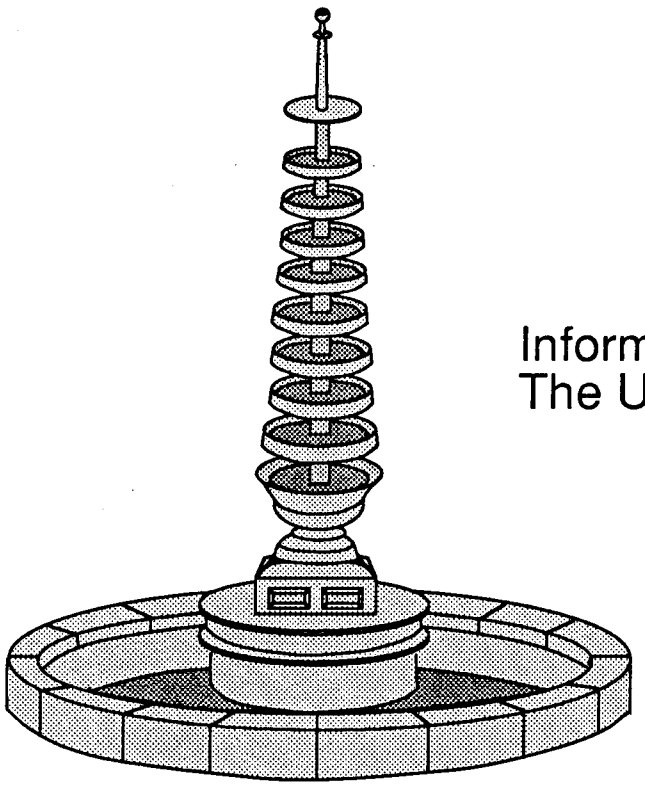
TECHNICAL REPORT

TRIE-91-7

Application of Boolean Unification to Combinational Logic Synthesis

Yuji Kukimoto and Masahiro Fujita



Information Engineering
The University of Tokyo

TRIE-91-7

Application of Boolean Unification to Combinational Logic Synthesis

Yuji Kukimoto and Masahiro Fujita

Information Engineering
The University of Tokyo

Application of Boolean Unification to Combinational Logic Synthesis *

Yuji Kukimoto
University of Tokyo

Masahiro Fujita
FUJITSU LABORATORIES LTD.

October 1, 1991

Abstract

Boolean unification is an algorithm to obtain the general solution of a given Boolean equation. Since the general solution provides a way to represent complete don't care sets in a functional form, Boolean unification can be a powerful method when applied to logic synthesis. In this paper we present various applications of Boolean unification to combinational logic synthesis. Three topics of combinational logic synthesis: redesign, multi-level logic minimization and minimization of Boolean relations are discussed. All these problems can be uniformly formalized as Boolean unification problems. Experimental results are also reported.

*To appear in Proceedings of IEEE International Conference on Computer-Aided Design, Santa Clara, November 1991

1 Introduction

Boolean Unification[9, 8] is a procedure to obtain the general solution of a given Boolean equation or formula. In the field of CAD for integrated circuit design, it has been applied to logic verification and test pattern generation [11, 4] combined with logic programming.

In this paper we present various applications of Boolean unification to combinational logic synthesis. Three topics: redesign [5], multi-level logic minimization and minimization of Boolean relations [12] are discussed. All these problems can be uniformly formalized as Boolean unification problems. The general solutions of these problems can be obtained by using Boolean unification algorithms [9, 8]. These solutions express complete don't cares, which enable us to explore larger design space.

In section 2, we briefly review a Boolean unification problem and its algorithm and show that the algorithm can be easily implemented by Binary Decision Diagrams [3]. In section 3, we present three applications of Boolean unification to combinational logic synthesis and show how to formalize them in a unified framework. A method to synthesize circuits from the general solutions of Boolean unification is given in section 4. In section 5, an implementation of the unification algorithm and experimental results are discussed. Section 6 gives concluding remarks.

