HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning

Wolfgang Kunz*

Max-Planck-Society
Fault-Tolerant Computing Group
University of Potsdam, 14415 Potsdam, Germany

Abstract

This paper introduces a new approach to logic verification of combinational circuits, which is based on recursive learning [1]. In particular, the described method efficiently extracts equivalencies between *internal* nodes of the two circuits to be verified. We present a tool, HANNIBAL, which is very efficient for many practical verification problems where such internal equivalencies exist. The presented method can also be used to drastically accelerate other verification tools. Experimental results clearly show the efficiency of HANNIBAL. For example, HANNIBAL verifies the multiplier c6288 against the redundancy-free version c6288nr in only 48 seconds on a Sparc Workstation ELC.

1. Introduction

In recent years, a lot of progress has been made in the automation of the design process for highly integrated circuits. Tools for automatic synthesis play an increasingly important role in the microelectronics industry. Nevertheless, also in the future there will be cases where the designer must resort to manual modifications or use some self-made software in order to fulfill the special requirements of his particular design. This phase of the design process is particularly prone to errors as the size of the circuits is growing continuously and the human designer has little insight into the functionality of his design, especially, after applying automatic synthesis procedures in an earlier design phase. Therefore, verification techniques have become of great importance and a lot of research is currently conducted to develop efficient verification methods for the various steps in the design process.

This paper deals with the (formal) logic verification problem for combinational circuits; i.e., with the problem of identifying whether two circuits are functionally equivalent or not. Two combinational circuits are functionally equivalent iff they respond to an arbitrary input pattern with the same output pattern. This work only considers completely specified functions, although this is no restriction for the approach to be presented.

Unfortunately, the problem of logic verification belongs to the most difficult problems in the field of computer-aided circuit design. This is particularly true for sequential circuits. A good survey of the state-of-the-art in sequential logic verification can be found in [2]. However, also for combinational circuits even the verification of small designs can lead to enormous computation times and require large amounts of memory. Although a lot of progress has been made with the introduction of Binary Decision Diagrams (BDDs) [3] and Ordered Binary Decision Diagrams (OBDDs) [4] logic verification still remains an active field of research since the construction of BDDs is not an easy task.

Since no general verification technique is known yet, which performs well for all classes of circuits, it is important to investigate the use of special properties of a given verification problem as they result from the nature of the design process. What properties can be used? A mathematical model to describe the design process is presented in [5]. In [5], the design process is viewed as a sequence of "atomic operations" that constitute the different design steps. Each design step must be followed by an accurate and efficient verification step in order to achieve high design quality at low costs. Motivated by the incremental nature of the design process, this paper presents a verification method which makes efficient use of the fact that many synthesis steps perform only local modifications in a given design. (Obviously, this is always true if we limit the number of atomic operations in each synthesis step.) Intuitively, the verification problem should become a lot easier, if the two circuits of comparison have some "structural similarity"; i.e. if they contain *internal* nodes that are functionally equivalent. It can be expected that there are many cases of practical verification problems where the two circuits contain equivalent nodes other than just the output nodes. Surprisingly, in spite of the incremental nature of the design process, only little research has been conducted on how to accelerate verification techniques by making use of the similarity between circuits. To the best knowledge of the author, only the techniques in [6] and [7] have addressed this practically important problem. In [7], a method is given to

* Most of the reported research was conducted while the author was with Institut für Theoretische Elektrotechnik, University of Hannover, Germany

establish "cross-relations" between well selected cuts through the two circuits. The philosophy of [6] is closely related to the approach in this work: a general technique is presented which is able to identify functionally equivalent nodes. With this technique, *any* other verification algorithm can be accelerated.

However, as pointed out [6], there is a serious difficulty when trying to exploit the "similarity" of two circuits: assume that the two circuits contain functionally equivalent nodes and that we simplify the verification problem by treating these nodes as independent input signals. As an example, let the two circuits contain structurally identical parts at the primary inputs. We are tempted to simplify the problem by "cutting" these parts off and to reduce the verification problem to the smaller, remaining part of the circuit, thus treating functionally equivalent nodes of the original circuit as independent primary inputs of the new, reduced circuit. It is important to realize that this or similar approaches can lead to *false negatives* [6]; i.e., in cases where the original, complete circuits are functionally equivalent, the reduced circuits may turn out to be *non*-equivalent. Therefore - against popular thought - even if two circuits have a lot of known internal equivalencies, it is not trivial to use this fact in order to reduce the complexity of the verification problem. The goal of this paper is to present a new approach to logic verification which can efficiently determine and exploit internal equivalencies:

