Implicant Expansion Methods Used in The Boom Minimizer

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

Abstract

The principles and functional properties of a new Boolean minimization system BOOM are presented. The paper concentrates above all on one of the key phases of the minimization, namely the implicant expansion. Different expansion techniques and strategies are presented and compared. Their properties are evaluated above all from the point of view of overall performance measured by the quality of the result and the time needed to reach it.

The resulting minimization method is efficient especially for functions with several hundreds of input variables, whose values are defined only for a small part of their range. The BOOM minimization tool has been tested on an extensive set of problems, which proved that for large problems the new algorithm delivers better results than the state-of-the-art ESPRESSO, and that the size of problems to which it is applicable by far exceeds that of ESPRESSO.

1. INTRODUCTION

The problem of two-level minimization of Boolean functions is old, but surely not dead. It is encountered in many design environments, e.g., multi-level logical design, FPGA design, etc. In addition, it is a tool for the solution of problems in the area of artificial intelligence, software engineering, etc. The minimization methods started with the papers by Quine and McCluskey [6], [9], which formed the basic two phases known as prime implicant (PI) generation and covering problem (CP) solution. Some more modern methods, including the well-known ESPRESSO [13], [4] try to combine these. This is motivated above all by the fact that the problems encountered in modern application areas like design of control systems, design of built-in self-test equipment, etc., often require minimization of functions with hundreds of input variables, where the number of PIs is prohibitively large. Also the number of don't care states is mostly so large that modern minimization methods must be able to take advantage of all don't care states without enumerating them.

One of the most successful Boolean minimization methods is ESPRESSO and its later improvements. The original ESPRESSO generates near-minimal solutions, as can be seen from the comparison with the results obtained by using alternative methods – see Section 6. ESPRESSO-EXACT [10] was developed in order to improve the quality of the results. The improvement consisted above all in combining the PI generation with set covering. Finally, ESPRESSO-SIGNATURE [7] was developed, accelerating the minimization by reducing the number of prime implicants to be processed by introducing the concept of a "signature".

A combination of PI generation with solution of the CP, leading to a reduction of the total number of PIs generated, is also used in the BOOM (BOOlean Minimization) approach proposed here. The most important difference between the approaches of ESPRESSO and BOOM is the way they work with the on-set received as function definition. ESPRESSO uses it as an initial solution, which has to be modified (improved) by expansions, reductions, etc. BOOM, on the other hand, uses the input sets (on-set and off-set) only as a reference, which determines whether a tentative solution is correct or not. This allows us to remain to a great extent independent of the properties of the original function coverage. The second main difference is the top-down approach in generating implicants. Instead of expanding the source cubes in order to obtain better coverage, BOOM reduces the universal hypercube until it no longer intersects the off-set while the coverage of the source function is satisfied. The basic principles of the proposed method and the BOOM algorithms were published in some previous reports [3], [5]. The present BOOM system was constructed using the same approach, but we added some new heuristics allowing us to control the iterative mode in order to meet the quality requirements and runtime limitations. BOOM was programmed in Borland C++ Builder and tested under MS Windows NT.

This paper has the following structure. After a formal problem statement in Section 2, the principles of the proposed method and its implementation in the BOOM system are described in Section 3. The initial generation of implicants is described in Section 4 and their expansion into prime implicants is studied in Section 5. Experimental results are evaluated and commented in Section 6.

2. PROBLEM STATEMENT

Let us have a set of m Boolean functions of n input variables $F_1(x_1, x_2, \dots x_n)$, $F_2(x_1, x_2, \dots x_n)$, … $F_m(x_1, x_2, \dots x_n)$, whose output values are defined by truth tables. These truth tables describe the **on-set** $F_i(x_1, x_2, \dots x_n)$ and **off-set** $R_i(x_1, x_2, \dots x_n)$ for each of the functions F_i. The terms not represented in the input field of the truth table are implicitly assigned don't care

values for the corresponding output function, i.e., they represent the **don't care set** $D_i(x_1, x_2, \ldots x_n)$. 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 with a large number of input variables, because in these cases the size of the don't care set largely exceeds the two care sets. We will assume that n is of the order of hundreds and that only a few of the 2^n minterms have an output value assigned, i.e., the majority of the minterms are don't care states.

