Computing and Informatics, Vol. 22, 2003, 19–51

BOOM — A HEURISTIC BOOLEAN MINIMIZER

Petr Fišer, Jan Hlavička[†]

Czech Technical University in Prague
Dept. of Computer Science and Engineering
Karlovo nám. 13
121 35 Prague 2, Czech Republic
e-mail: `fiserp@fel.cvut.cz`

Manuscript received 8 November 2002; revised 5 March
Communicated by Norbert Frištacký

Abstract. This paper presents an algorithm for two-level Boolean minimization (BOOM) based on a new implicant generation paradigm. In contrast to all previous minimization methods, where the implicants are generated bottom-up, the proposed method uses a top-down approach. Thus, instead of increasing the dimensionality of implicants by omitting literals from their terms, the dimension of a term is gradually decreased by adding new literals. The method is advantageous especially for functions with many input variables (up to thousands) and with only few care terms defined, where other minimization tools are not applicable because of the long runtime.

The method has been tested on several different kinds of problems and the results were compared with ESPRESSO.

Keywords: Boolean minimization, logic functions, sparse functions, implicant expansion, group minimization, covering problem solution, mutations

1 INTRODUCTION

The problem of two-level minimization of Boolean functions is old, but surely not dead. It is encountered in many design environments like PLA design, multi-level logic design, design of control systems, or design of built-in self-test (BIST) equipment, and also in software engineering, artificial intelligence problems, etc. The systematic Boolean minimization methods mostly copy the structure of the original method by Quine and McCluskey [1, 2], implementing the two basic phases known as prime implicant (PI) generation and covering problem (CP) solution. Some more modern methods, including the well-known ESPRESSO [3, 4], try to combine these two phases because the problems encountered in up-to-date application areas often require minimization of functions with a prohibitively large number of PIs. Also the number of don't care states is mostly very large, hence the 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 10. ESPRESSO-EXACT [5] was developed in order to improve the quality of the results, mostly at the expense of much longer runtimes. Finally, ESPRESSO-SIGNATURE [6] was developed, accelerating the minimization by reducing the number of prime implicants to be processed by introducing the concept of a "signature", which is an intersection of all primes covering one minterm. This in turn was an alternative name given to the concept of "minimal implicants" introduced in [7]. Other Boolean minimization methods exploiting the implicit set manipulation techniques were proposed in, e.g., [8, 9]. The idea of meta-products was proposed, which allows the manipulation with extremely large sets of PIs.

A sort of 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 Minimizer) approach proposed here. An important difference between the approaches of ESPRESSO and BOOM is the way they work with the on-set received as a 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 that determines whether a tentative solution is correct or not. This allows us to reduce the dependence on 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 n-dimensional hypercube until it no longer intersects the off-set, while it covers as many 1-terms of the source function as possible. This phase is denoted as a CD-Search and represents the most innovative idea of the proposed method. Beyond this, some other commonly known algorithms (Implicant Expansion, Covering Problem solution, etc.) are used together with the CD-Search to obtain the final solution.

The algorithm is advantageous above all for functions with a large number of input variables, where other minimization tools often fail to give a result in a reasonable time.

Some features of the proposed method were published in several conference proceedings [10–14]. BOOM was programmed in Borland C++ Builder and tested under MS Windows 2000.

This paper has the following structure. After a formal problem statement in Section 2, the structure of the BOOM system is described in Section 3 and the initial implicant generation through a coverage-directed search is described in Section 4. The iterative use of the method is described in Section 5. The expansion of the obtained implicants into prime implicants is given in Section 6. Section 7 briefly describes the covering problem solution. The extension of the method to multi-output functions is described in Section 8. Section 9 presents the use of mutations that sometimes improves the result. Experimental results are presented and commented in Section 10, and Section 11 evaluates the time complexity of the algorithm. Section 12 concludes the paper.

2 PROBLEM STATEMENT

2.1 Boolean Minimization

Let us have a set of m Boolean functions of n input variables $\mathcal{F}_1(x_1, x_2, \ldots, x_n)$, $\mathcal{F}_2(x_1, x_2, \ldots, x_n)$, $\ldots$, $\mathcal{F}_m(x_1, x_2, \ldots, x_n)$, whose output values are defined by truth tables. These truth tables describe the *on-set* $\mathrm{F}_i(x_1, x_2, \ldots, x_n)$ and *off-set* $\mathrm{R}_i(x_1, x_2, \ldots, x_n)$ for each of the functions $\mathcal{F}_i$. The terms not represented in the input field of the truth table are implicitly assigned don't care values for all output functions. The *don't care set* $D_i(x_1, x_2, \ldots, x_n)$ of the function $\mathcal{F}_i$ is thus represented by all the terms not used in the input part of the truth table and by the terms to which don't care values are assigned in the i^{th} output column. The don't care values can be also specified explicitly in the truth table. 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 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. Moreover, using off-set in the function definition simplifies checking whether a term is an implicant of the given function. Without the explicit off-set definition, more complicated methods using, e.g., tautology checking as in ESPRESSO [3], must be used, which slows down the minimization process.

Our task is to formulate a synthesis algorithm which will for each output function $\mathcal{F}_i$ produce a sum-of-products expression $G_i = g_{1i} + g_{2i} + \ldots + g_{ti}$, where $\mathrm{F}_i \subseteq G_i$ and $G_i \cap \mathrm{R}_i = \emptyset$. The expression $T = \sum_{i=1}^{m} t_i$ should be kept minimal.

This formulation of the minimization process uses the number of product terms (implicants) 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 [16]. Thus, e.g., for PLAs, the number of product terms is what counts, whereas the total number of literals has no importance. In some other cases, like in custom design, the total number of literals and the output cost (the number of inputs into all second-level OR gates), may be important. Hence we will formulate the method in such a way that all criteria can be used on demand and allow the user to choose among them.

2.2 Motivation

An example of a design problem with many input variables and many don't care states can be found in the design of built-in self-test (BIST) devices for VLSI circuits. A very common method of BIST design is based on the use of a linear feedback shift register (LFSR) generating a code whose code words are used as the input patterns for the circuit under test. However, before being used as test patterns, these words usually have to be transformed into the patterns needed for fault detection [17, 18]. The LFSR may have more than one hundred stages and the sequence used for testing may have several thousands of states. Thus, e.g., for a circuit with 100 LFSR stages and 1000 test patterns the design of the decoder is a problem with 100 input variables and $2^{100} - 1000$ don't care states.

Another typical application is the design of control systems, where the circuit is described by its behavior. Here, again, the truth table defines the values of the outputs (or internal variables) for the corresponding values of the input variables (internal variables).

3 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 for single-output functions is performed in two steps: first the *Coverage-Directed Search* (CD-Search) generates a sufficient set of implicants needed for covering the on-set of the source function, and the succeeding *Implicant Expansion* (IE) phase converts them into PIs.

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 (group implicants). Here, *Implicant Reduction* takes place. Then the *Group Covering Problem* is solved and *Output Reduction* is performed. Figure 1 shows a block diagram of the BOOM system, where each block corresponds to one minimization step and the data sets described between these blocks correspond to the products of these steps.

The BOOM system may improve 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 all the output functions. In each of the following iterations, another sufficient set is generated and new implicants are added to the previous ones (if the solutions are not equal). This process is treated more closely in Section 5.

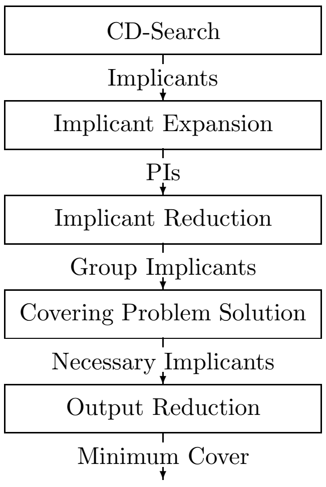

Fig. 1. Structure of the BOOM system

4 COVERAGE-DIRECTED SEARCH

4.1 Basis of the Method

The idea of combining implicant generation with the solution of the covering problem was the basis of the Coverage-Directed Search (CD-Search) method used in the BOOM system. This consists in a 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 $\mathcal{F}_i$. This happens at the moment when the resulting hypercube does not intersect with 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 may be an implicant, if it does not intersect with any 0-term. If there are some 0-minterms covered, we add another literal and verify whether the new term already corresponds to an implicant by comparing it with 0-terms that can intersect with this term. We continue adding literals until an implicant is generated, then we record it, remove the 1-terms that are covered by this term, 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 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 yet uncovered terms. Now we repeat the procedure from the beginning and apply it to these terms, selecting the most frequently used literal, until another 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. This algorithm is greedy and thus the obtained implicants need not be prime. In order to expand them into primes the IE phase must be performed after the CD-Search.

