Two-Level Boolean Minimizer BOOM-II

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

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 the required runtime and/or the quality of the solution. It is applicable to functions with an extremely large number of both input and output variables.

1. Introduction

The problem of two-level Boolean minimization is quite old, but surely not dead. It occurs in almost every area of the logic design, as is the design of control systems, design of build-in self-test (BIST) for VLSI circuits [1] and in the VLSI synthesis in general [2]. It has been studied for many decades and plenty of minimization methods and algorithmic minimizers were developed. In 50's the classical Quine-McCluskey method [3, 4] was proposed, and laid the basis for subsequent Boolean minimization algorithms. MINI [5], ESPRESSO [6] and its modifications [7] were proposed, 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.

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 process and the runtimes grow with the number of output variables rapidly as well. Lately we have developed an algorithm called FC-Min [11] solving this problem efficiently. The solution is being constructed from the necessary group implicants only, which makes the algorithm extremely fast and with low memory demands. On the other hand, FC-Min does not produce good results for function with a low number of output variables.

In this paper we propose a method where the FC-Min and the original BOOM algorithms are combined together to achieve better results. 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 makes the system a good minimizer for all kinds of problems. Since the system is a successor of BOOM to some extent, we have named it *BOOM-II*.

The paper has the following structure: Section 2 defines the problem statement, the structure and major principles of BOOM-II will be described in Section 3. Section 4 describes the experiments with BOOM-II. 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 are defined by a truth table. Thus, each function is specified by its on-set and off-set. 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* ***I***, the rows of the input matrix will be denoted as *input vectors*. The part defining the output values of the terms 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* (which means the number of care terms).

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 rest are don't cares. 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].

Our task is to synthesize a two-level circuit implementing the multi-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.

3. Principles of the Method

As it was stated in the Introduction, BOOM-II is a composition of two previously published minimization algorithms - BOOM [9, 10] and FC-Min [11]. 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 grows rapidly, and the algorithm begins to be less efficient as well. This is due to the demanding implicant reduction phase above all. BOOM is based on a generation of prime implicants (PIs), and thus it is strong for problems whose solution is consisted 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. 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, thus problems with many output variables. On the other hand, it is not suitable for functions with only 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 the fact that some minimization phases are driven by random events. Hence, two runs of the same algorithm on the same problem 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 run several times, while all the different 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 size of implicant set 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 (FC-Min and BOOM-II differs only in the amount of implicants produced; the shape of the curve is the same). Theoretically, the more implicants we have, the better the solution that can be found after solving the covering problem. In reality, 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 further. This fact can be observed in Fig. 1 (thick line).

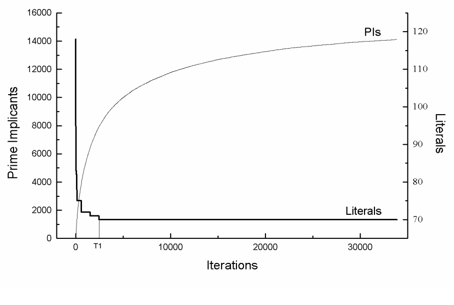


Figure 1: Growth of PI number and decrease of SOP length during iterative minimization

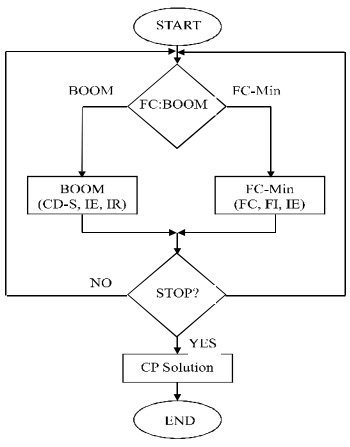


Figure 2: Flowchart BOOM-II

The idea of combining implicants from different minimization runs gave rise to BOOM-II. Same problem is solved both by BOOM and FC-Min (repeatedly), all the implicants are put together and the covering problem is solved at the end. The solution will be some combination of the implicants obtained from the two algorithms. Intuitively, prime implicants are more likely being picked up 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.

In order to enlighten the principles of BOOM-II and especially the differences of the two algorithms, we will briefly describe the major notions of the algorithms used.

