

CAFE: Connection Arrays From Equations

Department of Electrical and Computer Engineering
University of Wisconsin–Madison
May 1997

CAFE is a program for converting Boolean functions specified in any of a variety of ways to 1) truth tables, 2) connection arrays or equations, both of which express switching functions in multiple-output reduced-cost, sum-of-products form, or 3) other specialized forms of expression. In both connection arrays and equations CAFE attempts to minimize the number of product terms for expressing the set of output variables, and pays less attention to minimizing the size of the product terms.

Connection arrays are a (multiple output) sum-of-products expression of output variables that might be used as programs for programmable logic arrays (PLA); equations give the same minimized expression, but in Boolean notation. Truth tables give a canonical, sum-of-minterms expression that may be used as read-only memory (ROM) programs.

1 The CAFE Language

Data for CAFE is a series of **specifications**. One or more **function specifications** are followed by a single **process specification**.

Function specification(s)
Process specification

Specifications begin on a new line with a less-than (<) followed by a word that gives the type of the specification. All specifications end with a period (.). The body of most specifications consists of a series of **function definitions**; each begins on a new line and terminates with a comma (,), except the last which is terminated by the period that also terminates the specification.

```
< spec-type >      [ comment ]  
  function_definition , [ comment ]  
  ... ,  
  function_definition . [ comment ]
```

1.1 General Rules

The CAFE language is largely free format. Ends of lines are converted to blanks so that function definitions may be split over lines wherever blanks are permitted. But, some entries must begin on new lines, and the end of line is recognized as terminating comments.

- ★ Boolean variables consist of consecutive letters and/or digits, except the one-digit sequences “0” and “1” which express logic constants. All characters from *A* to *z* in the ASCII code are taken as letters, so underlines, brackets, and other characters may be used to write more meaningful variable names.
- ★ Boolean variables are case sensitive, i.e. “ab” and “Ab” are different variables. Command entries (discussed below) are case insensitive.

- ★ Everything following the comma, or period, to the end of the line is a **comment**, and is ignored by CAFE. Lines outside, either before or after, specifications are ignored as comment lines. Blank lines may be introduced to make the data file more readable.
- ★ Syntax is set in sans serif font as seen above. Brackets ([and]) enclose optional entries.
- ★ If the care and don't care conditions for a Boolean function cover the same minterm(s), that common minterm(s) is removed from the don't care list in processing the function.
- ★ Control characters are converted to blanks.

1.2 Specification Heads

A specification begins with a line that begins with the less-than character “<” followed by the type of the specification. The type may be followed by a “>”. Anything beyond the first letter of the type is a comment. The specification types supported at this time are:

Equation	Boolean equations
Arrays	On and don't cares in cube notation
Minterms	Minterm numbers for care and don't cares
Xterms	MaXterm numbers for care and don't cares
Tables	Tabular presentation of care and dc's
Process	Process commands

1.3 Equations

A Boolean equation has the form:

$$\text{variable} = \text{Boolean_expression} [= \text{Boolean_expression}] ,$$

where don't care conditions for the variable on the left may be expressed by using the second, optional equal sign and Boolean expression in a Boolean equation. The last equation must be terminated with a period rather than a comma, and since all equations must be terminated with either a period or comma, all Boolean equations must begin on a new line.

Boolean expressions consist of variables, and operator and punctuation symbols with the usual grammar. Note in Table 1 that multiple operator symbols are available for expressing some logic operations and that operators are listed in hierarchical order; i.e., if parentheses do not dictate otherwise, perform all NOTs first, then all ANDs, ..., then all ORs.

TABLE 1. CAFE Boolean Operators

Symbol	Operation	
- ~ /	NOT	(unary, prefix)
* &	AND	(binary, infix)
\$ ~&	NAND	(binary, infix, non-associative)
@	XOR	(binary, infix)
#	A*~B	(binary, infix, non-commutative)
! ~	NOR	(binary, infix, non-associative)
+	OR	(binary, infix)
()	punctuation	
=	connect	
,	end equation	
.	end last equ.	

Example:

```
<equations>      Full Adder
  p = a @ b,
  g = a * b,
  sum = p @ c,
  carry = g + p * c.
```

1.4 Input Lists

Most of the types of specification described below require the specification head line to be followed by an **input list** line which gives an integer, n , followed by n Boolean variables followed by a period.

```
n n_names . [ comment ]
```

The integer and names must be separated by one or more blanks. The names are used as the input variables for all of the functions defined in the body of the specification.

