

Fast Boolean Minimizer for Completely Specified Functions

Petr Fiser¹, Premysl Rucky¹, Irena Vanova²

¹ Dept. of Computer Science and Engineering, Czech Technical University, Karlovo nam. 13, CZ-121 35 Prague 2

² UTIA, CAS, Pod Vodarenskou vezi 4, CZ-182 08, Prague 8

e-mail: fiserp@fel.cvut.cz, ruckyp@fel.cvut.cz, vanovai@utia.cas.cz

Abstract: We propose a simple and fast two-level minimization algorithm for completely specified functions in this paper. The algorithm is based on processing ternary trees. A ternary tree is proposed as a structure enabling a very compact representation of completely specified Boolean functions. It is efficient especially for functions having many on-set terms. The minimization algorithm is thus most suited for functions described by many on-set terms. Such functions emerge as a result of many algorithms used in logic synthesis process, e.g., multi-level network collapsing, algebraic manipulation with logic functions, etc. When these functions are to be minimized, most of the state-of-the-art minimizers (Espresso) need prohibitively long time to process them, or they are even completely unusable, due to their very high memory consumption. Our algorithm is able to minimize such functions in a reasonable time, though the result quality does not reach the quality of other minimizers. Here our minimizer found its application as a pre-processor that, when run prior to, e.g., Espresso, significantly reduces total minimization time, while fully retaining the result quality.

I. INTRODUCTION

The two-level Boolean minimization problem occurs in many areas of the logic design [1], in the built-in self-test (BIST) design [2], in a design of control systems, etc. Starting with the basis of minimization algorithms stated in 50's by Quine and McCluskey [3], many different minimizers have been developed (MINI [4]), ending up in Espresso [5] with its later improvements [6, 7]. Lately, two two-level Boolean minimizers Boom [8, 9] and FC-Min [10] have been developed, which were afterwards combined together, into BOOM-II [11, 12].

All these methods suffer from their own specific drawbacks when solving problems of different kinds. For example, Espresso cannot solve functions with a large number of inputs (>100) in a reasonable time. Moreover, functions having a large portion of don't cares (i.e., heavily incompletely specified functions) make troubles here as well. BOOM (-II) solves these problems efficiently, but, on the other hand, it needs to have the function's off-set specified explicitly, which limits its usability in some cases.

During our research in the area of logic synthesis, namely the collapsing of multi-level networks, i.e., deriving their two-level description, the need of a minimization of completely specified two-level functions emerged. These functions are specified by their on-set only, in the sum-of-products (SOP) form. The number of terms defining

the on-set is extremely high (up to millions). Thus, there was a need of a compact representation of two-level descriptions of these functions that consumes low memory and, simultaneously, retains the structure of the product terms. The ternary tree, firstly proposed in [9, 13] as a "tree buffer", has been found to be a very good choice. The principles and properties of the ternary tree structure are described in this paper into detail.

Similar tree-like structures, like binary decision diagrams (BDDs) [17], ternary decision diagrams (TDDs) [18], modified decision diagrams MDDs [19] or term trees [20] are not suitable for our purposes, since they do not retain the structure of the product terms they have been constructed of. Moreover, their size heavily depends on the nature of the function and the variable ordering in the tree. On the contrary, the size of the proposed ternary tree grows only with the number of product terms stored in it.

A minimization method *Pupik*, based on a reduction of the ternary tree, is proposed in this paper. It is based on merging terms neighboring in the Boolean space. However, it can be further augmented to support more Boolean manipulations. The comparison of results obtained by Espresso, the proposed method and by their combination is presented.

The paper is structured as follows: after the introduction, the problem statement is given. Section III describes the ternary tree structure and its properties. The ternary tree based minimization method is described in Section IV, the minimization results are shown in Section V. Section VI concludes the paper.