2 Boolean Unification

2.1 Problem Formulation

In this section, we present a Boolean unification problem and its algorithm. Detailed explanations can be found in [9, 8]. In this manuscript, the complement of a formula is expressed with $'$ (prime), i.e. p' represents the complement of p. $\oplus$ denotes an exclusive-or operation.

Boolean unification is a problem to find the most general unifier (or substitution) for $p_1, p_2, \ldots, p_n$ which satisfies the following equation no matter what functions or variables $q_1, q_2, \ldots, q_m$ are.

$$f(p_1, p_2, \ldots, p_n, q_1, q_2, \ldots, q_m) = 0 \tag{1}$$

In general, a most general unifier can be represented by functions of $q_1, q_2, \ldots, q_m$, and some newly introduced variables, $r_1, r_2, ..., r_n$, i.e.,

$$\begin{aligned} p_1 &= mgu_1(q_1, q_2, ..., q_m, r_1, r_2, ..., r_n) \\ &\vdots \\ p_n &= mgu_n(q_1, q_2, ..., q_m, r_1, r_2, ..., r_n) \end{aligned} \tag{2}$$

Variables r_i's can be arbitrary functions of $q_1, q_2, \ldots, q_m$ or simply constants. In other words, Boolean unification translates a relation expressed in equation(1) into

```
unify(f(p)){
  if (p = ()) then {
    if (f(p) = 0) then return () else fail
  }
  else {
    G(y) = unify(f(0,y) · f(1,y))
    return(
      ((f(0,G(y)) ⊕ f(1,G(y)) ⊕ 1) · r1 ⊕ f(0,G(y)),G(y))
    )
  }
}
```

Note: $\mathbf{p} = (p_1, p_2, \ldots, p_n), \mathbf{y} = (p_2, \ldots, p_n)$

Figure 1: Boole's unification algorithm for $f(\mathbf{p}) = 0$

the corresponding functional description in (2) by introducing new variables r_i's for representing don't cares.

For example, suppose we unify the following formula.

$$f(p_1, p_2, q_1, q_2) = p_1' \oplus p_2' \oplus p_1'p_2' \oplus q_1 \oplus q_2 = 0 \tag{3}$$

Most general unifiers for p_1 and p_2 are:

$$\begin{aligned} p1 &= q_1r_1r_2' \oplus q_2r_1r_2' \oplus q_1 \oplus q_2 \oplus 1 \\ p2 &= q_1r_2 \oplus q_2r_2 \oplus q_1 \oplus q_2 \oplus 1 \end{aligned} \tag{4}$$

The equation (3) is always satisfied no matter what functions r_1 and r_2 are in (4). (The unification procedure is discussed in the following section). To get a particular solution, we have only to assign some functions of q_1, q_2 (or constant) to r_1 and r_2. If we assign q_1 to r_1 and q_2 to r_2, we get:

$$\begin{aligned} p_1 &= q_1q_1q_2' \oplus q_2q_1q_2' \oplus q_1 \oplus q_2 \oplus 1 = q_1 + q_2' \\ p_2 &= q_1q_2 \oplus q_2q_2 \oplus q_1 \oplus q_2 \oplus 1 = q_1' + q_2 \end{aligned}$$

2.2 Boolean Unification Algorithm

There are two major Boolean unification algorithms: Boole's method and Löwenheim's method [9, 8]. Here we introduce only Boole's method because of space limitation.

Boole's method is based on Shannon's expansion of a given formula, and recursively applies the unification procedure to its sub-formulas. The unification procedure is shown in figure 1. It first checks whether there remains a variable to be unified in the formula. If there is no such variable, the formula is checked to be 0. If it is 0, the unification succeeds and returns, otherwise it fails, i.e. there is no solution for the

given formula. If there are remaining variables to be unified, one variable is selected from these as a splitting variable (in figure 1, the first variable, p_1, is selected) and the unification is applied recursively to a new formula $f(0,\mathbf{y}) \cdot f(1,\mathbf{y})$. The most general unifier for the splitting variable can be obtained as follows, where $G(\mathbf{y})$ is the result of the unification of the sub-formula $f(0,\mathbf{y}) \cdot f(1,\mathbf{y})$.

$$p_1 = (f(0,G(\mathbf{y})) \oplus f(1,G(\mathbf{y})) \oplus 1) \cdot r_1 \oplus f(0,G(\mathbf{y}))$$

Since each operation in figure 1 is a logic operation, we can easily implement the unification procedure using BDD's.