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. The most important part of the algorithm, the *Coverage-Directed Search (CD-Search)*, generates a sufficient set of implicants needed for covering the on-set. The implicants are then passed to the *Implicant Expansion (IE)* phase, which converts them into PIs. The PIs are then being reduced in the *Implicant Reduction (IR)* phase 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 some previously constructed term. 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 $\boldsymbol{F}_i$. 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 that covers as many 1-terms as possible. To do this, we start implicant generation by selecting the most frequent input literal 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 in 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 adding literals until an implicant is generated, then we record it, remove 1-terms that are covered by this term, and start searching for other implicants. This algorithm is greedy and thus the obtained implicants need not be prime, so they have to be further expanded.

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

3.2. Principles of FC-Min

The FC-Min minimizer generates a 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 necessary irredundant set of implicants is found in order to cover the on-sets of all the functions. Such an approach might be very demanding (in time and space) for functions with a large number of input and output variables, since the number of both the prime and group implicants is often extremely large.

In FC-Min, the process of generating implicants is conducted in a reverse way. Firstly the cover of the on-sets that is 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. FC-Min does not produce any PIs, since the necessary group implicants are directly generated. As the group implicants are highly important especially for problems with many outputs, this makes FC-Min 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 because the cover of the on-set is being generated partially ad-hoc 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.

An example problem is shown in Fig. 3. Both the input and output matrices are shown here. The 5-input and 5-output function is defined by 10 care terms. An example of a rectangle cover of the **O** matrix is shown in Figure 4. Here all the “1”s are covered by six implicants t_1 - t_6.

```
0 11010 10000
1 10000 11100
2 01001 01100
3 01111 01010
4 00110 00111
5 01110 00000
6 10110 00011
7 00001 01101
8 10101 10111
9 11100 10100
```

Figure 3: The input and output matrices

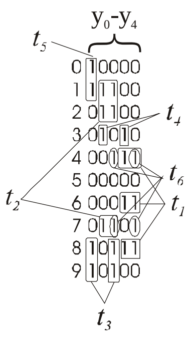


Figure 4: Rectangle cover of the output matrix

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 in the 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 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.

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 corresponding to the rows of the cover of t_i. Moreover, this supercube must not intersect any **I** matrix term that is not included in the particular cover, since it would cover some zeros then. Let us assume our example. The term t_1 covers vectors *4, 6* and *8*. Thus, the minimum term that can be a candidate for t_1 must be constructed as a minimum supercube of the terms *4, 6* and *8* in the input matrix, thus:

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

Figure 5: The implicant t_1

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

Figure 6: The final solution

The term `(-01--)` has been found as a candidate for an implicant t_1. Similarly, we will obtain the minimum implicants t_1 - t_6. Figure 6 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.

3.3. Covering Problem Solution

We saw in 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. With a large number of implicants it is impossible to obtain an exact solution, since it is an NP-hard problem, thus some heuristic must be used. Here a large number of implicants may misguide the CP solution algorithm and thereby lead to a non-minimal solution.

After an extensive testing we have decided for a greedy additive heuristic method based on computing the contributions (scoring functions) of terms as a criterion for their inclusion into the solution [15]. We construct a covering matrix *A*, its dimension will be denoted as (*r*, *s*). The columns correspond to the implicants, rows to the individual on-set terms that have 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

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

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

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

After that 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 Experimental Results

4.1. Standard MCNC Benchmarks

We have conducted a vast number of experiments to evaluate the performance and scalability of the BOOM-II system. In this subsection we will present a comparison of BOOM and FC-Min on several "harder" MCNC benchmarks [16]. Both the algorithms were run one iteration only. Here FC-Min always found a minimal solution, often in a shorter time than BOOM. Thus, presented results of BOOM-II would be meaningless, since it has to be run more than one iteration to take effect (BOOM and FC-Min is being alternated according to the *FC-Min:BOOM* ratio).

The results are presented in Table 1. The "*i / o / p*" column indicates the numbers of the benchmark's input and output variables and the number of care terms, the "*lit / out / terms*" shows the quality of the respective solution, in terms of the number of literals in the SOP form, the output cost and the number of product terms. The minimum solutions and smaller times are shadowed. It can be seen that running BOOM on these benchmarks would be ineffective (only a speedup is reached in some cases), however further experiments prove the contrary. For more details on the MCNC benchmarks see [10, 11], where the comparison with ESPRESSO results was presented.

All the experiments were conducted on an Athlon XP2500+ PC, Windows XP.

