Flexible Two-Level Boolean Minimizer BOOM-II and Its Applications

Petr Fišer, Hana Kubátová
Czech Technical University
Dept. of Computer Science and Engineering, Karlovo nám. 13, 121 35, Prague 2
e-mail: fiserp@fel.cvut.cz, kubatova@fel.cvut.cz

Abstract

We propose a novel two-level Boolean minimizer coming in succession to our previously developed minimizer BOOM, so we have named it BOOM-II. It is a combination of two minimizers, namely BOOM and FC-Min. Each of these two methods has its own area where it is most efficiently applicable. We have combined these two methods together to be able to solve all kinds of problems efficiently, independently on their size or nature. The tool is very scalable in terms of required runtime and/or quality of the solution. It is applicable to functions with an extremely large number of both input and output variables. The minimization process is very flexible and can be driven by miscellaneous user-defined constraints, such as low-power design, design-for-testability and decomposition constraints. Some of the application areas are described in the paper.

1. Introduction

The two-level Boolean minimization problem occurs in many areas of the logic design [1], in the build-in self-test (BIST) design [2], in a design of control systems, etc. Since the time when the basis of minimization algorithms was laid in 50's by Quine and McCluskey [3], the minimization process is divided into two phases: generation of implicants and solution of the covering problem (CP). A representative of these principles is, e.g., ESPRESSO [4]. Lately we have developed the BOOM minimizer [5, 6], capable to handle functions with a very large number of input variables, basically based on the same principle.

A common drawback of the previously mentioned 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 process and the runtimes grow with the number of output variables rapidly as well. Lately we have developed an algorithm called FC-Min [9] solving this problem efficiently. The solution is being constructed from group implicants only, which makes the algorithm extremely fast and having low memory demands. On the other hand, FC-Min does not produce good results for functions with a low number of output variables.

A method where the FC-Min and the original BOOM algorithms are combined together to achieve better results is proposed in this paper. Implicants are being produced both by BOOM and FC-Min and then they are put together into a common implicant pool. The final solution is then constructed by solving a covering problem using all the implicants. The ratio of runs of both algorithms can be freely adjusted, which produces good solution for all kinds of problems. Since the system is a successor of BOOM to some extent, we have named it *BOOM-II*. Since FC-Min primarily produces group implicants, the extremely time-demanding implicant reduction phase can be omitted in BOOM-II. These effects are documented in this paper.

This paper then discusses how the minimization process can be influenced by constraints, like designing easily decomposable functions, easily testable functions, low-power design, etc.

The paper is structured as follows: Section 2 defines the problem statement, the structure and major principles of BOOM-II will be described in Section 3, the experimental results are shown in Section 4. The ways to influence the BOOM-II run are described in Section 5. Section 6 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, $0 \leq I < n$, the output variables as y_j, $0 \leq j < m$. The output values of the care terms (both minterms and terms of a higher dimensions may be used) are defined by a truth table. To the minterms that are not present in

the truth table are implicitly assigned don't care values. The number of defined terms 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 [7, 8].

Our task is to synthesize a two-level circuit implementing a multiple-output Boolean function described by a truth table, whereas this implementation should be as small as possible. Thus, we perform a group two-level minimization. The result will be in form of a set of *m* SOP (sum-of-the-products) forms.

3. BOOM-II Principles

As it was stated in the Introduction, BOOM-II is a composition of two previously published minimization algorithms - BOOM [5, 6] and FC-Min [9]. Both the algorithms have their advantages and drawbacks. BOOM is suitable for problems with a large number of input variables, but it is somewhat limited regarding the number of output variables; for a large number of outputs the runtime rapidly grows and the algorithm begins to be less efficient as well. This is due to the time-demanding implicant reduction (IR) phase above all. BOOM is based on generation of prime implicants (PIs), and thus it is efficient for problems whose solution is constructed mostly of PIs. Thus, BOOM is very efficiently applicable to problems with many input variables and a low number of outputs.

