2002

Formal Methods for Functional Verification

Randal E. Bryant
Carnegie Mellon University

James H. Kukula
Synopsis, Inc

Published In
The Best of ICCAD: 20 Years of Excellence in Computer-Aided Design, A. Kuehlmann, ed. Springer, 3-16.

FORMAL METHODS FOR FUNCTIONAL VERIFICATION

Randal E. Bryant
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

James H. Kukula
Advanced Technology Group
Synopsys, Inc.
Hillsboro, OR 97124

Abstract
Formal hardware verification ranges from proving that two combinational circuits compute the same functions to the much more ambitious task of proving that a sequential circuit obeys some abstract property expressed in temporal logic. In tracing the history of work in this area, we find a few efforts in the 1970s and 1980s, with a big increase in verification capabilities the late 1980s up through today. The advent of efficient Boolean inference methods, starting with Binary Decision Diagrams (BDDs) and more recently with efficient Boolean satisfiability (SAT) checkers has provided the enabling technology for these advances.

1. Introduction

Functional hardware verification involves determining whether or not a logic design matches a specification of its intended behavior. Most commonly, the design consists of a combinational or sequential logic gate circuit (possibly derived from an RTL description), and so the analysis can be performed purely at the Boolean level. Furthermore, the circuit is generally assumed to be either fully combinational or fully synchronous, and hence the functionality can be verified without any consideration of the circuit timing.

In many applications, the specification is also given as a logic circuit. This form of verification is referred to as *equivalence checking*. For example, a designer might want to verify that some optimizations to a netlist did not alter its functionality. Even when verifying that a gate-level netlist implements a specification given in a hardware description language such as Verilog or VHDL, the first step is typically to expand the HDL description into gate-level form and then use this as the specification.

Equivalence checking can be further categorized as *combinational* or *sequential*. With combinational equivalence checking, the two circuits are acyclic, gate-level circuits, and the task is to determine whether they compute the same

Boolean functions. Note that combinational equivalence can be used to prove the equivalence of two sequential circuits, as long as these circuits use the same encodings of their states. In fact, the commercial equivalence checkers now in widespread use follow this approach. With sequential equivalence, we are given two sequential circuits that could be using totally different state encodings. The task is to determine whether the two circuits would ever differ in their output values when they are started in their respective initial states and run with some arbitrary input sequence.

Historically, and even to this day, the most common approach to functional verification is to perform extensive simulation. For equivalence checking, this simply involves simulating the two circuits over many patterns and seeing whether they ever produce different values. In principle, combinational circuits could be fully verified by this means, if we were willing to run enough simulation (typically exponential in the number of primary inputs). For sequential equivalence checking, there is no practical bound on the amount of simulation required to prove the two circuits will have identical behavior for all possible input sequences. In this commentary, we will focus instead on *formal* verification, where mathematically based techniques are used to prove equivalence or other properties for all possible input sequences.

Going beyond equivalence checking, a more ambitious task is to prove that a circuit satisfies some general requirement, such as that there should never be a deadlock, or that any bus request will eventually be granted. The most widely studied class of tools for this form of verification are *model checkers*, where the program determines (checks) whether the circuit (model) obeys a property given as a formula in some type of temporal logic. Such formulas can express properties that involve the behavior of the system over time, cases where we use English words such as “always” and “eventually.”

2. Early Work in Verification

Formal hardware design verification appears to have developed in the 1970’s from earlier work in hardware testing and in software verification [23]. Testing-based approaches were applied to equivalence checking. Roth initially proposed [59] unrolling sequential logic to perform bounded sequential equivalence checking. Later [60] he introduced the assumption of a tight correspondence between the registers of the circuits to be compared, thus reducing the sequential problem to a combinational one. Kawato et al. [40] developed similar equivalence checking methods around the same time.

Early formal hardware verification methods based on software methods [29] included symbolic simulation [66, 18, 24], user-defined inductive invariants [56] and inductive invariants derived from assertions [50]. These early software-

based methods generally required user interaction to guide expression simplification.