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

Algorithm 1 (The CD-Search)**.**

```
CD_Search(F, R) {
  $H = \emptyset$                    // $H$ is being created
  do
    F$'$ = F                 // $F'$ is the reduced on-set
    $t$ = true               // $t$ is the term in progress
    do
      $v$ = most_frequent_literal(F$'$)
      $t = t.v$
      F$'$ = F$'$ − cubes_not_including($t$)
    while ($t \cap R \neq \emptyset$)
    $H = H \cup t$
    F = F − F$'$
  until ($F == \emptyset$)
  return $H$
}
```

4.2 Immediate Implicant Checking

When selecting the most frequent literal, it may happen that two or more literals have the same frequency of occurrence. In these cases we either select one at random or we must apply another decision criterion — namely the immediate implicant checking. The idea consists in constructing terms as candidates for implicants by multiplying all newly selected literals (those with the same frequency) by the previously selected one(s). Among these terms we select only *the implicants* (if any) and reject the rest. When there are still more possibilities to choose from, we select one at random.

Sometimes this feature prevents a term from being unnecessarily prolonged, because it would have to be shortened during the IE. The effects of using this additional criterion are the following:

- it reduces the runtime of CD-Search and the whole minimization
- it reduces the number of PIs that are generated.

This can be illustrated by Table 1. A single-output function with 20 input variables and 500 defined terms was minimized in 1000 iterations. In the first experiment, immediate implicant checking was not used, while in the second one it was.

	not used	used
Total CD-Search time [s]	318,9	265,1
Total minimization time [s]	6688,3	4782,8
Number of PIs found	2719	21741

Table 1. Immediate implicant checking effects

4.3 CD-Search Example

Let us have a partially defined Boolean function of ten input variables $x_0, \ldots, x_9$ and ten defined minterms given by a truth table Table 2. The 1-minterms are highlighted.

```
var:  0123456789
  0.  0000000010 1
  1.  1000111011 1
  2.  0000011001 1
  3.  1111011000 0
  4.  1011001100 0
  5.  1111000100 1
  6.  0100010100 0
  7.  0011011011 0
  8.  0010111100 1
  9.  1110111000 1
```

Table 2

As the first step we count the occurrence of literals in the 1-minterms. Lines "0" and "1" in Table 3 give the counts of x_i' and x_i literals respectively. In this table we select the most frequent literal.

var:	0123456789
0.	3435322444
1.	3231344222

Table 3

The most frequent literal is x_3' with five occurrences. This literal alone describes a term that is not an implicant, because it covers the 6th minterm (0-minterm) in the original function. Hence another literal must be added. When searching for the next literal, we can reduce the scope of our search by suppressing 1-minterms containing the selected literal with the opposite polarity (shaded dark in Table 4). An implicant containing a literal x_3' cannot cover the 5th minterm, because it contains the x_3 literal. Thus, we temporarily suppress this minterm. In the remaining 1-minterms we find the most frequent literal.

var:	0123456789	
0.	0000000010	1
1.	1000111011	1
2.	0000011001	1
3.	1111011000	0
4.	1011001100	0
5.	1111000100	1
6.	0100010100	0
7.	0011011011	0
8.	0010111100	1
9.	1110111000	1

var:	0123456789
0:	343-211433
1:	212-344122

Table 4

As there are several literals with maximum frequency of occurrence 4 (x_1', x_5, x_6, x_7'), the second selection criterion must be applied. We use these literals tentatively as implicant builders and create four product terms using the previously selected literal x_3': $x_3'x_1'$, $x_3'x_5$, $x_3'x_6$, $x_3'x_7'$. Then we check which of them are already implicants. The term $x_3'x_5$ is not an implicant (it covers the 6th minterm), so it is discarded and among the remaining three terms one is selected at random, e.g., $x_3'x_6$. This implicant is stored and the search continues.

The search for literals of the next implicants is described in Table 5. We omit minterms that are covered by the selected implicant $x_3'x_6$ (dark shading) and select the most frequent literal in the remaining minterms.

As seen in the lower part of Tab. 5, we have four equal possibilities, so we choose one randomly — e.g. x_5'. In a similar way we can find another literal (x_6') needed to

var:	0123456789	
0.	0000000010	1
1.	1000111011	1
2.	0000011001	1
3.	1111011000	0
4.	1011001100	0
5.	1111000100	1
6.	0100010100	0
7.	0011011011	0
8.	0010111100	1
9.	1110111000	1

var:	0123456789
0:	1111222112
1:	1111000110

Table 5

create an implicant covering the remaining two 1-minterms.

The resulting expression covering the given function is $x_3'x_6 + x_5'x_6'$.

4.4 Weights

The fact that the input file may contain both 1-minterms and 1-terms of higher dimension may complicate 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 literals used in the original term 2^k times. Strictly speaking, each of these literals should be assigned a weight corresponding to 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. However, the best results were obtained when no weights were assigned to the literals in connection with the dimensions of the input terms. This can be explained to some extent by the fact that when searching for the proper literal for inclusion, the algorithm does not try to cover maximum 1-minterms, but the maximum number of 1-terms in the function definition.

5 ITERATIVE MINIMIZATION

Most current heuristic Boolean minimization tools use deterministic algorithms. The minimization process then 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 and thus we can pick out the best solution. Moreover, the PIs and group implicants can be cumulated during the process and afterwards the CP solved using all of them, which enables us to reach a better final result.

5.1 The Effect of Iterative Approach

The iterative minimization concept takes advantage of the fact that each iteration produces a new set of prime implicants sufficient for covering all 1-terms of all output functions. This 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 Figure 2 (thin line). This curve plots the values obtained during the solution of a problem with 20 input variables and 200 care minterms. Theoretically, the more primes we have, the better the solution can be found after solving the covering problem, but the maximum set of primes is often extremely large. In reality, the quality of the final solution, measured by the number of literals in the resulting SOP form, 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 Figure 2 (thick line).

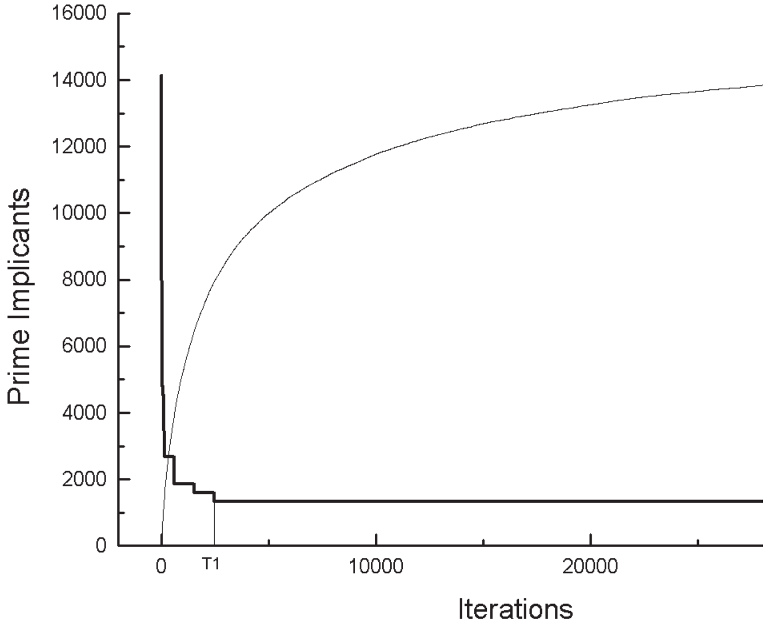

Fig. 2. Growth of PI number and decrease of SOP length during iterative minimization

It is obvious from the curves in Figure 2 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.

The iterative minimization of a group of functions $\mathcal{F}_i$ $(i = 1, 2, \ldots, m)$ 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, and the output is a minimized disjunctive form $G = (G_1, G_2, \ldots, G_m)$.