The second minimizer FC-Min was developed to handle problems with many output variables. It is extremely fast - the runtime grows almost linearly with growing number of both the input and output variables [9]. The solution is being constructed of group implicants only (particularly it does not distinguish between PIs and group implicants). Hence, FC-Min is good for problems whose solution is constructed of many group implicants, i.e., problems with many output variables. On the other hand, it is not suitable for functions with few outputs, since the cover of the on-set is being generated purely at random in this case.

Both the algorithms were developed in their iterative versions. The *iterative minimization* is based on a fact that some minimization phases are driven by random events. Hence, two runs of the same algorithm on the same problem instance need not produce equal results. Moreover, a better solution can sometimes be achieved by *combining* implicants from two or more different solutions. In practice, the algorithm is executed several times, while all the implicants obtained are put together into a common implicant buffer. Then the covering problem (CP) is solved using all of them.

A typical growth of the number of prime implicants as a function of the number of iterations is shown in Fig. 1 (thin line). This curve plots the values obtained during the solution of a single-output problem with 20 input variables and 200 minterms, using BOOM only. Theoretically, the more implicants we have, the better is the solution that can be found after solving the covering problem. In practice, the quality of the final solution, measured by the number of literals in the resulting SOP form, improves rapidly during first few iterations and then remains unchanged, even though the number of PIs grows. See the thick line in Fig. 1.

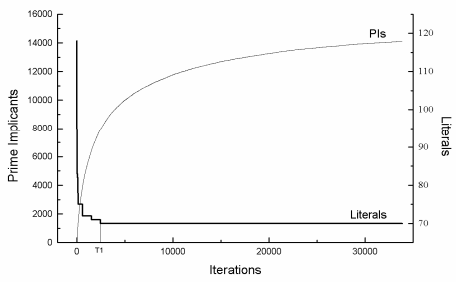


Figure 1. Growth of the number of PIs and decrease of SOP length during iterative minimization

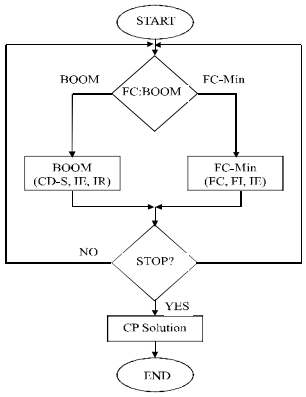


Figure 2: BOOM-II flowchart

The idea of combining implicants from different minimization runs gave rise to BOOM-II. The same problem is solved both by BOOM and FC-Min (repeatedly), all the implicants are put together and the covering problem is solved. Prime implicants are more likely being picked out from the implicants obtained by BOOM, while the group implicants are produced by FC-Min. The ratio of the two algorithms can be adjusted manually by a *FC-Min:BOOM* factor. For example, when this factor is set to 1:1, half of the

iterations will be conducted by BOOM and half by FC-Min in average. The problem of the distribution of implicants produced by BOOM and FC-Min is studied more thoroughly in Subsection 4.3. The flowchart of the BOOM-II system is shown in Fig. 2.

3.1. Brief Summary of BOOM

Like most other Boolean minimization algorithms, BOOM consists of two major phases: *generation of implicants* and the subsequent *solution of the covering problem*. At the beginning the *m*-output function is split into *m* single-output functions and a set of PIs is computed for each function. The most important part of the algorithm, the *Coverage-Directed Search (CD Search)*, generates a sufficient set of implicants needed to cover the on-set of a single function. The implicants are then passed to the *Implicant Expansion (IE)* phase, which converts them into PIs. In the subsequent *Implicant Reduction (IR)* phase the PIs are being reduced to obtain group implicants. Then the covering problem is solved to obtain the final solution.

