A Heuristic Method of Two-Level Logic Synthesis

Jan Hlavička, Petr Fišer
Department of Computer Science and Engineering
Czech Technical University
Karlovo nám. 13, 121 35 Prague 2
e-mail: hlavicka@fel.cvut.cz, fiserp@fel.cvut.cz

ABSTRACT

An original method of logic function minimization is presented. Its implementation denoted as BOOM minimizer is described an evaluated. The method is applicable to the problems of logic design, artificial intelligence, software engineering and graph theory, etc.

The innovative feature of the proposed method consists above all in the top-down approach to implicant generation by inclusion of literals. The selection of these newly included literals is based on heuristics using the frequency of literal occurrence. The proposed method is efficient especially for functions with several hundreds of input variables, whose values are defined only for a relatively small part of their range. The method has been tested on several different problems including standard design benchmarks, but also on problems of a much larger dimension, generated randomly. These experiments proved that the new algorithm is very fast and that for large circuits it delivers better results than the state-of-the-art ESPRESSO.

Keywords: Boolean Minimization, Logic Design, Implicant Generation, Heuristics, Covering Problem, PLA

1. INTRODUCTION

The problem of two-level minimization of Boolean functions is by far not limited to the area of switching circuit design, where it was first identified [8, 12]. The new implementation technologies of digital circuits like, e.g., multi-level custom design, FPGAs, and above all the PLAs require in some phase this minimization, sometimes referred to as PLA minimization. The same problem is encountered in many other modern application areas like design of on-line real-time control systems, design of built-in self-test equipment for VLSI circuits, in mathematical problems like graph coloring [11], during solution of problems in the area of artificial intelligence, in software engineering, etc. These problems are mostly characterized by a large number of input variables, but by a limited number of input states for which the output value is determined (care states). The number of don't care states reaches then astronomical values, and the quality of a minimization method is thus determined by its ability to take advantage of their existence without enumerating them. Hence the proposed method accepts functions defined by on-set and off-set, whereas the don't care set is defined implicitly. An efficient minimization method must be further able to cope with the existence of large number of prime implicants (PIs) of the given function, while most of them are not needed for the minimal solution.

There are scores of texts treating the problem of Boolean minimization, hence it is impossible even to mention them. Some books like e.g. [1, 6] give a good survey of the important methods and their lists of references can be used as pointers to most of the important papers. Although these methods differ in many practical aspects, they mostly preserve the main feature of the original method, namely the use of two basic phases known as PI generation and covering problem (CP) solution. Some more modern methods, including the well-known ESPRESSO [1, 6] with its later improvements ESPRESSO-EXACT and ESPRESSO SIGNATURE [9], combine these two phases, reducing the number of implicants to be processed.

A sort of combination of PI generation with the solution of the CP, leading to a reduction in the total number of prime implicants generated, is also used in the BOOM (BOOlean Minimization) approach proposed here. However, the principal improvement consists in speeding up PI generation by applying the top-down approach instead of the commonly used bottom-up approach. Thus instead of starting from some 1-minterm (whose selection mostly represents the key innovative idea in different minimization methods), and increasing its dimension by deleting some literals, we reduce the *n*-dimensional hypercube by adding literals to its term, until it becomes an implicant of the given function. Another - maybe even more important - feature of the new method is the fact, that the terms (be it minterms or terms of higher dimension) given in the definition of the function to be minimized, are not used as a basis for the solution. This means that the minimal solution is not influenced by the primary solution and can be completely independent of any implicants delivered within the input data.

The basic principles of the proposed method were published in [7], the structure of the BOOM system in [4] and the implicant expansion method was presented in [5]. The present paper introduces some new amendments of the method introduced into the system in order to achieve a higher performance. BOOM was programmed in Borland C++ Builder and tested under MS Windows NT.

