Boolean Minimizer FC-Min: Coverage Finding Process

Petr Fišer, Hana Kubátová
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

This paper describes principles of a novel two-level multi-output Boolean minimizer FC-Min, namely its Find Coverage phase. The problem of Boolean minimization is approached in a reverse way than common minimizers do. First, the cover of the on-set is found, and after that the appropriate implicants are being constructed to satisfy this cover. Thus, only the necessary group implicants are being generated, which makes FC-Min an extremely fast and efficient minimizer for functions with many output variables. An essential phase of the algorithm is the Find Coverage procedure. This phase determines the number of terms in the final solution, which has to be reduced to minimum. It solves an NP-hard problem, thus some heuristic has to be applied. We propose our heuristic method to solve this problem and study the influence of parameters on the final solution quality and runtime.

1. Introduction

The problem of two-level Boolean minimization occurs in many areas of logic design, in logic synthesis in general [1], in build-in self-test (BIST) design [2], in a design of control systems, as so on. The basis for all the conventional minimization algorithms was laid in 50's by Quine and McCluskey [3, 4]. Here the minimization process was divided into two successive phases: the generation of implicants and the solution of the covering problem (CP). Mostly the prime implicants (PIs) are looked for at the beginning, and then they have to be further reduced to obtain group implicants for multi-output functions. The representatives of these principles are MINI [5], ESPRESSO [6] and its modifications [7], later Scherzo [8] with its improved CP solution algorithm was introduced. Lately we have developed a Boolean minimizer BOOM [9, 10], which is able to handle functions with an extremely large number of input variables.

A common drawback of the previously mentioned algorithms is the way they handle multi-output functions. The implicant reduction phase is often very time consuming and produces many unnecessary implicants (i.e., implicants that will not be a part of the solution). A large number of prime implicants of all the output functions is being generated as well, which then complicates the CP solution. Generally, minimization of functions with a large number of output variables is a very time-consuming process and the results are often suboptimal.

Our Boolean minimizer FC-Min [11] generally solved this problem. Here the implicants are being generated in a completely different way: first the cover of the on-set is found (in the Find Coverage phase) and after that the implicants satisfying this cover are computed from the input terms. Only the necessary set of implicants is generated; no implicants that will not be a part of the solution are produced. The covering problem solution phase can be completely omitted here. Basically, the Find Coverage phase can be apprehended as a covering problem solution, but in a reverse way.

The basic FC-Min algorithm was then extended to support the iterative minimization. The principle exploits the fact that during the minimization some decisions are being done at random. Thus, multiple runs of the same algorithm yield different results. All the obtained implicants are then put together and the regular CP is solved. This enables us to reach better results for a price of a longer runtime.

We have extensively tested the minimizer on standard MCNC benchmarks and on randomly generated problems. Our results were compared with ESPRESSO and BOOM. FC-Min is superior to these methods in both the runtime and result quality, especially for functions with many output variables. On the other hand, it is not advantageous to use FC-Min for single-output functions, since the cover is being generated partially at random in this case, so the result is not optimal.

The paper is structured as follows: Section 2 contains the problem statement, Section 3 describes the principles of FC-Min, with emphasis on the Find Coverage phase. The comparison of the method with other algorithms is given in Section 4, Section 5 concludes the paper.

2. Problem Statement

Let us have a set of m Boolean functions of n input variables. The input variables will be denoted as x_i, $0 \le i < n$, the output variables as y_j, $0 \le j < m$. The functions will be referenced as $\boldsymbol{F}_1(x_1, x_2, \dots x_n)$, $\boldsymbol{F}_2(x_1, x_2, \dots x_n)$, … $\boldsymbol{F}_m(x_1, x_2, \dots x_n)$. The output values of the care terms (both minterms and terms of a higher dimensions may be used) are defined by a truth table. Thus, each function is specified by its on-set $\mathrm{F}_i(x_1, x_2, \dots x_n)$ and off-set $\mathrm{R}_i(x_1, x_2, \dots x_n)$. To the minterms that are not present in the truth table are implicitly assigned don't care values. The don't cares can be given explicitly too, both in the input matrix (thus defining a term of a higher dimension) and the output matrix (specifying an output don't care value for a given input term). The part of a truth table representing the terms will be denoted as an *input matrix* ***I***, 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* ***O***; similarly, the rows of this matrix *output vectors*. Each row of the output matrix defines values of the output variables for the values of input variables specified by the corresponding row in the input matrix. The number of **I** matrix columns correspond to the number of input variables n, the number of **O** matrix columns is equal to the number of output variables m, the number of **I** and **O** matrix rows will be denoted as p.