The principle of the Coverage-Directed Search consists in selecting most suitable literals that should be added to a term under construction. Thus, instead of increasing the dimension of an implicant starting from a minterm, we reduce an *n*-dimensional hypercube by adding literals to the term, until it becomes an implicant of the processed function. This happens at the moment when the resulting hypercube does not intersect any 0-term. The search for suitable literals that should be added to a term is directed towards finding an implicant covering as many 1-terms as possible. To do this, implicant generation starts by the most frequent input literal selection from the given on-set, because the (*n-1*) dimensional hypercube covering the most 1-minterms is described by the most frequent literal appearing in the on-set. The (*n-1*) dimensional hypercube found by this way is an implicant, if it does not intersect any 0-term. If there are some 0-minterms covered, we add another literal (the second most frequent one) and verify whether the new term already corresponds to an implicant by comparing it with 0-terms that might intersect with this term. We continue by adding literals until an implicant is generated, then it is recorded. Then we start searching for other implicants.

More thorough description of CD-Search and the subsequent phases can be found, e.g., in [9, 10].

3.2. Principles of FC-Min

The FC-Min minimizer generates the solution in a completely different way. As it was said before, classical minimization methods consist of two major phases: the generation of implicants and the subsequent covering problem solution, where the irredundant set of implicants is found in order to cover the on-set. Such an approach might be very demanding (both in time and space) for functions with a large number of input and output variables.

In FC-Min, the process of generating implicants is conducted in a reverse way. Firstly the cover of the on-set is found. Then implicants corresponding to this cover are generated. This reverse approach allowed us to make a fast Boolean minimizer with extremely low memory demands. FC-Min does not produce PIs; only necessary group implicants are directly generated. As the group implicants are highly important for problems with many outputs, this makes FC-Min superior to other minimizers for such problems. On the other hand, FC-Min is not suitable for problems with a *small* number of output variables. It is because the cover of the on-set is being generated partially ad-hoc here and thus proper implicants often cannot be found. For such functions our algorithm mostly cannot outperform the others (ESPRESSO, BOOM).

The FC-Min algorithm consists of two major phases: the *Find Coverage* phase, in which the rectangle cover [2] of the on-set is found, and the *Implicant Generation* phase producing the very implicants from this cover.

3.3. Covering Problem Solution

It can be seen from Fig. 1 that even a small subset of PIs may give the minimum solution. However, the quality of the final solution strongly depends on the CP solution algorithm. It is impossible to obtain exact solutions when having a large number of implicants, since it is an NP-hard problem. Thus some heuristic must be used.

After an extensive testing we have chosen a greedy method based on computing the contributions (scoring functions) of terms as a criterion for their inclusion into the solution [17]. We construct a covering matrix *A*, its dimension will be denoted as (*r*, *s*). The columns correspond to the implicants, rows to the on-set terms that are to be covered. *A[i, j]* = 1 if the implicant *j* covers the on-set term *i*, *A[i, j]* = 0 otherwise. For each row its *strength of coverage* is computed as

$$\mathrm{SC}(x_i) = \frac{1}{\sum_{j=1}^{s} A[i,j]} \qquad (1)$$

Then the *column contribution* is computed for each column:

$$\mathrm{CC}(y_j) = \sum_{i=1}^{r} A[i,j] \cdot SC(x_i) \qquad (2)$$

The implicant (column) with the maximum contribution value is selected into the solution, the contribution values are recomputed and the process is repeated until the whole on-set is covered.

4. BOOM-II Results

4.1. Study of the Structure of the Solution

One possibility how to estimate the "usefulness" of the incorporation of FC-Min into BOOM is to analyze implicants in the solution. Particularly, we have studied the origin of the implicants in the final solution, and analyzed which one of the two algorithms contributes to it most.