1.5 Minterm Specification

Following an input list line, each function is defined with a line of the form:

```
variable = list_of_integers [ = list_of_integers ] ,
```

The unsigned integers must be blank separated. They are taken as minterm of the variable on the left of the equal sign. When a second list is given, it provides don't care conditions for the variable. If a minterm number exceeds $2^n - 1$ no error is reported. The rightmost n bits of the number are used. The last function definition must be followed by a period.

Example:

```
<minterm>
  4 a s longname f. Optional comment
  w = 0 1 2 = 3 4,
  x = 12 13 14,
  , y = 4 5 7 11, Comma makes this a comment line
  z = 0 2 20. Really 0 2 4
```

1.6 Maxterm Specification

Maxterm specifications have the same appearance as minterm specifications. The numbers listed are interpreted differently, of course.

1.7 Array Specification

Minterms written in binary could comprise the function definitions of an array section, but greater generality is provided. Think of a function written as a sum of product terms. Then encode each product term with an n -tuple of ternary digits (**trits**). A complemented variable is encoded with a "0"; a true variable is encoded with a "1". If the variable does not appear in the product term, its place is marked with the third of the three ternary digits; any of "x", "X" and "-" are accepted by CAFE. Such an n -tuple written without the commas and parentheses that usually punctuate a tuple is called a **cube**.

$$a \cdot \bar{b} \cdot d \Rightarrow 10x1$$

$$b \cdot c \cdot \bar{d} \Rightarrow x110$$

A function is then defined in array form with:

```
variable = list_of_cubes [ = list_of_cubes ] ,
```

The first list of cubes (an **array**) is commonly known as the **ON-array** of the function. The second array, if present, is the **DC-array**, the don't-care array. The last function definition is terminated with a period rather than a comma.

Cubes are commonly listed in a vertical fashion giving the appearance of a matrix. Blanks may separate trits to make an array more readable.

Example:

```
<Array>
 5 h i j k m.   Input variables for these functions
a = 00x11
    100x1
    1100x
    =
    x1100
    0x110,
A = 1xxxx
    x1xxx.      A = h + i
```

1.8 Tabular Specification

Two lists must precede a table giving on and don't care conditions. The first is an input list as discussed above. The second has the same syntax, but the integer, m , reveals the number of output variables in the table, and the names name those variables. The table then consists of cubes of $n + m$ trits. The last cube is followed by a period. The first n trits specify a product term as illustrated above. The remaining m trits indicate that the product term is an implicant of an output variable with a "1", and that it is a don't care condition with a "x", "X" or "-". Zeros, "0", are NOT taken to indicate that the output variable has value zero when the product term evaluates to logic 1. Thus the table is not restricted to the rigorous definition of a truth table. For example, two listed product terms may cover the same minterm, but have different, and apparently conflicting, output parts.

```
<table>
 4 a b c d.      Input variables
 3              x y z. Output variables
 0 0 0 x 1 0 -
 0 0 x 0 0 1 1.
```

Output variable $x = \bar{a} \cdot \bar{b} \cdot \bar{c}$, and takes the value 1 for minterm 0 despite the appearance of a 0 in its column in the second cube.

Many input conditions are not presented in the above example. They are taken as having an output part of all zeros. The following cubes could be included above to make this explicit, but there is clearly an advantage to not having to do so.

```
1 x x x 0 0 0
x 1 x x 0 0 0
x x 1 1 0 0 0
```

1.9 Process Specification

The body of a process specification consists of one or more **commands**. A command consists of one of the letters of Table 2 followed by an output list.

`command_letter output_list , [comment]`

Either upper or lower case command letters may be used. Each command must start on a new line of the source file. Variables on an output list are **output** variables for the processing of that list. Variables used only in Boolean expressions (to the right of the equal sign) and as input variables in array, minterm, etc. specifications are **input** variables. In general, output variables are functions of some but not all input variables. The number of input variables needed for an output list is limited to 127 on Unix systems, and 63 on DOS systems.

TABLE 2. CAFE Command Letters

Letter	Process Commanded
C	Form Connection Array
E	Form Equations
K	Form description for Kim
L	Form Espresso description
T	Form Truth Table
R	Form ON- and DC-arrays

An output list consists of one or more variables, optionally preceded by a NOT symbol (-, ~ or /) to express that the complement of the variable is to be formed, separated by blanks. The last variable **MUST** be followed by a comma (period for the last command.) The variables on an output list must appear on the left of the equal sign in a Boolean equation or minterm, maxterm or array specification, or on the output line in a tabular specification, but not all such variables need appear on an output list. An array is formed for each command with the exceptions that truth tables are formed only if eleven or fewer input variables are required to express the output variables.