Specifying a Boolean function by its on-set and off-set, rather than 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 rest are don't cares. A typical example of a use of such functions can be found, e.g., in the build-in self-test (BIST) design [12, 13, 14].

Our task is to synthesize a two-level circuit implementing a multiple-output Boolean function described by a truth table, whereas the implementation of the circuit should be as small as possible. Thus, we perform a group two-level Boolean minimization where a set of functions is given by their on-sets and off-sets. The result will be in a form of a set of m SOP (sum-of-the-product) forms implementing m output functions. In practice, the output of an algorithm is a PLA file ("type fd") describing the structure of the PLA implementation of the required circuit. In our algorithm we try to take advantage of the group implicants, thus terms that imply more than one output function.

3. Principles of FC-Min

The FC-Min algorithm produces implicants of a function in a way that is reverse to other common minimization algorithms. First, the cover of the on-set is found and after then the implicants are being derived from this cover. Hence, there are two major phases of this algorithm: the *Find Cover* phase and the *Find Implicant* phase. After that, the implicants should be yet expanded in order to reduce the total number of literals. Description of this process exceeds the scope of this paper, for more information see [11].

3.1. Find Cover Phase

In the Find Cover phase we try to find a rectangle cover [1] of the whole on-set. This means that we try to find potential implicants that could be a part of the solution, if they met the requirements for the stated cover. This phase is the most essential one, since it determines the number of implicants in the final solution.

Before we describe the principles of the algorithm we have to state four Definitions.

Definition 1

Let t_i be an implicant. The *coverage set* $C(t_i)$ of the implicant t_i is a set of vectors (rows) of the **O** matrix, in which at least one "1" value is covered by this implicant. In other words, the coverage set is a set of vectors of the output matrix for which t_i is an implicant for at least one output variable.

■

Definition 2

The *coverage mask* $M(t_i)$ of the implicant t_i is a set of columns in the **O** matrix, in which all the vectors included in $C(t_i)$ have one or more "1" value. The coverage mask $M(t_i)$ can also be expressed as a vector in the resulting output matrix corresponding to the term t_i. In the following text we will use both these representations of the coverage mask.

■

Definition 3

The *coverage of an implicant* t_i is a pair of the sets $C(t_i)$ and $M(t_i)$ for which the following equation holds:

$$\forall a \in C(t_i), \forall b \in M(t_i) : \mathbf{O}[a,b] \neq "0"$$

The "1" or don't care values covered by t_i are identified by the Cartesian product $C(t_i) \times M(t_i)$.

■

Definition 4

The *coverage of the matrix* **O** is a set of implicant coverages $\{C(t_i), M(t_i)\}$ such that

$$\forall a < p, \forall b < m, \mathbf{O}[a,b] = "1" : \{a,b\} \in \bigcup_{\forall i} C(t_i) \times M(t_i)$$

■

The individual implicant coverages will be also denoted as *coverage elements* with respect to the coverage of the matrix.

An example of such a coverage of the **O** matrix is shown in Figure 1. Here all the "1"s are covered by six implicants t_1 - t_6. Their respective coverage sets and masks are listed in Table 1.

This output matrix **O** 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 cover no zero. For example the *group* term (implicant) t_1 covers the ones of the fourth and fifth output variable of vectors *4, 6* and *8*. Let us note that the structure of the terms is not known yet; only the set of covered "1"s is known. However, 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 respective "1"s), the solution will consist of six implicants.

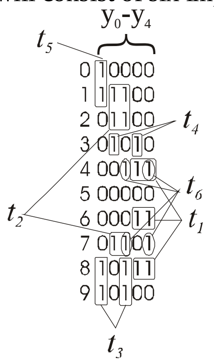

Figure 1. Cover of the output matrix

Table 1. Coverage sets and masks

Implicant	$C(t_i)$	$M(t_i)$
t_1	*{4, 6, 8}*	$\{y_3, y_4\} \equiv 00011$
t_2	*{1, 2, 7}*	$\{y_1, y_2\} \equiv 01100$
t_3	*{8, 9}*	$\{y_0, y_2\} \equiv 10100$
t_4	*{3}*	$\{y_1, y_3\} \equiv 01010$
t_5	*{0, 1}*	$\{y_0, y_1\} \equiv 11000$
t_6	*{4, 7}*	$\{y_2, y_4\} \equiv 00101$

Finding an optimum rectangle cover is a NP-hard problem, thus some heuristic must be used. There exist many efficient algorithms finding a cover consisting of the minimum of elements [1]. However, such a cover needs not be always optimal for solving our minimization problem. The solution will be then consisted of a minimum of product terms, however, it needs not be minimal with respect to the number of literals.