Algorithm 2 (Minimization of a group of functions)**.**

```
BOOM(F[1..$m$], R[1..$m$]) {
  $G = \emptyset$
  do
    $I = \emptyset$
    for ($i = 1$; $i \leq m$; $i$++)
      $I'$ = CD_Search(F[$i$], R[$i$])
      Expand($I'$, R[$i$])
      Reduce($I'$, R[1..$m$])
      $I = I \cup I'$
    $G'$ = Group_cover($I$, F[1..$m$])
    Reduce_output($G'$, F[1..$m$])
    if (Better($G'$, $G$)) then $G = G'$
  until (stop)
  return $G$
}
```

5.2 Accelerating Iterative Minimization

When the CD-Search phase is being repeated, identical implicants are quite often generated in different iterations. These are then passed to the Implicant Expansion phase (see Section 6), 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). A diagram of the whole minimization algorithm for a multi-output function is shown in Figure 3.

Each newly generated implicant is first looked up in the I-buffer and, if it is already present, its further processing is stopped. Otherwise it is stored in both the I-buffer and E-buffer (Expansion buffer). The E-buffer serves as storage of implicants that are candidates for expansion into PIs. After expansion, they are removed from the E-buffer. Then they are reduced to group implicants and, after duplicity and

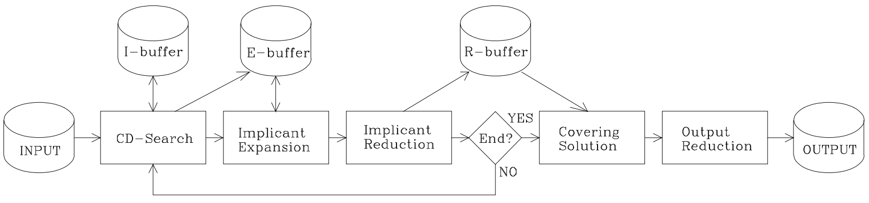

Fig. 3. Iterative minimization schematic plan

dominance checks, the newly created group implicants are stored in the R-buffer (Reduced implicants buffer). Finally, the covering problem is solved using all the implicants from the R-buffer. For multi-output functions there are separate I- and E-buffers for each output. The R-buffer is common.

The main implementation requirement for the I-buffer is its high look-up speed, enhanced especially by early detection of the absence of a term. The buffer is structured as a ternary tree with depth n. During the search in the tree, the direction at the k^{th} level is chosen according to the type of occurrence $(0, -, 1)$ of the k^{th} variable in the searched term. The presence of an implicant is represented by the existence of its corresponding leaf. The tree is dynamically constructed during the addition of implicants into the buffer. An example of such a tree for $n = 3$ is shown in Figure 4. The buffer contains implicants 0−0, 10− and 11−. If, e.g., an implicant 0−1 is looked for, the search will fail in the node 0−, where no path leading to 0−1 is present.

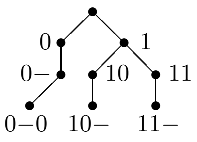

Fig. 4. I-buffer tree structure

6 IMPLICANT EXPANSION

As mentioned above, the implicants constructed during the CD-Search need not be prime. To reduce the number of implicants needed to cover all 1-terms of the given function, we have to increase their size by IE, which is done by removing literals (variables) from their terms. When no literal can be removed from the term any more, we get a prime implicant (PI).

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.

6.1 Checking the Removal of a Literal

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 and checked whether the expanded term does not intersect the off-set (therefore the DC terms need not be enumerated explicitly). If a non-empty intersection with a 0-term is found, the removal is rejected. The checking is done by a simple comparison of the term with all the off-set terms.

6.2 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 systematically tried until all primes are obtained. Unfortunately, the complexity of this algorithm is exponential. Hence this method is usable only in problems with up to 20 input variables.

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 Expansion*, 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 as above, but if the removal is successful, we make it permanent. If, on the contrary, some 0-minterm is covered, we put the literal back and proceed to the next one. After removing all possible literals we obtain one prime implicant covering the original term. This algorithm is greedy, i.e., we stay with one PI even if there is more than one PI that can be derived from the original implicant. The complexity of this algorithm is linear with the number of input variables and the number of processed terms.

A sequential expansion obviously cannot reduce the number of product terms, but it reduces the number of literals. The experimental results have shown that this reduction may reach approximately 25 %.

With a *Multiple Sequential Expansion* 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 $O(n \cdot p)$, where p is the number of processed terms.

6.3 Evaluation of Expansion Strategies

The properties of the proposed IE methods and their influence on the minimization process (runtime and quality of the final solution) will be discussed in this section.

Figure 5 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 expansion is linear, which means that an equal time is needed for each iteration. The time for the multiple sequential expansion and the exhaustive expansion grows faster at the beginning and then 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 not executed. This causes simple sequential expansion, which is seemingly the fastest, to become the slowest after a certain number of iterations.

Figure 6 illustrates the growth of the PI set as a function of time. We can see that Sequential Expansion 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. The implicants are repetitively expanded, even if they have already been expanded in all possible ways. The two other methods achieve higher values, because they put an implicant into the E-buffer only once and then they are blocked by the I-buffer. Hence when the same implicants are generated repetitively by the CD-Search, they are not processed any more, which speeds up the whole minimization. We can see that the most complex method, namely exhaustive expansion, produces PIs at the fastest rate.

Practice shows that the more complex IE methods are advantageous for functions with large care sets, where the number of implicants in the final solution is big, while the simplest sequential expansion is better for very sparse functions.

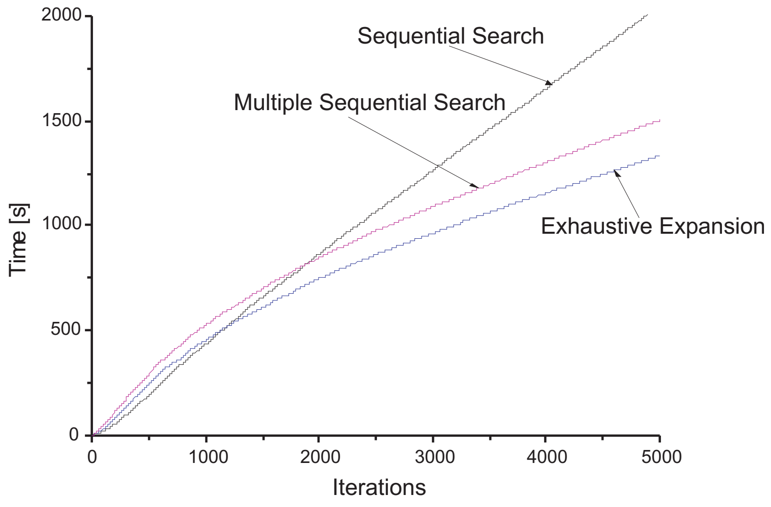

Fig. 5. Growth of time for different IE methods

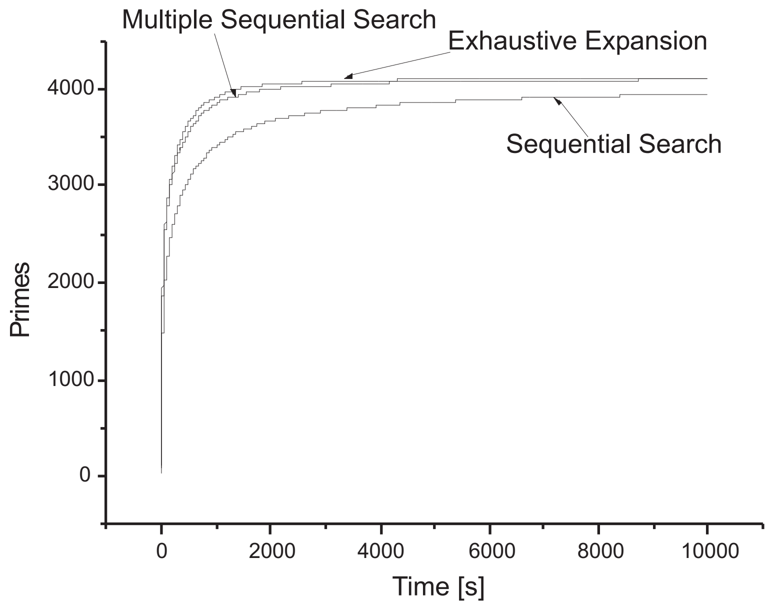

Fig. 6. Growth of PIs for different IE methods

7 SOLUTION OF THE COVERING PROBLEM

We saw in subsection 5.1 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 PIs it is impossible to obtain an exact solution and thus some heuristic must be used. Here the large number of implicants may misguide the CP solution algorithm and thereby lead to a non-minimal solution.