Our task is to formulate a synthesis algorithm which will for each output function F_i produce a sum-of-products expression $G_i=g_{1i}+g_{2i}+\ldots+g_{ti}$, where $F_i \subseteq G_i$ and $G_i \cap R_i = \varnothing$. The expression G should be kept minimal, whereas the criterion of minimality (number of product terms, number of literals, output cost, etc.) can be chosen in accordance with the intended application.

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 source 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 several times, and records all different implicants that were found. Then the CP is solved using all obtained primes. A simple algorithm was used for solving the CP at this stage. It is based on some heuristics, which prefer PIs covering 1-terms covered by the lowest number of other implicants and those covering the highest number of yet uncovered 1-terms.

Multi-output functions are minimized in a similar manner. Each of the output functions is first treated separately; the CD-search and IE phases are performed in order to produce primes covering all output functions. However, to obtain the minimal solution, we may need implicants of more than one output function that are not primes of any. Here, **Implicant Reduction** takes place. Then the **Group Covering Problem** is solved and **Output Reduction** (corresponding to the ESPRESSO's MAKE_SPARSE procedure [4]) is performed.

3.2 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. In reality, the quality of the final solution improves rapidly during the first few iterations and then remains unchanged. This fact can be observed in Fig. 1 (thick line).

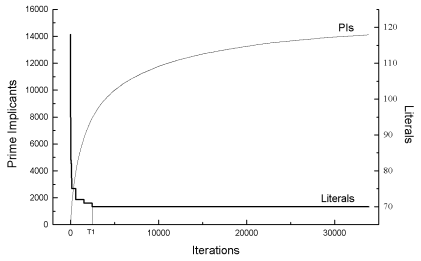

Fig. 1: Growth of PI number and decrease of SOP length during iterative minimization

The whole iterative minimization process can be described by the following pseudo-code. The inputs are the on-sets F_i and off-sets R_i of the m functions, the output is a minimized disjunctive form G of all F_i.

```
BOOM(F[1..n], R[1..n]) {
       G = ∅
       do
              I = ∅
              for (i = 1; i < n; i++)
                     I′ = CD_Search(F[i], R[i])
                     Expand(I′)
                     Reduce(I′, R[1..n])
                     I = I ∪ I′
              G′ = Group_cover(I, F[1..n])
              Reduce_output(G′, F[1..n])
              if (Better(G′, G)) then G = G′
       until (stop)
       return G
}
```

3.3 Accelerating Iterative Minimization

When the CD-search phase is repeated, identical implicants are quite often generated in different 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 flow diagram of the whole minimization algorithm for a multi-output function is shown in Fig. 2.

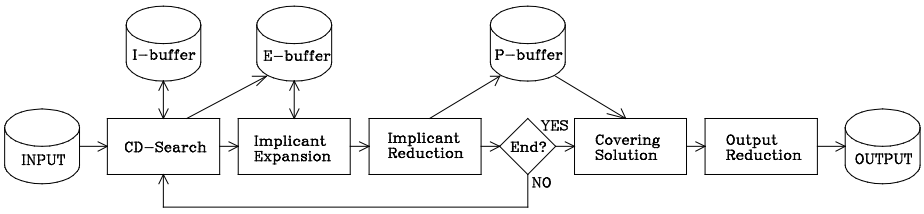

Fig. 2: Iterative minimization schematic plan

The CD-search generates the set of implicants necessary for covering the function. Each implicant is first looked up in the I-buffer and, if it is not present, it is stored both in the I-buffer and E-buffer (Expansion buffer). Otherwise it is discarded. The E-buffer serves as a storage of implicants that are candidates for expansion into PIs and after expansion they are removed. Then primes are reduced to group implicants and stored in the P-buffer. Finally, the covering problem is solved using the implicants from the P-buffer.

The main implementation requirement for the I-buffer is its high look-up speed, hence it is structured 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.

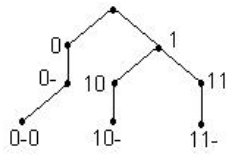

Fig. 3: I-buffer tree structure

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 linear linked lists.

4. 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 an *n* dimensional hypercube by adding literals to its term, until it becomes an implicant of F_i. 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. 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 those terms that are 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 does not intersect with any 0-term.