3 Application of Boolean Unification

3.1 Redesign

A redesign problem [5] arises when we encounter a slight change of specification. In this situation we should use an existing circuit as a part of a new circuit. To attach an extra circuitry to the original circuit is promising in this context. Here we show how to formalize this redesign problem with Boolean unification. We assume that an existing circuit has a physical design, where only external inputs and outputs of the circuit can be connected from the outside. We also assume that some internal nets connected to the boundary of the layout can be reached from the outside of the circuit. To compensate for the difference between the existing design and the new specification, we attach an extra circuit to these easily accessible terminals or nets, as shown in figure 2.

Now we explain formalization methods. In the following, let f be the logic function realized by an existing circuit, i.e., $o = f(i)$, (i, o, and t are inputs, outputs, and intermediate variables respectively. These can be vectors of variables) and s be the logic function for the specification: the one which must be realized by the redesigned circuit.

When we attach an extra circuit to the input part of the existing circuit, as shown in figure 2(a), the problem to be solved is:

$$o = f(t,i) = s(i) \tag{5}$$

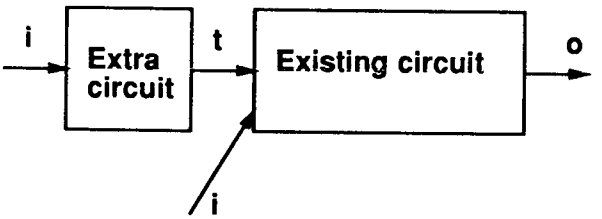
(5) can be transformed to the normal form of (1),

$$f(t,i) \oplus s(i) = 0$$

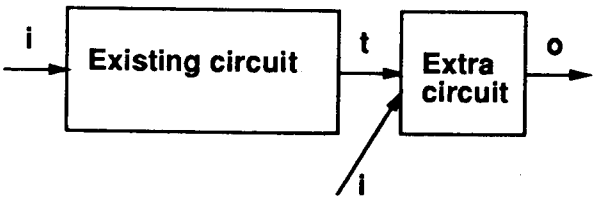
where t and i correspond to p_i's and q_i's in (1) respectively. As a result of Boolean unification, t expresses requirements for the extra circuit in the most general form.

In the case where an extra circuit is attached to the output part of the existing circuit, as shown in figure 2(b), we have to find a logic function o such that:

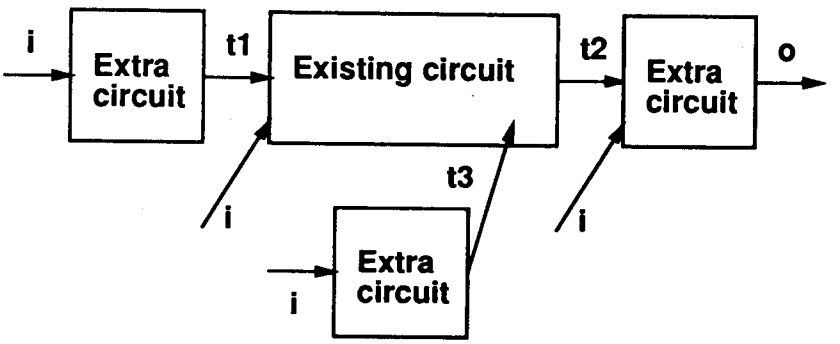
$$(o = s(i))(t = f(i)) = 1$$



(a) Add an extra circuit to an input part



(b) Add an extra circuit to an output part



(c) Combination of various types of redesign

Figure 2: Redesign after layout

When we can access some internal nets, a similar formalization is possible. Clearly the combination of these redesign methods(cf. figure 2(c)) can be formalized in the same way, which enables us to solve any types of redesign problems.