The paper has the following structure. After a formal problem statement in Section 2, the structure of the BOOM system is presented in Section 3 together with the most important algorithms used for the solution of individual phases of the minimization. The results of experimental verification of the BOOM system are evaluated and commented in Section 4.

2. THE PROBLEM OF LOGIC SYNTHESIS

We will be solving the standard problem of two-level minimization [1, 6]. A Boolean function of n input variables is defined by a truth table describing the **on-set** $F(x_1, x_2, \dots x_n)$ and **off-set** $R(x_1, x_2, \dots x_n)$. Here the on-set (off-set) is the set of terms to which the output value 1 (0) is assigned. We will assume that both minterms and terms of higher dimension may be used for defining the on-set and off-set. This means that individual lines of the truth table may contain don't care entries in the input portion. The terms not represented in the input field of the truth table are implicitly assigned don't care values of the output function, i.e., they represent the **don't care set** $D(x_1, x_2, \dots x_n)$. Listing the two care sets instead of an on-set and a don't care set, which is usual, e.g., in MCNC benchmarks, is more practical for problems where n is of the order of hundreds, because there the size of the don't care set usually largely exceeds all other sets.

Our task is to formulate a synthesis algorithm producing a sum-of-products expression $G = g_1+g_2+\dots+g_t$, where $F \subseteq G \subseteq F+D$ and t is minimal. In case of a set of m functions we will minimize the total number of implicants of all functions, while some of them can be used for more output functions. According to this specification, the number of product terms (implicants) is used as a universal quality criterion. This is mostly justified, but it should be kept in mind that the measure of minimality must correspond to the needs of the intended application. ESPRESSO uses the sum of the number of literals and the number of inputs into all output OR gates (also denoted as the output cost).

3. PRINCIPLE OF THE METHOD

3.1. BOOM Structure

Like most other Boolean minimization algorithms, BOOM consists of two major phases: **generation of implicants** (PIs for single-output functions, group implicants for multi-output functions) and the subsequent **solution of the covering problem**. The generation of implicants consists of two steps: first the **Coverage-Directed Search (CD-Search)** generates a sufficient set of implicants needed for covering the given function and these are then passed to the **Implicant Expansion (IE)** phase, which converts them into PIs.

The BOOM system improves the quality of the solution by repeating the implicant generation phase several times and recording all different implicants that were found. At the end of each iteration we have a set of implicants that is sufficient for covering the function. In each following iteration, another sufficient set is generated and new implicants are added to the previous ones (if the solutions are not equal). After that the covering problem is solved using all obtained primes using the heuristic method suggested in [3, 14].

3.2. Coverage-Directed Search

The idea of combining implicant generation with the covering problem solution gave rise to the coverage-directed search (CD-search) method used in the BOOM system. This consists in a directed search for the most suitable literals that should be added to some previously constructed term. Thus instead of increasing the dimension of an implicant starting from a 1-minterm, we reduce the n-dimensional hypercube by adding literals to its term, until it becomes an implicant of the given function. This happens at the moment when this hypercube does not intersect with any 0-term.

The implicant generation method aims at finding a hypercube 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. The selected literal describes an *n-1* dimensional hypercube, which may be an implicant, if it does not intersect with any 0-term. If there are some 0-minterms covered, we add one literal and verify whether the new term already corresponds to an implicant by comparing it with all 0-terms. We continue adding literals until an implicant is generated, then we record it and start searching for other implicants.

During the CD-search, the key factor is the efficient selection of literals to be included into the term under construction. After each literal selection we temporarily remove from the on-set the terms that cannot be covered by any term containing the selected literal. These are the terms containing that literal with the opposite polarity. In the remaining on-set we find the most frequent literal and include it into the previously found product term. Again we compare this term with all 0-terms and check if it is an implicant. After obtaining an implicant, we remove from the original on-set the terms covered by this implicant. Thus we obtain a reduced on-set containing only uncovered terms. Now we repeat the procedure from the beginning and apply it to the uncovered terms, selecting the next most frequently used literal, until the next implicant is generated. In this way we generate new implicants, until the whole on-set is covered. The output of this algorithm is a set of product terms covering all 1-terms and not intersecting with any 0-term.