An exact CP solution is mostly rather time-consuming, especially when it is performed after several iterations during which many implicants had accumulated. In this case, a heuristic approach is the only possible solution. Out of several possible approaches we used two. The first one, denoted as the LCMC cover (Least Covered, Most Covering) is a common heuristic algorithm for solution of the covering problem. The implicants covering minterms covered by the lowest number of other implicants are preferred. If there are more than one such implicants, implicants covering the highest number of yet uncovered 1-minterms are selected.

More sophisticated heuristic methods for CP solution are based on computing the contributions (scoring functions) of terms as a criterion for their inclusion into the solution [19, 20, 21]. Such a method is also used in BOOM as we found it very effective and not too time-consuming.

8 MINIMIZING MULTI-OUTPUT FUNCTIONS

To minimize multi-output functions, only a few modifications of the algorithm must be done.

At the beginning, each of the output functions $\mathcal{F}_i$ is treated separately, and the CD-Search and IE phases are performed. After that, we have a set of primes sufficient for covering all m functions. However, for obtaining the minimal solution we may need implicants of more than one output function that are not primes of any $\mathcal{F}_i$. Here the next part of the minimization — *Implicant Reduction* (IR) finds place. After the IR the group covering problem is solved.

Its solution is a set of implicants needed to cover each of the output functions $\mathcal{F}_1, \ldots, \mathcal{F}_m$. These implicants are assigned to the individual output functions, so they do not intersect the functions' off-sets. 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 gate. Sometimes this is harmless (e.g. in PLAs); moreover it could 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 for all m functions independently. This phase corresponds to ESPRESSO's MAKE_SPARSE procedure.

8.1 Implicant Reduction (IR)

All obtained primes are tried for reduction by adding literals in order to become implicants of more than one output functions. The method of implicant reduction is similar to the CD-Search. Literals are sequentially being added to the previously obtained implicants until there is no chance that the implicant will be used for more output functions. Preferably, literals that prevent intersecting with most of the terms of the off-sets of all $\mathcal{F}_1, \ldots, \mathcal{F}_m$ (i.e., yielding reduced terms that cover the least zeros in all the functions) are selected. When no further reduction leads to any possible improvement, the reduction is stopped and the term is recorded. A term that no longer intersects with the off-set of any $\mathcal{F}_i$ becomes its implicant. All implicants that were ever found are stored and output functions are assigned to them. Then simple dominance checks are performed in order to eliminate implicants that are dominated by other implicants. Figure 7 shows the typical growth of the number of group implicants (non-primes) as a function of the number of iterations. Here the function of 13 input variables, 13 output variables and 200 defined terms was used for demonstration. We can see that the number of the reduced implicants first grows rapidly, but then it falls to approx. 15 % of the maximum value. This is due the fact that new prime implicants are being constantly produced and they absorb most of the previously generated group implicants in the preliminary dominance checks.

When group implicants are generated, the Group Covering Problem is solved using the same heuristic as described in Section 7.

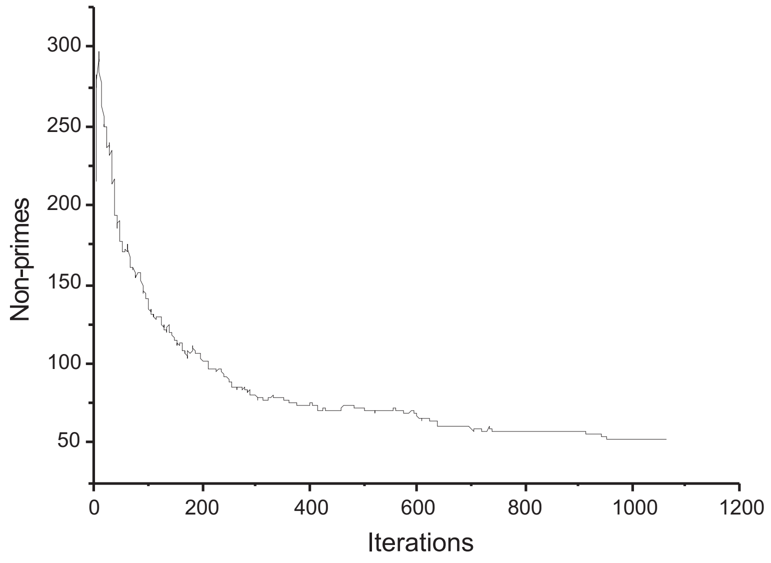

Fig. 7. Growth and fall of the number of non-primes

9 MUTATIONS

The heuristics used to implement individual steps of this procedure are based on a study of the statistical properties of the given Boolean functions. In some cases this selection criterion may prevent reaching the minimum solution. In other words, there may exist implicants that are unreachable by a strict CD-Search, although they are necessary for obtaining the minimum solution. In such cases *mutations*, implemented as a random choice used in place of a deterministic decision, may be of help. These mutations may be used in several places in the minimization process, namely in the CD-Search and IR phases. This section will investigate the usefulness of the mutations, i.e., the quality of the solution obtained and the time needed to find the solution.

During the strict CD-Search the terms are constructed by selecting literals with the maximum frequency of occurrence in the reduced on-set. Selection of a literal with a lower than maximum frequency of occurrence will be denoted as a *mutation*. The probability of occurrence of a mutation will be denoted as a *mutation rate* k. The implementation of mutations consists in a selection of a random literal with a non-zero frequency of occurrence with the probability k. For $k = 0\,\%$ no mutations are present; on the other hand, for $k = 100\,\%$ the CD-Search is driven by random events only. Thus, introducing mutations into the CD-Search randomizes the literal selection to some extent.

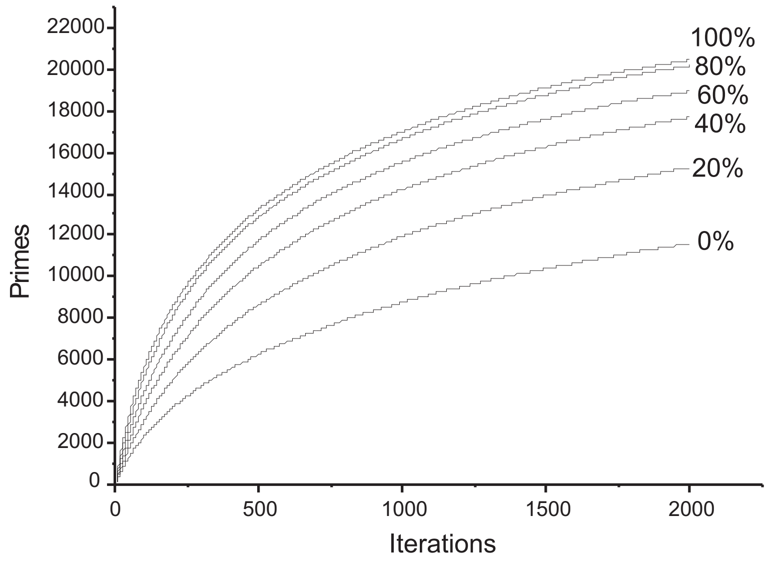

Fig. 8. PI set growth with the number of iterations

The more mutations are implanted, the faster is the growth of the number of primes found during iterative minimization. This is caused by the variety of implicants that are being produced. Figures 8 and 9 show the growth of the number of prime implicants as a function of the number of iterations and the time, respectively. The mutation rate k was changed as a parameter from 20 to 100 %. The problem solved was the minimization of a single-output function of 20 input variables with 300 defined minterms.

Although the number of PIs grows faster for higher mutation rates, the CD-Search is slowed down. This is because implicants that cover fewer 1-terms are produced and thus more of them must be generated to cover all the on-set. The time needed for one pass of the CD-Search as a function of the mutation rate is shown in Figure 10.

