BoolTool: A Tool for Manipulation of Boolean Functions

Petr Fišer, David Toman
Czech Technical University in Prague
Department of Computer Science and Engineering
Karlovo nám. 13, 121 35 Prague 2
e-mail: fiserp@fel.cvut.cz, tomand1@fel.cvut.cz

Abstract

A new tool for manipulation of logic functions is presented in this paper. The source functions are described by an algebraic expression (or a set of expressions), in a VHDL-like style, by a truth table (PLA) or as a CNF form. In particular, any multilevel network of logic gates can be used as a source for the tool. The tool is capable of performing basic Boolean operations on the source functions, like negating the function, performing AND, OR, XOR operations, etc., between two or more functions. Then, the tool is able to mutually transform the CNF and DNF function representations, which also enables to solve a satisfiability (SAT) problem as a byproduct. Last, but not least, the tool performs a function collapsing, i.e., it transforms a multi-level Boolean network into its two-level description (truth table).

Some Boolean operations, like computing a negation of a function in DNF, are rather time and memory consuming. For this reason a special structure called a "ternary tree" is introduced. The ternary tree is used to store the function's terms and to perform a basic minimization of the function representation. Basic principles of the ternary tree representation of a function and its minimization principles are described in this paper too.

The proposed tool was tested on several different problems (minimization of a function, SAT solving, collapsing) and compared with state-of-the-art tools.

1. Introduction

The logic synthesis process has undergone a great progress since 1960's, mostly due to a rapid development of EDA (Electronic Design Automation) tools. There are many logic synthesis tools available, either commercial or open-source, like Espresso [1] and Boom [2, 3] for a two-level minimization of combinational functions, SIS [4], MVSIS [5] and ABC [6] for a multi-level sequential synthesis. Very sophisticated synthesis algorithms are employed there, offering the end user powerful synthesis tools. However, all these tools are primarily targeted towards conducting standard synthesis processes. To our best knowledge, there is no tool available enabling the user to perform elementary Boolean operations upon logic functions or Boolean networks. Therefore we have developed such a tool (*BoolTool*) enabling us to efficiently manipulate Boolean functions. There always is a possibility to "hard-wire" the required operation in some multi-level function specification format, like VHDL or BLIF [7], and to synthesize the resulting function by, e.g., SIS [4] or ABC [6]. However, this procedure is rather inconvenient in most cases and it cannot be easily performed using a script. We offer an easy-to-use tool able to perform any simple Boolean operation upon Boolean functions, or, better, Boolean expressions. The tool can be very efficiently exploited using a script file, so that the overall design process would be maximally automated.

Some of Boolean operations, like computing a complement of a function, involve exponential growth of the computational time and, more importantly, exponential growth of the memory consumption. Moreover, many duplicate terms are being produced in the computation process. Thus, a sophisticated structure to store the function's on-set terms is needed, so the duplicities are avoided and the number of terms could be reduced using a fast minimization algorithm. We propose a *ternary tree* structure enabling both.

The paper is structured as follows: the BoolTool principles and its capabilities are described in Section 2. The ternary tree structure and the ways of its minimization are shown in Section 3. Section 4 contains some experimental results and Section 5 concludes the paper.

2. The BoolTool

2.1. Basic Description

BoolTool is a powerful tool for manipulation of logic functions which are described either by a two-level representation (PLA [1, 4]) or as a multi-level Boolean network, in a structural VHDL-like format. The DIMACS [8, 16] format is also supported, to allow employing BoolTool as a SAT solver.

The source function (or, better, a set of functions) is then processed either by interactively giving commands to BoolTool, or by a script. The result is returned in the PLA [1, 4], VHDL, BLIF [7] or DIMACS CNF format [16].

These operations are supported:

- Application of basic Boolean functions (NOT, AND, OR, NAND, NOR, XOR, XNOR)
- Transformation of an arbitrary Boolean network into an AND-OR-NOT representation
- Transformation of an arbitrary Boolean network into a network consisted of NAND or NOR gates only
- Transformation of an arbitrary Boolean network into a CNF or DNF representation, thus, collapsing the multi-level network to obtain a two-level representation
- Satisfiability (SAT) solving
- Cofactor computation

Using these operations, any Boolean function or transformation can be performed upon any given Boolean network and the result may be obtained in most of commonly used Boolean function or network descriptions.