Table 1: MCNC Benchmarks

		BOOM		FC-Min	
bench	*i / o / p*	*time [s]*	*lit / out / terms*	*time [s]*	*lit / out / terms*
alcom	15 / 38 / 90	0.7	177 / 45 / 42	0.1	174 / 49 / 40
apex1	45 / 45 / 1440	38.4	1915 / 1025 / 229	15.1	1739 / 1103 / 206
apex2	39 / 3 / 1576	4.7	14489 / 1065 / 1041	17.3	14453 / 1075 / 1035
apex3	54 / 50 / 1036	13.0	2537 / 821 / 326	17.7	2270 / 1022 / 280
apex4	9 / 19 / 1907	2.9	4268 / 1426 / 530	20.5	3688 / 1731 / 436
apex5	117 / 88 / 2710	161.5	6089 / 1192 / 1088	242.6	6089 / 1192 / 1088
b4	33 / 23 / 680	1.8	472 / 96 / 59	0.4	437 / 109 / 54

		BOOM		FC-Min	
bench	*i / o / p*	*time [s]*	*lit / out / terms*	*time [s]*	*lit / out / terms*
chkn	29 / 7 / 370	0.4	1598 / 141 / 140	0.6	1598 / 141 / 140
cordic	23 / 2 / 2105	2.7	13825 / 914 / 914	15.3	13825 / 914 / 914
cps	24 / 109 / 855	11.1	2139 / 739 / 187	13.7	1890 / 946 / 163
e64	65 / 65 / 327	8.8	2145 / 65 / 65	0.2	2145 / 65 / 65
ex4	128 / 28 / 654	8.4	1649 / 279 / 279	6.7	1649 / 279 / 279
exep	30 / 63 / 643	2.8	1175 / 110 / 110	1.65	1175 / 110 / 110
ibm	48 / 17 / 499	0.8	882 / 173 / 173	1.2	882 / 173 / 173
signet	39/8/3627	0.9	500 / 143 / 122	4.6	490 / 146 / 119
soar	83/94/779	37.4	2570 / 508 / 379	17.7	2455 / 549 / 353

4.2. Randomly Generated Problems

As the second set of experiments randomly generated problems with varying *n* and *p* (number of inputs and care terms) were solved, the number of outputs was fixed to 15. For each problem size ten different instances were solved and the average of all the values computed. This measurement has been done in order to compare the quality of the final result. Each problem was solved by ESPRESSO first, and then by BOOM-II with different *FC-Min:BOOM* ratios, while the runtime was set equal to the runtime of ESPRESSO.

The results are shown in Table 2. The number of input variables (*i*) increases horizontally, the number of defined terms (*p*) vertically. The first line in each cell shows the ESPRESSO result. The first number indicates the runtime, the number of literals in the SOP form follows, the third number is the output cost and the number of product terms is the last one. The second row describes the result reached by running BOOM only (no FC-Min). The runtime is omitted here, since all the runtimes are equal. On the other hand, the number in brackets indicates the number of iterations processed, which is a good measure of the speed-up. The third row describes the situation where the *FC-Min:BOOM* ratio was set to 1:1. Finally, the last row shows the results of a pure FC-Min, thus without running BOOM.

The observations can be summarized as follows:

- With increasing *FC-Min:BOOM* ratio (towards FC-Min) the speedup increases. Even when only three extreme ratios were used (FC-Min only, 1:1 and BOOM only), we have observed that the runtime grows almost linearly with the ratio.
- FC-Min produces solutions with very few terms, especially for functions with many input variables (> 50), where BOOM-II outperforms ESPRESSO
- The number of literals decreases was well, mostly due to the decreasing number of terms.
- The output cost depends on the *FC-Min:BOOM* ratio only slightly, but it is always much lower than the output cost reached by ESPRESSO. In general, BOOM produces a solution with lower output cost. This is mainly due to the fact, that the solution is consisted of fewer group implicants.