The effects documented above can be summarized in the following way: the mutations slow down the whole minimization process and make it less effective; hence we can conclude that selecting literals with maximum frequency of occurrence is the best method of literal selection. The necessary set of implicants for covering the on-set is then reached in the shortest time, and any deviation from this rule will slow down the algorithm.

However, experiments show that 2–5 % of mutations can improve the result by producing some originally unreachable implicants. For example, it can easily be proved that for the problem given by the truth table shown in Table 6 a simple CD-Search without the use of mutations cannot reach the minimum two-term solution shown below.

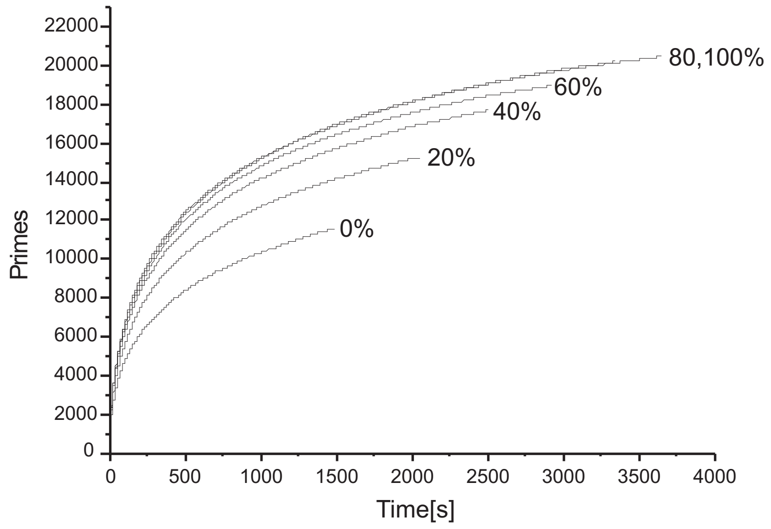

Fig. 9. PI set growth as a function of time

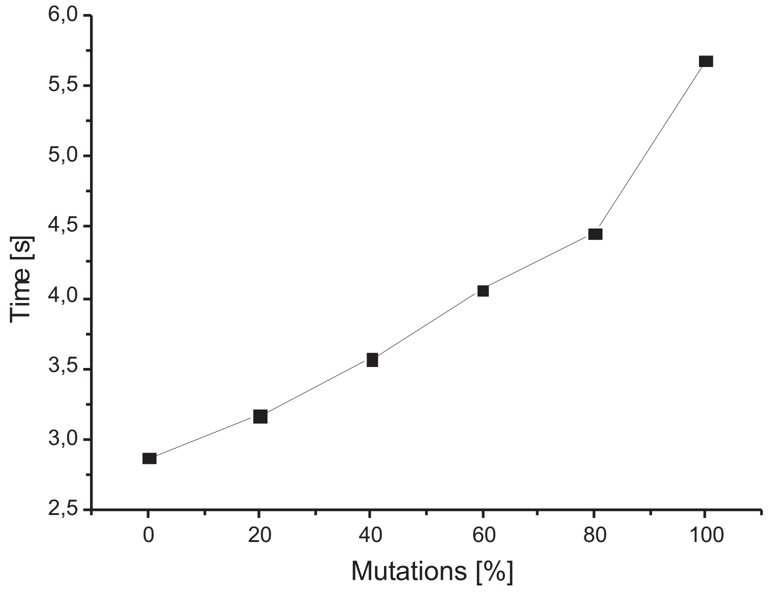

Fig. 10. CD-Search run-time growth with increasing mutation rate

abcdef	
001000	0
000101	0
111011	0
010111	0
101011	0
101110	1
001110	1
000110	0
100000	1
010000	1

exact solution: $cd + c'd'$
solution without the use of mutations: $af' + bf' + cef'$

Table 6. Mutations example

9.1 Implicant Reduction Mutations

Not only the CD-Search, but also implicant reduction can be enhanced by mutations. Here the mutation is a random selection of a literal that prevents intersecting a given term with at least one 0-term. The number of group implicants (non-primes) grows very rapidly with increasing mutation rate — see Figure 11. However, the experimental results show that the final result of the minimization (the quality of the solution) depends only slightly on the rate of IR mutations. The group implicants that arise from the mutations are mostly not necessary for reaching the minimum

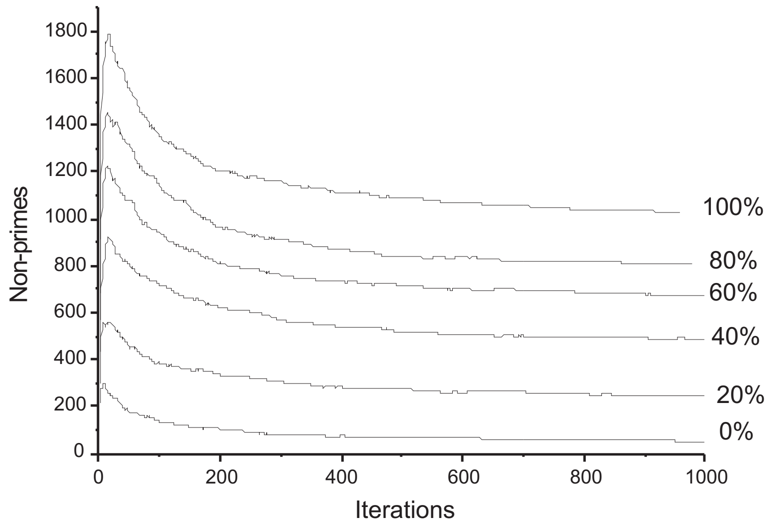

Fig. 11. Dependence of the number of non-primes for different IR mutation rates

solution. Still, there may exist functions that require the use of mutations in order to find the minimum solution.

10 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 quality of the results was measured by three parameters: total number of literals, output cost, and number of product terms (implicants). One of them or the combination of two (sum of the number of literals and output cost) had always to be chosen as a minimization criterion. The results of the experiments are listed in the following subsections. All the experiments were performed on a standard PC with a 900 MHz Athlon processor and 256 MB RAM.

10.1 Standard MCNC Benchmarks

A set of 139 standard MCNC benchmarks [22] was solved by ESPRESSO v2.3 [15], ESPRESSO EXACT and BOOM [23]. As the benchmark functions were specified by the on-set and don't care set (type fd PLA), the source files had to be converted into the format where the on-set and off-set are defined (type fr PLA). This was done by ESPRESSO (by the -ofr switch) before the minimization was performed. Thus, both ESPRESSO and BOOM could use the same input files. The runtimes indicated in the paper do not include the time needed for the conversion.

Of all the 139 standard problems, 67 (48.2 %) were solved by BOOM in a shorter time than by ESPRESSO. In all cases only one iteration of BOOM was used, and thus the results may not be optimal. However, this option was chosen in order to have a better comparison of the results. In 52 cases (37.4 %) BOOM gave the same result as ESPRESSO, while 30 (57.7 %) of these equal results were reached faster than by ESPRESSO.

Tables 7 and 8 show the minimization results of a selected set of the benchmarks. The benchmarks were also solved by ESPRESSO-EXACT in order to obtain the minimum solution for comparison. Note that in this case the minimality criterion is only the number of terms and thus some "exact" solutions are even worse than those reached by ESPRESSO or BOOM. Some benchmarks were not solved by ESPRESSO-EXACT because of extremely long runtimes (blank entries in Table 8). ESPRESSO solutions that are equal to the exact ones are shaded in the ESPRESSO column. The column i/o/p describes the number of input/output variables and care terms of a particular benchmark, the time columns indicate the computational time in seconds, the lit/out/terms columns show the quality of the results, i.e., the number of literals in the final SOP form, the output cost and the number of terms. The shadowed cells indicate that the benchmark was solved by BOOM in a shorter time than by ESPRESSO, or the same result was reached, respectively.

As the MCNC benchmark circuits mostly have a relatively low number of inputs and many care terms defined, the features of BOOM couldn't be fully exploited here. Thus, the results are not optimal comparing with ESPRESSO. However, BOOM is much more advantageous for more complex problems (see the following Subsections), which ESPRESSO often cannot solve in a reasonable time.