The basic CD-search algorithm can be described by the following function in pseudo-code. The inputs are the on-set (F) and the off-set (R), and the output is the sum of products (H) that covers the given on-set.

```
CD_Search(F,R) {
        H = ∅                   // H is the cube that is being created
        do
                F′ = F          // F′ is the reduced on-set
                t = 1           // t is the term in progress
                do
                        l = most_frequent_literal(F′)
                        t = t * l
                        F′ = F′ – cubes_not_including(t)
                while (t ∩ R ≠ ∅)
                H = H ∪ t
                F = F - F′
        until (F == ∅)
        return H
}
```

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 can change the current term into an implicant. When there are still more possibilities to choose from, we select one at random. A detailed example of the CD-search algorithm was published in [3].

The fact that the input file may contain both 1-minterms and 1-terms of higher dimension complicates to some extent the search for the most frequent literal. In fact, every term with k don't care input values (representing a k-dimensional hypercube) might be replaced by 2^k minterms, thus increasing the occurrence of the used literals 2^k times. Strictly speaking, the weight assigned to these literals should be multiplied by this factor. This is, however, not feasible, because for functions with several hundreds of input variables the number of vertices of any hypercube may reach astronomic values. Different approaches to the solution of this problem have been evaluated and tested. The best results were obtained when no corrections of the literal weight were made with respect to the dimensions of the input terms.

A certain drawback of the CD-search algorithm is that it is greedy. Thus, once a literal is selected, it is kept till the end of implicant generation, and therefore the obtained implicants need not be prime. Hence we have to check whether some literals can be removed without losing the property of an implicant. Here the second part of the PI generation algorithm, namely Implicant Expansion (IE), finds its application.

5. IMPLICANT EXPANSION

As mentioned above, the implicants constructed during the CD search 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 is a very sensitive operation in the sense that much effort may be wasted if a bad strategy is chosen, and the result may be far from optimum at that.

There are basically two problems to be solved in connection with implicant expansion. One of them is the mechanism that effectively checks whether a tentative literal removal is acceptable. The other is the selection of the literals and the order in which they are to be removed from the implicant term. First let us discuss the checking mechanism.

5.1 Checking a Literal Removal

Removing a variable from a term doubles the number of minterms covered by the term. The newly covered minterms may be 1-minterms or DC-minterms, but none of them should be a 0-minterm. In BOOM, individual literals are tried for removal by checking whether the expanded term does not intersect the off-set. This means that the DC terms need not be enumerated explicitly, because every newly created implicant is compared with the 0-terms. If an intersection is found, the removal is cancelled.

5.2 Expansion Strategy

The second problem is the selection strategy for the literals to be removed. The expansion of one implicant may yield several different prime implicants. To find them all, we have to try systematically to remove each literal, whereas the order of the literals selected plays an important role. Trying all possible sequences of literals to be removed will be denoted as an **Exhaustive Implicant Expansion.** Using recursion or queue, all possible literal removals can be tried until all PIs are obtained. Unfortunately, the complexity of this algorithm is exponential. Hence this method is usable only for smaller problems. Nevertheless, the modification of this method called Distributed Exhaustive Implicant Expansion is generally usable and quite effective.

There exist several other IE methods differing in complexity and quality of results obtained. Some of them that are used in BOOM are described below.

The simplest one, namely a **Sequential Search**, systematically tries to remove from each term all literals one by one, starting from a randomly chosen position. Every removal is checked against the off-set and if the removal is successful, we make it permanent. Otherwise we put the literal back and proceed to the next one. After removing all possible literals we obtain one PI covering the original term. This algorithm is greedy, i.e., we stay with one PI even if there are more than one PIs that can be derived from the original implicant. A sequential search only reduces the number of literals, the experimental results show that this reduction may reach approximately 25%. The complexity of this algorithm is linear.

With a **Multiple Sequential Search** we try all possible starting positions and each implicant thus may expand into several PIs. The upper bound of the number of PIs that can be produced from one implicant is n-d, where n is the number of input variables and d is the dimension of the original implicant. The complexity of this algorithm is $O(n*p)$, where p is the number of defined on-terms.

Some IE algorithms, especially Exhaustive Implicant Expansion are rather time consuming for large problems. Therefore a distributed version for each of them was proposed and tested. **Distributed Expansion** is based on the idea of distributing the expansion among several consecutive iterations. The advantage of this approach is the possibility to stop the expansion at the moment when an acceptable result is reached and save a considerable amount of time, because the quality of the solution is checked after each iteration.