Our heuristic is based on a gradual search for coverage elements consisting of the maximum number of "1"s. Firstly the output vector containing the most of not covered "1"s is selected as a basis for a new cover - in our example (Fig. 1) it is the row no. 8 with four "1"s. Now we continue the search for a next row to add in order to increase the number of the covered ones. In our example, when the row 6 is added to the row 8, 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 4, the number of covered ones increases to six.

After finding one 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 a cover consisting of many "1"s in the output matrix is advantageous, however it often means that it contains many vectors. This fact complicates the subsequent phase – finding the structure of a 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 cumulating of vectors, we can decide after each 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 = 1:1, 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 fewer 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 generally advantageous for functions with many outputs: the group implicants of many output functions are very easy to find 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. Thus, using the algorithm for few-output functions is disadvantageous.

The Find Cover algorithm can be described into detail by the following pseudo-code:

```
FindCover(O) {
   C = ∅;
   M = all_output_variables;
   k = number_of_ones_in_O;
   do {
      v = vector_with_maximal_x_for(0 ≤ i < p)
           x = |C+1|*|M ∩ O[i]| - |C|*|M|;
      C = C ∪ {v};    // include v into C
      M = M ∩ O[v];   // reduce M
      Assign_as_Covered(O, C × M)
      k = k – (|C+1|*|M ∩ O[i]| - |C|*|M|);
   } while (k > 0);
   return (C, M);
}
```

Algorithm 1. Find Cover phase

3.2. Incremental Implicant Generation

After the coverages and the coverage masks are produced, the implicants have to be derived using these coverages and the **I** matrix. 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. Therefore the *minimum term* t_i satisfying the particular cover must be constructed as a *minimum supercube* of all the input vectors corresponding to $C(t_i)$. Moreover, this supercube must not intersect any **I** matrix term that is not included in its coverage set, since it would cover some zeros then. This process will be denoted as an *Implicant Generation* phase. For more detailed description see [11].

Here we assume that a term created as a minimum supercube of the proper **I** matrix terms is a valid implicant of all the output functions included in its respective coverage mask. But some of such terms may intersect the off-set (i.e., some of the **I** matrix terms that are not included in its coverage set), and thus they are not valid. In this case there is no solution, particularly, it is impossible to find implicants for the computed cover. Therefore, the cover must be recomputed somehow. One possibility is to try to split the cover in order to make supercubes of fewer terms. This approach leads to a rapid growth of the number of terms in the final solution. The other possibility is to recompute the whole cover, thus repeat the phases until a solution is found. Such an approach causes a great growth of the run-time and also the algorithm often gets into an insolvable state.

We have found that the best way how to solve this problem is an *incremental implicant generation*. In this approach the two main phases are not separated; firstly one coverage element is generated, then immediately its minimum implicant is created and, if it is not valid (it covers some zeroes), just the last coverage element is discarded and a different one is found. This approach is very fast and produces best results. It can be proved that this approach always yields a solution; there is never a chance to reach an insolvable state.

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 a PLA matrix *G*.

```
FC_Minimize(I, O) {
   G = ∅;
   do {
       do {
           c = FindOneCoverElement(O);
           t = GenerateMinimumTerm(I,
                 c);
       } while !IsValid(t, I);
       G = G ∪ t;
   } while !AllCovered();
   return G;
}
```

Algorithm 2. The minimization algorithm

3.3. Influence of DF on the Solution

Since the depth factor DF significantly drives the generation of the cover, it is important to choose a proper value to obtain a desirable solution. Bigger values of DF force the algorithm to generate “deeper” terms, i.e., terms that cover many on-set terms. These terms are then implicants of a smaller number of output variables, since the cardinality of the coverage mask of a term tends to decrease with an increasing cardinality of its respective coverage set (see Alg. 1). Unfortunately, these covers often are not be valid (see the previous Subsection), thus they have to be often recomputed (see Alg. 2). This means a rapid increase of a runtime. The ratio of the total number of cover computations (tries) to the number of valid coverage elements (hits) as a function of DF is shown in Fig. 2, Fig. 3 depicts the total runtime as a function of DF. The problem solved was a 20-input, 20-output function with 500 care terms. In both the input and output matrices, 10% of explicit don’t cares were included.