When we get a general solution for the extra circuit, the remaining thing to do is to synthesize the circuit from it. This is discussed in section 4.

We can effectively apply the proposed redesign techniques to logic synthesis where a part of the internal structure of a circuit to be synthesized is determined in advance by a designer. For example, it is well known that one can design a compact adder using a carry-chain composed of pass transistors. Figure 3 shows a one-bit slice adder, where q_1, q_2, and q_3 are inputs; s_1 and s_2 are a carry output and an internal signal; p_1 and p_2 are the outputs of circuit1 and 2 respectively. In this case, a designer has already determined to use this structure and the remaining task is to design the circuit1 and 2. We can apply the redesign method to this problem in the following manner. First we specify the circuit to be designed in its logic operation

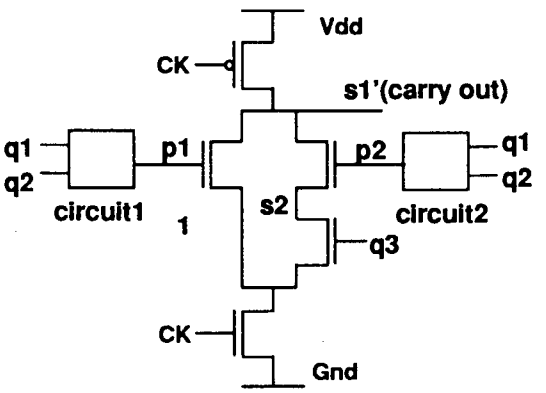


Figure 3: Carry-chain adder with pass transistors

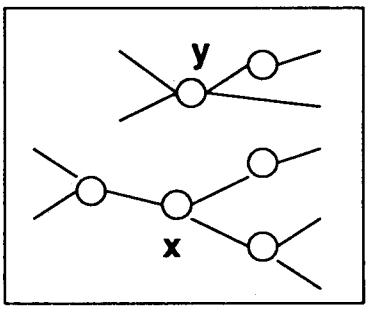


Figure 4: Multi-level Logic Minimization

phase(CK=1) by its characteristic function.

$$(s_1 = q_1q_2q_3 + q_1'q_2q_3 + q_1q_2'q_3 + q_1q_2q_3')(p_1 \rightarrow s_1' = 0)$$
$$(p_2 \rightarrow s_1' = s_2)(q_3 \rightarrow s_2 = 0) = 1 \qquad (6)$$

where p_1 and p_2 are variables to be unified. After smoothing out the variables s_1 and s_2 from (6), we can apply the unification procedure to the formula and get the general solutions for p_1 and p_2 in terms of q_1, q_2, and q_3. Using a synthesis method presented in section 4, we can synthesize circuit1 and 2 from these general solutions: in this case an AND gate for the circuit1 and an EXOR gate for the circuit2.

3.2 Multi-level Logic Minimization

A general solution of a given equation obtained by Boolean unification can represent don't care sets in much broader sense than those proposed so far [10, 2, 1]. Don't care sets used in logic synthesis methods in [10, 2, 1] are basically defined for a single net or gate, and only one logic function corresponding to a net can be modified at the same time during the minimization process. (CSPF [10] is an exception,

but it is a very limited don't care set). For example, MSPF [10] is defined for each net on condition that the logic functions of all other nets do not change.

However, we can modify circuit structures much more dramatically if we use don't care sets derived from the condition that the logic functions of more than one net or gate are changed simultaneously, which leads us to a better minimization result. For example, as shown in figure 4, if we allow the logic functions of x and y to change simultaneously on condition that the logic functions of all other nodes do not change, the derived don't care sets for x and y capture higher degrees of freedom for optimizing a given network.