II. PROBLEM STATEMENT

Let us have a single-output Boolean function of n input variables. The input variables will be denoted as x_i , $0 \leq i < n$. Output values of the on-set terms (both minterms and product terms of higher dimensions may be used) are defined by a truth table. The function is completely specified, thus values of minterms that are not contained in terms in the truth table are assigned to zero.

Thus, we have an n -variable Boolean function defined by a sum-of-product (SOP) form as an input. The number of product terms will be denoted as p . Our aim is to minimize the number of terms (p). The secondary aim could be the reduction of the number of literals in the terms, thus increasing the dimension of the terms.

III. TERNARY TREE

The proposed minimization algorithm is based on processing of a *ternary tree* structure. The ternary tree has been proposed for the first time in [13] as a *tree buffer*. The ternary tree structure was used to store and, more importantly, quickly look up product terms. The main implementation requirement for the buffer was its high look-up speed. However, all the capabilities of the structure were not discovered at that time.

The ternary tree depth is equal to the number of inputs of the function (n). The tree is gradually constructed by adding product terms into it. Let us define a total ordering $<$ over the set of input variables, the function $var(i)$ gives the input variable of the function in the i -th order. Each level of the ternary tree corresponds to one variable, according to the ordering. Each non-terminal node in a level i corresponds to a “partial” product term, where values of only $var(0) \dots var(i-1)$ variables are defined. Terminal nodes correspond to completely described terms.

An example of a ternary tree for a 3-input function is shown in Fig. 1. Three terms are contained in the tree, namely 0-0, 10- and 11-. Each non-terminal node u may have three potential children, $lo(u)$, $dc(u)$, $hi(u)$. In our example in Fig. 1, $lo(u)$ is the left child, $dc(u)$ the middle one and $hi(u)$ the right one.

When inserting a term into the tree, at the i -th level of the tree the branch is chosen according to the polarity (0, -, 1) of the i -th variable in the term. If the corresponding branch is present, we follow it, when not, the branch is newly created.

Checking for a presence of a term is of a complexity $O(n)$, however, if the term is not present in the tree, the search terminates in less than n steps. If, e.g., term 011 being is looked for, the search will fail in the node ‘0’ where no path leading to ‘01’ is present.

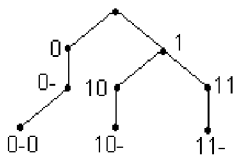


Figure 1. Ternary tree example

A. Comparison with Other Representations

The ternary tree buffer was primarily designed to store product terms and quickly look-up them. Thus, there has been put a big emphasis on its lookup speed. Since the depth of the tree is equal to the number of variables, the maximum number of steps needed to find out if a particular term is present in the tree is n . This happens in a case when the term is present in the tree, thus the tree has to be traversed up to the leaf. On the other hand, the *minimum* number of comparisons needed to find the presence of an implicant in the tabular representation is n . This happens when the searched term is the first one in the table of terms. The minimum and maximum look-up speeds for both representations are summarized in Table I. We can see that the maximum look-up

speed for the tabular representation is $n.p$, while the maximum look-up speed of ternary tree is always n .

TABLE I
COMPARISON OF LOOK-UP SPEEDS

	on success	on failure
Ternary tree	n	$1 \dots (n-1)$
Truth table	$n \dots n.p$	$n.p$

Let us note that inserting a term into the tree always takes n steps as well. Here the ternary tree brings another advantage w.r.t. a standard truth table: there is no chance for duplicate terms to occur. If a term that is already present in the tree is being inserted, it is found during the insertion process. This cannot be done in the tabular representation, unless the term is looked up first, i.e., the with a complexity of $O(n.p)$.