At any time, the set of implicants in the common implicant buffer (and, of course, in the final solution too) can be divided into these six groups:

1. Prime implicants (of at least one output function) found by BOOM only
2. Prime implicants found both by FC-Min and BOOM
3. Prime implicants produced by FC-Min and which were not found by BOOM (these had to be identified by a subsequent analysis, since FC-Min does not recognize any PIs)
4. Group implicants found by BOOM only
5. Group implicants that have been found both by FC-Min and BOOM
6. Group implicants produced by FC-Min only

These sets make a decomposition of the set of all implicants. The union of these six subsets gives all the implicants, the subsets are disjoint, see Fig. 3. It can be better visualized by a Venn's diagram:

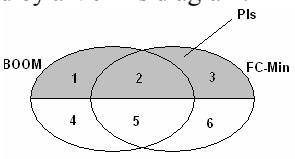
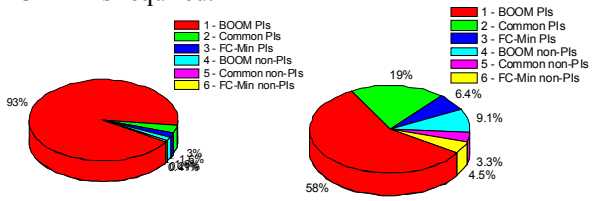


Figure 3. BOOM-II implicants

We have minimized a randomly generated function of 20 input variables, 20 outputs, having 10% of explicit both input and output don't cares and 500 defined terms. The ratio *FC-Min:BOOM* was set to 1:1.

Figure 4 shows the distribution of *all the implicants* that were ever produced after 50 iterations. We can see that 93% of them are prime implicants produced by BOOM, which seemingly puts the rest (i.e., all the group implicants) into an unimportant minority. However, the distribution of implicants in the final (and thus also the best) solution is shown in Fig. 5. Here, these make only 58% of the solution, while the group implicants begin to play an important role. The most important observation is that FC-Min significantly contributes to the solution both by group implicants and PIs. The majority of implicants was found by BOOM, however we must consider significantly shorter runtime of FC-Min comparing to BOOM (especially the IR phase).

Let us note that the total number of implicants generated in 50 iterations was more than 40000 (in Fig. 4), the solution consisted of 516 implicants (in Fig. 5). Thus, we can claim that BOOM often produces many unnecessary PIs, while FC-Min produces a low number of implicants, which often could form a significant part of the solution. However, to reach the best results, running both the BOOM and FC-Min is required.



Figures 4, 5. Distribution of all the implicants and the implicants in the solution respectively

4.2. Comparison Results

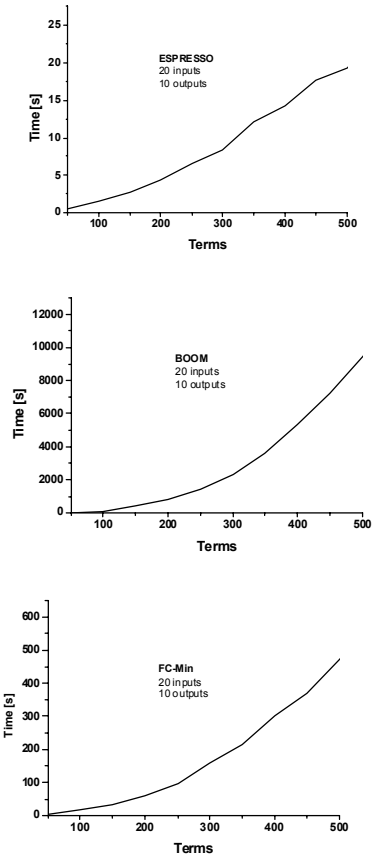
In order to estimate the efficiency of the method, we have compared the results obtained by different methods. The result quality and minimization runtime are compared in Table 1 (see the end of the paper). Several artificial benchmarks were run by ESPRESSO [1], BOOM, FC-Min, BOOM-II and BOOM-II where the implicant reduction phase was turned off. All the BOOM and FC-Min algorithms were run for 100 iterations. The sizes of benchmarks are indicated in the leftmost column by the number of input variables (n), output variables (m) and number of terms (p). The resulting circuit size is given in gate equivalents. For each instance size 20 benchmarks were processed and the obtained results averaged.