For a low value of DF, implicants that cover a small number of terms are more likely produced, however they are implicants of a large number of outputs. In a trivial case, solution of a problem with *p* terms could consist of *p* implicants created just by an expansion of the input terms, while the coverage masks of the terms would be equal to the output vectors of the respective source terms. Here the Find Cover phase is, de facto, omitted, the minimization consists just of the input expansion phase (see [11]).

In general, a low value of DF generates a solution in an extremely short time, but the cost of the solution is high. High values of DF produce solutions having a low number of implicants (product terms) and literals as well, for a cost of slightly longer runtimes. These dependencies are shown in Figures 4 and 5. The same problem as in Fig. 2 and 3 was solved. Figure 4 shows the dependence of the number of terms in a solution on DF, Fig. 5 shows the total cost of the solution, measured as a sum of the number of literals and the output cost (the output cost is the number of inputs into the OR-gates, if the circuit is implemented as an AND-OR net).

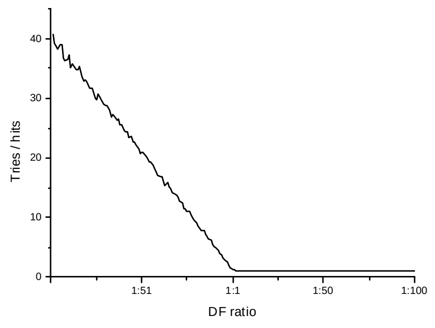

Figure 2. Ratio of the tries to the hits

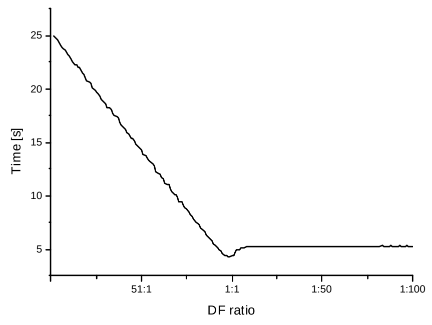

Figure 3. FC-Min runtime

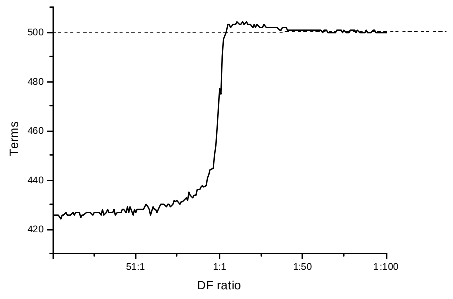

Figure 4. Number of terms

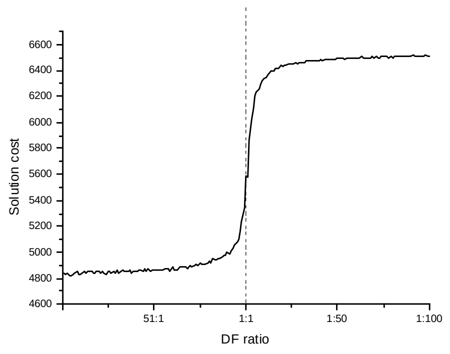

Figure 5. Total solution cost

We have concluded from our experiments that the most efficient value of DF is about 10:1. Higher values mean just the growth of a runtime, while the improvement of the quality of the solution is negligible.

3.4. Iterative FC-Min

In most cases the FC-Min algorithm is not deterministic – the progress of the Find Coverage phase is controlled by a random number generator when deciding whether to continue the "prolongation" of the processed cover element or not. Secondly, when two or more equally advantageous steps are possible, one is chosen at random. Thus, repeated run of FC-Min could produce different results. The idea of an *Iterative FC-Min* consists in repeating the FC-Min several times, while all the different implicants are put together and stored. At the end the final solution is constructed by solving the covering problem using all the implicants. Even if each of the single FC-Min runs produces a valid solution, a properly selected *combination* of the implicants obtained from different iterations might produce a better solution. It is paid by a longer runtime.

