FC-Min: A Fast Multi-Output Boolean Minimizer

Petr Fiser, Jan Hlavicka[†], Hana Kubatova
Department of Computer Science and Engineering
Czech Technical University
Karlovo nam. 13, 121 35 Prague 2
e-mail: fiserp@fel.cvut.cz, kubatova@fel.cvut.cz

Abstract

We present a novel heuristic algorithm for two-level Boolean minimization. In contrast to the other approaches, the proposed method firstly finds the coverage of the on-sets and from that it derives the group implicants. No prime implicants of the single functions are being computed; only the necessary implicants needed to cover the on-sets are produced. This reverse approach makes the algorithm extremely fast and minimizes the memory demands. It is most efficient for functions with a large number of output variables, where the other minimization algorithms (e.g. ESPRESSO) are too slow. It is also very efficient for highly unspecified functions, i.e. functions with only few terms defined.

1. Introduction

The problem of two-level Boolean minimization is quite old but surely not dead. It is encountered in many areas of logic design, e.g. the design of control systems, design of build-in self-test for VLSI circuits and in the VLSI synthesis in general [1]. Many Boolean minimization tools were developed, e.g., the classical Quine-McCluskey [2, 3] algorithm, MINI [4] and ESPRESSO [5, 6] with its modifications [7, 8], lately Boom [9], Scherzo [10] and Rondo [11]. The major drawback of these algorithms is the limited size of the problems they can solve in a reasonable time. When the number of input variables grows to hundreds (such problems occur, e.g., in the BIST design), the minimization times are extremely long. This problem was partially solved by Boom. However, the same problem can be encountered for functions with many outputs – the group minimization is quite a demanding problem and the runtimes grow with the number of output variables rapidly as well. The need of a minimization of functions with a large number of input and output variables forced us to develop a fast heuristic algorithm that can handle very complex problems in a reasonable time while the result is near by the optimum.

The classical minimization algorithms consist of two major phases (or they are combined somehow): the generation of implicants of a function and the subsequent covering problem (CP) solution, where the necessary irredundant set of implicants is found. Such an approach may be very demanding (in time and space) for functions with a large number of input and output variables, as the number of both the prime and group implicants is often extremely large.

The process of generating implicants can be also conducted in the reverse way, as it is being done in our algorithm. Firstly the cover of the on-sets independent on the source terms is found, and then the implicants corresponding to this cover are looked for. This reverse approach allowed us to make a fast Boolean minimizer with extremely low memory demands. The main part of our algorithm is a “Find Coverage” procedure, therefore our algorithm was named ***FC-Min***.

Our algorithm does not produce any prime implicants (PIs), since the necessary group implicants are directly generated. As the group implicants are highly important especially for problems with many outputs, this makes our algorithm superior to the others for such problems. On the other hand, FC-Min is not suitable for problems with a small number of output variables. It is due to the cover of the on-set is being generated partially ad-hoc and thus the proper implicants often cannot be found. For such functions our algorithm mostly cannot outperform the others (ESPRESSO, BOOM).

Another feature of FC-Min is that it needs not the don’t care set of the functions to be specified. Thus, the source function is given by its on-set and off-set only.

The algorithm was thoroughly tested on many kinds of problems and the results and runtimes were compared with ESPRESSO. The MCNC [10] benchmarks were solved in order to test the algorithm on practical problems, as well as a set of artificial randomly generated problems were solved to estimate the scalability of the algorithm.

The paper has the following structure: Section 2 defines the problem statement, the main principles of the method are described in Section 3, Section 4 shows the experimental results, Section 5 contains conclusions.

2. Problem Statement

Let us have a set of *m* Boolean functions of *n* input variables $F_1(x_1, x_2, \ldots x_n)$, $F_2(x_1, x_2, \ldots x_n)$, … $F_m(x_1, x_2, \ldots x_n)$; the output values of the care terms (both minterms and terms of a higher dimensions may be used) are defined by the truth table. Thus, each function is specified by its on-set $F_i(x_1, x_2, \ldots x_n)$ and off-set $R_i(x_1, x_2, \ldots x_n)$. To the minterms that are not present in the truth table are implicitly assigned don't care values. The part of a truth table representing the terms will be denoted as an *input matrix*, the rows of the input matrix will be denoted as *input vectors*. The part defining the output values of the functions will be called an *output matrix*, similarly, the rows of this matrix *output vectors*. Each row of the output matrix defines the values of the output variables for the values of input variables specified by the corresponding row in the input matrix.