2.2. Internal Representation of a Network and the Network Manipulation

A binary tree was chosen as an internal representation of a Boolean expression. One binary tree is constructed for each function (Boolean expression). Internal nodes of the tree represent binary operators, leaves represent input variables, see Fig. 1. Here a tree for the *(x_1 nor x_3) or (x_0 and x_2')* expression is shown.

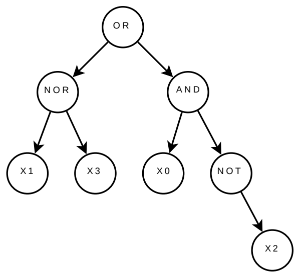

Figure 1. Boolean expression tree

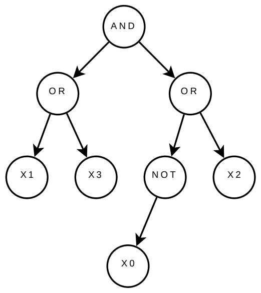

Figure 2. Negated expression tree

Such a representation offers us many benefits when performing Boolean operations upon it, since it can be processed recursively very easily. For example, performing a negation of a function is done by recursively traversing the tree from its root to the leaves. Nodes are gradually being substituted by their negated versions (AND to NAND, etc.). Moreover, DeMorgan's rules may be simultaneously applied (without any significant performance loss), to produce a tree constructed of AND-OR-NOT nodes only. When a NOT node is encountered, it is removed from the tree and the recursion is terminated. The resulting negated tree from Fig. 1 is shown in Fig. 2.

Simple Boolean operations (like AND, OR, XOR, ...) between two functions (expression trees) are being conducted in a straightforward way: the respective operator becomes a root of the resulting tree and the operand trees are appended to it as its successors.

2.3. Network Collapsing

The conversion of a Boolean network into a two-level representation of a function (i.e., into a CNF or, more frequently into a DNF) is an essential operation for many BoolTool applications. It is known that the multi-level network collapsing is computationally extremely demanding, since the memory consumption grows exponentially with the number of the function's input variables. En exemplary function to demonstrate it is an Achilles' heel function [9], whose CNF representation is polynomial in size, whereas its DNF representation is exponential.

The network collapsing process is conducted in the following way: first, the expression tree is processed by recursively applying DeMorgan's laws to it, to produce a tree constructed of AND, OR and NOT nodes. Simultaneously all the NOT nodes are moved to the leaves. Then distributivity laws are applied to the tree, to obtain the required DNF form.

Since an extreme growth of the number of the expression tree nodes is usually observed, a fast intermediate minimization of sub-trees should be performed, to reduce time and memory demands of the collapsing process. For this purpose a simple and fast two-level minimization algorithm is employed. Since the tree is being transformed into DNF from leaves to the root, a two-level minimization can be performed at each transformation step. A very fast ternary tree based minimization algorithm is thus being run after each collapsing step, to reduce the overall expression tree size. The algorithm will be described in Section 3 into details.

2.4. SAT Solving

In order to solve the Boolean satisfiability (SAT) problem, the expression tree is first converted into a DNF (i.e., sum-of-products) representation. Then the function's satisfiability check can be performed in linear time, by traversing the AND-OR tree. A '1' value is being distributed from the root to the leaves. Variables are assigned a value at the respective leaves. The conflicting variable is identified when a value opposite to already assigned value is to be assigned to the variable.

Let us note that, in contrast to many well known and commonly used SAT-solvers [10, 11], BoolTool is able to compute all the solutions of the solved SAT, not only to perform a satisfiability check. This makes the SAT solver very time and memory consuming. However, the obtained results may be advantageously exploited in many areas of logic design, e.g., in automatic test pattern generation (ATPG) process.

3. Ternary Tree

The two-level minimization algorithm employed in BoolTool is based on processing of a *ternary tree* structure. The ternary tree has been proposed for the first time in [12] 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 to 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 at a level i corresponds to a "partial" product term, where values of only $var(0)\dots var(i\text{-}1)$ variables are defined. Terminal nodes correspond to completely described terms.

An example of a ternary tree of a 3-input function is shown in Fig. 3. Three terms are contained in the tree, namely 0-0, 10- and 11- (a dash stands for a don't care, i.e., the respective variable is not present in the term). Each non-terminal node u may have three potential children, $lo(u)$, $dc(u)$, $hi(u)$. In our example, $lo(u)$ is the left-hand child, $dc(u)$ the middle one and $hi(u)$ the right-hand 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 in a tree shown in Fig. 3, the search will fail in the node '0' where no path leading to '01' is present.