1) Unlike [6], the presented verification tool can determine and use internal equivalencies *without* suffering from the false negative problem. If many internal equivalencies exist, the verification process is speeded up drastically.
2) The presented method to identify internal equivalencies is complete; i.e., all internal equivalencies can be identified, if given enough time.
3) The presented method can be used as a preprocessor to accelerate *other* state-of-the-art verification techniques (e.g., BDDs). In this case however, the problem of false negatives has to be addressed.
4) Unlike in [6], [7], it is not necessary to identify "candidate nodes" for functional equivalence. (More precisely, [7] determines "good cuts" in order to determine "cross relations" between the two circuits.) The technique to be presented performs one pass from the inputs to the outputs of the circuits, heuristics to focus on certain signals are not needed.
5) Memory requirements grow only linearly with the size of the considered circuits.

The method to be presented has been incorporated in the verification tool HANNIBAL (HANNover Implication tool BAsed on Learning). HANNIBAL is based on the recursive learning technique presented in [1] and [8]. Recursive learning is a general method to solve the Boolean satisfiability problem and hence also represents a complete algorithm for logic verification. Since recursive learning is a recent development a short review of the technique is given.

2. Review of recursive learning

By recursive learning, it is possible to make precise implications[1] for a given set of previously specified signals. Making precise implications means to identify *all* signal values that are uniquely determined by the given situation of value assignments. In other words, the precise implication procedure determines all signal values that are *necessary* for the consistency of the given situation of value assignments. This task is NP-complete, because it requires to solve the Boolean satisfiability problem. The following definitions are needed to describe the procedure:

Def. 1: Given a gate G that has at least one specified input- or output signal: The *gate* G is called *unjustified*, if there are one or several unspecified input- or output signals of G for which exist a combination of value assignments that yields a conflict at G. Otherwise, G is called justified.

Unjustified gates describe the locations *where* learning has to be performed. The next definition determines *what* signal values have to be injected at the unjustified gates in order to perform learning:

Def. 2: A set of signal assignments, $J = \{f_1=V_1, f_2=V_2, \ldots f_n=V_n\}$, where $f_1, f_2, \ldots f_n$ are unspecified input- or output signals of an unjustified gate G, is called *justification* for G, if the combination of value assignments in J makes G justified.

At every unjustified gate it is necessary to examine the logical consequences for a *complete set of justifications*:

Def. 3: Let ${}^{G}C$ be a set of m justifications $J_1, J_2, \ldots J_m$ for an unjustified gate G. If there is at least one justification $J_i \in {}^{G}C$, i=1,2...m for any possible justification J^* of G, such that $J_i \subseteq J^*$, then set ${}^{G}C$ is called *complete*.

For a given unjustified gate it is straightforward to derive a complete set of justifications. In the worst case, this set consists of all consistent combinations of signal assignments that represent a justification of the considered gate. Often, the set can be smaller as explained in [1].

All learning operations rely on a basic implication technique. As in [1], these basic implications are called *direct implications*. A well-known example of direct implications in combinational circuits are the implications as performed in FAN [9].

With the same notation as in [1] the precise implication procedure is given in Table 1. The reader may refer to [1] in order to find examples that illustrate the procedure.

```
initially: r=0;
make_all_implications(r, r_max)
  {
  make all direct implications as well as all prestored indirect impl.
  and set up a list U^r of all gates that become unjustified
  if r<r_max : learning
  {
    for each gate G_x, x=1,2.., in U^r: justifications
    {
      set up list of justifications Gx_C^r
      for each justification J_k ∈ Gx_C^r:
      {
        - make the assignments contained in J_k
        - make_all_implications(r+1, r_max)
      }
      If there is one or several signals f_i in the circuit, which assume
      the same logic value V_i for all consistent justifications J_k ∈ Gx_C^r,
      then learn: f_i =V_i is uniquely determined in level r, make direct
      implications for all learned values, erase all other values in level r

      If all justif. are inconsistent, then learn: given situation of value
      assignments in level r is inconsistent, erase all values in level r
} } }
```

Table 1: Precise implication procedure

Direct implications are performed in an event-driven way by evaluating the truth tables of gates that have an event and by propagating the signal values according to the connectivity in the circuit. *Indirect* implications are performed by learning. Learning means that justifications for unjustified gates are *temporarily* injected in order to examine their logical consequences. By performing this process recursively, it is possible to determine *all* necessary assignments. As in [1], r_{max} is a parameter which allows a reasonable tradeoff between the level of precision of the implication procedure, i.e. its ability to derive necessary assignments, and its computational complexity. Note that there is a little difference to the procedure in [1]. In HANNIBAL, indirect implications are stored similarly as with the static learning procedure of [10]. Therefore, the above routine performs all direct implications along with indirect implications that have been identified and stored before.