Similar tree-like structures, like BDDs [17], MDDs [19], TDDs [18] or term trees [20] cannot be used for our purposes, since they have different properties than ternary trees. Generally, these structures are used as a representation of a function, not as a *storage of terms*. Their size heavily varies with the nature of the function they are describing and the variable ordering [17]. The size of the ternary tree changes with the number of stored terms only. Moreover, BDDs, etc., do not retain the structure of product terms they were constructed of – the original terms cannot be reconstructed. Term trees [20], on the other hand, do not have a fixed variable ordering, which is required for the minimization algorithm proposed below. All these competitive tree structures suffer from one common drawback: an operation of extension of the function by one product term usually involves a sometimes time-consuming resynthesis of the whole tree. Addition of a term into our ternary tree always takes n steps. For all these reasons, any comparison with these other tree structures becomes irrelevant.

The ternary trees most closely resemble SOP TDDs, briefly described in [17]. We basically extend the SOP TDDs notion by introducing new operations and their new application areas.

IV. THE MINIMIZATION ALGORITHM

The minimization algorithm, named *Pupik*, is very simple and straightforward. It is based on applying absorption and complement property rules of Boolean algebra only, targeting the reduction of the number of the ternary tree terminal nodes (leaves). Particularly, when a non-terminal node at the $(n-1)$ -th level has two successor nodes (which are thus terminals), they always may be merged into one DC terminal, either by applying the absorption rule (in a case of a 0- or 1-terminal together with a DC terminal) or a complement properties rule (in a case of a 0- and 1-terminal).

The principles of the reduction are illustrated by the following example. Let us consider a function $y = x_1 + x_2 + x_3$ described by its on-set minterms (see Table II). It is uniquely represented by a ternary tree shown in Fig. 2.

TABLE II
THE EXAMPLE FUNCTION

minterm	x_1	x_2	x_3	y
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

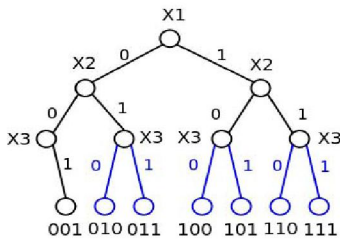


Figure 2. Minimization example (1)

There are seven terminal nodes representing the on-set minterms 1-7. It can be easily seen that minterm couples (010, 011), (100, 101) and (110, 111) may be merged, to obtain DC terminals. See Fig. 3.

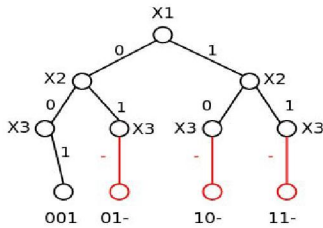


Figure 3. Minimization example (2)

The ternary tree now represents a function described by the list of on-set terms shown in Tab. III. The Complement property rule of Boolean algebra has been applied for the variable x_3 . No other tree reduction can be performed at this time, thus another phase of the minimization algorithm follows – the tree rotation.

TABLE III
THE FUNCTION FROM FIG. 3

x_1	x_2	x_3	y
0	0	0	0
0	0	1	1
0	1	-	1
1	0	-	1
1	1	-	1
1	1	-	1

A. Tree Rotation

The algorithm proposed in the previous subsection considers a minimization of the number of terminal nodes only, i.e., only the last variable (x_3) is being omitted, if possible. Thus, the next step to follow is obvious: the rotation of the tree, so that non-terminals become terminals. Then the terminal minimization procedure is performed again. The whole process is repeated n -times (where n is the number of input variables), so that all the variables are tried for removal. Moreover, the quality of the result may be improved by repeating the whole minimization process several times, i.e., running it for i iterations. See Section IV.B.

The tree rotation is done by cutting off the root node, which yields three separated trees (at most). Then, the root variable is appended to all leaves of the trees. The rotation of the tree from Fig. 3 is shown in Fig. 4. The tree is split into two trees only, since the root of the original tree had two successors.

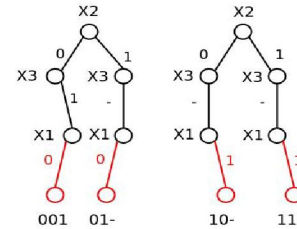


Figure 4. Minimization example (3)

Then the trees are merged together, by traversing these trees from the roots in parallel and merging nodes. The result is shown in Fig. 5. Notice that the four terminals remain unchanged; the rotated tree describe the same set of terms as in Fig. 3.