Specifying a Boolean function by its on-set and off-set, rather by its on-set and don't care set is advantageous especially for highly unspecified functions, i.e., functions that have the defined values of only few terms. The typical example of the use of such a function can be found, e.g., in the build-in self-test (BIST) design [12, 13, 14], for which the method was originally designed. The use of BIST in the nowadays circuits is becoming inevitable, since their external testing is extremely time demanding. Another application of BIST is the design of highly reliable circuits.

Our task is to synthesize a two-level circuit implementing the multi-output Boolean function described by the truth table, whereas the implementation of the circuit should be as small as possible. The result will be in a form of a set of *m* SOP (sum-of-the-product) forms implementing the *m* output functions.

3. Principles of the Method

3.1. Find Coverage

As it was stated in the introduction, the method firstly tries to find a coverage of the on-set by finding a rectangle cover [1] of all the "1" values in the output matrix and then generates implicants having the properties given by this coverage. An example of the coverage of an output matrix is shown in Fig. 1:

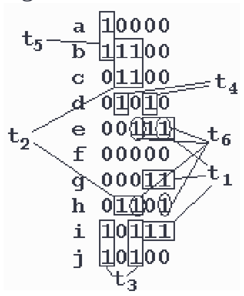

Figure 1. Coverage of the output matrix

The output matrix corresponds to a function with 5 output variables and 10 care terms defined. The potential $t_1 - t_6$ terms cover all the "1" values in the output matrix and no zero. For example, the *group* term (implicant) t_1 covers "1"s of the fourth and fifth output variable in the vectors *e*, *g* and *i*. Let us note that the structure of the terms is not known yet; only the set of covered "1"s is known. Now it is apparent, that if we succeed in finding the implicants having the properties of $t_1 - t_6$ (i.e., the terms cover the appropriate "1"s), the solution will consist of six implicants.

Finding the optimum cover is a NP-hard problem so the covers are looked for by a heuristic. It is based on a gradual search for rectangle covers consisting of the maximum number of "1"s. Firstly the output vector containing the most not yet covered ones is selected as a basis for a new cover - in our example it is the *i* row with four ones. Now we continue to search for the next row to add in order to increase the number of the covered ones. In our example when the row *g* is added, the number of covered ones will not increase (because the first and the third variable cannot be covered after that), however it does not decrease either. After adding the row *e*, the number of covered ones increases to six.

After finding the cover the "1"s that are included in it are marked as "covered" and we continue the search for other covers until all the ones in the output matrix are covered.

Finding the coverage consisting of many "1"s in the output matrix is advantageous indeed, however it often means that it contains many vectors. This fact complicates the succeeding phase – finding the structure of the term. A term whose cover consists of fewer vectors is easier to find. Thus, the heuristic algorithm is driven by a *depth factor* DF. Since each of the rectangle covers is being produced by a successive addition of vectors into it, we can decide after every addition whether to extend the cover to more vectors, or to terminate its generation, even if it could grow bigger. The decision is made at random with a probability given by DF. For instance, when DF = 1, there is an equal probability that the search will continue; when DF = 1/5, there is a probability 1:5 for a continual, and thus terms that cover less vectors and more outputs are more likely generated. In general: when the depth factor is low, the runtimes are shorter, while the complexity of the result is slightly higher.

Such a heuristic approach, where the implicants are looked for independently on the input matrix (source terms), explains why the algorithm is advantageous for functions with many outputs: the group implicants of many output functions are very easy to find by this way. On the other hand when it is used for single-output functions, the algorithm cannot find the primes – it generates the implicants entirely ad-hoc. So using the algorithm for few-output functions is disadvantageous.

3.2. Finding the Implicants

When the cover is generated, we have to find the implicants corresponding to the cover. The information about the output variables included in the cover is insignificant in this phase; the implicants are derived from the information about the covered vectors only. Obviously, when a term (cube) should cover a particular output vector, the corresponding input vector must be contained in this cube, since the input vector implies the output. From this results that the *minimum term* satisfying the particular cover can be constructed as a minimum *supercube* of all the input vectors in the same lines as are the output vectors included in the cover.