Table 2: Randomly generated problems

p / n	25	50	100
50	2.15/233/346/49	10.80/218/324/48	51.96/204/309/47
	340/246/70(2)	294/189/61(7)	247/139/53(27)
	307/257/62(3)	269/190/53(11)	231/151/46(38)
	290/264/58(8)	252/185/50(28)	214/150/43(81)
75	5.62/400/513/74	34.37/370/463/70	154.71/357/438/68
	525/381/95(3)	466/276/86(12)	423/218/79(35)
	502/382/90(5)	433/280/76(18)	373/223/66(48)
	465/394/83(13)	404/279/71(47)	357/223/62(99)
100	11.24/581/673/99	84.48/546/586/92	416.29/520/564/90
	768/528/127(4)	665/358/111(16)	600/287/102(44)
	712/529/117(5)	594/362/96(24)	524/301/84(62)
	659/543/110(19)	571/365/92(63)	498/301/80(118)
125	17.75/773/845/123	157.19/706/722/113	895.25/657/700/110
	1010/616/160(4)	872/459/137(17)	765/359/122(52)
	950/632/149(6)	783/464/120(27)	674/377/102(69)
	868/674/138(22)	745/456/115(71)	650/374/99(137)

Entry format: ESPRESSO (1st line): time [s] / #of literals / output cost / #of implicants
Next 3 lines: #of literals / output cost / #of implicants (iterations)

It could have been apparent from this example, that a pure FC-Min always produces better results than BOOM (-II) and ESPRESSO. This is not true in general, especially for functions with a low number of inputs.

Let us consider an example single-output function with 25 input variables and 500 defined terms. The results of the same minimization process are shown in Table 3. The data format is retained from Table 2.

Table 3: Results for single-output function

21.93/881/111/111
793/98/98(33)
852/106/106(19)
981/124/124(15)

Here the results are completely different – FC-Min is much slower than BOOM and the result quality is much worse as well. Thus, a proper *FC-Min:BOOM* ratio must always be found (e.g., experimentally on the particular circuits). In general, FC-Min is more advantageous for functions with many outputs, BOOM for low-output functions.

4.3. 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 the implicants in the solution of some problem. Particularly, we have studied the origin of the implicants in the final solution, and analyzed which of the two major algorithms contributes to it at 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 six groups:

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

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

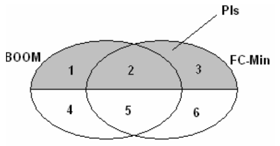
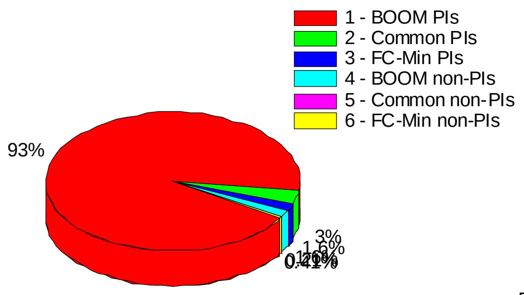


Figure 7: BOOM-II implicants

We have minimized a randomly generated function of 20 input variables, 20 outputs, 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 8 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. 9. 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. 8), the solution consisted of 516 implicants (in Fig. 9). 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 best results, running both the BOOM and FC-Min is required.



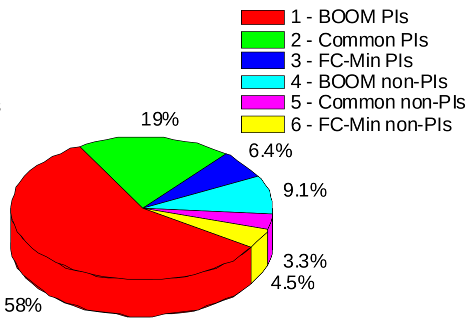


Figure 8, 9: Distribution of all the implicants and the implicants in the solution respectively

5. 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 in 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.

The BOOM-II minimizer can be downloaded for free from [17].

Acknowledgement

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

References

[1] 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

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

[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, 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, P. Fišer: BOOM - a Heuristic Boolean Minimizer, Proc. ICCAD-2001, San Jose, Cal. (USA), 4.-8.11.2001, 439-442

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

[11] P. Fišer, J. Hlavička, H. Kubátová: FC-Min: A Fast Multi-Output Boolean Minimizer, Proc. Euromicro Symposium on Digital Systems Design (DSD'03), Antalya (TR), 1.-6.9.2003, pp. 451-454

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

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

[14] P. Fišer, H. Kubátová: An Efficient Mixed-Mode BIST Technique, Proc. 7th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop 2004, Tatranská Lomnica, SK, 18.-21.4.2004, pp. 227-230

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

[16] ftp://ic.eecs.berkeley.edu

[17] http://service.felk.cvut.cz/vlsi/prj/BOOM/