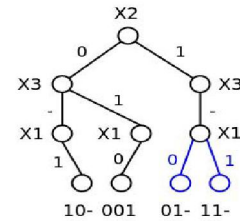


Figure 5. Minimization example (4)

Now the new terminals may be merged. Each terminal merging results in a removal of a particular (terminal) variable from a term. The tree is rotated n -times, the resulting ternary tree after 3 rotations is shown in Fig. 6, representing three terms (001, -1-, 10-), which is the minimum representation of the source function.

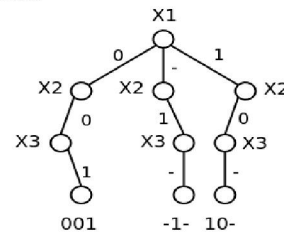


Figure 6. Minimization example (5)

The minimization method described above performs merely two operations of Boolean algebra:

The one-variable absorption rule: $a+ab = a$

The complement property: $ab + ab' = a$

It is not able to disclose more complex relations, like even the absorption of more than one variable. However, this can be solved at the end of the whole minimization process, by removing terms that are absorbed by any other term. Each term is checked against each tree terminal and, if it is absorbed by it, it is removed from the solution.

The minimization procedure (merging the terminals and rotation) could be repeated for an arbitrary number of iterations, to gradually improve the result quality, see Subsection IV.B.

The same minimization process may be performed upon the tabular representation of the function, nevertheless using the tree representation is more advantageous, in terms of the computational time. The leaf merging procedure corresponds to merging (or omitting) of rows in the truth table that differ in the last variable only (see Table II). Rotation of the tree corresponds to the rotation of the truth table columns.

The asymptotic complexity of the algorithm is computed as follows: let us assume that the algorithm runs for i iterations, each iteration comprising of n terminal merging operations and rotations. Each of these operations involve traversing the whole tree once, thus exploring $n \cdot p$ nodes at most (where n is the number of input variables and p the number of terms). Thus, the overall asymptotic complexity of the algorithm is $O(i \cdot n^2 \cdot p)$.

If the tabular representation were used, merging and omitting the rows involves a comparison of every pair of rows, which takes $O(n \cdot p^2)$ steps. This is performed for n columns i -times, thus the overall complexity of the minimization algorithm is $O(i \cdot n^2 \cdot p^2)$. The gain obtained by using the tree representation of the function is apparent now. The minimization method is advantageous especially for functions with many on-set terms, which it is targeted to.

The whole minimization algorithm can be described by the following pseudo-code:

```

Minimize (F) {
  t = CreateTree(F);
  for (i = 0; i < n*iterations; i++)
  {
    t->Merge_leaves(F);
    t->Divide_and_Merge(F);
  }
  F = Dump_Tree(t);
  RemoveAbsorbed(F, t);
  return F;
}

```

Algorithm 1. The minimization algorithm

B. The Iterative Process

In each leaf merging process one particular variable is removed from some terms, it is substituted by a don't care value. As a consequence of this, two terms that were not

neighboring terminals (i.e., having a common parent) may become neighbors after a tree rotation.

An example of such a function is shown in Fig. 7. We can see that even after 2 rotations ($n = 2$), there are still terms which can be merged.