Automated formal methods continued to be developed [32, 55, 20], but before efficient BDD algorithms were introduced most work was based on impoverished data representations (e.g., sum-of-products), or inefficient search routines (variants of the DPLL [31, 30] method used for Boolean satisfiability). The main problem with these approaches was that they did not exploit commonality of structure. Consequently, either the data representations would blow up, or the search routines would run too long.

A notable exception is work at IBM on equivalence checking. They developed programs for internal use that could handle very large combinational circuits.

2.1 Early Equivalence Checking at IBM

An algorithmic breakthrough was achieved at IBM with the Differential Boolean Analyzer using Equivalent Sets of Partials (DBA/ESP) [64]. This algorithm provided the satisfiability engine for the equivalance checking tool which was widely used at IBM in the late 1970's through the 1980's. DBA/ESP was inspired by Shannon decomposition and also the method of bifurcations given in Hammer and Rudeanu's book [37], the practical potential of which was recognized by Al Brown. DBA is also closely related to the Binary Decision Diagrams introduced by Akers [2].

DBA proceeds by successive elimination of variables using Shannon decomposition. The key advances that made DBA practical were effective variable ordering heuristics and the ESP feature suggested by Gordon Smith, which detected shared subproblems so that redundant analysis could be avoided. Variable ordering was inspired by the "longest equation" and "most frequent unknown" heuristics of Hammer and Rudeanu. The discovery of shared subproblems was made feasible by a representation that allowed hashing and efficient structural isomorphism checking. Shannon decomposition by itself would generate a binary tree, but common subproblem recognition transforms the tree into a directed acyclic graph, in particular a BDD. Since common subproblem recognition is done before the subproblem is analyzed and based on structural isomorphism rather than functional equivalence, the resulting BDD is not fully reduced. However, satisfiability of the initial problem is immediately determined once the diagram construction is complete or a 1 terminal is reached.

The IBM equivalence checker also provided means to indicate correspondence between internal combinational signals, reducing the complexity of the problems that the DBA/ESP algorithm needed to solve. The possibility of false negatives due to cutting at internal equivalence points was noted, and the use of a manually specified don't care signal to circumvent the problem is described.

Don't care signals could also be used to avoid false negatives due to unreachable states, with support for validating the unreachable state assertion during simulation.

2.2 Binary Decision Diagrams

The idea of encoding a Boolean function as a graph of decisions was first proposed by Akers [2], based on an earlier encoding as a straight-line program by Lee[44]. Akers coined the term "Binary Decision Diagram" (BDD) and also explored some of their properties, but he did not provide an efficient method for building BDDs from circuits. Akers' default strategy would be exhaustive simulation to build a complete binary tree, followed by reduction operations to exploit subtree sharing.

In late 1983, Bryant was inspired by the way concurrent fault simulators evaluate the gates in a circuit for both the good and many faulty behaviors in a single pass through the circuit, using lists to encode the multiple different values at each gate. He thought of replacing the list representation with a tree to encode all possible input combinations and then realized the subtrees could be merged to form a directed acyclic graph. This led him to formulate the Reduced Ordered Binary Decision Diagram (ROBDD, but often simply referred to as "BDD") representation and to devise algorithms for performing operations on Boolean functions based on graph algorithms operating on BDDs. He first published these ideas in 1985 [12], with a more complete description in the well known 1986 paper [13] (submitted for publication in 1984).

The success of BDDs stems from an interrelated set of issues:

- The BDD data structure is based on a maximal sharing of substructure. They are not as prone to exponential blow up as are other representations.

- Boolean operations can be performed using simple graph algorithms. The complexity of these operations are polynomial in the graph sizes. They gain efficiency by exploiting the sharing within BDDs.

- They provide a single, homogeneous representation of the problem space. For example, with symbolic model checking BDDs are used to represent the system being modeled, and the sets of possible states of the system. By contrast, many EDA programs shift back and forth between many different data structures.

- By providing a general purpose Boolean manipulation engine, BDDs help application developers think in more abstract terms. Looking at earlier work, we can often see where the application developer muddles concerns about the problem with how the problem is represented.