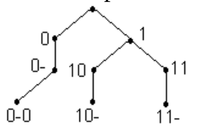

Figure 3. Ternary tree example

3.1. The Tree Minimization Algorithm

The minimization algorithm used in BoolTool 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 1). It is uniquely represented by a ternary tree shown in Fig. 4. 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. 5. The Complement property rule of Boolean algebra has been applied to 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 1. 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

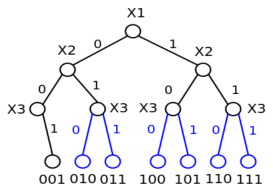

Figure 4. Minimization example (1)

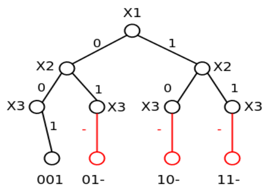

Figure 5. Minimization example (2)

3.2. Tree Rotation

The algorithm proposed in the previous subsection considers a minimization of the number of terminal nodes only, i.e., only the leaf variable (x_3) is being removed from the terms, 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 (which involves $n.i$ rotations).

The tree rotation is done by cutting off the root node, which yields three separate trees at most, each for one subtree rooted in $lo(u)$, $dc(u)$, $hi(u)$, where u is the root node. Then, the root variable is appended to all leaves of the three trees. The rotation of the tree from Fig. 5 is shown in Fig. 6. Here the tree is split into two trees only, since the root of the original tree has two successors (*lo* and *hi*).

Then the trees are merged together, by traversing these trees from their roots in parallel and merging nodes. The result is shown in Fig. 7. Notice that the four terminals remain unchanged; the rotated tree describes the same set of terms as in Fig. 5.

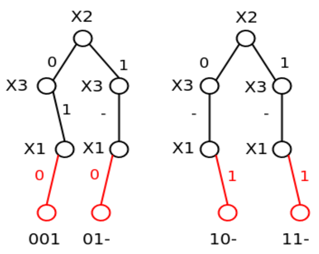

Figure 6. Minimization example (3)

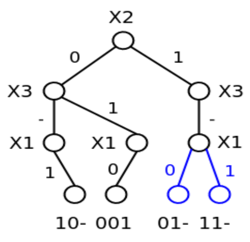

Figure 7. Minimization example (4)

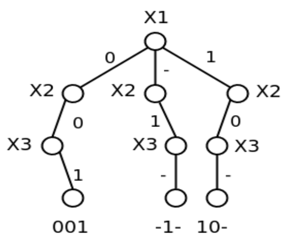

Figure 8. Minimization example (5)

Now newly obtained terminals may me 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. 8, representing three terms (001, -1-, 10-), which is the minimum representation of the source function.

For more details on the ternary tree minimization see [13].

4. Experimental Results

The experimental results presented in this section are shown just to give an insight into the capabilities and performance of BoolTool. The collapsing and SAT solving comparison results are not too positive for BoolTool. However, this is to be expected: more efficient structures and algorithms are used in the tools the comparison is done with. However, let us remind that BoolTool is a more general tool with different aims. Its collapsing and SAT solving capabilities have arisen as a byproduct here; we do not aspire to overpower other mature and, more importantly, dedicated tools. Unfortunately, as it was said in the introduction, we are not aware of any tools similar to BoolTool, thus no relevant comparison can be made.

Minimization capabilities of the built-in ternary tree based minimization algorithm are more impressive. However, presenting of these results is beyond the scope of this paper. For details see [13].

All the experiments have been performed on a 2 GB Core 2 Duo, 2 GB RAM PC, Windows XP.

4.1. The Collapsing Results

The results of collapsing of MCNC benchmark circuits [14] converted into their CNF form are shown in this subsection. A comparison with MVSIS [5] is made in Table 2. The computational times and the numbers of terms in the resulting DNF forms are shown. The “Terms” columns indicate the number of terms in the obtained sum of products expression.