Left variables may appear in other Boolean expressions, or as input variables for minterm, maxterm, array or table specifications. When such variables do not appear on the output list, and hence are not output variables, they serve as **indirection** variables. Back substitution is performed to relate output variables to input variables only; values of indirection variables do not appear in result arrays.

However, if a variable depends indirectly upon itself, a loop exists in the logic network. When loops are encountered a message is printed, and the loop is broken and logic 1 is placed on the opened input terminal. If the network is truly combinational, then the injected value is of no consequence. If the network is sequential, then the array produced does depend on the injected value. However, that array does not describe a realization of the sequential switching function. Therefore, when sequential networks are to be processed, loops should be broken in the function definitions by introducing dummy input variables.

Example:

```
<process>                               Ok, lets get some answers
  c p g sum carry,                       Look at the Full Adder
  c sum carry,                           p and g are indirect
  c sum carry /sum /carry. Both active high and low
```

2 Semantics of Results

2.1 Connection Arrays

Only input variables that actually affect the value of one or more output variables appear as labels on the left columns of connection arrays. Output variables appear on the right in connection arrays unless they are trivial (logic 0, logic 1, equal an input variable); an equation is printed for trivial variables. Input variables are assigned left columns according to the order in which they appear in the equation set. If a specific order is desired, then the first equation might be a dummy equation that introduces the variables in the desired order. Output variables are assigned columns in the output list order.

If the following file is supplied to CAFE, the results immediately below it are obtained. The NAND operator is not associative and $a\$b\c therefore expresses $a \cdot b + \bar{c}$. Using the don't care condition $a\#b = a \cdot \bar{b}$, y simplifies to $a + \bar{c}$, which is what is reported in the connection array. Variable b is not shown as an input variable for this output list. The \$ cost reported is the number of 0's and 1's in the connection array.

```
<equations>
dummy = a*b*c*d*e,
x = a*c*d + (a!c)*d,
y = a$b$c = a#b,
z = c @ y.
<process>
c x y z .
q
```

```
Connection Arrays from Equations   May 20 1997  16:13:23   Page 1
      File:  ex1                               Run Options:  e
```

```
3 Input Variables
3 Output Variables
5 Cube Connection Array: $ = 15
=====
acd xyz
=====
001 1--
111 1--
1xx -1-
0xx --1
x0x -11
```

The left (input) part of each row of a connection array describes a logic product (AND gate). True variables in the product are indicated with a 1; complements are specified with a 0. The x indicates that the input variable is not included in the product. The right (output) part of each row indicates the output variable(s) to which the product term applies. The 1's in an output variable column reveal the product terms that must be logically summed to express the output variable in sum-of-products form. The array above indicates that x is the logic sum of two product terms, the first being $\bar{a} \cdot \bar{c} \cdot d$. A 1 therefore specifies a connection between the AND gate (input part of row) and the OR gate (output variable column) that provides the output variable. A connection array then provides the program

for a PLA realization of the output variables, just as a truth table provides the program for a ROM realization.

CAFE attempts to minimize the number of rows in a connection array and the number of connections, but does not always give absolute minimum results.

2.2 Espresso

The **L** (espresso) command produces an array with rather different semantics. The example results below, for the same equation set but command “**L x y z .**”, show that a number of lines give information about the switching function before the array that details the (multiple-output) function. The **.i** and **.o** lines give the number of input and output variables respectively. Names of the input and output variables are listed on the **.ilb** and **.ob** lines respectively. The espresso array provided is of type **fd**; only ON and DON'T CARE information is provided. Finally the **.p** line indicates the number of cubes in the array that immediately follows.