Boolean unification provides a way to handle these don't care sets uniformly. In the case of figure 4, we first represent the output logic function of the circuit in terms of inputs and also x and y. (s is a logic function for specification):

$$o = f(x, y, i) = s(i)$$

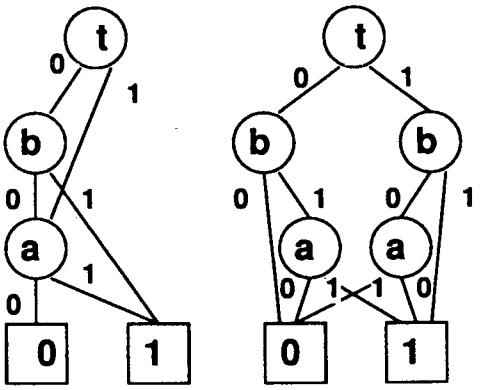
By corresponding x, y and i to p_1, p_2 and q_1 in equation (1) respectively, we can apply Boolean unification to obtain the general solutions for x and y. The general solutions for x and y are logic functions in terms of i and newly introduced variables r_1 and r_2. These variables r_1 and r_2 express complete don't care sets for x and y and the relationship between the don't care set of x and that of y. Based on this don't care set, we can transform the circuit structure on condition that the logic functions for x and y after the transformation remain in the general solutions of Boolean unification. Note that if only one node x is considered instead of two nodes, the general solution expresses the same don't care set as MSPF for the net x.

3.3 Boolean Relation

Boolean relations are the generalizations of incompletely specified functions and arise in many applications [12]. There are several approaches to the minimization of Boolean relations in sum-of-product forms [12, 6, 13], but no methods for multi-level minimization of Boolean relations have been published so far. Here we present how to apply Boolean unification to the multi-level minimization of Boolean relations. Since a Boolean relation is a relation between inputs and outputs, we can represent it with its characteristic function:

$$f(o_1, o_2, ..., o_n, i_1, i_2, ..., i_m) = 1 \tag{7}$$

In (7) o_i and i_i correspond to p_i and q_i in (1) respectively. If we apply Boolean unification to this formula, we can get the general solution for primary outputs o_i's as functions of both inputs and newly introduced variables. This solution completely expresses the don't care set of the original Boolean relation. Since we can represent compatible outputs as functional forms not as relational, it is easy to apply multi-level logic minimization methods proposed so far.



(a) BDD for x (b)BDD for y

Figure 5: BDD for equation (8)

4 Logic Synthesis from the Results of Boolean Unification

In this section, we consider logic synthesis from the results of Boolean unification. Suppose we get the following general solution after Boolean unification. (a and b are inputs, x and y are outputs, and t is a newly introduced variable):

$$\begin{aligned} x &= a + bt' \\ y &= ab + a't \end{aligned} \tag{8}$$

The remaining task is to generate a simple circuit from the general solution of (8). We take the following two steps:

1. Generate an initial circuit by appropriately assigning constants to newly introduced variables.
2. Minimize the initial circuit.

Let us apply the above method to (8). Since we implement the unification procedure by BDD's, the first step uses the number of nodes in BDD's as a measure for the complexity of logic functions. Figure 5 shows BDD's for (8). We order newly introduced variables first so that we can see how large the number of nodes below those variables is. By counting the number of nodes below the nodes of the variable t, we can get the complexity of the logic functions when we assign constant values to t. In this case, if we assign 0 to t, the total number of nodes below t is 8, and if we assign 1 to t, it is 7. Thus the latter substitution is expected to give a more efficient result.

$$\begin{aligned} x &= a \\ y &= a' + b \end{aligned}$$

	literals	CPU time(sec)
rd53-73	5	0.99

Table 1: Redesign

	GYOCRO + MIS	BU + MIS	
	literals(fac/sop)	literals	CPU time(sec)
int3	13/19	12/15	0.03
int9	26/30	23/27	0.26
int15	437/588	447/560	700.5
gr	270/300	371/447	113.7
b9	162/284	163/278	5.69
vtx	130/2780	67/85	12.66

Table 2: Minimization of Boolean Relations

However, this is not the simplest solution, i.e., if we assign b to t, we get:

$$x = a$$
$$y = b$$

This shows that the quality of the initial circuit is restricted if the substitutions to newly introduced variables are limited to constants. To exploit don't care sets of Boolean unification completely, minimization of the initial circuit is essential. An extended transduction method is a promising procedure for this purpose, where the general solutions of Boolean unification are used as permissible functions [10]. The details can be found in a forthcoming paper.