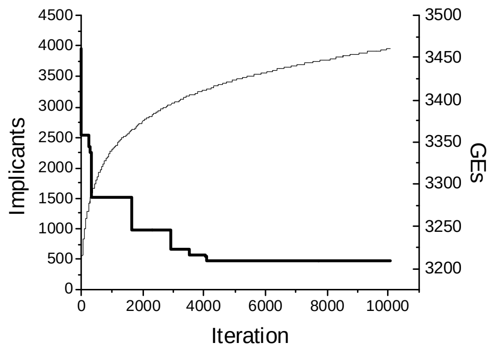

Figure 6. Iterative minimization example

An example of an iterative FC-Min run is shown in Fig. 6. The sample problem solved was a randomly generated function of 20 input and 20 output variables, with 200 terms defined. The input matrix contained 10% of explicit don't cares. The depth factor was set to 2:1. The thin line indicates the growth of the number of implicants. We can observe that in 10000 iterations the total number of different implicants increased from 200 to 4000. The thick line shows the quality of the result after each iteration. It is measured in gate equivalents (GEs), which is a good approximation of a complexity of the physical implementation of the circuit [16]. The number GEs for an n input NAND or NOR gate is computed as $n/2$.

The number of GEs was reduced by 8% in 4000 iterations, and then it remains unchanged, even if the number of implicants is still increasing.

Figure 7 illustrates the influence of the depth factor DF on the implicant growth rate. For an extremely low DF (1:100) the number of implicants remains almost unchanged. On the other hand, many different implicants are being generated when increasing DF. This allows us to reach a better result in a shorter time. However, with increasing DF also the runtime grows rapidly. Thus, some kind of a tradeoff must be found to achieve the fastest implicant growth. It is illustrated by Fig. 8. Here the same problem was solved and the increase of the number of implicants was measured as a function of a time. We can observe that the fastest growth was achieved for DF = 10:1. For higher DF the seemingly faster implicant production is suppressed by longer runtimes.

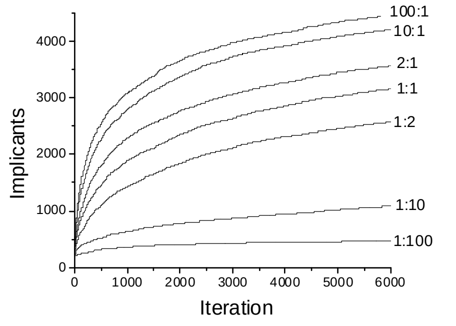

Figure 7. Influence of DF on the number of iterations

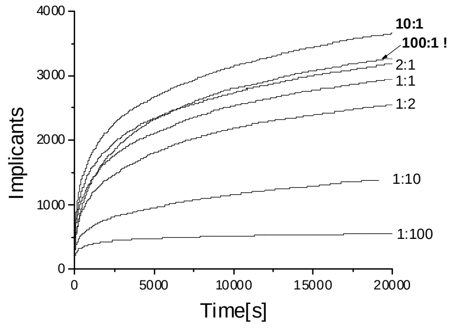

Figure 8. Influence of DF - growth of the number of implicants with time

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 standard MCNC benchmarks [15] and compared the results and runtimes with ESPRESSO v2.3. Since 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 a 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%) from 120 of them 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. For more detail information see [11].

4.1. Randomly Generated Problems – One Iteration

The second set of problems on which we have tested FC-Min were randomly generated functions, functions with no special properties (no aggregated ones in the output matrix, etc.). With a help of such problems we can easily observe the properties and scalability of the algorithm. One of the reasons why FC-Min was developed was a need to synthesize the combinational logic for BIST, namely the output decoder transforming the LFSR patterns into test patterns pre-generated by an ATPG tool. Both the LFSR and ATPG patterns mostly have a random nature, and thus the randomly generated benchmarks simulate these practical problems very well [12-14].

We have generated problems with a varying number of input variables and terms, the number of outputs was fixed to 15. These artificial benchmarks were solved by FC-Min, BOOM [9, 10] and ESPRESSO [6] to compare the performance. Here only one iteration of BOOM and FC-Min was performed, the FC-Min depth factor was set to 10:1.