2.3 The Effectiveness of BDDs for Design Automation

Although not among the selected papers for this volume, ICCAD papers by Malik et al. [48] and Fujita et al. [35] put BDDs on the map to the larger CAD community. They showed 1) that fairly elementary heuristics could select reasonably good variable orders for combinational circuits, and 2) that BDDs could then be constructed for large benchmark circuits enabling tasks to be performed (e.g., equivalence checking) that far exceeded what had been done before.

2.4 Dynamic Variable Ordering

Rudell's work [62] strongly reinforces the advantage of having a separate Boolean manipulation engine. Previously, others had shown that this engine could handle housecleaning tasks such as automatic garbage collection [52] and cache management [10]. Rudell showed that it could also handle the task of continuously improving the ordering to minimize storage requirements. While a program is running, the BDDs pointed to by the application keep changing in structure (without changing the underlying function being represented.) But, the user need not be concerned with this.

The work is also a masterpiece of careful engineering. Rudell recognized that the BDD transformations could be made without having the update any of the pointers from external sources into the BDD data structure. This allowed many BDD-based applications to be "retrofitted" to use dynamic variable ordering with only minor changes. In practice, dynamic variable ordering often greatly increases the runtime of BDD-based applications, but it can also enable successful completion in cases that would otherwise fail due to excessive memory requirments.

2.5 Combinational Equivalence Checking with BDDs

Researchers at Bull Research, headed by François Anceau, were "early adopters" of BDDs. They showed that BDDs could be used to perform equivalence checking of combinational circuits in an industrial setting in 1987, published in a paper [46] at DAC in 1988. (The earlier work at IBM was not widely known, and did not use conventional BDD algorithms.) Their ICCAD paper [47] showed that once you have good equivalence checking and a powerful Boolean engine, you could go beyond a basic Yes/No equivalence check and attempt to determine why the two circuits are not equivalent. It is based on looking for small variants of the circuit (e.g., inserting one inverter or changing one Nand to Nor) to see if the circuits could be made equivalent. They encode these variants symbolically, so that one run of the engine can test all candidate variations. Although this

particular application of BDDs has not had widespread use, the paper illustrates a general principle of using BDDs. By using symbolic encoding, one can often replace a long series of tests with a single symbolic evaluation.

Further developments [41, 38] have continued in diagnostic methods for combinational equivalence checking.

2.6 Sequential Verification

Sequential equivalence checking requires proving that two state machines have identical behavior. A common approach is to cast this as a reachability problem. First, a composite circuit is constructed consisting of the two original circuits, plus logic that indicates whether the two circuits have different output values. The task then becomes to determine whether, starting in a state where the two original circuits are in their initial states, the composite circuit can ever reach a state where the comparator circuit indicates that the two circuits have generated different outputs. If so, the circuits are inequivalent. This is equivalent to performing model checking on the composite circuit with the temporal logic query EF *t* ("is it ever possible for *t* to hold?"), where *t* is the output of the comparator circuit. Thus, we can see that sequential equivalence checking is a special case of model checking.

Model checking was first developed by Clarke and Emerson [21, 22] as a way to automatically verify properties of synchronization programs. They also coined the term "model checking." The first implementations of model checkers used an *explicit state* representation, encoding each state as a node in a graph. For most hardware designs, the number of states is far too large (exponential in the number of state variables) to be represented in such a fashion, and hence model checkers were originally only applied to very small circuits.

The major breakthrough for the application of model checking to hardware design came with the advent of *symbolic model checking*, where both the circuit model and the set of reachables states are encoded implicitly, typically with BDDs.

The history of symbolic model checking and symbolic FSM equivalence is more difficult to trace, with a number of researchers coming up with similar ideas independently. It is generally acknowledged that Ken McMillan originated the idea of BDD-based symbolic model checking in 1987 and implemented one. But, he did not publish any papers about this work at the time. In 1989, Coudert, Berthet, and Madre presented two papers [26, 25] on using symbolic state machine traversal to verify the equivalence of two finite state machines. Bose and Fisher presented a paper [9] describing a symbolic model checker and its implementation. Bahnsen and Kukula [3] also sketched some ideas for BDD-based state traversal, but they did not have any implementation.