It can be seen that for circuits having many input variables BOOM significantly overpowers ESPRESSO both in time and result complexity. ESPRESSO is faster for benchmarks having many outputs, however BOOM yields better results for this case. BOOM-II efficiently combines the advantages of BOOM and FC-Min. A good result quality is retained from BOOM, while there is a significant speed-up given by-FC-Min. Moreover, additional speed-up is reached, when the implicant reduction phase of BOOM is omitted in BOOM-II. It can be seen that it is fully substituted by FC-Min, since there is no quality loss.

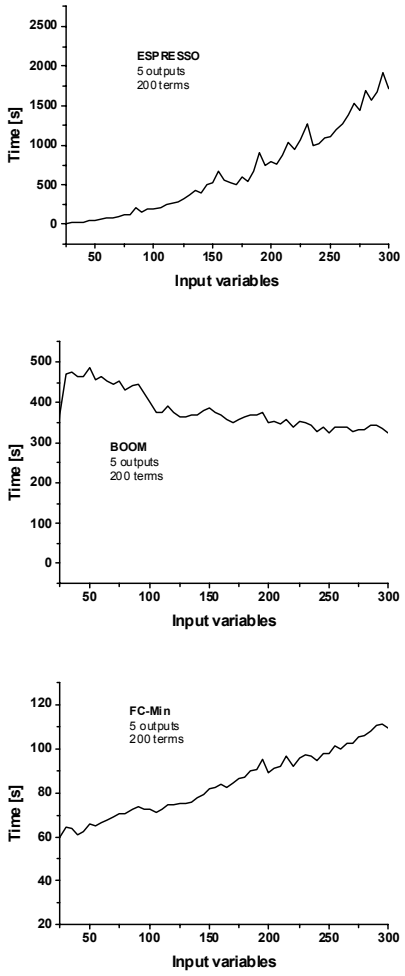
Notice that all the algorithms (except for ESPRESSO) were run for 100 iterations. If they were run for one iteration only, the runtime would be more than 100-times shorter, whereas the result quality only slightly worse.

4.3. BOOM-II Average Time Complexity

In order to properly estimate the dependency of the runtime on the instance sizes, we have minimized 20 different problem instances of the same size and averaged the results. The graphs in Figures 6a, b, c, show the average dependency of the runtime on the number of defined terms, for ESPRESSO (a), BOOM (b) and FC-Min (c). The values of the fixed parameters are indicated inside of the figures. It can be seen that the ESPRESSO complexity grows almost linearly, while the curves representing BOOM and FC-Min grow apparently faster. On the other hand, the runtime growth, with respect to the number of input variables, is very positive for BOOM and FC-Min, see Figures 7a, b, c. The ESPRESSO runtime grows almost exponentially, while for BOOM it remains almost constant. In FC-Min, the runtime grows linearly.



Figures 6a, 6b, 6c. Dependency on the number of terms



Figures 7a, 7b, 7c. Dependency on the number of input variables

5. Influencing BOOM-II Run

Sometimes it is advantageous to influence the minimization process to satisfy, or at least near to some technological constraints. Here are some of them:

Decomposition constraints. A two-level minimizer can "help" the subsequent decomposition phase to perform better. Particularly, when the circuit is to be divided into several stand-alone *blocks*, using, e.g., the bi-decomposition [16] or functional decomposition [18], these blocks should share minimum inputs.

The simplest case of such decomposition is the *single-level partitioning*. It is based on dividing the circuit into a given number of blocks, so that their two-level nature is retained. The blocks may share the primary inputs, each block generates several outputs. The SOP terms cannot be shared among the blocks. This is illustrated by Figure 8. Here a logic function

of 7 inputs x_1-x_7 and 6 outputs y_1-y_6 is decomposed into two 5-input and 3-output blocks while each block is a two-level (AND-OR) circuit.

