設 計 自 動 化 52－3
（1 9 9 0. 5. 2 2）

大規模回路の多段論理簡単化について

藤田昌宏　　松永裕介　　角田多苗子

富士通研究所　人工知能研究部

我々は許容関数の内部表現に２分決定グラフを用いた多段論理回路簡単化プログラムを開発し、ベンチマーク回路による評価を行なってきた。２分決定グラフは他の論理関数表現手法と比べ、より大きな回路がコンパクトに表現できるため、他の多段回路簡単化手法と比較してより大きな回路を簡単化できる。本稿では、さらに大規模回路の効率的な簡単化を実現するために、簡単化の際に使用するドントケアの効率的な絞り込み手法を提案し、実際の評価結果を述べる。元の許容関数では、回路の外部出力の論理関数が変化しないという条件を用いていたが、ここで提案する手法では、各ゲートごとに許容関数を定義する際に、そのゲートの出力の論理関数が変化しないという条件で許容関数を生成するようにする。これは、回路の構造から生じるドントケアを一部捨てることに相当し、結果として許容関数が簡単になり、大規模回路に適用できるようになる。また、実験結果から、簡単化の品質もそれほど悪くならないことが分かった。

Multi-level logic minimization for large circuits

Masahiro Fujita, Yusuke Matsunaga, and Taeko Kakuda

Artificial Intelligence Labs. FUJITSU LABORATORIES LTD.

1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan

We have developed multi-level logic minimization programs based on the transduction method with Binary Decision Diagram and also developed a Boolean resubstitution algorithm with permissible functions. In this paper, we present a method which enable us to minimize larger circuits. We calculate permissible functions for each gate on the condition that only the output of that gate does not change, which is a more strict condition than that of the original permissible functions which guarantees that the primary output logics do not change. The sizes of permissible functions are smaller, which means we can handle larger circuits, although some don't cares are ignored. We present some experimental results.

1. Introduction

In logic synthesis, multi-level logic minimization is very important to get high quality synthesized circuits in terms of area and testability, and several methods have been proposed and developed [2,3,4,5,6,7]. In all of them, the key point of multi-level minimization is the use of don't care sets. MIS [2] uses satisfiability don't cares (filtered for speed-up [3]), and BOLD uses full don't cares by checking the logical equivalence between the original circuit and the modified circuit. Another method to utilize don't cares effectively is to use permissible functions used in the transduction method [5].

We have developed a multi-level logic minimization program based on the transduction method [6] with Binary Decision Diagram [8] and also developed a Boolean resubstitution algorithm with permissible functions [7]. Permissible functions are defined on each gate and express don't care sets which do not change the primary output logics. These programs show equal or superior performance compared with other multi-level logic minimization programs, such as MIS and BOLD, especially for large circuits.

In this paper, we present a method which enables us to minimize larger circuits. It is a kind of filtering methods which are also used in MIS [3]. We calculate permissible functions for each gate on the condition that only the output of that gate does not change. It is a more strict condition than that of the original permissible functions which guarantees that the logics of the primary outputs do not change, and so the sizes of permissible functions become smaller (when expressed in BDDs), which means we can handle larger circuits. Since some don't cares are ignored in this method, minimization results may be worse. However, the don't care sets obtained by this method still gives equal or larger don't care sets than MIS's filtering [3]. Also, as seen from the experimental results presented in the later section, the results are not so bad compared with other minimization methods.

In section 2, we briefly review permissible functions expressed in BDDs and the Boolean substitution algorithm with it. In section 3, we present the new method with some exterimental results, and section 4 is a concluding remark.

2. Boolean resubstitution with permissible functions and BDD

2.1 Permissible Functions

The key concept of permissible functions is that each node in a circuit is an incompletely-specified function of the primary inputs due to the don't care sets obtained from network topologies, and permissible functions represent possible implementations at such nodes [5]. Permissible functions are sets of logic functions that do not change any output logic, and they are defined as follows. Assume Vi is an intermediate node in a network. The logic function of any output variable in the network may not change even when the logic function Fi of node Vi is replaced with another logic function PF. Then the logic function PF is called a permissible function of node Vi.