The results of the minimization are shown in Table 2. The number of inputs increases in the horizontal direction (*i*), the number of care terms in the vertical direction (*p*). Each of the cells contains average values of ten problems of the same size that were solved, to ensure steady statistical values. The first row of each cell in the table contains results obtained by ESPRESSO, the second one the result obtained by BOOM and the third by FC-Min. The first number in each line indicates the runtime, the second one the number of literals in the SOP form, the output cost follows and last number indicates the number of terms. We can see that in all the cases FC-Min completed the minimization in a significantly shorter time than ESPRESSO and BOOM, while the result quality is comparable.

4.2. Randomly Generated Problems – Same Time

Next, we have solved the same set of problems, but taking advantage of the iterative minimization this time. The functions were minimized by ESPRESSO first, and then both by BOOM and FC-Min, while the runtime was set to meet the runtime that ESPRESSO needed to reach a solution.

The results are shown in Table 3. The format of the table is retained from the previous example, except of that only the ESPRESSO runtime is shown, while the number of iterations (to meet that time) is given in the parentheses for BOOM and FC-Min.

Here FC-Min gives much better results than ESPRESSO, especially for problems with many input variables. For most of these problems FC-Min outperforms BOOM as well. However, for problems with a low number of output variables BOOM is faster and the result quality is better too. Thus, for an efficient minimization we have to decide whether to use BOOM or FC-Min, judging by the number of outputs.

4.3. Time Complexity Evaluation

Since *FC-Min* is a heuristic method, it is difficult to determine its time complexity exactly. In order to estimate the time complexity of the method FC-Min was run on a large number of randomly generated problems with one parameter varying each time, while the minimization times were recorded.

Figures 9-11 show the time dependencies on the number of input variables (Fig. 9), output variables (Fig. 10) and the number of care terms (Fig. 11). The values of the fixed parameters are indicated in the figures, the depth factor was set to 10:1. No exponential growth of time can be observed in any of the curves, thus the method can be scaled to very large problems while the runtime remains minimal.

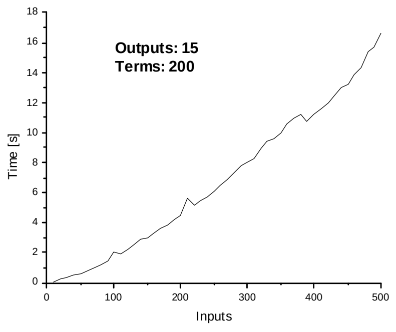

Figure 9. Time complexity as a function of the number of inputs

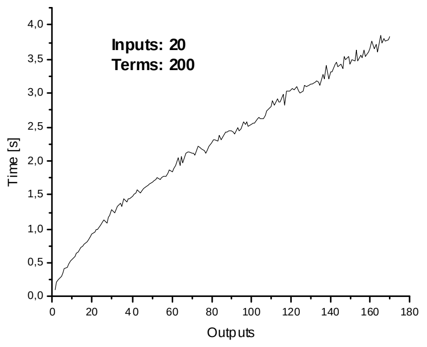

Figure 10. Time complexity as a function of the number of outputs

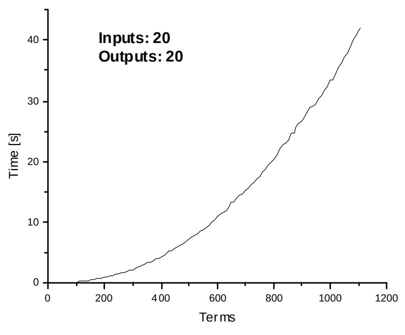

Figure 11. Time complexity as a function of the number of the care terms

5. Conclusions

We have presented a new two-level Boolean minimization method called FC-Min. It is based on the idea that the coverage of the output matrix is found at first, and then the structure of implicants is derived from this cover. The algorithm is extremely fast and it is very efficient, especially for functions with a large number of outputs. Particularly, the Find Coverage phase, which is essential for the algorithm, is more thoroughly described in this paper. The influence of the Depth Factor on the runtime and the quality of the solution was studied here.

The method was tested on the MCNC benchmarks and randomly generated problems. We have shown that for problems with higher number of output variables it is faster than both ESPRESSO and BOOM and gives better results as well. The algorithm was tested on randomly generated problems in order to estimate the statistical properties and scalability of the method. We have found that the method can be easily applied to very large problems without a significant growth of a runtime.