		ESPRESSO		ESPRESSO-EXACT	
bench	i/o/p	time	lit/out/terms	time	lit/out/terms
alu2	10/8/241	0.07	268/79/68	0.18	268/79/68
alu3	10/8/273	0.08	279/70/65	0.19	278/74/64
alu4	14/8/1184	0.59	4445/644/575	12.24	4495/648/575
b9	16/5/292	0.08	754/119/119	0.89	754/119/119
br1	12/8/107	0.05	206/48/19	0.07	206/48/19
br2	12/8/83	0.06	134/38/13	0.07	134/38/13
chkn	29/7/370	0.14	1598/141/140	0.25	1602/142/140
cordic	23/2/2105	1.86	13825/914/914	3.59	13843/914/914
ex4	128/28/654	0.62	1649/279/279		
e64	65/65/327	0.11	2145/65/65	0.11	2145/65/65
exep	30/63/643	0.17	1175/110/110	0.55	1170/108/108
ibm	48/17/499	0.11	882/173/173		
mark1	20/31/72	0.25	97/57/19	1.45	97/57/19
misex2	25/18/101	0.07	183/30/28	0.06	183/30/28
misex3c	14/14/1566	0.98	1306/253/197		
misj	35/14/55	0.07	54/48/35		
shift	19/16/200	0.07	388/105/100		
spla	16/46/837	0.71	2558/643/251	6.65	1564/450/181
vg2	25/8/304	0.08	804/110/110	0.54	804/110/110
x9dn	27/7/315	0.08	1138/120/120	0.49	1138/120/120

Table 7. Runtimes and minimum solutions for the standard MCNC benchmarks

10.2 Test Problems with $n \geq 50$

The MCNC benchmarks have relatively few input terms, few input variables (only for 9 standard benchmarks does n exceed 50) and also 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 200 input variables and up to 200 terms was solved. In order to accomplish this we created a set of artificial

		BOOM – 1it.	
bench	i/o/p	time	lit/out/terms
alu2	10/8/241	0.02	268/79/68
alu3	10/8/273	0.02	279/68/66
alu4	14/8/1184	1.02	4449/636/577
b9	16/5/292	0.09	754/119/119
br1	12/8/107	0.02	215/45/20
br2	12/8/83	0.01	134/38/13
chkn	29/7/370	0.41	1598/141/140
cordic	23/2/2105	4.05	13825/914/914
ex4	128/28/654	14.01	1649/279/279
e64	65/65/327	15.06	2145/65/65
exep	30/63/643	3.66	1175/110/110
ibm	48/17/499	0.82	882/173/173
mark1	20/31/72	0.04	93/46/23
misex2	25/18/101	0.10	183/30/28
misex3c	14/14/1566	0.59	1335/242/209
misj	35/14/55	0.03	54/48/35
shift	19/16/200	0.06	388/105/100
spla	16/46/837	1.54	2821/517/285
vg2	25/8/304	0.15	804/110/110
x9dn	27/7/315	0.22	1138/120/120

Table 8. Runtimes and minimum solutions for the standard MCNC benchmarks

benchmark problems, which we denoted as BOOM Benchmarks [24, 25]. The truth tables of these problems were generated by a random number generator, for which only the number of input variables and the number of care terms were specified. The number of outputs was set equal to 5, and the input matrix contained 20 % of don't cares. The on-set and off-set of each function were kept approximately of the same size. For each problem size (# of variables, # of terms) in Tables 9, 10 and 11, ten different samples were generated and solved and average values of the ten solutions were computed.

The randomness of the benchmarks used here was chosen in order to have functions with no special properties. This allows us to determine more easily the properties and scalability of the algorithms. One of the main reasons why BOOM was developed was the need to synthesize the combinational logic for BIST, namely the output decoder transforming the LFSR patterns into test patterns pre-generated by an ATPG tool [26]. Both the LFSR and ATPG patterns mostly have a random nature, and thus the randomly generated benchmarks simulate these practical problems very well.

First the minimality of the result was compared. BOOM was always run iteratively, using **the same** *total runtime* as ESPRESSO needed to obtain a solution. In the following three tables, the number of input variables n increass horizontally and the number of input terms p is increased vertically. The first row of each cell in Table 9 contains the BOOM results, the second row shows the ESPRESSO results. The quality criterion selected for BOOM was the sum of the number of literals and the output cost, which approximates the gate equivalents (GEs) [27]. We can see that for all but one problem size (shaded cell) BOOM found a better solution than ESPRESSO.

p/n	50	100	150	200
50	110/41/25 (58) 122/54/27/3.89	96/35/23 (90) 104/45/23/10.29	90/32/21 (147) 92/41/21/24.87	84/29/20 (199) 89/39/20/41.99
100	284/86/52 (46) 289/104/51/19.31	229/68/42 (94) 231/84/42/77.07	217/61/40 (140) 213/80/39/199.17	207/57/38 (140) 201/74/37/246.21
150	474/132/76 (43) 481/158/76/54.76	389/101/63 (101) 384/125/62/282.80	362/92/61 (116) 345/113/56/646.20	381/90/64 (64) 322/107/52/1066.14
200	678/177/101 (51) 686/209/101/162.62	553/137/83 (116) 539/165/81/730.91	492/125/75 (207) 480/149/72/1913.65	469/110/71 (277) 450/136/68/3372.66

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

Table 9. Solution of Boom Benchmarks — comparing the result quality

A second group of experiments for $n \geq 50$ was performed to compare the runtimes. Again, the randomly generated problems from [24] were solved, but this time BOOM was running until a solution of *the same or better quality* as ESPRESSO was reached. The quality criterion selected was again the sum of the number of literals and the output cost. The results given in Table 10 show that for all samples the same or better solution was found by BOOM in a shorter time than by ESPRESSO.

p/n	50	100
50	170/0,64 (12) 176/3,89	145/1,89 (21) 149/10,29
100	388/7,15 (23) 393/19,31	313/25,5 (48) 315/77,07
150	631/20,38 (25) 639/54,76	506/153,84 (70) 509/282,8
200	890/71,97 (31) 895/162,62	697/467,63 (86) 704/730,91

p/n	150	200
50	131/14,52 (73) 133/24,87	126/3,26 (25) 128/41,99
100	291/38,91 (56) 293/199,17	273/86,51 (83) 275/246,21
150	456/374,68 (105) 458/646,20	427/974,40 (161) 429/1066,14
200	625/1026,28 (149) 629/1913,65	582/1759,27 (220) 586/3372,66

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

Table 10. Solution of Boom Benchmarks — comparing the runtime

10.3 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 very large test problems was generated and solved by BOOM. Each problem was a single-output function in this case. For problems with more than 200 input variables we could not use ESPRESSO, because the runtimes were too long (several hours). 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 Table 11, 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 and 2000 care minterms was solved by BOOM in about 20 seconds.

11 TIME COMPLEXITY EVALUATION

As for most heuristic and iterative algorithms, it is difficult to evaluate the time complexity of the proposed algorithm analytically. An experimental evaluation has been therefore performed.

p/n	200	400	600	800	1000
200	0.06	0.11	0.17	0.26	0.26
400	0.25	0.34	0.52	0.77	0.88
600	0.45	0.80	1.15	1.44	1.96
800	0.88	1.43	2.05	2.69	3.35
1000	1.32	2.10	3.07	4.21	4.42
1200	1.91	3.28	4.69	6.30	7.27
1400	2.69	4.48	6.04	7.72	8.96
1600	3.56	5.78	8.58	10.57	11.69
1800	4.51	7.73	10.56	12.52	16.34
2000	5.64	10.02	13.17	17.45	20.17

Table 11. Time for one iteration on very large problems

11.1 Influence of the Problem Size

The average time needed to complete one pass of the algorithm for various sizes of the input truth table was measured. The number of experiments of each size was 10. The truth tables were generated randomly, following the same rules as in paragraphs 10.2 and 10.3. Figure 12 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 Figure 12 can be approximated with the square of the number of care minterms.