When selecting the most frequent literal, it may happen that two or more literals have the same frequency of occurrence. In these cases we select a literal that makes an implicant from the current term. When there are still more possibilities to choose from, one is selected at random.

3.3. Implicant Expansion (IE)

The disadvantage of the CD-search is that it is greedy and the constructed implicants need not be prime. To increase the chance that fewer implicants will be needed to cover all 1-terms of the given function, we have to increase their size by IE, which means by removing literals (variables) from their terms. When no literal can be removed from the term any more, we get a PI.

The expansion of implicants into PIs can be done by several methods differing in complexity and quality of results obtained. We tested several approaches, from the simplest sequential search (which is linear) to the most complex exhaustive (exponential) search.

A **sequential Search** systematically tries to remove from each term all literals one by one, whereas the first literal is chosen randomly. Every removal is made permanent if no 0-minterm is covered. Only one PI is generated from each implicant, even if it could yield more PIs. A Sequential Search obviously does not reduce the number of product terms. On the other hand, experimental results show that it reduces the number of literals by approximately 25%.

With a **Multiple Sequential Search** we try all possible starting positions within an implicant, which thus leads to expansion into several PIs. This method produces more primes than a Sequential Search, while the time complexity is acceptable.

Even the Multiple Sequential Search algorithm cannot expand an implicant into all possible PIs. To do so, an **Exhaustive Implicant Expansion** must be used. Using recursion or queue, all possible literal removals are then tried until all primes are obtained. Unfortunately, the complexity of this algorithm is exponential.

3.4. Minimizing Multi-Output Functions

To minimize multi-output functions, only a few modifications of the algorithm need to be made. First, each of the output functions is treated separately: the CD-search and IE phases are performed. After that, we have a set of PIs sufficient for covering all *m* functions. However, to obtain the minimum solution we may need group implicants, i.e., implicants of more than one output function that are not primes of any. Here, the next part of minimization – **Implicant Reduction** - takes place.

All obtained primes are tried for reduction (by adding some literals) in order to become implicants of more output functions. The method of implicant reduction is similar to a CD-search. Literals are repetitively added to each term until there is no chance that the implicant will be used for more functions. We prefer literals that prevent intersecting with most of the terms of the off-sets of all functions (i.e., covering the least zeros). When no further reduction yields any possible improvement, the reduction is stopped and the implicant is recorded. If a term no longer intersects with the off-set of any function, it becomes its implicant. All implicants that were ever found are stored, assigned to the output functions and then the **Group Covering Problem** is solved.

As a solution of the covering problem we get a set of implicants needed to cover all output functions. For each output we may find all implicants that do not intersect the off-set of the output function. However, to generate the required output values, some of these implicants may not be necessary. These implicants would create redundant inputs into the output OR gates. Sometimes this is harmless (e.g., in PLAs), or it can even prevent hazards. Nevertheless, for hardware-independent minimization the redundant outputs should be removed. This is done at the end of the minimization by solving *m* covering problems once again (for each output function independently).

3.5. Iterative Minimization

Most current heuristic Boolean minimization tools, including ESPRESSO, use deterministic algorithms. Here the minimization process always leads to the same solution, never mind how many times it is repeated. On the contrary, in the BOOM system the result of minimization depends to a certain extent on random events, because when there are several equal possibilities to choose from, the decision is made randomly. Thus there is a chance that repeated application of the same procedure to the same problem would yield different solutions.

The iterative minimization concept takes advantage of the fact that each iteration produces a new set of implicants satisfactory for covering all minterms of all output functions. The set of implicants gradually grows until a maximum reachable set is obtained. The typical growth of the size of a PI 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 problem with 20 input variables and 200 minterms. Theoretically, the more primes we have, the better the solution that can be found, but the maximum set of primes is often extremely large. In reality, the quality of the final solution improves rapidly during the 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).