3. The algorithm

Look at Fig. 1 in order to see how HANNIBAL proceeds to verify the functional equivalence of two combinational circuits A and B. For reasons of simplicity, assume that the two circuits have only one output. Obviously, by combining the two circuits in the way shown, the logic verification problem is reduced to solving the Boolean satisfiability problem for the output signal E.

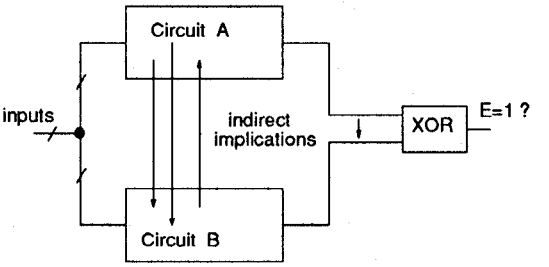

Fig. 1: Logic verification with indirect implications

In principle, the precise implication procedure of Fig. 1 represents a simple method to check the Boolean satisfiability of E: If the precise implications for E=1 produce a conflict, then it follows E=0 and the two circuits are equivalent. If no conflict occurs, the precise implication procedure determines all value assignments that are necessary to generate a distinguishing vector. However, what maximum depth of recursion is required in order to solve the problem? As pointed out in [8], the maximum depth of recursion required to identify all necessary assignments is closely related to the *size* of the redundancies in the circuit. In practical circuits, i.e. circuits that realize a certain function, the redundant structures are usually rather small, so that only few recursions are needed to perform precise implications. In the case of Fig. 1 however, things are different: we are dealing with an "artificial" circuit which does not serve any practical function. If circuits A and B are functionally equivalent, then the resulting circuit represents a large redundancy and it seems generally intractable to perform precise implications in this case.

Fortunately, as mentioned before, recursive learning can make use of a special aspect: In many cases, there are "similarities" between the two circuits under consideration. Logically, these similarities can be expressed as (usually indirect) implications between signals of different circuits. This is schematically shown in Fig. 1. Recursive learning is a very powerful technique to identify these implications, if they exist. Note that these implications immediately indicate the functional equivalence of internal nodes (as a special case).

Fig. 2 shows a flowchart of the algorithm. At first, HANNIBAL reads the description of the two circuits and combines their networks as shown in Fig. 1. The verification process essentially consists of two phases that can be called repeatedly. In phase 1, HANNIBAL passes through both circuits in order to identify and store indirect implications. At every gate, the algorithm assigns the logic signal value which makes the gate unjustified (e.g. '0' at AND). Then, *make_all_implications()* is called to perform the implications. If signal values are learned, i.e., if indirect implications exist, these implications are stored at the respective gate. This procedure is repeated for every gate in both circuits. There are

two aspects, which are very important for the efficiency of this preprocessing phase:

1) The gates G_i have to be picked in an appropriate order. Before some gate G_i is analyzed in the described way, it must be guaranteed that all gates in the cone of influence of G_i have been treated before. This ordering is necessary in order to make maximum use of prestored indirect implications.
2) In phase 1, *make_all_implications* is not only used to learn and store indirect implications for later reference in phase 2, but it is crucial that *make_all_implications()* itself makes use of previously stored indirect implications as given in Table 1.