Permissible functions are defined for each node. Usually, there is more than one permissible function for a node. Therefore, the don't care mark (*) is used to represent a set of permissible functions as a single vector at that node. Figure 1 shows an example of permissible functions. In this figure, v1 and v2 are input nodes, v4 is an output node, and v3 is intermediate node. The Fi vector in the truth table represents a logic function of each node vi. G4 is an OR gate. Since the first and second values of F4 are 0s, the first and second values of F3 must remain to be 0s. The third and fourth values of F4 are 1s and the third and fourth values of F1 are 1s. Thus, the third and fourth values of F3 may be either 0 or 1, and the logic function of F4 does not change even when logic function F3 is replaced with PF3. Then PF3 is the set of permissible functions of v3. Though the logic functions and permissible functions in this figure are represented in terms of truth tables, in our implementation these are represented in BDD.

Permissible functions can be calculated by traversing networks from outputs to inputs. The details can be found in [5,6].

2.2 Binary Decision Diagrams

Binary Decision Diagrams (BDD or sometimes called Ordered BDD) were proposed by Bryant [8]. A BDD is a kind of decision graph for representing Boolean functions with restrictions on the ordering of variables in the graph. Boolean functions are represented by directed, acyclic graphs with a vertex set containing two types of vertices. A non-terminal vertex has as attributes an input variable index and two children. A terminal vertex has as attributes a constant value 0 or 1 (to express permissible functions, we added one more constant '*' to express don't care value). Ordered means that if xi<xj then all nodes with xi precede all nodes with xj. A path from the root to the terminal vertex with value 0 (or 1) gives a condition when Boolean function f=0 (or f=1).

Figure 2 shows an example of BDD representation of a Boolean function F=v1&v2 + v3. In this figure, a rectangle indicates a terminal node with logical values, and a circle indicates a non-terminal node containing the variable index with the two children indicated by branches labeled 0 and 1. The variable ordering of this graph is v1 > v2 > v3.

A graph can be shared with many logic functions and permissible functions, and the negative edge can be used to indicate an inverted logic [11]. This improvement enables the graph to be copied only by operating the pointer. The effective use of graph sharing and negative edges reduces CPU-time and memories [7]. Figure 3 shows an example of shared BDD with negative edges.

2.3 Boolean resubstitution with permissible functions

We have implemented Boolean resubstitution based on the Bold algorithm [4] with permissible functions [7]. Boolean resubstitution generalizes the Reduction, Expand and Irredundant operations [10] of two-level logic minimizer to a multi-level context. This process removes the redundancies from the network, as well as modifies the network configuration. Thus, the possibilities of further optimization are increased.

Figure 4 shows the general flow of our Boolean resubstitution. Even though we are manipulating Boolean networks, each cover comprises connections of AND gates and one OR gate. We use the following conditions to execute operations shown in Figure 4:

(1) Pruning condition: The connection whose set of permissible functions consists of 1 and don't care only or 0 and don't care only can be set to constant 1 or 0. Then this connection may be removed from the network.

(2) Connectable condition: A connection can be added to a gate as a new fan-in when the new logic function of that gate's output is included in its permissible functions.

The Boolean resubstitution procedure shown in Figure 4 was implemented by the above operations using BDDs [7]. In this paper, this implementation is used to evaluate the two methods which we propose in this paper to increase the power of our minimization method.

3. Minimization by subsets of permissible functions

This section present a minimization method using a subset of permissible functions. The original permissible functions (we call it PF here) are defined on each node of Boolean networks so that it is the set of logic functions which do not change the primary output logics, even if the logic of that node is replaced by a logic function within the permissible functions. The subset of permissible functions we use here (we call it SPF) is the set of logic functions which do not change the logic of that node. This is a more strict condition, and so gives smaller don't care sets, which means it can be expressed in smaller BDDs. Let us explain by Figure 5.