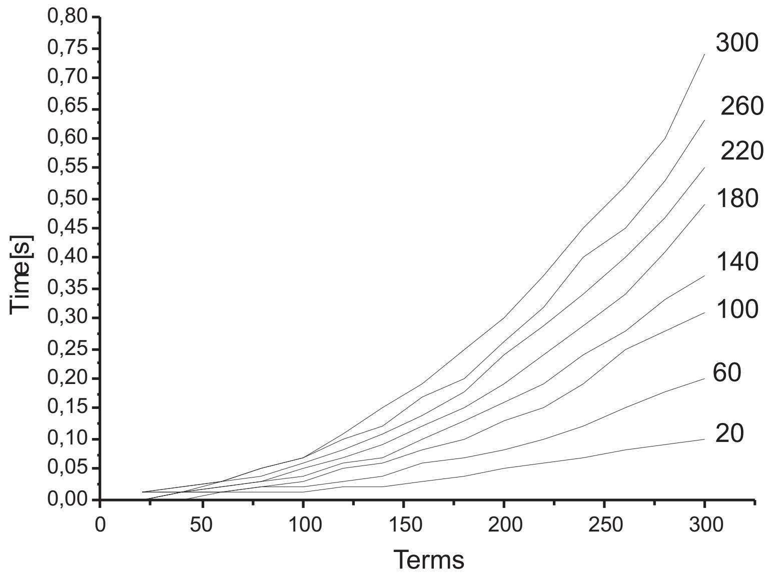

Fig. 12. Time complexity (1)

The minimization time thus grows quite rapidly with the number of care terms. This fact complicates the minimization of functions with a large number of defined terms to some extent. Hence, BOOM is more suitable for minimizing very sparse functions, where the number of care terms is low.

Figure 13 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. Figure 14 shows a three-dimensional representation of the above curves.

The fact that the time complexity grows linearly with the number of input variables (while keeping the number of defined terms) expresses the main advantage of the BOOM algorithm. As the size of the Boolean space of the function grows exponentially with the number of input variables, the time complexity of most of the common minimization algorithms grows exponentially too. In BOOM there is no chance for an exponential time grow, as there are no algorithms with an exponential complexity used in BOOM (except of the situation when the exhaustive IE is used). This allows us to minimize functions with an extremely large number of inputs very efficiently.

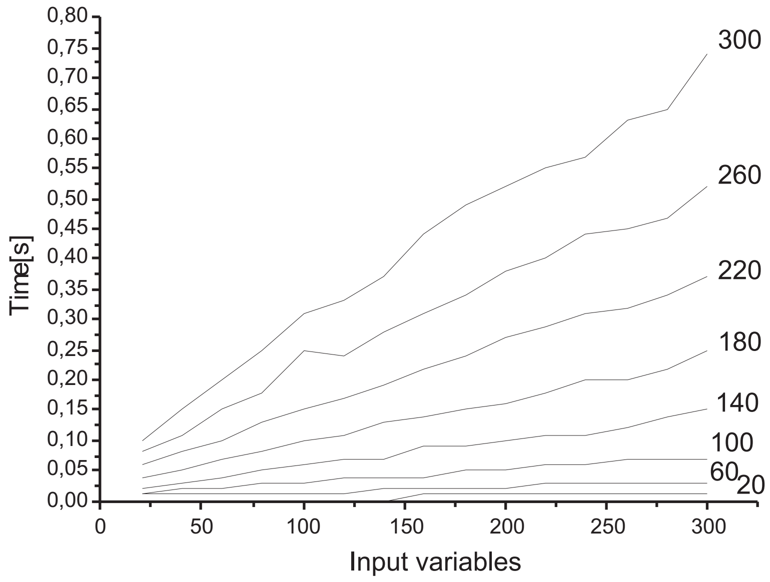

Fig. 13. Time complexity (2)

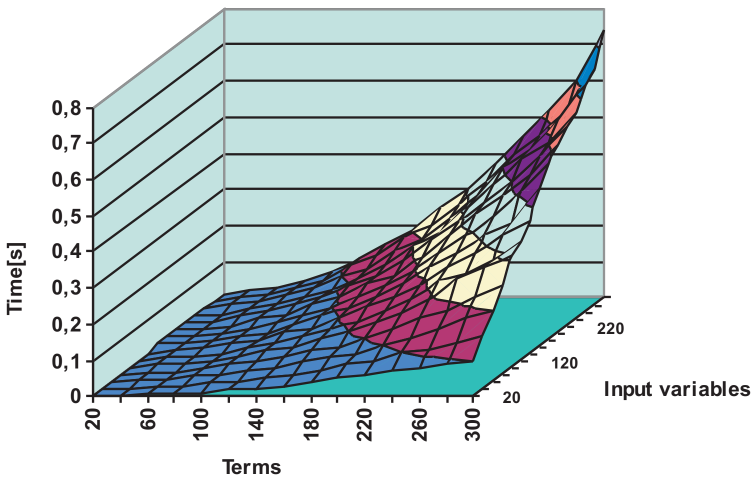

Fig. 14. Time complexity (3)

11.2 Influence of Don't Cares

The influence of the number of don't care states in the field of input variables on the runtime was studied on a set of problems generated by a random number generator for $n = 20$, 50 and 80, with $m = 5$ and 200 terms. The percentage of don't cares was changed in the range from 0 to 35 %. Here 0 % denotes the situation when only minterms were used in the function definition. At the other end, 35 % of don't cares means that slightly more than one third of all values of the input variables were undefined.

The growth of the runtime for ESPRESSO and for BOOM is shown in Figure 15, where the number of input variables is indicated in parentheses. We can see that although ESPRESSO runtime grows to 5000 s for 80 input variables, the BOOM runtime remains almost constant within the used scale for all problem sizes. The influence on the runtime is visualized even more clearly in Figure 16, showing the relative slowdown of BOOM and of ESPRESSO caused by the don't cares. We can see that the relative slowdown of BOOM for the highest percentage of don't cares is about 7.5, whereas for ESPRESSO it is up to 100.

We can conclude from this observation that ESPRESSO is extremely sensitive to the dimensionality of the source terms; the minimization time grows rapidly with the growing number of input don't cares. On the other hand, BOOM is almost insensitive to the dimension of the terms. Thus, BOOM can be efficiently used to minimize functions with a large portion of don't cares in the source terms.

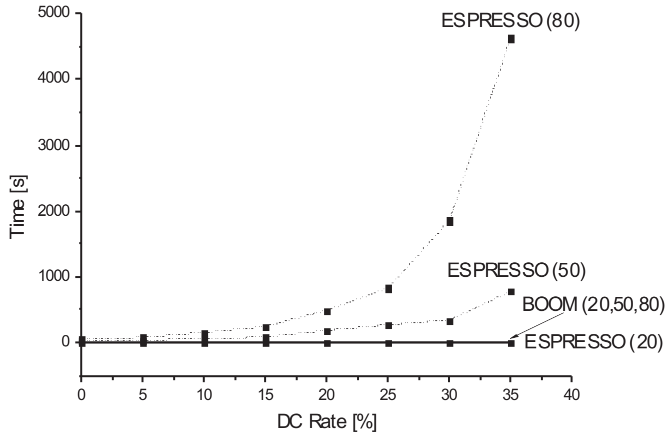

Fig. 15. Runtimes for ESPRESSO (dashed lines) and BOOM (solid line) for various percentages of DCs

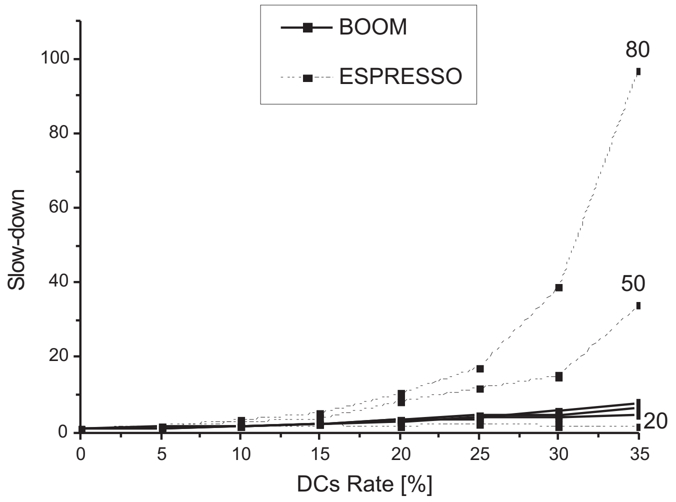

Fig. 16. Relative slowdown [%] for various percentages of DCs

11.3 Duration of Individual Steps

To optimize the time needed for the minimization, it is important to know the contribution of the individual steps of BOOM introduced in Figure 1 to the overall runtime. This aspect was studied on different types of problems. A typical structure of the runtime composition is shown in Figure 17, which plots the data collected during the minimization of a problem with 20 input variables, 5 outputs and 300 care terms. The percentage of input don't care values was set to 10 %. We can observe that the first three steps (CD-Search, IE, IR) represent a very small portion of the overall time, and that the duration of these steps sinks both absolutely (from 0.5 to 0.2 s) and relatively in proportion to the rest of the minimization run.