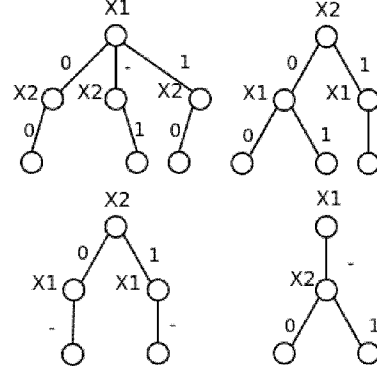


Figure 7. Minimization with number of rotations $> n$

Thus, rotating the tree more than n -times could improve the result quality. Figure 8 shows how fast the number of terms in the tree decreases with increasing number of rotations. The function shown in the figure has 36 variables (the ISCAS'85 c432 benchmark circuit) and as we can see, the most significant decrease of the number of terms happen in the initial n (36) rotations. After that, the number of terms decreases only slightly. Thus, n rotations (i.e, one iteration) will be considered in the following experimental section. However, the result quality may be further improved by increasing the number of iterations.

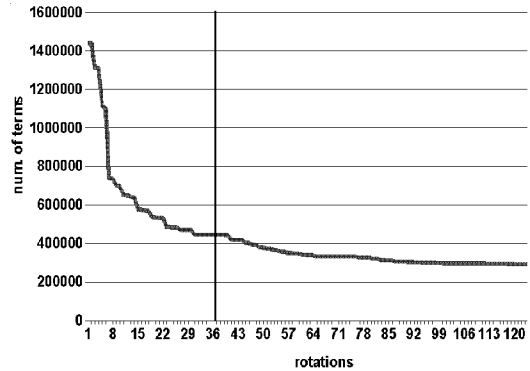


Figure 8. Relation between the number of rotations and the number of terms

V. EXPERIMENTAL RESULTS

The experimental results will be presented in this section. As it was stated before, the proposed minimization algorithm performs best in a combination with another Boolean minimizer, Espresso in our case. *Pupik* is run first, to quickly reduce the number of product terms. Then a more powerful minimizer is applied to the resulting function, to further reduce the number of products.

When our minimizer is run prior to Espresso, the result quality remains unchanged (w.r.t. Espresso), but the total runtime is significantly reduced.

A. ISCAS Benchmarks

We will present experimental results obtained by practical examples, particularly by the collapsing of multi-level ISCAS [14, 15] benchmarks into their two level representations, each output forming a stand-alone function. The network collapsing problem is known to be one of the most computationally demanding problems in logic synthesis [1]. The number of product terms obtained by the process grows exponentially with the number of inputs of the circuit. Thus, any reduction of the number of product terms in the resulting representation is extremely beneficial. Here the ternary tree based minimizer found one of its application areas. First of all, many duplicate product terms have been produced during the collapsing process [16]. The ternary tree structure used to store these terms prevented an occurrence of duplicities. Then, the number of product terms was reduced by applying the minimization algorithm. The results are shown in Tab. IV. The benchmark name is followed by the ordinal number of its output that has been collapsed. The size of the two-level function that is to be minimized is indicated in the second column (# of inputs / # of product terms). Then, respective number of product terms the computational time obtained by Espresso, *Pupik* and their combination is shown.

The used benchmark circuits and their outputs that were collapsed were chosen according to the effort needed to collapse them. The easy-to-collapse benchmarks/outputs were not considered for their simplicity, as well as the very hard-to-collapse ones, due to extreme time and memory demands.

It can be seen that our minimization method significantly lacks in the result quality when compared to Espresso, however our minimizer runtimes are always negligible. When *Pupik* and Espresso are combined, i.e., *Pupik* is run prior to Espresso as a pre-processor, the result quality is retained, and the total runtime is significantly reduced, especially for the most complex benchmark circuits. Thus, here *Pupik* finds its actual application area – to be run as a pre-processor to another Boolean minimizer, to reduce the total minimization runtime.

The experiments have been performed on PC with Athlon64 1.8GHz CPU and 1GB RAM.

B. Randomly Generated Functions

In order to estimate the limits of usability of our minimizer and Espresso, we have tested these two minimizers on randomly generated function, with a very high number of on-set terms. Different extreme-case benchmarks were randomly generated, for different numbers of input variables and on-set terms. The percentages of don't care states in the terms varied linearly from 0 to 90% for each circuit. This often makes the circuit hard to minimize, moreover such functions resemble real circuits.