Here we assume initial circuits for multi-level minimization are obtained by the weak-division algorithm [12], and so each internal node is expressed as sum-of-product of fan-ins of that node. Let us concentrate on the minimization of a node Vi, an internal node of the circuit. PFs guarantee that logic of primary outputs do not change, while SPF for node Vi guarantees that the logic of the node Vi does not change. This means SPF has no don't cares relating to the sub-circuit which is transitive fan-outs of the node Vi (the area which is lined horizontally in Figure 5. This is sometimes called as observability don't cares), but SPF still considers the don't cares relating to the sub-circuit which is transitive fan-ins of the node Vi (the area which is lined vertically in Figure 5. This is sometimes called as satisfiability don't cares). PF considers don't cares relating to both areas. Hence, SPF is smaller than PF, which means the size of BDD for SPF is smaller than the size of BDD for PF..

The above discussion can be understood in a different way by considering how to calculate permissible functions. We can get permissible functions for the fan-ins of a node Vi from the permissible functions for the output of Vi and the logic functions for the fan-ins of Vi [5]. This means that permissible functions can be calculated by traversing a circuit from outputs to inputs. This is the way to calculate PFs. However, when calculating SPFs for a node Vi, we start from Vi, not the primary outputs, and assume the SPF for the output of Vi is the logic function of Vi, which means there is not don't care values in SPF for the output of Vi. This is the contrast to the fact that PF for the output of Vi may have many don't care values. We calculate SPFs for the transitive fan-ins of Vi in the same way as PFs. So, SPFs is generally smaller than PFs in the sense that there are fewer don't care values.

Of course, since SPF is smaller than PF, minimization results by SPFs may be worse than that by PFs. But sizes of BDDs which are used to represent permissible functions determines the speed of minimization and how large circuits can be minimized. So, by using SPFs, we can expect not only faster execution times but also can minimize larger circuits than those by PFs, although the minimization results may be worse.

We did some experiments using MCNC benchmark circuits. The results are shown in Table 1 and Table 2. 'unable' in the tables means the required memory space exceeds 30M Bytes, which means BDDs are too large. Table 1 shows the results of multi-level examples and Table 2 shows the results of two-level examples. The initial circuits for the two-level examples are generated by the weak-division algorithm [12], except alu4. alu4 is synthesized by the combination of weak-division and Boolean resubstitution in order to speed up synthesis time, i.e., we first weakly divides alu4 only when the saved literals exceed 10, then apply the Boolean resubstitution procedure, then weakly divides it only when the saved literals exceed 5, then then apply the Boolean resubstitution procedure, then weakly divides it with no restriction, and finally apply the Boolean resubstitution procedure one more time. This is because the results after simply applying the weak-division algorithm has many redundancy, so it takes much time to minimize (although it is possible). So we should execute the Boolean resubstitution procedure during the weak-division process.

In both tables, we use the variable orderings obtained from the heuristics in [1]. As for Table 1, larger ISACAS benchmark circuits than c432 (and also the circuit rot) could not be minimized by PF nor SPF, i.e., they need variable ordering optimization presented in the next section. We can see from Table 1 that we get 40 - 65% CPU time reductions, although saved literals are increased by 10 - 50 %. However, the reduction ratios of CPU time is larger than the reduction ratios of saved literals.

As for the results of two-level examples, the reduction ratios of CPU time are not necessarily larger than the reduction ratios of saved literals. However, if an initial circuit has much redundancy, e.g., alu4, minimization by SPF can eliminate large amount of redundancy. This shows that minimization using SPF followed by minimization using PF is a very good strategy, if initial circuits have much redundancy.

4. Conclusion

In this paper a new method for minimizing large circuits is presented. The experimental results show the method enables us to minimize larger circuits with comparable quality than other methods.