A number of advances in model checking were reported in 1990. Burch, Clarke, McMillan, and Dill presented their seminal papers [15, 14]. Their work was the most general of all, showing that they could symbolically evaluate formulas in a mu-calculus logic that can express many other forms of logic, including several different temporal logics. Coudert, Madre, and Berthet presented a paper [28] on a symbolic, computation-tree-logic (CTL) model checker. In the same conference, Pixley [57] presented a sequential equivalence checker, which followed a different approach than the reachability approach sketched above. Pixley's method involved determining which pairs of states are equivalent to each other, eliminating the need to specify initial states. He also presented detailed algorithms for symbolic model checking, although his only experimental results were for equivalence checking.

The included paper by Coudert and Madre [27] was the first appearance in ICCAD of a paper on symbolic model checking. The subject of the paper was some refinements on how to perform the preimage computation more efficiently than had done before. Coudert and Madre's model checker was based on a *function vector* approach, where the set of reachable states is encoded as the range of a set of Boolean functions. This approach has not proved as popular as the *characteristic function* approach, where the set of reachable states is encoded as a single Boolean function indicating whether or not a given state is reachable. The Coudert and Madre paper also introduced the operations *constrain* and *restrict*, which were later explored for other applications [45, 63, 1].

Since this early research on symbolic model checking, there has been a continuous stream of research on ways to improve efficiency. The most successful techniques exploit the modularity of circuits by representing and applying the transition relation in a partitioned form [16, 53].

3. SAT and ATPG Methods

Binary Decision Diagrams continue to serve as the foundation for many CAD algorithms. They provide a canonical representation, which is compact across a wide range of practical Boolean functions and efficiently supports a wide range of operations such as intersection, inversion, and quantification. In some situations, however, a less powerful approach can be more efficient. For example, some applications require finding only a subset of the solutions of a Boolean equation. Significant advances and fresh applications in dynamic search approaches such as SAT and automatic test pattern generation (ATPG) algorithms have paralleled those in BDDs.

3.1 Efficient Search Space Pruning

GRASP [49] introduced a new generation of SAT solvers that were designed and tuned using EDA benchmarks. GRASP and its successors[54, 36] are based

on the same DPLL search procedure that has been known for decades, but they are much more efficient at pruning the search space to avoid fruitless search. In the case of GRASP, the main contribution was "conflict diagnosis," where the solver analyzes the conditions leading to a dead end in the search and infers from this a general condition that will make the formula unsatisfiable. This analysis enables *nonchronological backtracking*, where the search engine backtracks through multiple levels of decisions. It also enables *clause learning*, where the SAT solver can add information about a failed search (in the form of a new clause) to its database and thereby avoid repeating a fruitless search. SAT solvers have now supplanted BDDs for many EDA applications where simple Boolean operations are required.

SAT checking has recently flourished as a research area. Each successive generation of SAT checker typically outperforms its predecessors by an order of magnitude, both in terms of the speed on existing benchmarks and the ability to handle larger problems. Much of the recent focus has been on efficiently organizing and maintaining the internal data structures, particularly the set of clauses. The CHAFF solver [54] demonstrated the value of using clever structures to minimize the number of clauses that need to be checked during constraint propagation. Other ideas seen in modern SAT checkers include: *restarts*, where the solver abandons the current search tree and starts a new one, as well as refined techniques for deciding which elements to discard from the clause set, which variable to split on next, and which value (0 or 1) to try first for a splitting variable [36].

One important application of SAT checkers has been to a limited form of model checking, known as a *bounded model checker*[6]. Bounded model checking involves running the circuit for a fixed number of steps from its initial state and determining whether it satisfies the specified temporal properties. This does not guarantee that the properties would then hold indefinitely, but it serves as a useful check, often quickly finding counterexample traces when they exist. The main advantage of bounded model checking is that it can be applied to very large circuits [7, 8].