A relatively steady part of the time is represented by the Dominance Check, which lasts for some 2 s. The most important part of the overall time is consumed by the solution of the Covering Problem. This is the only phase that tends to grow with the number of iterations, because the number of implicants to be processed grows with every iteration. The two interesting discontinuities (zero values of CP solution time) are due to the fact that no new implicants were generated in iterations 625 and 655.

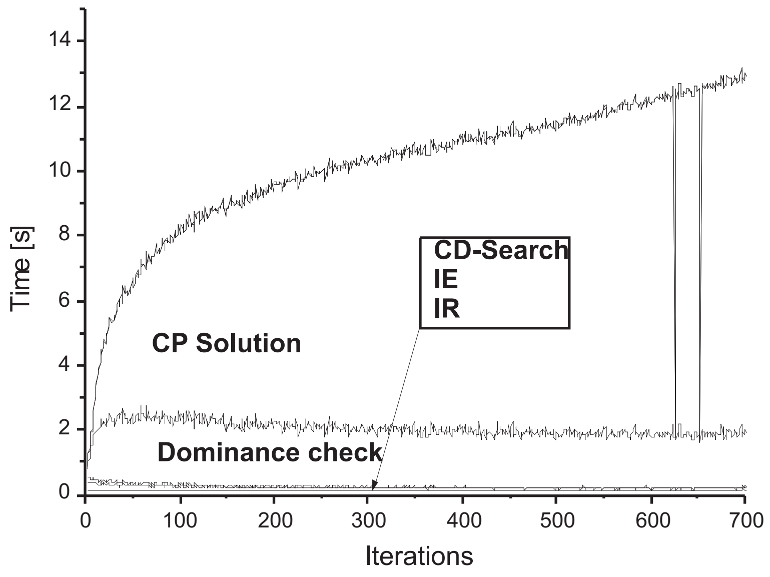

Fig. 17. Execution time of individual steps

12 CONCLUSIONS

An original Boolean minimization method, based on a new approach to implicant generation, has been presented. Its most important features are its applicability to functions with several hundreds of input variables and very short minimization times for sparse functions. The function to be minimized is defined by its on-set and off-set (which may consist of minterms and terms of higher dimensions), whereas the don't care set needs not be specified explicitly. The properties of the BOOM minimization tool were demonstrated on examples. Its application is advantageous above all for functions with a large number of input variables 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 and can easily be used in an iterative manner. Extensive tests on different benchmarks (MCNC, randomly generated problems) were performed in order to determine the strengths and weaknesses of the BOOM system.

The dimension of the problems solved by BOOM can easily be increased over 1000, because the runtime grows linearly with the number of input variables. For problems of very high dimension, success largely depends on the size of the care set, because the runtime grows roughly with the square of its size.

The BOOM minimization tool was programmed in C++ and is available for a public use at [23].

REFERENCES

[1] Quine, W. V.: The Problem of Simplifying Truth Functions. Amer. Math. Monthly, Vol. 59, 1952, No. 8, pp. 521–531.

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

[3] Brayton, R. K. et al.: Logic Minimization Algorithms for VLSI Synthesis. Boston, MA, Kluwer Academic Publishers, 1984.

[4] Hachtel, G. D.—Somenzi, F.: Logic Synthesis and Verification Algorithms. Boston, MA, Kluwer Academic Publishers, 1996, pp. 564.

[5] Rudell, R. L.—Sangiovanni-Vincentelli, A. L.: Multiple-Valued Minimization for PLA Optimization. IEEE Trans. on CAD, Vol. 6, 1987, No. 5, pp. 725–750.

[6] McGeer, P. et al.: ESPRESSO-SIGNATURE: A New Exact Minimizer for Logic Functions. In Proc. of the Design Automation Conf. '93.

[7] Nguyen, L.—Perkowski, M.—Goldstein, N.: Palmini — Fast Boolean Minimizer for Personal Computers. In Proc. of the Design Automation Conf. '87, pp. 615–621.

[8] Coudert, O.—Madre, J. C.: Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions. Proc. of 29th DAC, Anaheim CA, USA, June 1992, pp. 36–39.

[9] Coudert, O.—Madre, J. C.—Fraisse, H.: A New Viewpoint on Two-Level Logic Minimization. Proc. of 30th DAC, Dallas TX, USA, June 1993, pp. 625–630.

[10] 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.

[11] 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, pp. 291–298.

[12] Hlavička, J.—Fišer, P.: A Heuristic Method of Two-Level Logic Synthesis. Proc. The 5th World Multiconference on Systemics, Cybernetics and Informatics SCI'2001, Orlando, Florida (USA) 22–25. 7. 2001, Vol. XII, pp. 283–288.

[13] Fišer, P.—Hlavička, J.: On the Use of Mutations in Boolean Minimization. Proc. Euromicro Symposium on Digital Systems Design, Warsaw (Poland) 4.–6. 9. 2001, pp. 300–307.

[14] Hlavička, J.—Fišer, P.: BOOM — a Heuristic Boolean Minimizer. Proc. ICCAD-2001, San Jose, Cal. (USA), 4.–8. 11. 2001, pp. 439–442.

[15] `http://eda.seodu.co.kr/ chang/download/espresso/`.

[16] McCluskey, E. J.: Logic Design Principles. Prentice-Hall, Englewood Cliffs, 1986.

[17] Chatterjee, M.—Pradhan, D. J.: A Novel Pattern Generator for Near-Perfect Fault Coverage. Proc. of VLSI Test Symposium 1995, pp. 417–425.

[18] Touba, N. A.—McCluskey, E. J.: Transformed Pseudo-Random Patterns for BIST. CRC Technical Report No. 94-10, 1994.

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

[20] Servít, M.: A Heuristic Method for Solving Weighted Set Covering Problems. Digital Processes, Vol. 1, 1975, No. 2, pp. 177–182.

[21] Coudert, O.: Two-Level Logic Minimization: An Overview, Integration. The VLSI Journal, 17-2, pp. 97–140, Oct. 1994.

[22] `ftp://ic.eecs.berkeley.edu`.

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

[24] `http://service.felk.cvut.cz/vlsi/prj/BoomBench`.

[25] Fišer, P.—Hlavička J.: A Set of Logic Design Benchmarks. Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS'02), Brno (Czech Rep.), 17.–19. 4. 2002, pp. 324–327.

[26] Fišer, P.—Hlavička J.: Column-Matching Based BIST Design Method. Proc. 7th IEEE European Test Workshop (ETW'02), Corfu (Greece), 26.–29. 5. 2002, pp. 15–16.

[27] Hartmann, J.—Kemnitz, G.: How to Do Weighted Random Testing for BIST. Proc. of International Conference on Computer-Aided Design (ICCAD), pp. 568–571, 1993.

Petr Fišer received the MSc. degree in electrical engineering at the Czech Technical University in Prague in 2002, currently is a PhD. student at the same university, faculty of Computer Science and Engineering. His main areas of interest are Boolean minimization, decomposition, build-in self-test (BIST) and design for testability (DFT).

Jan Hlavička (1942–2002) graduated from Faculty of Electrical Engineering, Czech Technical University in 1964. He received his PhD and DSc degrees from the same university in 1971 and 1987, respectively, and was appointed Associate Professor and Professor by the same institution in 1985 and 1991, respectively. During his fruitful scientific career, he was with the Research Institute for Mathematical Machines in Prague, Siemens Munich, and Department of Computer Science and Engineering of the Faculty of Electrical Engineering, CTU Prague. He was the visiting professor at TH Ilmenau (Germany), Universit de Montréal (Canada) and Hochschule fr Bauwesen Cottbus (Germany). He is the author and co-author of numerous scientific papers. His research interests indluded fault-tolerant computing testing and diagnostics of digirtal circuits an systems,computer architecture, error coding, self-checking circuits.