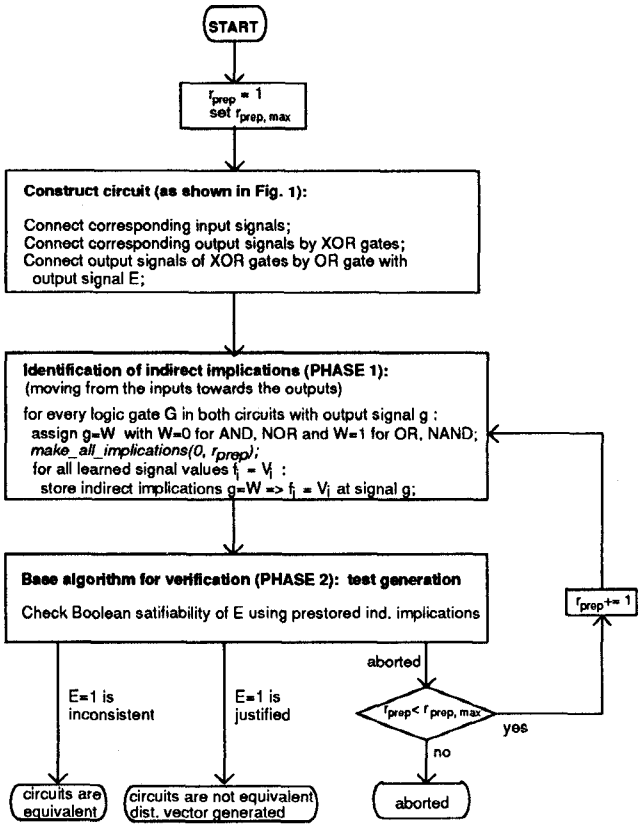

Fig. 2: Flowchart of HANNIBAL

After phase 1 has been finished, the actual verification algorithm is started in phase 2. In the current version of HANNIBAL, a test generator is invoked to justify E=1. This test generator as published in [1] is based on the FAN-algorithm, but uses *make_all_implications()* instead of a decision tree. Since this test generator performs implications with *make_all_implications()*, it can easily make use of all prestored indirect implications without the problem of false negatives. Test generation is aborted, if the maximum recursion depth for the test generation process has reached a user defined maximum. Note, that this is only one possibility to perform verification in phase 2. Importantly, since the prestored indirect implications immediately yield the internal equivalencies of the two circuits, any other base algorithm for logic verification, such as [11] or [12], can profit from recursive learning in phase 1. However, as pointed out before, depending on the verification method used, it is essential to develop efficient strategies to avoid false negatives. A method to avoid false negatives when using OBDDs has been outlined in [6].

If the verification problem is still too complex and the base algorithm for logic verification has to abort the problem, phase 1 is repeated with higher depth of recursion in order to identify more indirect implications. This is continued, until either a distinguishing vector is generated or the circuits are proved equivalent or the search is aborted because a user-defined maximum recursion depth for the preprocessing phase is exceeded. Note that both, phase 1 and phase 2, each represent a complete algorithm for logic verification which complement each other.

4. Example

In order to illustrate, how HANNIBAL performs logic verification, the example circuits of [6] are used as shown in Fig. 3. In phase 1, HANNIBAL derives indirect implications between the two circuits that have to be compared. In the above example the procedure is as follows:

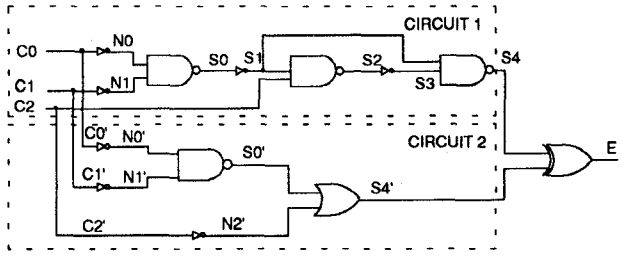

Fig. 3: Example circuits (Berman, Trevillyan 1989 [6])

CIRCUIT 1:
Assign the appropriate value at every output signal of a gate (with at least two input signals) and use *make_all_implications(0,1)* to determine indirect implications. For reasons of simplicity, internal signal values are only listed if they belong to a gate with more than one input:

S0=1: => S2=1 => S4=1
1. justif.: N0=0 => N0'=0
=> S0'=1
=> S4'=1
=> E=0
2. justif.: N1=0 => N1'=0
=> S0'=1
=> S4'=1
=> E=0
we learn: S0'=1, S4'=1, E=0

indirect implications: **S0=1 => S0'=1**
(stored at S0) **S0=1 => S4'=1, S0=1 => E=0**
S2=1: => S4=1
1. justif.: S1=0 => S0=1
=> S0'=1(by prestored ind. implication)
=> S4'=1 (by prestored ind. implication)
=> E=0
2. justif.: C2=0 => N2'=1
=> S4'=1, => E=0
we learn: S4'=1, E=0
indirect implications: **S2=1 => S4'=1**
(stored at S2) **S2=1 => E=0**
S4=1:
1. justif.: S1=0 => S0=1
=> S0'=1 (by prestored ind. implication)
=> S4'=1 (by prestored ind. implication)
=> E=0
2. justif.: S3=0 => S2=1
=> S4'=1 (by prestored ind. implication)
=> E=0 (by prestored ind. implication)
we learn: S4'=1, E=0
indirect implications: **S4=1 => S4'=1**
(stored at S4) **S4=1 => E=0**