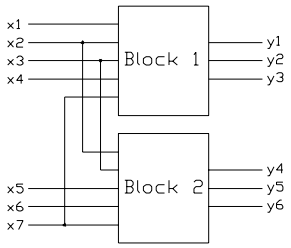


Figure 8. A single-level partitioning

In a case of functions having many input variables, there is a big freedom in choosing implicants (and consequently literals) to the solution. Thus, the minimization should be influenced so that the input variables are not shared between blocks. On the other hand, the FC-Min Find Cover phase can be modified to produce results satisfying the output grouping.

Design-for-testability (DFT). To synthesize an easily testable circuit, we try to reduce the sizes of the cones. Thus, the number of inputs driving each output variable should be minimized.

Load Balancing. In some cases it is desirable to design circuits with balanced loads of its inputs. This means that the number of branching of the circuit's inputs should be kept balanced. Moreover, for a low-power design the number of branching should be kept minimal.

5.1. BOOM Algorithm Modification

CD-Search. This is the essential phase that has to be modified. The algorithm is based on a gradual addition of literals into the terms. Recall that the candidate literals are being selected using a scoring function, which is the frequency of occurrence. Thus, it is easy to modify this scoring function to manifest the constraints.

For the partitioning purposes we modify the scoring function, so that the frequency of the literal that is already included in the processed block is multiplied by some factor, and is thus preferred to other literals. The higher this factor is, the smaller is the number of input variables entering the blocks.

For DFT purposes, further modification is very similar to the previous one: input variables that are already included in the current partial SOP form of the currently processed variable are preferred.

When applying the load balancing, we penalize variables entering other blocks.

Implicant Expansion. In this phase the literals are being removed from the terms. It could be modified to adopt some constraints as well, e.g., by preferring a removal of the literal that would yield a reduction of the number of inputs entering the block (for partitioning).

Implicant Reduction. Here, as well, many group implicants are being produced from one PI. It is possible to modify the scoring function defining the candidate literals for inclusion, however we have found that the effect of this modification is negligible.

5.2. FC-Min Algorithm Modification

Find-Coverage. Since this phase does not directly influence the selection of what literals would be included in the solution, its modification would be meaningless.

Implicant Generation. This phase is fully deterministic and cannot be influenced in any way.

Implicant Expansion. In this phase the number of literals in the final set of SOP forms is being significantly reduced, by up to 70%. Thus, here we can decide what literals will be included in the solution. For the partitioning based minimization, literals of input variables that are not included in the currently processed block are tried for removal at first, and only when no such a removal is possible, literals of variables that are entering the processed block are tried for removal.

For the load-balancing and minimization, literals of variables included in other blocks are removed first.

In a DFT design, we remove literals that are not included in current SOP forms of output variables, for which the currently processed term is an implicant.

5.3. Modification of CP Solution Algorithm

Modifying the CP solution algorithm is of a key importance to reach good results. Consider that the CP solver selects only a small number of implicants from a huge implicant pool and constructs the final solution. Thus, if the algorithm were not modified, it could spoil all the effort made in the previous phases.

In fact, any CP solver can be used, whereas only one condition has to be fulfilled: the solver has to be a greedy additive heuristic algorithm, i.e., the implicants have to be added to the solution one by one. Modifying an exact solver could also be possible, however it would complicate the construction of a cost function here. Moreover, the common state-space pruning techniques [19] cannot be used here.

For the partitioning purposes, the CP is solved for each block individually, so that we prevent sharing the implicants among the blocks. We will consider the CP

algorithm described in Subsection 3.3. To apply the partitioning, additional weights are assigned to the implicants, thus the weights modify the contributions. Input variables used in particular blocks are recorded during the process. The weights of the implicants are proportional to the number of new input variables they would add to the currently processed block if they were selected into the solution. The more input variables are newly added into a given block by a term, the less likely will be this term selected.

For the load minimization purposes the weights can be modified so that implicants containing inputs entering other blocks will be penalized.

