A graph-based approach for Boolean matching

[1]Anderson Santos da Silva, [2]Vinicius Callegaro, [1,2]Renato Ribas, [1,2]André Reis
{assilva,vcallegaro,rpribas,andreis}@inf.ufrgs.br

[1]Institute of Informatics, UFRGS, Porto Alegre, Brazil
[2]PPGC, UFRGS, Porto Alegre, Brazil

Abstract

This paper proposes a new method for Boolean matching. During the technology mapping, the matching associates Boolean functions with logic cells present in a library. This new approach for Boolean matching is based on graph theory. The method constructs a bipartite graph from a truth table and an identifier is generated to associate the graph to the function. The resulting identifier can be used to match Boolean functions with library cells that can implement them directly.

1. Introduction

Boolean matching is used in synthesis of logic circuits, more specifically in technology mapping task [3]. In such process, a given circuit is divided into several sub-circuits that are implemented with logic cells from a target library, while improving an optimization criterion and keeping the functionality of the original logic function.

Hence, it is quite important the use of mechanisms to decide if a determined sub-circuit is logically equivalent to another. Generally, logic cells are organized in a library that associates their Boolean function to a signature. This signature is generated under input permutation in order to retrieve the key function. One way to provide such key is calculating a signature for function [1][2]. This signature must be an exclusive and irredundant form to represent a specific class of functions.

It is desirable that these methods optimize some aspects in their searching for signatures and in the data structure. In this sense, optimized data structure can be understood as effective manner for representing a specified problem that reduces the computing time and used space (memory) while increasing the scalability [2].

In next sections, a new method for generating the function matching is presented. It is based on graph built directly from truth tables. This graph is intrinsic to a determined function and represents its minterms together with its variables.

2. Background

Truth tables and mintems: Every Boolean function has input values and the combination of these values defines a logic output of this function. To investigate all possible logic outputs of a given function, one way is by enumerating all possible entries. This enumeration of all possible values is named truth table and is usually represented as shown in Fig. 1.

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 1- Illustration of a truth table.

In this case, the inputs are named with the letters A and B, called **variables**. It represents all possible entries for this function, and *F* represents the value of the output when the entries are applied.

Each line can be seen as a binary number, where we have for all lines, in decimal notation, 0, 1, 2 and 3 as possible entries, but only the lines 1 and 2 associate *F* to value 1. The cases that the function *F* is equal to 1 are named **minterms.** For a function with *n* variables, the maximum number of lines in the truth table is 2^n and, consequently, the maximum number of minterms either.

Equivalent classes: Considering a given function, if all possible permutations are done in its variables, a class of functions that are permutation-equivalent to the original is obtained. This class of functions is named P-class. If the variables are complemented or permuted, then we have a NP-class for the

original function. Finally, if the variables are complemented or permuted, and the output of function can be also complemented, we have the NPN-class for the original function.

Signature for function: Of all the functions in a class P, we can choose one function to be a representative of the whole class. For example, we pick the smallest function value (in hexadecimal) to be the representative of the class P.

Graph adjacency matrix: Given a graph G= (V, E) with *n* vertices, where V represents a set of vertices and E represents a set of edges in the graph. The matrix of adjacency is defined as a matrix $n \times n$: B= $[b_{ij}]$, where b_{ij} is 1 if v_i and v_j are adjacent in E, and 0 otherwise.

3. Proposed method

Several methods of finding equivalent classes of function have been proposed in the literature [1][2]. The method proposed by Debnath and Sasao [1] enumerate all truth tables and find the smaller function as signature that represents such class. As it suffers of high memory space consumption, they use a tree structure to accelerate the time of searching and optimize memory consumption. Another method proposed by Hinsberger and Kolla [2] uses properties of functions like symmetry to infer which truth table should be stored in memory, discarding the ones that do not represent a smaller signature for the original function.

A new way to treat this problem is the method proposed in this paper. Truth tables can be viewed as graphs, and a new form to represent a Boolean function is presented herein. If we consider that each value 1 in a column represents an adjacent relation between the minterm in the line and the variable in correspondent column.

Hence, we can discuss such hypothesis:

- every truth table can be represented as a unique graph;
- the lines that do not correspond to a minterm do not participate in the graph.

Therefore, we observe that any value 1 in truth table keeps the same relation with its variable and minterm in a permutation. And the lines that do not correspond to a minterm do not influence in the final result because their values do not belong to the function output. In fact, we are only interested in the information shown in Fig 2, which maintain the minterms.

	A	B	F
1:	0	1	1
2:	1	0	1

Fig. 2- Minterms of the function in Fig. 1.

With this reduced truth table, we can build a graph considering each value 1 as adjacency signals with the variables in the same column. For example, in the truth table shown in Fig.2, the minterm in the line 1 is adjacent to B and the minterm in line 2 is adjacent to A. By viewing the truth table like that, one generates the graph shown in Fig. 3 for 2-inputs exclusive-OR (XOR2).