3.2 Exploiting Similarity

Often two combinational circuits being compared for equivalence will have many functionally equivalent internal signals. This observation was already exploited quite early [33, 4] in the evolution of formal equivalence checkers and has since evolved into the most successful approach to improving performance and capacity.

The equivalence checker developed by Berman and Trevillyan [5] detects equivalent points within the two circuits and then partitions the circuits, treating these equivalence points as primary inputs and outputs of the subcircuits. The

key idea in this paper is the use of a weighted graph min-cut algorithm to minimize the number of subcircuit primary inputs, thus improving efficiency of the equivalence check. Treating internal equivalences as primary inputs introduces the possibility of false negatives. This paper outlines a method for eliminating false negatives by incrementally shifting cutpoints back toward the original primary inputs, usings BDDs and the compose operation.

While [5] relies primarily on exhaustive simulation, Kunz [43] used ATPG methods to detect and exploit internal equivalences. This more powerful method can be used without introducing internal cutpoints, thus avoiding the false negative problem. Advances in this area continue to be made by many researchers, e.g., [58, 39, 51, 42, 17]. Sequential methods that exploit internal equivalences have also been developed [65, 34].

3.3 Leveraging Observability

To further improve the efficiency of combinational equivalence checking, one can extend the notion of internal equivalence to include signals that are different functions of the primary inputs but whose difference is unobservable at primary outputs or state registers. Cerny and Mauras [19] introduced such a method that relied on BDD characteristic functions across full cuts of a circuit.

Brand [11] provided improved efficiency with an ATPG-based combinational equivalence checker. This paper introduced the concept of the joining two candidate equivalent signals with a "Miter" circuit, which indicates whether a difference in the signal values is observable at the primary outputs. The complexity of the equivalence check is reduced by repeatedly transforming this joined circuit based on detected equivalences. Broadening the class of equivalences detected, from signals which are identical functions of the primary inputs to signals whose functional difference is undetectable at the primary outputs, allows further simplifications of the problem so that ATPG can be used on larger problems without introducing internal cutpoints with their risk of false negatives.

4. Conclusions

The progress in formal verification from the late 1980s through today has been remarkable. The research community has developed and refined both the underlying symbolic manipulation engines as well as the verification tools that use these engines to reason about complex combinational and sequential circuits. In addition, a number of different equivalence and property checkers have become available commercially. Formal combinational equivalence checking tools have become robust enough to be incorporated routinely into industrial methodologies. Still, the needs of the electronics industry in terms of capacity, performance, and functionality far exceed the capabilities of current sequential verification tools, and even combinational checking occasionally fails due to in-

adequate capacity. We can anticipate that this part of the EDA community will continue to flourish in its ideas and importance.

Acknowledgments

The authors wish to thank Al Brown and Gordon Smith for their help in outlining the early history of formal equivalence checking at IBM.

References

[1] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Formal verification using parametric representations of Boolean constraints. In *Proc. Design Automation Conference*, pages 402–407, 1999.

[2] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27:509–516, 1978.

[3] R. J. Bahnsen and J. H. Kukula. Technique for verifying finite state machines. *IBM Technical Disclosure Bulletin*, 32(3A):166–169, 1989.

[4] L. Berman. On logic comparison. In *Proc. Design Automation Conference*, pages 854–861, 1981.

[5] L. Berman and L. Trevillyan. Functional comparison of logic designs for VLSI circuits. In *Proc. of the International Conference on Computer Aided Design*, pages 456–459, 1989.

[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 1579, pages 193–207. Springer, 1999.

[7] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a Power PC microprocessor using symbolic model checking without BDDs. In *Computer-Aided Verification*, Lecture Notes in Computer Science 1633. Springer-Verlag, pages 60–71, 1999.

[8] P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an Alpha microprocessor using satisfiability solvers. In *Computer-Aided Verification*, Lecture Notes in Computer Science 2102. Springer-Verlag, pages 454–464, 2001.

[9] S. Bose and A. L. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In *Proc. IFIP Int. Workshop on Applied Formal Methods for Correct VLSI Design*, pages 759–764, 1989.