Semantics of espresso and connection arrays differ. In the espresso array, both absent input variables (x's) and don't cares (d's) are expressed with “-”. A one in an output column indicates that the product term of the row is a part of the ON-array of the output variable. A “-” indicates a don't care condition for the output variable. Zeros (0's) in output parts are really just place-markers since type **fd** arrays do not report OFF information. Note that *z* has a don't care condition, although none was listed in the equation for *z*. It inherits the condition from *y* via its care dependency on *y*. This result file may be used as data for espresso with or without editing, even if source lines are echoed.

```
Connection Arrays from Equations   May 20 1997  16:17:35   Page 1
      File:  ex1                      Run Options:  e
.i 4
.o 3
.ilb a b c d
.ob  x y z
.type fd
.p 6
    101- 0--
    0-01 100
    1-11 100
    11-- 010
    0-1- 001
    --0- 011
.end
```

2.3 On and Dc Arrays

The **R** command gives ON- and DC-arrays for the output variables in a simple form suitable for reading with the **readarray()** function from the CLSS library used in course ECE751. The **K** (Kim) command gives similar results, but control information required by the Kim multiple-level synthesizer program is inserted. If **K** is the first command given, the CAFE result file can be fed directly to the synthesizer, even though source lines are echoed. The first name following the **K** is taken as the name of the network, and not as an output variable. It must not be one of the variables that appears in a function definition.

3 Running CAFE

At the command prompt, just type:

```
cafe [ source_file [ result_file ]] [ options ]
```

CAFE reads equations and commands from `stdin`, and writes results to `stdout` and error messages to `stderr`. `Stdin` and `stderr` may be redirected via the operating system (< and >) or by listing source and result file names on the command line.

3.1 Options

Any of the options of Table 3 may be selected. They may appear before or after file names; more than one letter may be given following the hyphen (-).

TABLE 3. CAFE Options

Letter	Effect
-e	Suppress echoing equations with results
-f	Fast connection arrays
-r	Repeat reduction of connection arrays
-x	Extraction algorithm minimization

Connection arrays and equations are produced very rapidly with the `-f` option, but they may not be very desirable as almost no effort is made to minimize them. With the `-r` option, some of the minimization steps in forming connection arrays and equations are repeated until the number of cubes does not decrease. In some cases the cost of the results are reduced by a small amount.

When forming connection arrays and equations, each output variable is normally reduced to a nonredundant sum-of-prime-implicants form. With the `-x` option the minimum sum-of-prime-implicants expression is found with the extraction algorithm (find extremals, throw away less-than prime implicants, branch when cyclic structures are encountered). Relatively innocent-looking equations can demand hours of processing with this algorithm, and often it does not provide a lower cost connection array.

3.2 Errors

No further CAFE commands are processed when an error is found in a function definition, or input or output list. The offending line is echoed with a pointer to the point where the error was detected and a message about the error, as illustrated here:

```
Connection Arrays from Equations   May 20 1997  16:49:14   Page 1
      File:  ex2                               Run Options:

1:  <equations>
2:   x = a*c*d + (a!c)*d),
ERROR: -----^ Matching ( not found
3:   y = a$b$c = a#b,
4:   z = c @ y.
```

CAFE terminated because of 1 errors!

4 Examples of CAFE Use

4.1 Flattening a Multi-level Description

A file named `ripple4` listed below describes a 4-bit wide ripple-adder. ON- and DC-arrays, a truth table, equations and a connection array are requested.

```
<Equations>
dummy = a3*a2*a1*a0 + b3*b2*b1*b0 + Cin,
s0 = a0 @ b0 @ Cin,
c0 = a0*b0 + (a0 + b0)*Cin,
s1 = a1 @ b1 @ c0,
c1 = a1*b1 + (a1 + b1)*c0,
s2 = a2 @ b2 @ c1,
c2 = a2*b2 + (a2 + b2)*c1,
s3 = a3 @ b3 @ c2,
c3 = a3*b3 + (a3 + b3)*c2.
<Process>
r s0,
t c0 s0 -c0 -s0,
e s1 c1,
c c3 s3 s2 s1 s0 .
```

In the results that follow the ON- and DC-arrays and truth table are rather transparent. In preparing the equations for s_1 and c_1 variables s_0 and c_0 serve as indirection variables and are eliminated by back substitution. The final connection array requested has 135 cubes and hence is not shown. The count of 135 may be expected: $31 = 2^{n+1} - 1$ cubes for c_3 , and $60 + 28 + 12 + 4$ cubes for the *sum* output variables ($4(2^{n+1} - 1)$ for s_n). Check the equations for s_1 and c_1 to understand these numerical results.

```
Connection Arrays from Equations   May 20 1997  16:57:43   Page 1
File: ripple4                      Run Options:  er
```

```
Names:  a0 b0 Cin
3 0 ON(s0)
    111
    001
    100
    010
end
0 0 DC(s0)
end
```

```
3 Input Variables
4 Output Variables
8 Cube Truth Table
```

```
-----
abC cs--
00i 00cs
  n  00
-----
000 0011
001 0110
010 0110
011 1001
100 0110
101 1001
110 1001
111 1100
```