5.4. Output Grouping

The FC-Min Find Cover phase directly derives group implicants, or respectively only the set of output variables they the implicants share. This information can be advantageously exploited to derive an *output grouping*. It is a set of output variables that should be grouped together in one block (see Fig. 8). Several function's outputs are generated by each of these blocks, but the blocks cannot share any internal signals. Then it will be advantageous to group the outputs into blocks in such a way, so that output functions sharing many implicants are grouped in one block.

A *Grouping Matrix* based approach has been proposed in [11]. It involves a symmetric matrix of dimensions [m, m], where m is the number of output variables. The value G[i, j] defines the strength binding the two output variables i and j together. The G matrix is constructed from the cover obtained in the *Find Cover* phase. Here a set of coverage masks is produced. The *coverage mask* is a binary vector representing a set of output variables sharing a group implicant. Its size is equal to m (number of outputs), a '1' value in the i-th position means that the i-th output variable is implicated.

At the beginning, the matrix is filled with zeros. Then all coverage masks are processed one by one. For each coverage mask and each two pairs (i, j) of '1's, the value of G[i, j] is increased by one. After processing all the fault masks, the G-matrix contains values describing the "*binding force*" of the output variables. The value of a cell G[i, j] represents the number of group implicants that are common to output variables i and j. Thus, outputs having high binding force should be more likely grouped together.

Thus, after the G-matrix is computed, distribution of the function's outputs among the blocks is generated. Outputs having the highest binding force should be grouped together, thus we process the matrix starting by finding the highest value (let it be G[i, j]) and grouping the respective two variables (i, j) together into one block. Then the second maximum value in the i-th and j-th rows (columns) is found and the respective variable is included into the block. This is being done until the maximum number of the block's outputs is exhausted. Then the search continues from the start, while the already assigned variables are removed from the matrix.

The precision of the G matrix can be improved by using a repetitive FC-Min, i.e., by recording many more implicants. For details see [11].

To illustrate the effectiveness of the proposed output grouping method, the results of the decomposition of several "hard" MCNC benchmarks are shown in Table 2. The benchmarks were decomposed into several blocks, each of them having approximately 10 outputs. In the first experiment, the outputs were distributed among the blocks randomly, in the second experiment the FC-Min based grouping was used. Then the divided two-level benchmarks were synthesized using SIS [12] and decomposed into 2-input NAND gates. After the benchmark name the number of its output is shown (m), then the number of blocks (B). The number of two-input NAND gates obtained after the synthesis, where the random grouping is used is shown next (*random*) and the results of the proposed method are presented in the *"FC-Min*" column. The percentage improvement is computed in the last column.

Table 2. Output grouping results

bench	*m*	*B*	*random*	*FC-Min*	*impr.*
duke2	29	3	676	517	24%
jbp	57	6	755	600	21%
mainpla	54	6	5742	4506	22%
mish	43	5	156	136	13%
misj	14	3	78	75	4%
soar	94	10	1424	970	32%
spla	46	5	1033	811	22 %
ti	72	7	1496	1173	22%
x2dn	56	6	331	206	38%

6. Conclusions

We have presented a flexible two-level Boolean minimizer constructed as a combination of two previously proposed methods. Each of the single methods excels for different problem sizes, and the nature of the solution obtained by the two algorithms differs as well. Joining them together in an adjustable manner allowed us to make a universal minimizer suitable for all kinds and sizes of problems. The time demanding implicant reduction phase can be often completely omitted and fully substituted by FC-Min. Criterion of the quality of the solution can be selected too, which makes BOOM-II a good minimizer for any hardware implementation of the circuit. The iterative

minimization allows us to find a trade-off between the runtime and the quality of the solution.

Individual minimization algorithms can be modified so that many technological constraints, like decomposition, DFT or low power design, can be satisfied. The FC-Min main phase is exploited to derive the output grouping, i.e., the main decomposition parameters.

The algorithm was tested on random and standard MCNC benchmarks.

Acknowledgement

This research was supported by a grant GA 102/04/0737 and MSM6840770014.