CIRCUIT 2:
Similarly, we learn the indirect implications:
S0'=1 => S0=1, S0'=1 => S2=1
S0'=1 => S4=1, S0'=1 => E=1, and
S4'=1 => S2=1, S4'=1 => S4=1, S4'=1 => E=0

OUTPUT SIGNAL:
E=1: 1. justif.: S4=1 and S4'=0
=> E=0 (by prestored ind. implication), inconsistent
2. justif.: S4=0 and S4'=1
=> E=0 (by prestored ind. implication), inconsistent
=> E=0, circuits are equivalent

In this example, phase 1 with recursion depth 1 has been sufficient to prove that the circuits are equivalent. Illustrations of the implication procedure at higher recursion depths can be found in [1]. It is straightforward to derive the equivalent nodes from the prestored indirect implications.

5. Experimental results

A prototype version of HANNIBAL has been implemented in C. In order to examine its performance, logic verification was performed for the ISCAS 85 benchmarks[13]. For a practical evaluation of HANNIBAL's performance, the ISCAS 85 benchmarks were verified against their non-redundant versions, which were also obtained from MCNC. The non-redundant circuits resulted from the original circuits by redundancy elimination as described in [14].

Circuits to verify	max. rec. level		CPU - time	# equ. nodes
	Ph.1	Ph. 2	[min:s]	
c432, c432nr	1	0	00 : 03	110
c499, c499nr	1	0	00 : 06	112
c1355, c1355nr	1	0	00 : 19	328
c1355, c499	1	0	00 : 09	118
c1908, c1908nr*	1	0	00 : 26	424
c2670, c2670nr*	2	0	03 : 51	496
c3540, c3540nr*	2	5	34 : 17	778
c5315, c5315nr*	2	0	13 : 17	1267
c6288, c6288nr	1	0	00 : 48	2332
c7552, c7552nr*	2	0	78 : 44	1814

Table 2: Experimental results (Sparc Station ELC)

Table 2 shows the results for the verification experiments. The first column gives the circuits that have to be verified. The non-redundant circuits are indexed 'nr'. The second column gives the maximum depth of recursion for phase 1, denoted r_{prep} in Fig. 2. Column 3 depicts the maximum recursion depth that was necessary in phase 2, i.e. during the test generation phase with the test generator [1]. Column 4 shows the total CPU-time in minutes and seconds that was needed to solve the verification problem. The last column shows the number of functionally equivalent nodes identified.

Surprisingly, for some of the initial circuits obtained from MCNC, HANNIBAL proved that the non-redundant circuits were *not* equivalent to the original ISCAS 85 benchmarks. These benchmark circuits are marked by an asterisk. In fact, these results have been confirmed and the circuits were corrected. The shown results are for the corrected circuits which are all functionally equivalent. As far as the author knows, by now, the corrected circuits are available at MCNC. This "real life" experience shows that synthesis tools can indeed contain bugs and that efficient verification tools can be most helpful to debug them.

Furthermore, the results impressively show that HANNIBAL is able to solve logic verification problems in very short time, if there is some "similarity" between the circuits. The recursion depths for both, phase 1 to store indirect implications and phase 2 for test generation are very small. Consequently, the CPU-times are very short in most cases. It is very encouraging to note that a recursion depth of 1

or 2 in the preprocessing phase has been sufficient in almost all cases to complete the verification task by the preprocessing phase alone. In all circuits but c3540, the equivalence of the primary output signals was already established during preprocessing. This shows that this kind of preprocessing based on recursive learning is indeed very efficient to identify internal equivalencies and clearly superior to a structural comparison which must stop as soon as the first structural difference is encountered.

In general, if the designer uses some standard synthesis tool to derive a multi-level logic description of the circuit, we can expect that this logic description is correct because the standard tools are carefully debugged. However, design errors are more likely to be introduced at a later stage when the designer makes (possibly manual) further modifications to comply with his particular requirements (e.g. timing). The results show that HANNIBAL is very useful for the designer to detect errors that are introduced by such incremental changes.