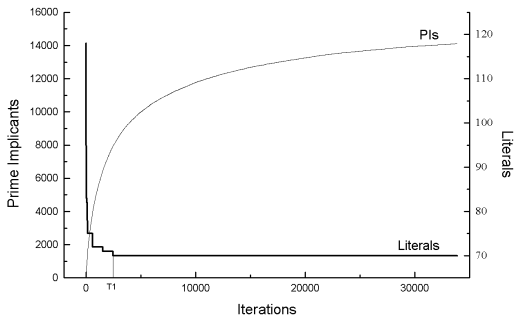

Fig. 1: *Growth of PI number and decrease of SOP length during iterative minimization*

From the curves in Fig. 1 it is obvious that selecting a suitable moment T1 for terminating the iterative process is of key importance for the efficiency of the minimization. The approximate position of the stopping point can be found by observing the relative change of the solution quality during several consecutive iterations. If the solution does not change during a certain number of iterations (e.g., twice as many iterations as were needed for the last improvement), the minimization is stopped. The amount of elapsed time may be used as an emergency exit for the case of unexpected problem size and complexity.

3.6 Accelerating Iterative Minimization

When the CD-search phase is repeated, identical implicants are quite often generated in various iterations. These are then passed to the Implicant Expansion phase, which might be unnecessarily repeated. To prevent this, all implicants that were ever produced by the CD-search are stored in the I-buffer (Implicant buffer). Each new implicant is looked up in this buffer, and if it is already present its further processing is stopped. A schematic plan of the whole minimization algorithm for a multi-output function is shown in Fig. 2.

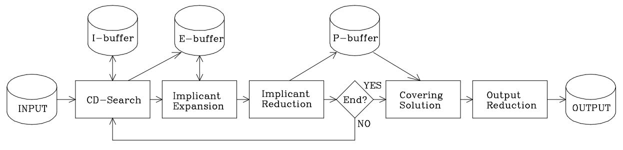

Fig. 2: *Schematic plan of iterative minimization*

First, the CD-search generates the set of implicants necessary for covering the function. These are looked up in the I-buffer. Implicants that are not present there are stored both in the I-buffer and E-buffer (Expansion buffer). Implicants already present are discarded. The E-buffer serves as a storage of implicants that are candidates for expansion into PIs. After expansion, the implicants are removed from the E-buffer. Then they are reduced to group implicants and the newly created group implicants are stored in the P-buffer (after duplicity and dominance checks). Finally, the covering problem is solved using the primes from the P-buffer.

The main implementation requirement for the I-buffer is its high look-up speed. Thus it was implemented as a ternary tree whose depth is equal to *n*. At the *k*-th level of the tree the direction is chosen according to the polarity (0,1,-) of the *k*-th variable in the searched term. The presence of a term is represented by the existence of its corresponding leaf. The tree is dynamically constructed during the addition of implicants. An example of such a tree is shown in Fig. 3.

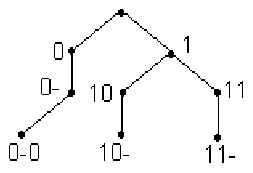

Fig. 3: *I-buffer tree example*

The example shows the structure of a three-variable I-buffer containing terms 0-0, 10- and 11-. If e.g., term 0-1 is looked for, the search will fail in the node 0- where no path leading to 0-1 is present. The maximum number of steps needed to look up or to insert a term is equal to *n*. The E-buffer and P-buffer are represented as a linear linked list.

4. EXPERIMENTAL RESULTS

Extensive experimental work was done to evaluate the efficiency of the proposed algorithm, especially for problems of large dimensions. Both runtime in seconds and result quality were evaluated. The processor used was a Celeron 433 MHz with 160 MB RAM. The quality of the results was measured by three parameters: total number of literals, output cost and number of product terms (implicants). Three groups of experiments, listed in the following three subsections, were performed.