Table 2. Collapsing results

	Time [s]		Terms	
Benchmark	BoolTool	MVSIS	BoolTool	MVSIS
b12	0.155	0.33	42	34
cordic	318.546	3.19	1757	1191
cps	5.566	1.23	4810	1870
duke2	2.883	0.37	2330	452
ex4	29.580	0.78	579	334
ex1010	64.530	2.62	11196	1415
misex2	0.065	0.08	286	146
misex3c	11.577	0.34	2502	508
pdc	204.837	30.26	2694	897
rd84	1.661	0.53	482	239
spla	197.931	32.27	1900	855

4.2. SAT Solving Results

Here we present SAT solving results, obtained by BoolTool and by a BDD-based approach [15]. This method was chosen as the only available candidate for a meaningful comparison, since it also generates all the solutions to the solved SAT. The benchmarks have been obtained from [16].

Table 3. SAT solving results

	Time [s]	
Benchmark	BoolTool	BDDCUDD
uf20-0500.cnf	4.162	0.780
uf20-0501.cnf	2.712	0.733
uf20-0502.cnf	2.584	0.749
uf20-0503.cnf	3.338	0.733
uf20-0504.cnf	2.408	0.718
uf20-0505.cnf	1.799	0.717
uf20-0506.cnf	4.335	0.733

	Time [s]	
Benchmark	BoolTool	BDDCUDD
uf20-0507.cnf	3.290	0.734
uf20-0508.cnf	2.544	0.733
uf20-0509.cnf	1.140	0.733
uf20-0510.cnf	3.077	0.733
uf20-0511.cnf	1.717	0.718

5. Conclusions

We have introduced a new tool for manipulation of logic functions. The tool is able to efficiently perform basic Boolean operations upon Boolean functions, like computing the complement of the function, computing unions and products of functions, cofactors, collapsing any Boolean network into its two level representation or solving a SAT problem. The input file format can be a PLA description (truth table) of a function, a CNF form or a VHDL-like multi-level network description. The resulting file format is, again, PLA, VHDL, BLIF or DIMACS CNF format.

Some of the implemented features require a multi-level network collapsing, which is computationally very intensive. For this reason, a very fast and efficient two-level minimizer is employed to reduce the size of intermediate results, yielding a significant reduction of the overall time and memory consumption. Basic principles of this minimizer are briefly presented in this paper.

The tool has been tested on several different problems on standard benchmark circuits and the performance evaluated and compared with other tools. However, since a tool of abilities comparable to BoolTool probably does not exist, we were not able to perform a sufficient comparatory evaluation of its performance. Anyway, BoolTool has found its applications in many areas of logic design up to now, as an essential part of the design process.

BoolTool is available for publics at [17].

Acknowledgment

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

References

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

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

[3] 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

[4] E.M. Sentovich et al.: SIS: A System for Sequential Circuit Synthesis, Electronics Research Laboratory Memorandum No. UCB/ERL M92/41, University of California, Berkeley, CA 94720, 1992

[5] D. Chai, J. Jiang, Y. Jiang, Y. Li, A. Mishchenko, and R. Brayton: MVSIS 2.0 User's Manual, UC Berkeley. Technical report, 2003

[6] Berkeley Logic Synthesis and Verification Group: ABC: A System for Sequential Synthesis and Verification, Release 70930. http://www.eecs.berkeley.edu/~alanmi/abc/

[7] Berkeley Logic Synthesis and Verification Group: Berkeley Logic Interchange Format (BLIF)

[8] http://www.qbflib.org/qdimacs.html

[9] T. Sasao, Switching Theory for Logic Synthesis. New York: Kluwer Academic, 1999.

[10] D. McAllester, B. Selman, H. Kautz: Evidence for invariants in local search. In Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97), pages 321–326, Providence, Rhode Island, 1997.

[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik: Chaff: Engineering an Efficient SAT Solver. In Proceedings of the 38th Design Automation Conference (DAC'01), 2001.

[12] P. Fišer, J. Hlavička: On the Use of Mutations in Boolean Minimization. Proc. Euromicro Symposium on Digital Systems Design, Warsaw (Poland) 4.-6.9.2001, pp. 300-305

[13] P. Fišer, P. Rucký, I. Váňová: Fast Boolean Minimizer for Completely Specified Functions, Proc. 11th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop 2008 (DDECS'08), Bratislava, SK, pp. 122-127

[14] S. Yang. Logic Synthesis and Optimization Benchmarks User Guide, Technical Report 1991-IWLS-UG-Saeyang, MCNC, Research Triangle Park, NC, January, 1991

[15] F. Somenzi: CUDD: CU Decision Diagram Package, Release 2.4.1, University of Colorado at Boulder (http://vlsi.colorado.edu/~fabio/CUDD/)

[16] http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

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