In the **Distributed Multiple Sequential Search** only one pass of a sequential search is made for every implicant. After that, these implicants are stored in the E-buffer. In the following iteration they are processed again together with the newly created implicants, while another starting position for the sequential search is used. When all meaningful starting positions are exhausted, the corresponding implicant is removed from the buffer. In other words, if the multiple sequential search produces j primes from some implicant in one pass, the distributed multiple sequential search will find all of them in j iterations.

Distributed Exhaustive Implicant Expansion uses the same mechanism as the Distributed Multiple Sequential Search. In this case, also the partially expanded implicants are stored in the E-buffer. This ensures the exhaustiveness of the expansion.

5.3 Evaluation of Expansion Strategies

The properties of the proposed IE methods and their influence on the minimization process (time and quality of the final solution) will be discussed in this section. The distributed mode of the implicant expansion methods will not be studied separately, as it is always a simple modification of the original algorithm. Hence their properties in the given example are similar.

The choice of IE method may influence two properties of the minimization process: the time of minimization and the quality of the result obtained. Fig. 4 shows the time of the minimization of a single-output function of 30 input variables and 500 defined minterms as a function of the number of iterations. The growth for the sequential search is linear, which means that an equal time is needed for each iteration. The time for the multiple sequential search and the exhaustive expansion grows faster at the beginning and then it turns to linear with a slower growth. At this point the CD-search no longer produces new implicants and thus the IE and the following phases are no longer executed. This causes the simple sequential search, which is seemingly the fastest, to become the slowest after a certain number of iterations.

Fig. 5 illustrates the growth of the PI set as a function of time. We can see that the Sequential Search achieves the lowest values, although it is the fastest implicant expansion method. However, when this method is used we cannot take advantage of the I-buffer and the implicants are repetitively expanded, even if they have been expanded in all possible ways. We can see that the most complex method, namely exhaustive expansion, produces PIs at the fastest rate.

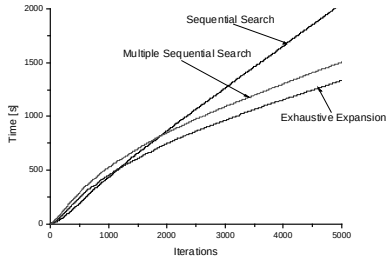

Fig. 4: Growth of time for different IE methods

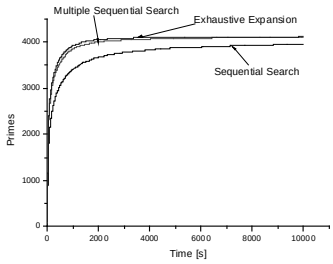

Fig. 5: Growth of PIs for different IE methods

We saw in subsection 3.2 that only a small number of PIs may be enough for the minimum solution. Moreover, the quality of the final solution strongly depends on the CP solution algorithm. With a large number of PIs exact solving is impossible and some heuristic must be used. Here the large number of implicants may misguide the CP solution algorithm and thereby prevent the minimum solution from being achieved. Practice shows that the more complex IE methods are more advantageous for less sparse functions, where the number of implicants in the final solution is big, while the simplest sequential search is better for very sparse functions.

6. 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, 160 MB RAM. Three groups of experiments, listed in the following three subsections, were performed.

6.1. Solution of MCNC Benchmark Problems

First a group of several MCNC benchmark problems was solved by ESPRESSO 2.3 and by BOOM [11]. 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). To check the quality of the result, all problems were also solved by ESPRESSO-EXACT. The results obtained from BOOM, measured by the number of literals, output cost and number of implicants (lit/out/impl) were in all cases equal or better. Tab. 1 shows that problems with a large number of defined terms (*p*) were often solved by ESPRESSO in a shorter time. This is due to the quadratic dependence of runtime on the number of terms in BOOM [3].

TABLE 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

		ESPRESSO		BOOM	
Bench	*n*/*m*/*p*	time	lit/out/impl	time	lit/out/impl
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

6.2. 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. 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 for all problems. The on-set and off-set of each function were kept approximately of the same size. For each problem size ten different samples were generated and solved. Tab. 2 contains the average values of the ten solutions.

BOOM was always run iteratively, using **the same total runtime** as ESPRESSO needed for one pass. The quality criterion selected for BOOM was the sum of the number of literals and the output cost. The first row of each cell contains the BOOM results, the second row shows the ESPRESSO results. The missing ESPRESSO results in the lower right-hand corner indicate the problems for which ESPRESSO could not be used because of the long runtimes. Hence only one iteration of BOOM was performed, and its duration in seconds is given as a last value.