5 Implementation and Results

We have implemented Boole's method using shared BDD's with negative edges [7] on Sparc station 2. As the variable ordering of BDD's, we select variables to be unified(p_i) first followed by non-unified variables(q_i). A case splitting variable in figure 1 is selected in the same order as the variable ordering of BDD's. The synthesis method shown in the previous section has been also implemented using BDD's.

Experimental results for redesign are summarized in table 1. We assume that we have a circuit `rd73` as an existing circuit, which is rectified to function as `rd53` by attaching an extra circuit to the input part of the circuit.

Table 2 shows experimental results on multi-level minimization of Boolean relations. An initial circuit is obtained from a general solution by substituting appropriate single literals or constants to newly introduced variables and minimized with

MIS2.2 standard script. These are compared with the minimization results of heuristic minimizer GYOCRO [13]. The two-level minimization results of GYOCRO are processed with MIS2.2 standard script to get multi-level circuits.

The CPU time on these two tables consists of the processing time for Boolean unification and that for selecting good substitutions to new variables. Most of the time is spent on the latter computation.

6 Conclusion

We have shown that Boolean unification gives powerful methods for various problems in combinational logic synthesis, and also presented some experimental results using BDD's. We are now working on constructing a more general logic synthesis method from the results of Boolean unification, and on the application of Boolean unification to sequential logic synthesis.

Acknowledgment

The authors are grateful to Mr. Y. Watanabe of UC Berkeley for providing us Boolean relation benchmarks and his minimization results.

References

[1] D. Bostick, G. D. Hachtel, R. Jacoby, M. R. Lightner, P. Moceyunas, C. R. Morrison, and D. Ravenscroft. Boulder optimal logic design system. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 62–65, November 1987.

[2] R. K. Brayton, R. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level interactive logic optimization system. *IEEE Transactions on Computer-Aided Design*, 6(6):1062–1081, November 1987.

[3] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computer*, C-35(8):677–691, August 1986.

[4] W. Buttner and H. Simonis. Embedding boolean expressions into logic programming. *Journal of Symbolic Computation*, 4:191–205, 1987.

[5] M. Fujita, T. Kakuda, and Y. Matsunaga. Redesign and automatic error correction of combinational circuits. In *Proceedings of IFIP Working Conference on Logic and Architectural Synthesis*, pages –, May 1990.

[6] A. Ghosh, S. Devadas, and A. R. Newton. Heuristic minimization of boolean relations using testing techniques. In *Proceedings of IEEE International Conference on Computer Design*, pages 277–281, September 1990.

[7] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagrams with attributed edges for efficient boolean function manipulation. In

Proceedings of 27th ACM/IEEE Design Automation Conference, pages 52–57, June 1990.

[8] Ursula Martin and Tobias Nipkow. Unification in boolean rings. *Journal of Automated Reasoning*, 4:381–396, 1988.

[9] Ursula Martin and Tobias Nipkow. Boolean unification — the story so far. *Journal of Symbolic Computation*, 7:275–293, 1989.

[10] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The transduction method – design of logic network based on permissible functions. *IEEE Transactions on Computer*, 38(10):1404–1424, October 1989.

[11] H. Simonis. Test generation using the constraint logic programming language CHIP. In *Proceedings of 6th International Conference on Logic Programming*, pages –, June 1989.

[12] F. Somenzi and R. K. Brayton. An exact minimizer for boolean relations. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 316–319, November 1989.

[13] Y. Watanabe and R. K. Brayton. Heuristic minimization of boolean relations. In *Proceedings of MCNC International Workshop on Logic Synthesis*, May 1991.

Information Engineering Course
Graduates School of Engineering
The University of Tokyo
7-3-1, Hongo, Bunkyo-ku, Tokyo 113
JAPAN

the cover designed by Jun-ichi Tatemura, march 1990