It is very promising to combine the powerful concept of Binary Decision Diagrams with recursive learning by using a BDD approach like [11] or [15] as the base algorithm in Fig. 2. Depending on the approach, special strategies to avoid false negatives have to be developed, as e.g. outlined in [6]. Recursive learning and Binary Decision Diagrams can complement each other nicely: If the circuits are "similar", recursive learning is able to learn many indirect implications and the BDDs remain small. Only in the worst case, the BDDs have to be constructed to their full size.

6. Conclusion and Future Work

Motivated by the incremental nature of the design process, we have demonstrated the usefulness of recursive learning for logic verification. By recursive learning it is possible to identify internal logic equivalencies between two circuits. A new tool called HANNIBAL has been presented which makes use of this in order to verify the functional equivalence of two combinational circuits without suffering from the false negative problem. Results impressively show that HANNIBAL is a highly efficient tool for many practical verification problems. Future work will investigate, if this approach can also be used for sequential circuits. Furthermore, it is intended to incorporate BDD- techniques into HANNIBAL. For different approaches combining recursive learning and BDDs, it will be examined to what extent recursive learning can reduce the size of the BDDs and what are the best ways to avoid false negatives in such hybrid approaches.

Acknowledgments

The author wishes to thank Prof. Joachim Mucha for supporting this work and Jürgen Alt for his help with the fault simulator COMSIM [16], which was used to check the correctness of the generated distinguishing vectors.

References

[1] Kunz W., Pradhan D. K.: "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits", Proc. Intl. Test Conference, 1992, pp. 816-825.

[2] Ghosh A., Devadas S., Newton A. R.: "Sequential Logic Testing and Verification", Kluwer Academic Pub., 1992.

[3] Akers S.: "Binary Decision Diagrams", IEEE Transactions on Computers, vol. C-27, no. 6 June 1978, pp.509-516.

[4] Bryant R.: "Graph-Based Algorithms for Boolean Function Manipulation", IEEE Trans. on Computers, vol. C-35, no. 8, August 1986, pp. 677-691.

[5] Aas E. J., Klingsheim K., Steen T.: "Quantifying Design Quality: A Model and Design Experiments", Proceedings of EURO ASIC 1992, pp. 172-177.

[6] Berman C. L., Trevyllian L. H.: "Functional Comparison of Logic Designs for VLSI Circuits", Intl. Conf. on Comp.-Aided Design, 1989, pp. 456-459.

[7] Cerny E., Mauras C.: "Tautology Checking Using Cross-Controllability and Cross-Observability Relations", Intl. Conference on Computer-Aided Design, 1990, pp. 34-38.

[8] Kunz W., Pradhan D.K.: "Recursive Learning: A Precise Implication Procedure and its Application to Test Generation in Digital Circuits", accepted for publication in IEEE Trans. on Computer-Aided Design.

[9] Fujiwara H., Shimono T.: "On the Acceleration of Test Generation Algorithms", 13th Intl. Symp. on Fault Tolerant Comp., pp. 98-105, 1983.

[10] Schulz M., Trischler E., Sarfert T.: "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System", IEEE Trans. on CAD, vol. 7, Jan. 88, pp. 126-137.

[11] Malik S. et al.: "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment", Intl. Conference on Computer-Aided Design, 1988, pp. 6-9.

[12] Jain J., Bitner J., Fussell D. S., Abraham J. A.: "Probabilistic Verification of Boolean Functions", Journal of Formal Methods in Systems Design, vol.1,1992, pp.61-115.

[13] Brglez F., Fujiwara H.: "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in FORTRAN", Special Session on the 1985 IEEE International Symposium on Circuits and Systems.

[14] Tromp G. J., van de Goor A. J.: "Logic Synthesis of 100-percent Testable Logic Networks", IEEE Intl. Conference on Computer-Design, Sept. 1991.

[15] Jain J. et al.: "Indexed BDDs: Algorithmic Advances in techniques to represent and verify Boolean functions", U. of Texas at Austin, UT-CERC-TR-JAA-93-02

[16] Mahlstedt U., Alt J.: "Simulation of Non-Classical Faults on the Gate Level - The Fault Simulator COMSIM-" accepted for publication, Intl. Test Conference 1993.