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

p/n	20	60	100	140	180	220	260	300
20	26/16/11(41) 29/20/10/0.31	22/12/9(67) 23/15/9/1.01	18/11/8(96) 23/13/8/1.95	18/10/8(127) 22/14/8/3.47	16/10/8(161) 19/13/8/5.59	17/10/8(201) 19/12/7/9.52	16/10/8(219) 18/11/7/12.03	16/9/8(262) 20/12/8/16.44
60	109/43/30(19) 116/52/28/1.03	76/29/22(54) 86/40/21/6.54	68/24/20(77) 75/34/19/14.26	65/22/19(127) 73/34/19/28.99	61/21/19(151) 68/30/17/42.46	58/21/17(183) 62/28/16/57.53	56/20/17(218) 64/29/17/78.68	55/19/17(271) 65/27/17/111.62
100	206/64/48(15) 203/79/43/2.09	143/42/35(45) 150/61/33/13.85	127/38/32(74) 133/55/29/41.26	118/36/30(100) 127/52/28/69.02	110/32/28(157) 121/46/27/124.22	108/31/28(162) 116/46/26/152.44	105/31/27(215) 116/45/26/248.67	102/30/27(260) 112/44/25/328.37
140	298/87/65(13) 296/108/58/3.81	206/56/47(46) 215/80/43/28.70	190/50/44(70) 191/72/39/71.23	177/46/41(94) 177/66/36/129.66	165/44/39(127) 171/63/36/206.76	159/44/37(160) 164/60/34/273.15	154/39/36(210) 164/60/33/452.93	149/40/36(231) 156/55/32/516.63
180	407/108/83(12) 397/139/74/6.09	288/70/61(45) 284/101/54/48.70	251/61/54(79) 253/92/48/141.97	230/56/51(111) 233/84/44/261.95	220/55/49(139) 228/80/44/397.36	209/50/46(181) 220/77/42/630.53	255/49/48/1.36 -	250/48/48/1.60 -
220	531/132/102(11) 497/162/88/8.01	363/85/72(48) 352/120/63/80.68	310/74/64(88) 310/109/57/256.40	291/68/60(118) 290/103/53/392.86	273/65/57(146) 285/98/52/632.04	329/61/60/1.79 -	320/60/59/2.09 -	308/58/57/2.43 -
260	648/155/120(10) 594/193/100/10.95	436/98/84(46) 427/144/74/108.50	374/84/74(87) 382/124/67/336.32	353/81/70(116) 348/119/61/580.84	420/75/73/2.15 -	398/71/70/2.55 -	391/70/69/2.98 -	372/66/65/3.39 -
300	786/182/142(8) 710/226/118/11.59	521/109/96(40) 489/160/83/120.69	450/97/87(81) 447/149/75/427.72	422/88/81(107) 416/139/71/719.54	493/84/8/32.88 -	469/80/79/3.48 -	449/77/77/3.91 -	441/77/75/4.75 -

Entry format: BOOM: #of literals/output cost/#of implicants(# of iterations).

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

6.3 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 large test problems was generated and solved by BOOM. For each problem size (# of variables, # of terms) 10 different problems were generated and solved. Each problem was a group of 10 output functions. 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 average time in seconds needed to complete one iteration for various problem sizes is shown. We can see that a problem with 1000 input variables, 10 outputs and 1000 care minterms was solved by BOOM in less than 20 minutes.

TABLE 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

7 CONCLUSIONS

A new method for Boolean function minimization has been proposed and implemented as the BOOM minimization tool. Its application is advantageous above all for problems with large dimensions and a large number of don't care states. The PI generation method is very fast, hence it can easily be used in an iterative manner. On MCNC benchmark problems, the runtime improvement as compared with ESPRESSO-EXACT was significant, in most cases more than one order of magnitude. For large problems with several hundreds of input variables and several hundreds of care terms, the BOOM system beats ESPRESSO both in minimality of the result and in runtime.

The BOOM minimizer has been placed on a web page [11], 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

[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] 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

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

[5] Hlavička, J. - Fišer, P.: Algorithm for Minimization of Partial Boolean Functions, Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems (DDECS00) Workshop, Smolenice, (Slovakia) 5-7.4.200, pp.130-133

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

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

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

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

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

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

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

[13] ftp://eecs.berkeley.org