Let us assume our leading example. Fig. 2 shows both the input and output matrix.

```
a 11010 10000
b 10000 11100
c 01001 01100
d 01111 01010
e 00110 00111
f 01110 00000
g 10110 00011
h 00001 01101
i 10101 10111
j 11100 10100
```

Figure 2. The input and output matrices

The term t_1 covers vectors *e*, *g* and *i* – see Fig. 1. Thus, the minimum term that can be a candidate for t_1 must be constructed as a minimum supercube of the terms *e*, *g* and *i* in the input matrix, thus:

```
00110
10110
10101
-----
-01--
```

Figure 3. The implicant t_1

The term `(-01--)` was found as a candidate for an implicant t_1. However, since it has to cover *only* the vectors listed above, the term must not intersect with any of the other terms. We can find that it is a valid implicant by comparing the term with the other terms.

Similarly, we can obtain the minimum implicants t_1-t_6. Figure 4 shows all the minimum implicants obtained by finding the corresponding supercubes of the source terms, together with the output part of the resulting PLA matrix:

```
t1: -01-- 00011       t4: 01111 01010
t2: --00- 01100       t5: 1-0-0 10000
t3: 1-10- 10100       t6: 00--- 00101
```

Figure 4. The minimum implicants

The implicants obtained in this phase could form the final solution. However, they can be further expanded to improve the quality of the result. The final result is shown in Fig. 5; the literals removed by the expansion are highlighted.

```
t1: -01-- 00011       t4: ---11 01010
t2: --00- 01100       t5: 1-0-- 10000
t3: --10- 10100       t6: 00--- 00101
```

Figure 5. The expanded implicants

3.3. Incremental Implicant Generation

Until now the algorithm was strictly divided into three successive phases: finding the coverage, generating the implicants and their expansion. However, finding the implicants with the properties of the particular coverage often may not be possible, since some of the minimum implicants may cover some zeroes. In this case the cover must be recomputed somehow. The best way how to solve this problem is an *incremental implicant generation*. Here the first two phases are not separated; firstly one cover is generated, then immediately the minimum implicant is created and, if it is not valid (it covers some zeroes), just the last cover is recomputed.

The whole algorithm can be described by the following pseudo-code. The inputs of the algorithm are the input matrix *I* and the output matrix *O*, the output is the PLA matrix *G*.

```
Minimize(I, O) {
   G = ∅;
   do {
       do {
           c = FindOneCover(O);
           t = GenerateImplicant(I, t);
       } while !IsValid(O, t);
       G = G ∪ t;
   } while !AllCovered();
   Expand(G);
   return G;
}
```

Algorithm 1. The minimization algorithm

4. Experimental Results

Many experiments have been performed in order to evaluate the performance of the method and to compare the results with other up-to-date two-level minimization tools. The algorithm was programmed in C++ Builder under Windows XP, the computer used for tests was Athlon 900 MHz with 256 MB RAM.

We have tested the algorithm on the MCNC benchmarks [15, 16] and compared the results and runtimes with ESPRESSO v2.3 [17]. As the benchmark functions were originally specified by their on-sets and don't care sets (PLA type fd), the sources had to be converted by ESPRESSO into the format where the function is specified by its on-set and off-set. The time needed for the conversion was not included in the runtimes.

There was 120 benchmark problems solved, plus 19 so-called "hard" MCNC benchmarks. The 86 (72%) "non-hard" benchmarks were solved by FC-Min in a shorter time than by ESPRESSO. For 103 cases (86%) FC-Min reached the same or better result (in 8 cases the result was better) and in 80 cases (67%) the same or better result was reached in a shorter time than by ESPRESSO.

From the 19 "hard" benchmarks five were solved in a shorter time by FC-Min, in 14 cases we obtained the same or better result (better in one case).

Table 1 shows the results of the "suggested" MCNC benchmarks [16] and those where FC-Min has reached a better result than ESPRESSO (the bottom part of the table). The column *i/o/p* describes the number of input/output variables and the number of defined terms of the particular benchmark. The ESPRESSO and FC-Min columns contain the runtimes in seconds and the number of literals of the resulting SOP form, the output cost and the number of group terms. The shadowed cells indicate shorter runtime or the equal result respectively.