[10] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. Design Automation Conference*, pages 40–45, 1990.

[11] D. Brand. Verification of large synthesized designs. In *Proc. of the International Conference on Computer Aided Design*, pages 534–537, 1993.

[12] R. E. Bryant. Symbolic manipulation of Boolean functions using a graphical representation. In *22nd Design Automation Conference*, pages 688–694, June 1985.

[13] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[14] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. Design Automation Conference*, pages 46–51, 1990.

[15] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. Symposium on Logic in Computer Science*, pages 1–33, 1990.

[16] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proc. Design Automation Conference*, pages 403–407, 1991.

[17] J. R. Burch and V. Singhal. Tight integration of combinational verification methods. In *Proc. of the International Conference on Computer Aided Design*, pages 570–576, 1998.

[18] W. Carter, W. Joyner, Jr., and D. Brand. Symbolic simulation for correct machine design. In *Proc. Design Automation Conference*, pages 280–286, 1979.

[19] E. Cerny, and C. Mauras. Tautology checking using cross-controllability and cross-observability relations. In *Proc. of the International Conference on Computer Aided Design*, pages 34–37, 1990.

[20] M. S. Chandrasekhar, J. P. Privitera, and K. W. Conradt. Application of term rewriting techniques to hardware design verification. In *Proc. Design Automation Conference*, pages 277–282, 1987.

[21] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs Workshop*, Lecture Notes in Computer Science 131. Springer-Verlag, 1981.

[22] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *10th Annual ACM Symposium on Principles of Programming Languages*, 1983.

[23] W. E. Cory and W. M. vanCleemput. Developments in verification of design correctness: a tutorial. In *Proc. Design Automation Conference*, pages 156–164, 1980.

[24] W. E. Cory. Symbolic simulation for functional verification with ADLIB and SDL. In *Proc. Design Automation Conference*, pages 82–89, 1981.

[25] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using Boolean functional vectors. In *Proc. IFIP Int. Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, 1989.

[26] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems: International Workshop Proceedings*, in Lecture Notes in Computer Science 407, pages 365–373. Springer, 1989.

[27] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proc. of the International Conference on Computer Aided Design*, pages 126–129, 1990.

[28] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In *Computer Aided Verification*, pages 75–84, 1990.

[29] J. Darringer. The application of program verification techniques to hardware verification. In *Proc. Design Automation Conference*, pages 375–381, 1979.

[30] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[31] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[32] S. Devadas, H. K. T. Ma, and A. L. Sangiovanni-Vincentelli. On the verification of sequential machines at differing levels of abstraction. In *Proc. Design Automation Conference*, pages 271–276, 1987.

[33] W. E. Donath and H. Ofek. Automatic identification of equivalence points for Boolean logic verification. *IBM Technical Disclosure Bulletin*, 18:2700–2703, 1976.

[34] C. A. J. van Eijk. Sequential equivalence checking without state space traversal. In *Proc. Design Automation and Test in Europe*, pages 618–623, 1998.

[35] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of Boolean comparison method based on binary decision diagrams. In *Proc. of the International Conference on Computer Aided Design*, pages 2–5, 1988.

[36] E. Goldberg, Y. Novikov. *BerkMin: A Fast and Robust SAT Solver* In *Design Automation and Test in Europe*, pages 142–149, 2002.

[37] P. L. Hammer and S. Rudeanu. *Boolean Methods in Operations Research and Related Areas*. Springer Verlag, Berlin, New York, 1968.

[38] S.-Y. Huang, K.-C. Chen, and K.-T. Cheng. Error correction based on verification techniques. In *Proc. Design Automation Conference*, pages 258-261, 1996.

[39] J. Jain, R. Mukherjee, and M. Fujita. Advanced verification techniques based on learning. In *Proc. Design Automation Conference*, pages 420–426, 1995.

[40] N. Kawato, T. Saito, F. Maruyama, and T. Uehara. Design and verification of large-scale computers using DDL. In *Proc. Design Automation Conference*, pages 360–366, 1979.

