the simple pattern matching between an expansion of f and g . This can be most helpful when attempting to verify expressions similar to a straight ripple-carry adder and carry-select adder where the basic adder circuit remains the same. The pattern matching feature helps to reduce the depth-first search space of the expansion process. If two Boolean formulas do not describe the same Boolean function, failure will generally come quickly since a leaf node of the depth-first solution tree will generally contain a conflict before the remaining portion of the formulas is expanded.

Further work is being explored to go beyond the current Greedy algorithm method of the implementation. A type of generalized best-first search option is being considered. In this case, examination of the next variable for expansion is determined using a criterion for selecting the variable of greatest occurrence. Identification of the variable would be further based on reducing the size of the formulas early in the expansion process or the likelihood of causing failure early in the expansion process.

VII. References

1. Cousineau, G., G. Huet and L. Paulson. *The ML Handbook*. INRIA; 1986.
2. Gordon, Michael. *The HOL Manual*. 1987.
3. Shannon, C.E., "The synthesis of two-terminal switching circuits," *Bell System Technical Journal*, vol. 28, no. 1, pp. 59-98, 1949.
4. Boole, George. *An Investigation of the Laws of Thought*. New York: Dover Publications; pp. 72-78, 1854.
5. Brown, Frank Markham. *Boolean Reasoning*. Boston: Kluwer Academic Press; pp. 40-42, scheduled 1990.
6. Rudeanu, Sergiu. *Boolean Functions and Equations*. London: North-Holland Publishing; p. 9, 1974.
7. Clocksin, W. F. and C. S. Mellish. *Programming in Prolog*. New York: Springer-Verlag; 1987.