Acknowledgement

This research was supported by a grant GA 102/04/2137 and MSM 212300014

References

[1] S. Hassoun and T. Sasao, „Logic Synthesis and Verification", Boston, MA, Kluwer Academic Publishers, 2002, 454 pp.
[2] Agarwal, Kime, Saluja: A tutorial on BIST, part 1: Principles. IEEE Design & Test of Computers, vol. 10, No.1 March 1993, pp.73-83, part 2: Applications, No.2 June 1993, pp.69-77
[3] W.V. Quine, "The problem of simplifying truth functions", Amer. Math. Monthly, 59, No.8, 1952, pp. 521-531
[4] E.J. McCluskey, "Minimization of Boolean functions", The Bell System Technical Journal, 35, No. 5, Nov. 1956, pp. 1417-1444

[5] 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

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

[7] P. McGeer et al., “ESPRESSO-SIGNATURE: A new exact minimizer for logic functions”, Proc. DAC’93

[8] 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

[9] J. Hlavička and P. Fišer, „BOOM - a Heuristic Boolean Minimizer”, Proc. ICCAD-2001, San Jose, Cal. (USA), 4.-8.11.2001, 439-442

[10] P. Fišer and J. Hlavička, „BOOM - A Heuristic Boolean Minimizer“, Computers and Informatics, Vol. 22, 2003, No. 1, pp. 19-51

[11] P. Fišer, J. Hlavička and H. Kubátová, „FC-Min: A Fast Multi-Output Boolean Minimizer“, Proc. Euromicro Symposium on Digital Systems Design (DSD'03), Antalya (TR), 3.-5.9.2003

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

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

[14] P. Fišer, J. Hlavička and H. Kubátová, „Column-Matching BIST Exploiting Test Don't-Cares“. Proc. 8th IEEE Europian Test Workshop (ETW'03), Maastricht (NL), 25.-28.5.2003, pp. 215-216

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

[16] G. De Micheli, “Synthesis and Optimization of Digital Circuits”. McGraw-Hill, 1994

Table 2. Randomly generated problems – one iteration

p / i	25	50	100	150
50	2.27/232/341/49 1.30/413/220/87 0.27/315/305/59	11.34/219/318/48 1.40/428/156/94 0.29/304/241/58	46.35/202/301/46 1.59/412/121/90 0.34/293/195/56	94.64/203/303/47 1.73/371/96/84 0.41/283/181/54
100	10.22/577/687/98 2.59/998/506/168 0.50/716/648/110	100.14/537/576/91 2.56/1103/342/190 0.51/718/491/113	369.45/510/569/90 3.02/1050/244/186 0.64/676/413/106	883.24/488/554/88 3.53/943/186/168 0.79/647/372/102
125	14.44/772/849/123 3.51/1333/650/211 0.66/952/846/137	148.96/710/728/114 3.49/1468/449/237 0.65/927/642/137	756.21/666/704/110 4.23/1408/317/231 0.86/880/519/131	2146.03/652/674/108 4.82/1252/243/211 1.09/829/473/124
150	23.35/973/1005/147 4.71/1691/785/255 0.86/1182/1007/163	283.88/892/869/136 4.63/1849/563/285 0.84/1164/779/164	1111.23/833/800/129 4.72/1761/378/278 1.04/1098/638/157	3422.94/798/773/126 5.65/1613/295/256 1.33/1039/573/148

Entry format: time [s] / # of literals / output cost / # of implicants

Table 3. Randomly generated problems – same time as ESPRESSO

p/n	25	50	100
50	2.15/233/346/49 340/246/70(2) 290/264/58(8)	10.80/218/324/48 294/189/61(7) 252/185/50(28)	51.96/204/309/47 247/139/53(27) 214/150/43(81)
75	5.62/400/513/74 525/381/95(3) 465/394/83(13)	34.37/370/463/70 466/276/86(12) 404/279/71(47)	154.71/357/438/68 423/218/79(35) 357/223/62(99)
100	11.24/581/673/99 768/528/127(4) 659/543/110(19)	84.48/546/586/92 665/358/111(16) 571/365/92(63)	416.29/520/564/90 600/287/102(44) 498/301/80(118)
125	17.75/773/845/123 1010/616/160(4) 868/674/138(22)	157.19/706/722/113 872/459/137(17) 745/456/115(71)	895.25/657/700/110 765/359/122(52) 650/374/99(137)

Entry format: ESPRESSO: time [s] / # of literals / output cost / # of implicants
next lines: # of literals / output cost / # of implicants (iterations)