The results are shown in Table V. The number of input variables and defined care terms is given in the "Benchmark"

column. Next, the results obtained by Espresso and *Pupik* are shown (number of terms in the result, computational time). We can see that the computational time of *Pupik* is negligible for these functions, while Espresso times are usually prohibitively large, for most of functions it did not produce any result after one day of computation. Even though the results obtained by *Pupik* are far from optimum, any reduction of the number of terms is beneficial, when other minimizers fail.

VI. CONCLUSIONS

A novel two-level minimization method for single-output Boolean functions is presented. It is based on a reduction of a ternary tree, which we propose as an internal representation of a Boolean function, or, more precisely, of its on-set terms. The minimization process significantly reduces the number of product terms in the sum-of-product form, however it does not reach the qualities of Espresso. On the other hand, the method significantly outperforms Espresso in a runtime. For some randomly generated problems, Espresso was not able to even produce any result. The proposed minimization method has found its most beneficial application as a pre-processor of functions specified by a very large number of product terms (up to millions). When Espresso is used afterwards to minimize the function, a big total minimization time reduction is obtained, while the result quality remains the same, as if only Espresso was run.

The ternary tree structure with its properties is described into detail in this paper. It is a very compact and flexible representation of a completely specified Boolean function described by an SOP form. It offers many additional benefits with respect to the tabular form representation, namely in the linear-time searching capabilities. Since the ternary tree structure implicitly avoids duplicities, it could be advantageously used in applications where many duplicate product terms are gradually generated, such as multi-level Boolean network collapsing, algebraic manipulations with Boolean functions, etc.

The minimization method was tested on random and standard ISCAS benchmarks and the results compared.

Further research will be directed towards implementing several additional operations upon the ternary tree, namely a more sophisticated support for Boolean absorption. The method still does not produce an irredundant cover; redundant terms cannot be identified. Introduction of techniques removing redundancies will be beneficial too.

The method should be then extended to support multi-output and incompletely specified functions.

ACKNOWLEDGMENT

This research has been supported by MSMT under research program MSM6840770014.