$$s1 = -a1*b1*b0*Cin + -a1*b1*-b0*-Cin + -a1*-a0*b1*-Cin + -a1*-a0*b1*-b0 +$$

$$-a1*a0*-b1*Cin + -a1*a0*-b1*b0 + a1*-b1*-b0*-Cin + a1*b1*b0*Cin +$$

$$a1*-a0*-b1*-Cin + a1*-a0*-b1*-b0 + a1*a0*b1*Cin + a1*a0*b1*b0$$

$$c1 = b1*b0*Cin + a0*b1*Cin + a0*b1*b0 + a1*b0*Cin + a1*b1 +$$

$$a1*a0*Cin + a1*a0*b0$$

4.2 Finite-State Machine

A finite-state machine is designed using different types of flip-flops on pages 204-213 of LOGIC AND COMPUTER DESIGN FUNDAMENTALS by Mano and Kime. The tables and equations given are duplicated in the following source file for CAFE.

A finite state machine from LCDF, page 209.

<Table>

3	A	B	X.							
7				An	Bn	Ja	Ka	Jb	Kb	Y.
	0	0	0	0	0	0	x	0	x	0
	0	0	1	0	1	0	x	1	x	1
	0	1	0	1	0	1	x	x	1	0
	0	1	1	0	1	0	x	x	0	0
	1	0	0	1	0	x	0	0	x	0
	1	0	1	1	1	x	0	1	x	1
	1	1	0	1	1	x	0	x	0	0
	1	1	1	0	0	x	1	x	1	0.

<Equations>

$$Ta = A @ An,$$

$$Tb = B @ Bn.$$

<Process>

c An Bn Y, Try D-flip-flops

c Ja Ka Jb Kb Y, Try using JK-ff

c Ta Tb Y. Try using T-FF

The results obtained are not identical to those presented in the text. Specifically, CAFE found a three AND-gate solution when using JK-flip-flops, as opposed to the three AND-gate, one EXNOR gate solution developed there.

3 Input Variables

3 Output Variables

5 Cube Connection Array: \$ = 17

=====

ABX ABY

nn

=====

x10 1--

10x 1--

0x1 -1-

110 -1-

x01 -11

```
Jb = X
```

```
3 Input Variables
4 Output Variables
3 Cube Connection Array: $ = 13
=====
ABX JKKY
  aab
=====
010 1-1-
111 -11-
x01 ---1
```

```
3 Input Variables
3 Output Variables
3 Cube Connection Array: $ = 14
=====
ABX TTY
  ab
=====
010 11-
111 11-
x01 -11
```

4.3 Comparing Functions

The exclusive-or operator may be used to determine if two Boolean variables (functions) are identical, or not. On a recent examination a student claimed that the function x given by the following array could be realized in multiple-level form by the equation y .

```
<array>
  7 a b c d e f g
  x = 010111x
      0110101
      0110110
      1001101
      1001110
      101010x.
```

```
<equations>
  y = e*(a@b)*(c@d)*(f@g).
```

```
<Xterms>
  2 x y.
  Dif = 0 3.
```

```
<Process>
  c x y Dif. Lets see if they are identical
  k exam x. See what the Kim synthesizer can do
```

The exclusive-or of the two functions is expressed in maxterm form just to illustrate that type of specification.

The results below quickly reveal that the student was certainly close to a factorization of the function. On the other hand, feeding the results below directly to the Kim synthesizer gave an answer that was better than any student's (or the professor's) answer because we didn't consider factoring \bar{x} . The last array is the OFF-array of x , i.e., the ON-array of \bar{x} .

Connection Arrays from Equations May 21 1997 21:11:42 Page 1
 File: Kim Run Options: e

7 Input Variables
 3 Output Variables
 10 Cube Connection Array: \$ = 90

```

=====
abcdefg xyD
          i
          f
=====
0101110 11-
0110101 11-
0110110 11-
1001101 11-
1001110 11-
1010101 11-
0101111 1-1
1010100 1-1
0101101 -11
1010110 -11
  
```

```

. exam 7 1
x
010111x
0110101
0110110
1001101
1001110
101010x
%
xxxx0xx
00xxxxx
0x00xxx
0x0xx0x
0x11xxx
0x1xx00
0x1xx11
11xxxxx
1x00xxx
1x0xx00
1x0xx11
1x11xxx
1x1xx1x
%
%
```

Comments and corrections
 may be sent to:
 dld@engr.wisc.edu