BDD (we use as the internal representation of permissible functions) heavily depends on variable orderings, and the sizes of BDDs determine the performance of the minimization programs. Although we have already developed an effective variable ordering algorithm [1], we believe there is not a little space to improve it, which will be one of the future works.

References

[1] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluations and Improvements of a Boolean Comparison Method Based on Binary Decision Diagrams", *IEEE International Conference on Computer Aided Design '88*, Santa Clara, November, 1988.

[2] K.A. Bartlett, R.K. Brayton, G.D. Hachtel, R.M. Jacoby, C.R. Morrison R. Rudell, A.L.Sangiovanni-Vincentelli and A. Wang, "Multi-Level Logic Minimization Using Implicit Don't Cares", *IEEE Trans. Computer-Aided Design*, Vol. CAD-7, No. 6, pp.723-740 June 1988.

[3] A. Saldanha, A.R. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Multi-Level Logic Simplification using Don't Cares and Filters", Proc. 25th DAC, June 1989.

[4] D. Bostick, G.D. Hachtel, R.M. Jacoby, M.R. Lightner, P. Moceyunas, C.R. Morrison, and D. Ravenscrofit, "The boulder optimal logic design system", Proc. ICCAD '87, November 1987.

[5] S. Muroga, Y. Kambayashi, H.C. Lai and J.N. Culliney, "The Transduction Method - Design of Logic Networks based on Permissible Functions", *IEEE Trans. Comput., Vol.C-38, No.10, pp.1404-1424*, October 1989.

[6] Y. Matsunaga and M. Fujita, "Multi-level Logic Optimization Using Binary Decision Diagrams", *IEEE International Conference on Computer Aided Design '89*, Santa Clara, November 1989.

[7] H. Sato, Y. Yasue, Y. Matsunaga, and M. Fujita, "Boolean resubstitution with permissible functions and Binary Decision Diagrams", *To appear in 27th ACM/IEEE Design Automation Conference*, June 1990.

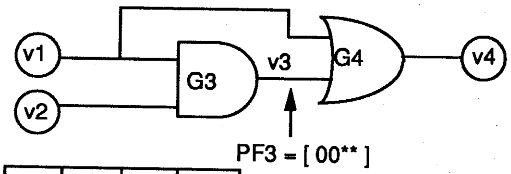
[8] R.E. Bryant, "Graph-based algorithms for boolean function manipulation", *IEEE Trans. Computer*, C-35(8):667-691, August 1986.

[9] F. Beglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran", Special session on ATPG and fault simulation, *IEEE International Symposium on Circuit and Systems '85*, June 1985.

[10] R.K. Brayton, G.D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli,"Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishers, 1984.

[11] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima, "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation", *To appear in 27th ACM/IEEE Design Automation Conference*, June 1990.

[12] R.K.Brayton, R.Rudell, A.Sangiovanni-Vincentelli and A.R.Wang, "MIS: Multi-level interactive logic Optimization system", *IEEE Trans. Computer-Aided Design*, Vol. CAD-6, No. 6, pp.1062-1081, November 1987.