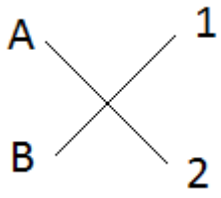

Fig. 3- Graph describing the XOR2.

Two functions are P equivalent if their signatures are equal after an algorithm of searching. As we only permute the variables from the truth table, we do not change the values in column below each variable on that. See illustration in Fig. 4.

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

B	A	F
0	0	0
1	0	1
0	1	1
1	1	0

Fig. 4- Illustration of permuted truth tables.

For a given value 1 in truth table, any permutation in its rows does not change the variable or minterm that the value 1 belongs. As a result, the graph generated is exactly the same, and then two hypotheses are created:

Hypothesis 1: For any input permutation in a certain function, a graph is obtained. All these graphs are isomorphic each other, and this is valid for all functions of the same P-class. And now, the problem of generate *n!* truth tables is reduced to graph-isomorphism complexity.

Hypothesis 2: Given two logic functions, if both are P equivalent, the both generated graph are isomorphic.

Experimental results have shown that such hypothesis seem to be true. However, the formal proof is on going. Currently, the following properties are known:

- never two variables share an edge;
- never two minterms share an edge;
- the degree of minterm is number of bits in 1 that it vector has;
- the degree of variable is number of minterms covered by the variable.

Therefore, the graph generated is a bipartite graph because the variables do not share edges themselves as well occurs with the minterms.

4. Proposed Algorithm

The algorithm that solves the problem of determining if two graphs are isomorphic is presented. The basic idea is generate a list of adjacency for both graphs. If both lists are equivalents, the graphs are isomorphic.

Algorithm Isomorphic Graphs (Function f1, Function f2)

1. Graph g1 = generate_graph_function(f1);
2. Graph g2 = generate_graph_function(f2);
3. List intersectionsMemoryG1;
4. List graph1_Code;
5. For (Vertex v: graph1) {
6. For (Vertex a: graph1) {
   6.1.1. intersectionsMemoryG1.insert (intersections (v.adjacences, a.adjacences) + a.size())
7. }
8. Graph1_code.insert (intersectionsMemoryG1);
9. }
10. Sort (intersectionsMemory);
11. List intersectionsMemoryG2;
12. List graph2_Code;
13. For (Vertex v: graph2) {
14. For (Vertex a: graph2) {
    14.1.1. intersectionsMemoryG2.insert (intersections (v.adjacences, a.adjacences) + a.size())
15. }
16. graph2_code.insert (intersectionsMemoryG2);
17. }
18. Sort (intersectionsMemoryG2);
19. Sort (graph1_Code)
20. Sort (graph2_Code)
21. Return graph1_Code.equals (graph2_Code);

The length of the signature can be increased depending on the number of minterms of the function.

5. Experimental Results

The results are resumed in generation of graphs and their codification in signatures. For graph with up to 4 variables the graph generation can be tested exhaustively. Thus, for all functions with 1, 2, 3 and 4 variables were generating the graphs and were tested how many graphs were necessary to represent the P-class. The number of these classes was reached when compared to Debnath and Sasao approach [1]. The time of searching was tested and the results are presented in Table 1.

Tab.1 – Times of P-class generation

Variables	1	2	3	4
Sasao	7 ms	7 ms	20 ms	409 ms
Graph-Function	4 ms	5 ms	28 ms	309 ms

These results are generated in Core i5-2400 CPU 3.10GHz, 8 GB of RAM.

6. Conclusion and Future Work

The proposed method is very fast and depends only on the minterms of function. In this way, functions with several variables and few minterms can be matched in very fast and efficient runtime. Moreover, functions with several minterms maintain a good runtime of processing. As future works, a method to find NP- and NPN-class intends to be proposed and the algorithm to be extended to other graphs to verify its validity, as well a formal proof intends to be explained. A study of length of graph with more than 4 variables will be made and compared to other methods.

7. Acknowledgements

Research partially funded by Nangate Inc. under a Nangate/UFRGS research agreement, by CAPES and CNPq Brazilian funding agencies, by FAPERGS under grant 11/2053-9 (Pronem), and by the European Community's Seventh Framework Programme under grant 248538 – Synaptic.

8. Referências

[1] U. Hinsberger and R. Kolla, “Boolean matching for large libraries,” Proc. Design Automation Conference (DAC), pp. 206–211, 1998.

[2] D. Debnath and T. Sasao, “Efficient computation of canonical form for Boolean matching in large libraries,” Proc. Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 591-596, 2004.

[3] Mishchenko, A; Chatterjee, S.; Brayton, R.; Wang, W. and Kam, T. 2005. Technology Mapping with Boolean Matching, Supergates and Choices. ERL Technical Report, EECS Dept., UC Berkeley, March 2005.