REFERENCES

- [1] S. Hassoun and T. Sasao, „Logic Synthesis and Verification”, Boston, MA, Kluwer Academic Publishers, 2002, 454 pp.
- [2] Agarwal, Kime, Saluja: “A tutorial on BIST, part 1: Principles”. IEEE Design & Test of Computers, vol. 10, No.1 March 1993, pp.73-83, part 2: Applications, No.2 June 1993, pp.69-77
- [3] E.J. McCluskey, “Minimization of Boolean functions”, The Bell System Technical Journal, 35, No. 5, Nov. 1956, pp. 1417-1444
- [4] S.J. Hong, R.G. Cain and D.L. Ostapko, “MINI: A heuristic approach for logic minimization”, IBM Journal of Res. & Dev., Sept. 1974, pp.443-458
- [5] R.K. Brayton et al., “Logic minimization algorithms for VLSI synthesis”, Boston, MA, Kluwer Academic Publishers, 1984, 192 pp.
- [6] R.L. Rudell and A.L. Sangiovanni-Vincentelli, “Multiple-valued minimization for PLA optimization”, IEEE Trans. on CAD, 6(5): 725-750, Sept.1987
- [7] P. McGeer et al., “ESPRESSO-SIGNATURE: A new exact minimizer for logic functions”, Proc. DAC’93
- [8] J. Hlavička, P. Fišer, „BOOM - a Heuristic Boolean Minimizer”, Proc. ICCAD 2001, San Jose, Cal. (USA), 4.-8.11.2001, 439-442
- [9] P. Fišer, J. Hlavička, „BOOM - A Heuristic Boolean Minimizer”, Computers and Informatics, Vol. 22, 2003, No. 1, pp. 19-51
- [10] P. Fišer, H. Kubátová, “Boolean Minimizer FC-Min: Coverage Finding Process”, Proc. 30th Euromicro Symposium on Digital Systems Design (DSD’04), Rennes (FR), 31.8. - 3.9.04, pp. 152-159
- [11] P. Fišer, H. Kubátová, “Two-Level Boolean Minimizer BOOM-II”, Proc. 6th Int. Workshop on Boolean Problems (IWSBP’04), Freiberg, Germany, 23.-24.9.2004, pp. 221-228
- [12] P. Fišer, H. Kubátová, “Flexible Two-Level Boolean Minimizer BOOM II and Its Applications”, Proc. 9th Euromicro Conference on Digital Systems Design (DSD’06), Cavtat, (Croatia), 30.8. - 1.9.2006, pp. 369-376
- [13] P. Fišer, J. Hlavička, “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
- [14] F. Brglez and H. Fujiwara, „A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran”, Proc. of ISCAS 1985, pp. 663-698
- [15] F. Brglez, D. Bryan and K. Kozminski, „Combinational Profiles of Sequential Benchmark Circuits”, Proc. of ISCAS, pp. 1929-1934, 1989
- [16] P. Rucký, “Multi-level Logical Circuit Collapsing Using Binary Decision Diagrams (BDDs)”, MSc. Thesis, Czech Technical University in Prague, 2007, 57 pp.
- [17] S. B. Akers, “Binary decision diagrams”, IEEE Trans.on Computers, Vol. C-27. No. 6, June 1978, pp. 509-516.
- [18] T. Sasao, “Ternary Decision Diagrams - A Survey”, Proc. of IEEE International Symposium on Multiple-Valued Logic, pp. 241-250, Nova Scotia, May 1997
- [19] R. P. Jacobi, “A Study of the Application of Binary Decision Diagrams to Multi-level Logic Synthesis”. Nivel de doutorado, Université Catholique de Louvain, Belgique, 1993.
- [20] L. Jozwiak, A. Slusarczyk and M. Perkowski, “Term Trees in Application to an Effective and Efficient ATPG for AND-EXOR and AND-OR Circuits”, VLSI Design, Volume 14, Number 1, 1 January 2002, pp. 107-122

TABLE IV
ISCAS BENCHMARKS

Benchmark	Size (n/p)	Espresso		Pupik		Pupik + Espresso	
		Terms	Time [s]	Terms	Time [s]	Terms	Time [s]
c880_3	60 / 212290	15673	470	57853	23.11	15673	305
c880_4	60 / 67136	440	37	12046	6.44	439	8.5
c1908_4	33 / 32768	7040	20	20990	1.1	7040	17
c1908_5	33 / 6464	2144	1	4640	0.19	2144	2
c1908_6	33 / 13700	3200	4	8704	0.41	3200	4.5
c3540_1	50 / 403298	32455	4740	101512	32.9	32396	1569
c3540_2	50 / 42568	10202	254	19620	4.2	10065	193
c3540_3	50 / 4464	1022	2	2112	0.34	1022	2.5
c3540_4	50 / 1912	184	0.1	459	0.07	184	0.1
c3540_5	50 / 6657	1516	5	3654	0.62	1522	4.5
c3540_6	50 / 159920	14397	900	43442	9.09	14397	382
c3540_7	50 / 6933	611	1	2360	0.37	611	1

TABLE V
RANDOM BENCHMARKS

Benchmark	Espresso		Pupik	
	Terms	Time [s]	Terms	Time [s]
30 / 5000	35	17.82	1004	0.63
35 / 5000	1169	1306.34	1719	1.01
35 / 8000	200	400.98	3703	2.41
40 / 5000	-	> 24 h	3033	5.5
40 / 10000	-	> 24 h	5054	7.0
40 / 20000	-	> 24 h	6606	13.72
50 / 30000	-	> 24 h	13232	33.6
60 / 50000	-	> 24 h	38203	94.9