4.1. Solution of MCNC Benchmark Problems

First a group of MCNC benchmark problems was solved by ESPRESSO 2.3 [16] and by BOOM [15]. The results of the comparison are shown in Tab. 1. The column *n*/*m*/*p* contains the parameters of the problem, namely the number of inputs, outputs and care terms. The benchmarks appearing in the table were solved by BOOM in one pass, hence the runtimes are very short (the 0.01 sec. value in most cases indicates a non-measurable runtime). Tab. 1 shows that problems with a large number of defined terms (*p*) were often solved by ESPRESSO in shorter time. This is due to the quadratic dependence of runtime on the number of terms in BOOM (see Subsection 4.4). In all these examples the quality of solutions was equal, in one case BOOM gave even better result than ESPRESSO. These solutions reached by BOOM and ESPRESSO are probably the minimum ones.

Tab. 1. MCNC Benchmark problems

		ESPRESSO		BOOM	
Bench	*n*/*m*/*p*	time	lit/out/impl	time	lit/out/impl
9sym	9/1/158	0.12	516/86/86	0.05	516/86/86
al2	16/47/139	0.15	324/103/66	0.66	324/103/66
Alu1	12/8/39	0.10	41/19/19	0.01	41/19/19
Alu2	10/8/241	0.20	268/79/68	0.04	268/79/68
b9	16/5/292	0.18	754/119/119	0.25	754/119/119
br1	12/8/107	0.12	206/48/19	0.02	206/48/19
br2	12/8/83	0.11	134/38/13	0.01	134/38/13
Clpl	11/5/40	0.12	55/20/20	0.01	55/20/20
Con1	7/2/18	0.10	23/9/9	0.01	23/9/9
dc1	4/7/25	0.12	27/27/9	0.01	27/27/9
dc2	8/7/101	0.13	207/52/39	0.01	206/51/39
dk27	9/9/24	0.10	31/15/10	0.01	31/15/10
dk48	15/17/64	0.24	115/28/22	0.02	115/28/22
ex7	16/5/292	0.19	754/119/119	0.22	754/119/119
in7	26/10/142	0.14	337/90/54	0.13	337/90/54
Max46	9/1/155	0.14	395/46/46	0.03	395/46/46
Misex1	8/7/41	0.12	51/45/12	0.01	51/45/12
Newpla	12/10/60	0.14	74/28/17	0.02	74/28/17
Newpla1	17/2/25	0.15	64/12/10	0.01	64/12/10
Newpla2	10/4/26	0.18	42/7/7	0.01	42/7/7
Newbyte	5/8/16	0.17	40/8/8	0.01	40/8/8
Newcond	11/2/72	0.16	208/31/31	0.01	208/31/31
Newcwp	4/5/24	0.18	31/19/11	0.01	31/19/11
Newill	8/1/18	0.13	42/8/8	0.01	42/8/8
Newtag	8/1/12	0.16	18/8/8	0.01	18/8/8
Newtpla	15/5/63	0.15	176/23/23	0.01	176/23/23
Newtpla1	10/2/15	0.16	33/4/4	0.01	33/4/4
Newtpla2	10/4/26	0.19	54/15/9	0.01	54/15/9
p82	5/14/74	0.17	93/56/21	0.02	93/56/21
rd53	5/3/67	0.09	140/35/31	0.01	140/35/31
rd73	7/3/274	0.14	756/147/127	0.08	756/147/127
sao2	10/4/137	0.11	421/75/58	0.04	421/75/58
sqrt8	8/4/66	0.11	144/44/38	0.01	144/44/38
squar5	5/8/65	0.12	87/32/25	0.01	87/32/25
vg2	25/8/304	0.15	804/110/110	0.47	804/110/110
xor5	5/1/32	0.08	80/16/16	0.01	80/16/16