Table 1. MCNC benchmarks

		ESPRESSO		FC-Min	
bench	*i/o/p*	*time*	*lit/out/terms*	*time*	*lit/out/terms*
b12	15/9/72	0.08	149/59/42	0.01	148/58/43
cordic	23/2/2105	1.86	13825/914/914	8.08	13825/914/914
cps	24/109/855	0.33	1890/946/163	1.30	1890/946/163
duke2	22/29/404	0.09	751/245/86	0.14	751/245/86
ex1010	10/10/1304	0.50	1974/746/284	0.44	1976/742/284
ex4	128/28/654	0.62	1649/279/279	2.98	1649/279/279
misex2	25/18/101	0.07	183/30/28	0.01	183/30/28
misex3c	14/14/1566	0.98	1306/253/197	0.61	1306/255/197
pdc	16/40/822	0.83	828/432/136	0.32	912/520/145
rd84	8/4/511	0.12	1774/296/255	0.15	1774/296/255
spla	16/46/837	0.71	2558/643/251	0.84	2648/749/260
alu4	14/8/1184	0.59	4445/644/575	1.49	4443/644/575
clip	9/5/271	0.10	630/162/120	0.05	621/162/120
dc2	8/7/101	0.05	207/52/39	0.01	206/51/39
in4	32/20/603	0.17	2151/411/212	0.61	2145/411/212
m4	8/16/329	0.16	640/518/105	0.06	640/509/105
newxcpla1	9/23/93	0.07	197/86/39	0.01	196/86/39
opa	17/69/382	0.11	559/540/79	0.17	560/524/79
soar	83/94/779	0.94	2454/549/353	8.01	2445/549/353
x6dn	39/5/310	0.08	641/177/82	0.04	640/177/82

5. Conclusions

A new two-level minimizer for a group of Boolean functions was presented. It is based on the idea that the coverage of the output matrix is found firstly and then the structure of implicants is derived from this coverage and from the vectors in the input matrix. The algorithm is extremely fast and it is very efficient especially for functions with a large number of outputs.

The method was tested on MCNC benchmarks. In most cases it was faster than ESPRESSO with the same result obtained. FC-Min was also tested on randomly generated problems in order to estimate the statistical properties and scalability of the method. The detailed description of these experiments exceeds the scope of this paper, however we have found that it can be easily applied to very large problems without a rapid growth of a runtime.

Acknowledgement

This research has been in part supported by the GA102/03/0672 grant and MSM 212300014, 1999 – 2003.

References

[1] S. Hassoun and T. Sasao, „Logic Synthesis and Verification", Boston, MA, Kluwer Academic Publishers, 2002, 454 pp.

[2] E.J. McCluskey, "Minimization of Boolean functions", The Bell System Technical Journal, 35, No. 5, Nov. 1956, pp. 1417-1444

[3] W.V. Quine, "The problem of simplifying truth functions", Amer. Math. Monthly, 59, No.8, 1952, pp. 521-531

[4] S.J. Hong, R.G. Cain and D.L. Ostapko, "MINI: A heuristic approach for logic minimization", IBM Journal of Res. & Dev., Sept. 1974, pp.443-458

[5] R.K. Brayton et al., "Logic minimization algorithms for VLSI synthesis", Boston, MA, Kluwer Academic Publishers, 1984, 192 pp.

[6] G.D. Hachtel and F. Somenzi, "Logic synthesis and verification algorithms", Boston, MA, Kluwer Academic Publishers, 1996, 564 pp.

[7] R.L. Rudell and A.L. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization", IEEE Trans. on CAD, 6(5): 725-750, Sept.1987

[8] P. McGeer et al., "ESPRESSO-SIGNATURE: A new exact minimizer for logic functions", Proc. DAC'93

[9] J. Hlavicka and P. Fiser, "BOOM - a Heuristic Boolean Minimizer", Proc. ICCAD-2001, San Jose, Cal. (USA), 4.-8.11.2001, 439-442

[10] O. Coudert, "Doing two-level logic minimization 100 times faster", Proc. of the sixth annual ACM-SIAM symposium on Discrete algorithms, 1995, pp.112-121

[11] http://www.ee.pdx.edu/~alanmi/research/

[12] N.A. Touba and E.J. McCluskey, "Transformed Pseudo-Random Patterns for BIST", CRC Technical Report No. 94-10, 1994

[13] M. Chatterjee and D.J. Pradhan, "A novel pattern generator for near-perfect fault coverage", Proc. of VLSI Test Symposium 1995, pp. 417-425

[14] P. Fiser and J. Hlavicka, „Column-Matching Based BIST Design Method", Proc. 7th IEEE European Test Workshop (ETW'02), Corfu (Greece), 26.-29.5.2002, pp. 15-16

[15] S. Yang, „Logic Synthesis and Optimization Benchmarks User Guide", Technical Report 1991-IWLS-UG-Saeyang, MCNC, Research Triangle Park, NC, January 1991

[16] ftp://ftp.mcnc.org/pub/benchmark/Benchmark_dirs/ LGSynth93/testcases/pla/

[17] http://eda.seodu.co.kr/~chang/ download/espresso/