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

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

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

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

[7] M. Chatterjee and D.K. Pradhan, „A BIST Pattern Generator Design for Near-Perfect Fault Coverage", IEEE Transactions on Computers, vol. 52, no. 12, 2003, pp. 1543-1558

[8] 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

[9] 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

[10] P. Fišer and H. Kubátová, "Boolean Minimizer FC-Min: Coverage Finding Process", Proc. 30th Euromicro Symposium on Digital Systems Design (DSD'04), Rennes (FR), 31.8. - 3.9.04, pp. 152-159

[11] P. Fišer and H. Kubátová, "Output Grouping-Based Decomposition of Logic Functions", Proc. 8th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop 2005 (DDECS'05), Sopron, HU, 13.-16.4.2005, pp. 137-144

[12] E.M. Sentovich, et al.: "SIS: A System for Sequential Circuit Synthesis", Electronics Research Laboratory Memorandum No. UCB/ERL M92/41, University of California, Berkeley, CA 94720, 1992

[13] P. Fišer and H. Kubátová, "Single-Level Partitioning Support in BOOM-II", Proc. 2nd Descrete-Event System Design 2004 (DESDes'04), Dychów, Poland, 15.-17.9.04, pp. 149-154

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

[15] P. Fišer and H. Kubátová, "Two-Level Boolean Minimizer BOOM-II", Proc. 6th Int. Workshop on Boolean Problems (IWSBP'04), Freiberg, Germany, 23.-24.9.2004, pp. 221-228

[16] A. Mishchenko, B. Steinbach and M. A. Perkowski, "An algorithm for bi-decomposition of logic functions", Proc. DAC '01, pp. 103-108.

[17] O. Coudert: Two-level logic minimization: an overview, Integration, the VLSI journal, 17-2, pp. 97-140, Oct. 1994

[18] L. Józwiak, A. Chojnacki: High-quality Sub-function Construction in Functional Decomposition Based on Information Relationship Measures, DATE'2001 - Design, Automation, and Test in Europe Conference, Munich, Germany, 13-16 March, 2001, ISBN 0-7695-0993-2, IEEE Computer Society Press, Los Alamitos, CA, USA, 2000, pp. 383-390.

[19] E. I. Goldberg, L. P. Carloni, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Negative Thinking in Branch-and-Bound: the Case of Unate Covering. IEEE Transactions on Computer-Aided Design, 19(3), pp. 281-294, March 2000.

Table 1. Comparison results

bench	ESPRESSO		BOOM		FC-Min		BOOM-II		BOOM-II, no IR	
n / m / p	GEs	Time [s]	GEs	Time [s]	GEs	Time [s]	GEs	Time [s]	GEs	Time [s]
10 / 5 / 200	646.33	0.37	647.75	7.30	666.92	29.93	640.73	24.19	640.90	23.20
50 / 5 / 200	318.73	51.36	276.35	486.53	300.18	65.54	287.43	272.37	290.00	226.90
100 / 5 / 200	256.90	194.68	241.90	399.80	254.30	72.37	252.15	216.70	254.82	176.74
200 / 5 / 200	220.88	786.50	216.40	349.41	220.72	89.16	223.05	190.16	221.60	154.64
300 / 5 / 200	205.40	1717.60	206.68	323.84	205.57	109.50	209.18	193.45	206.97	151.13
20 / 10 / 50	179.07	0.44	140.80	8.16	149.40	3.14	141.60	7.73	141.43	7.02
20 / 10 / 200	855.45	4.44	711.23	806.33	799.80	60.50	720.70	617.11	719.50	549.58
20 / 10 / 300	1349.55	8.43	1151.72	2331.87	1291.28	157.61	1159.83	1694.60	1160.90	1558.31
20 / 10 / 400	1880.55	14.20	1611.60	5352.55	1817.08	300.37	1628.33	3890.65	1627.25	3488.69
20 / 10 / 500	2444.25	19.34	2106.10	9410.28	2381.10	473.23	2121.80	6910.24	2118.70	6572.62