4.2. Test Problems with *n*>100

The MCNC benchmarks have relatively few input terms and few input variables (*n* never exceeds 128) and also have a small number of don’t care terms. In order to compare the performance and result quality achieved by the minimization programs on larger problems, a set of problems with up to 300 input variables and up to 300 minterms were solved. The truth tables were generated by a random number generator, for which only the number of input variables, number of care terms and number of don’t cares in the input portion of the truth table were specified. The number of outputs was set equal to 5. The on-set and off-set of each function were kept approximately of the same size. First, the problem was solved by ESPRESSO and then by BOOM, which ran until the solution of the same or better quality was reached. The quality criterion selected was the sum of the number of literals and the output cost. For all samples the same or better solution was found by BOOM in much shorter time than by ESPRESSO.

Tab. 2. *Solution of problems with n>100*

p/n	100	150	200	250	300
50	92/0.1 (1) 92/7.2	83/0.1 (1) 84/20.0	77/0.6 (4) 88/42.8	77/0.4 (2) 77/51.3	75/8.7 (35) 76/110.7
100	190/2.6 (7) 190/28.0	174/4.2 (9) 176/104.4	163/31.1 (35) 165/114.7	155/14.7 (19) 158/184.3	154/1.4 (2) 154/317.4
150	287/9.4 (10) 287/79.5	289/1.1 (1) 289/129.2	249/31.2 (20) 253/367.2	231/57.4 (29) 233/396.0	247/44.7 (19) 248/569.4
200	401/37.8 (15) 404/209.3	349/92.0 (25) 350/297.2	344/63.2 (20) 347/557.5	331/2.3 (1) 334/795.0	321/2.9 (1) 328/857.2
250	460/242.3 (36) 463/323.3	443/142.7 (23) 450/404.1	409/481.6 (50) 445/934.1	423/196.6 (27) 425/1607.5	385/507.2 (52) 389/2354.2
300	580/203.1 (22) 588/333.9	505/446.4 (38) 508/798.8	506/416.0 (34) 512/847.1	500/470.9 (38) 500/1822.0	465/205.8 (32) 466/3012.9

Entry format: BOOM: #of literals+output cost/time in seconds (# of iterations)

ESPRESSO: #of literals+output cost/time in seconds

4.3. Solution of Very Large Problems

A third group of experiments aims at establishing the limits of applicability of BOOM. For this purpose, a set of 10-output functions with up to 1000 input variables and 2000 defined minterms was generated and solved by BOOM. For problems with more than 300 input variables ESPRESSO cannot be used at all. Hence when investigating the limits of applicability of BOOM, it was not possible to verify the results by any other method. The results of this test are listed in Tab. 3, where the time in seconds needed to complete one iteration for various problem sizes is shown.

Tab. 3. *Time for one iteration on very large problems*

p/n	200	400	600	800	1000
200	3.67	6.26	9.87	12.79	30.40
400	17.25	28.25	45.44	59.32	156.38
600	42.66	76.54	133.37	235.19	379.94
800	91.77	168.67	300.23	379.36	816.28
1000	157.26	323.58	617.04	781.77	1101.85
1200	325.54	536.09	784.27	970.91	1182.07
1400	492.28	888.56	1181.41	1617.84	1785.01
1600	736.24	1167.49	1606.09	2064.99	2559.53
1800	988.79	1778.00	2457.84	2749.45	3437.51
2000	1488.81	2269.78	3339.00	4107.73	4835.20

4.4. Time Complexity Evaluation

As for most heuristic and iterative algorithms, it is difficult to evaluate the time complexity of the proposed algorithm exactly. We have observed the average time needed to complete one pass of the algorithm for various sizes of functions. For simplicity, only single-output functions are studied here. Fig. 4 shows the growth of an average runtime as a function of the number of care minterms (20-300) where the number of input variables is changed as a parameter (20-300). The curves in Fig. 4 can be approximated with the square of the number of care minterms. Fig. 5 shows the runtime growth depending on the number of input variables (20-300) for various numbers of defined minterms (20-300). Although there are some fluctuations due to the low number of samples, the time complexity is almost linear.