[41] A. Kuehlmann, D. Cheng, A. Srinivasan, and D. LaPotin. Error diagnosis for transistor-level verification. In *Proc. Design Automation Conference*, pages 218–224, 1994.

[42] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Proc. Design Automation Conference*, pages 263–268, 1997.

[43] W. Kunz. HANNIBAL: an efficient tools for logic verification based on recursive learning. In *Proc. of the International Conference on Computer Aided Design*, pages 538–543, 1993.

[44] C. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, July 1959.

[45] B. Lin, H. J. Touati, and A. R. Newton. Don't care minimization of multi-level sequential logic networks. In *Proc. of the International Conference on Computer Aided Design*, pages 414–417, 1990.

[46] J. C. Madre and J. P. Billon. Proving circuit correctness using formal comparison between expected and extracted behavior. In *Proc. Design Automation Conference*, pages 205–210, 1988.

[47] J. C. Madre, O. Coudert, and J. P. Billon. Automating the diagnosis and the rectification of design errors with PRIAM. In *Proc. of the International Conference on Computer Aided Design*, pages 30–33, 1989.

[48] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proc. of the International Conference on Computer Aided Design*, pages 6–9, 1988.

[49] J. P. Marques Silva and K. A. Sakallah. GRASP—a new search algorithm for satisfiability. In *Proc. of the International Conference on Computer Aided Design*, pages 220–227, 1996.

[50] F. Maruyama, T. Uehara, N. Kawato, and T. Saito. A verification technique for hardware designs. In *Proc. Design Automation Conference*, pages 832–840, 1982.

[51] Y. Matsunaga. An efficient equivalence checker for combinational gates. In *Proc. Design Automation Conference*, pages 629–634, 1996.

[52] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *Proc. Design Automation Conference*, pages 52–57, 1990.

[53] I.-H. Moon, J. Kukula, K. Ravi, and F. Somenzi. To split or conjoin: the question in image computation. In *Proc. Design Automation Conference*, pages 23–28, 2000.

[54] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proc. Design Automation Conference*, pages 530–535, 2001.

[55] G. Odawara, M. Tomita, O. Okuzawa, T. Ohta, and Z. Zhuang. A logic verifier based on Boolean comparison. In *Proc. Design Automation Conference*, pages 208–214, 1986.

[56] V. Pitchumani and E. Stabler. A formal method for computer design verification. In *Proc. Design Automation Conference*, pages 809–814, 1982.

[57] C. Pixley. A computational theory and implementation of sequential hardware equivalence. In *Computer Aided Verification*, pages 293–320, 1990.

[58] S. Reddy, W. Kunz, and D. Pradhan. Novel verification framework combining structural and OBDD methods in a synthesis environment. In *Proc. Design Automation Conference*, pages 414–419, 1995.

[59] J. P. Roth. Verify: an algorithm to verify a computer design. *IBM Technical Disclosure Bulletin*, 15:2646–2648, 1973.

[60] J. P. Roth. Hardware verification. *IEEE Transactions on Computers*, C-26:1292–1294, 1977.

[61] J. P. Roth. *Computer Logic, Testing, and Verification*. Computer Science Press, Potomac, 1980.

[62] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. of the International Conference on Computer Aided Design*, pages 42–47, 1993.

[63] T. R. Shiple, R. Hojati, A. L. Sangiovanni-Vincentelli, and R. K. Brayton. Heuristic minimization of BDDs using don't cares. In *Proc. Design Automation Conference*, pages 225–231, 1994.

[64] G. Smith, R. Bahnsen, and H. Halliwell. Boolean comparison of hardware and flowcharts. *IBM Journal of Research and Development*, 26:106–116, 1982.

[65] D. Stoffel and W. Kunz. Record and play: a structural fixed point iteration for sequential circuit verification. In *Proc. of the International Conference on Computer Aided Design*, pages 394–399, 1997.

[66] T. Wagner. Verification of hardware designs thru symbolic manipulation. In *Proc. International Symposium on Design Automation and Microprocessors*, pages 50–53, 1977.