F1	F2	F3	F4
0	0	0	0
0	1	~~0~~	0
1	0	0	1
1	1	1	1

PF3 is a set of permissible functions

Figure 1. An example of permissible functions

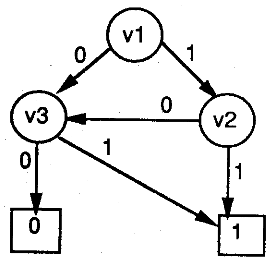
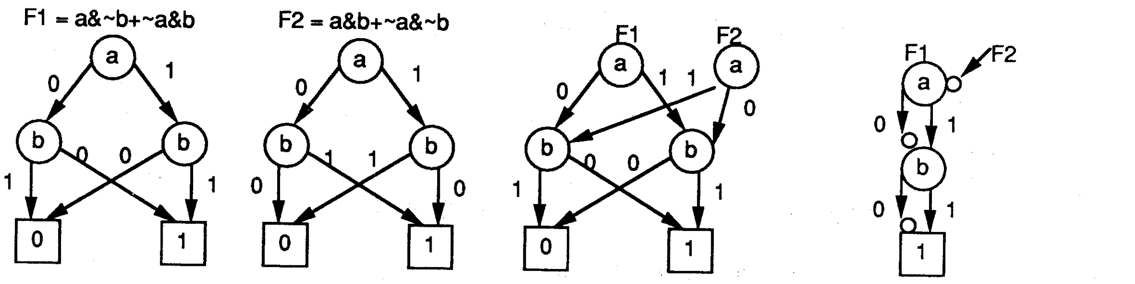


Figure 2. OBDD Representation of F=v1&v2 + v3



(a) BDD (b) Shared BDD (c) Shared BDD with negative edges

Figure 3. Shared BDD with negative edges

```
Boolean_resubstitution()
  /* CV1 and CV2: convers in the Boolean network  */
  /* G          : AND-gate in a cover CV2         */
{
  for each conver CV1 {
    for each cover CV2 {
      try to connect CV1 to G;               /* Reduction   */
      try to remove other fan-ins from G;    /* Expand      */
    }
    try to remove gates in CV2;              /* Irredundant */
  }
}
```

Figure 4. Boolean resubstitution procedure

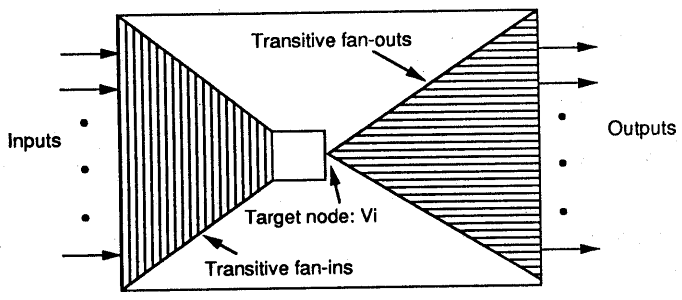


Figure 5. Don't cares drived from circuit topology

Circuits		Boolean resubstitution by PF/SPF				Saved literal reduction	CPU time reduction
		PF		SPF		((1)-(4))/((1)-(2))	(5)/(3)
Name	(1)Literals	(2)Literals	(3)CPU time	(4)Literals	(5)CPU time		
apex6	832	754	446.6	791	194.7	0.53	0.44
apex7	352	279	24.6	291	8.7	0.71	0.35
rot	1443	unable	-	unable	-	-	-
c432	272	194	71.3	204	43.2	0.87	0.61

Machine: SUN4/260
CPU time: seconds
Variable orderings are generated by the algorithm in [1], no additional optimization

Table 1. Results of Boolean resubstitution by PF/SPF for MCNC multi-level benchmark circuits

Circuits		Boolean resubstitution by PF/SPF				Saved literal reduction	CPU time reduction
		PF		SPF		((1)-(4))/((1)-(2))	(5)/(3)
Name	(1)Literals	(2)Literals	(3)CPU time	(4)Literals	(5)CPU time		
5xpl	152	103	3.2	122	3.0	0.61	0.93
9sym	234	223	20.8	231	7.1	0.36	0.34
alu4	2058	131	697.6	145	405.7	0.99	0.58
duke2	384	362	56.6	365	47.6	0.86	0.84
misex3	1288	679	1498.1	892	745.6	0.65	0.50
misez3c	522	443	117.1	492	76.8	0.38	0.66
rd84	256	147	11.4	190	8.4	0.61	0.74
seq	1754	1107	1846.0	1491	1179.2	0.41	0.64

Machine: SUN4/260
CPU time: seconds
Variable orderings are generated by the algorithm in [1], no additional optimization

Table 2. Results of Boolean resubstitution by PF/SPF for MCNC two-level benchmark circuits