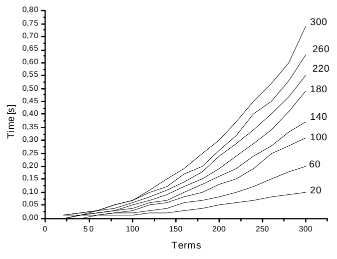

Fig. 4: *Time complexity (1)*

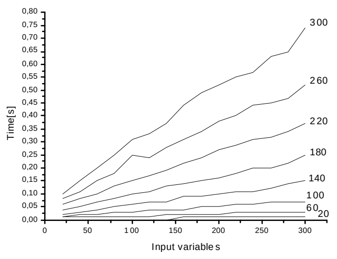

Fig. 5: *Time complexity (2)*

6. CONCLUSIONS

The proposed minimization method has several specific features. The function to be minimized is defined by its on-set and off-set. Thus the don't care set, which normally represents the dominant part of the truth table, need not be specified explicitly. The entries in the truth table may be minterms or terms of higher dimensions. The implicants of the function are constructed by reduction of *n*-dimensional cubes; hence the terms contained in the original truth table are not used as a basis for the final solution.

The properties of the BOOM minimization tool were demonstrated. Its application is advantageous above all for problems with large dimensions and a large number of don't care states where it beats other methods, like ESPRESSO, both in minimality of the result and in runtime. The PI generation method is very fast, hence it can easily be used in an iterative manner. However, in most cases it finds the minimum solution already in one iteration. For example, for most of the standard benchmark problems the runtime needed to find the minimum solution on a common PC was non-measurable. The dimension of the problems solved can be easily increased over 1000, because the runtime grows linearly with the number of input variables. For problems of very high dimension, the success largely depends on the size of the care set. This is due to the fact that the runtime grows roughly with the square of the size of the care set.

The BOOM minimizer has been placed on a web page [15], from where it can be downloaded by anybody who wants to use it.

Acknowledgment

This research was in part supported by the grant of the Czech Grant Agency GACR 102/99/1017.

REFERENCES

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

[2] Coudert, O. - Madre, J.C.: Implicit and incremental computation of primes and essential primes of Boolean functions, In Proc. of the Design Automation Conf. (Anaheim, CA, June 1992), pp. 36-39

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

[4] Fišer, P. – Hlavička, J.: Efficient minimization method for incompletely defined Boolean functions, Proc. 4th Int. Workshop on Boolean Problems, Freiberg, (Germany), Sept. 21-22, 2000, pp. 91-98

[5] Fišer, P. – Hlavička, J.: Implicant expansion method used in the BOOM Minimizer. Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS’01), Gyor (Hungary), 18-20.4.2001 (in print)

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

[7] Hlavička, J. - Fišer, P.: Algorithm for minimization of partial Boolean functions. Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS'00), Smolenice, (Slovakia) 5-7.4.2000, pp.130-133

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

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

[10] Nguyen, L. – Perkowski, M. – Goldstein, N.: Palmini – fast Boolean minimizer for personal computers. In Proc. DAC’87, pp.615-621

[11] Ostapko, D.L. - Hong, S.J.: Generating test examples for heuristic Boolean minimization. IBM Journal of Res. & Dev., Sept. 1974, pp. 459-464

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

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

[14] Rudell, R.L.: Logic Synthesis for VLSI Design, PhD Thesis, UCB/ERL M89/49, 1989

[15] http://cs.felk.cvut.cz/~fiserp/boom/

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

[17] ftp://ic.eecs.berkeley.org