The Use of Random Simulation in Formal Verification

Florian Krohm Andreas Kuehlmann Arjen Mets

IBM Thomas J. Watson Research Center
Yorktown Heights, N.Y., U.S.A.

Abstract

In this paper we present the application of random simulation in formal verification of functional equivalence of hardware designs. We demonstrate that random simulation can effectively complement BDD-based verification approaches in three areas: (1) quick generation of counter example pattern for miscomparing designs, (2) exhaustive comparison of small functions, and (3) providing meaningful signatures for design partitioning based on functionally equivalent cut-points. The presentation describes a smooth and efficient integration of a simulation algorithm into a general verification framework. In this framework the simulator can be applied as one of various engines for Boolean reasoning the outcome of which might be undecided.

1 Introduction

In recent years, formal hardware verification based on symbolic methods has gained significant acceptance in industrial design methodologies. In particular, Binary Decision Diagrams (BDD) [1] and their modifications are quite successful for proving functional equivalence of different hardware implementations for a large class of practically relevant designs. Intuitively, the multiplexor like structure of BDDs proved to be an efficient representation for many Boolean functions that hardware designers are interested in.

The major drawback of BDDs is their unpredictable memory consumption. Especially, if the two designs to be compared are not equivalent, the generation of a counter example by calculating the XOR between the two inequivalent functions often fails due to memory explosion. However, from a practical point of view, the computation of the complete set of counter example pattern using BDDs is exaggerated and not necessary. In most cases, a single input pattern that exercises the unwanted behavior is sufficient for efficient debugging of the faulty circuit.

A second problem of BDDs occurs for a specific class of equivalence checks which are common in hierarchical verification approaches. In order to successfully verify large systems, the two design representations to be compared are partitioned along the identical top part of their hierarchical description [2]. Verification is then performed in a bottom up manner, where pre-verified subcomponents are excluded at higher levels and replaced with black-boxes. This back-boxing scheme may result in millions of additional verification inputs and outputs, whereas the actual logic between the black-boxes is often simple and includes only straight interconnections or few logic gates. For those verification problems BDDs are inappropriate due to their relatively large overhead for maintaining the additional variables and comparing the simple functions.

The canonical function representation of BDDs is advantageous for comparing designs with completely different structures. As empirically shown for practical designs, the effective construction of BDDs hardly depends on the style or structure of the original representation. However, if the BDD construction fails for one side due to excessive memory usage, this brute force method cannot succeed, even if the two representations are structurally identical. For those cases, a general approach is desirable which combines: (1) the strength of BDDs to build a canonical representation for large parts independent of their original structure and (2) a general scheme that identifies structural similarities to partition large functions into smaller pieces that can be handled by BDDs. Internal nets which implement the same function on both sides can be used as cut-points for an identical design partitioning [3, 4].

In general, random simulation can be applied to effectively address the three problems mentioned above:

- For large sets of counter examples, simulating a few thousand random pattern is likely to distinguish the two miscomparing functions. Contrarily, if the set of counter example pattern is small, a BDD approach based on the XOR between the two functions will typically succeed.
- If the actual logic to be compared is small, exhaustive simulation can be used to prove their functional equivalence. As shown in this paper, if the number of functional comparisons is huge, an efficient simulator implementation can outperform a BDD-based method by more than a magnitude.
- Random simulation provides an inexpensive and efficient method to calculate signatures for all internal nets. These signatures can be used to classify all nets and to identify possible cut-points.

In this paper we present the integration of a random simulator into the verification tool Verity [2]. In contrast to other approaches, our implementation is emphasized on a smooth integration into a general verification framework, in which the simulator is applied as one of many Boolean reasoning engines. For this purpose, a unique set of basic operations was defined

for all engines which is similar to common operations supported in BDD packages. In the next section we will present the general verification approach used in Verity and discuss the set of basic operations that must be supported by the simulation algorithm. Section 4 describes the detailed implementation of the simulator. Section 5 and section 6 present results and conclusions, respectively.

2 Verification Approach in Verity

In this section we briefly describe those concepts of the verification tool Verity that are significant for the integration of the random simulator. The discussion will mainly focus on the operations that are needed for the functional extraction step. Further details can be found in [2].

Verity was designed for verifying the functional equivalence of two hardware designs modeled on register-transfer-level, gate-level, or switch-level. It employes a general functional extraction algorithm that works on a mixed design representation using any structural combination of switches and Boolean expressions. In the simple case of a static circuit technique, Verity computes for each functional net two Boolean functions F_0 and F_1, representing a driving condition to *Ground* and *Vdd*, respectively. Table 1 describes the meaning of the underlying four-value logic encoded by these two functions. In case of dynamic circuits and ratioed logic the extraction scheme is extended by using mutiple phases. These phases reflect the dynamic clocking scheme and the strength levels of switches.

F_0	F_1	Status of Functional Net
0	1	logical 1
1	0	logical 0
0	0	high impediance (floating)
1	1	collision

Table 1: Encoding of the four-value logic by F_0 and F_1.

The functions F_0 and F_1 are built in topological order from primary inputs to primary outputs. The main operations needed for this step are functions to create input variables and to perform basic Boolean operations (e.g. AND, OR, NOT). For the comparison of two outputs, the corresponding results for F_0 and F_1 need to be tested for equivalence. Depending on the circuit technique, the design methodology might prohibit certain conditions at internal nets. For example, a conservative static circuit technique might enforce well defined values at all nets, essentially disallowing $(F_0, F_1) \in \{(0,0),(1,1)\}$. The absence of such situations can be tested by a set of consistency checks expressed as Boolean formulas that need to be satisfied. In typical applications, the number of consistency checks might account for more than 90% of all verification problems.

A second important aspect of the functional extraction scheme is the handling of structural loops. In CMOS designs, structural loops are used to implement storage elements such as registers or latches. In Verity, these sequential elements are treated as state registers. In order to use a combinational verification approach, the storage loops are broken and corresponding register bits of the two designs are identified.

In addition to storage elements, structural loops are used in high speed CMOS designs to implement fast combinational circuits. Since there is typically no counter part for such loops, they must be treated specifically. Verity uses a loop extraction scheme to classify the loop type and to determine the correct function for all fan-out nets of the loop [2]. The loop extractor uses three specific operations:

First, once a strutural loop is detected, it is broken and a specific *loop variable* is introduced. As we will see later, it is important to distinguish between regular input variables and loop variables. Second, the depend function calculates the condition F_{depend} which is 1 if function F depends on a particular input variable v ($F_{depend} = F|_{v=0} \oplus F|_{v=1}$). This operation is used to determine the loop type. Third, the compose operation ($F_{comp} = F|_{v=G}$) replaces a particular input variable v in function F with some other function G. The compose operation is used to resubstitute the loop variables by the loop function. This effectively corrects the function for all fan-outs of the loop. It is important to notice that both, the depend and compose operations, are used for loop variables only. This has specific implications on their implementation in the simulation algorithm.

3 Verification Framework

The verification of functional equivalence of two design representations includes the comparison of all output functions and the completion of the consistency checks for all internal nets. In Verity, the list of these individual problems is maintained by a verification framework. This framework keeps track of the problem status and invokes the functional extraction algorithm with different Boolean reasoning engines to solve them. If a particular engine cannot solve a problem, it gets automatically passed to the next engine. For example, if the random simulator cannot prove functional equivalence through exhaustive simulation, the problem remains open and gets forwarded to the BDD package.

The functional extractor communicates through a well defined interface with the different Boolean reasoning engines. As mentioned above, the main components of this interface are functions for the basic Boolean operations, functional comparison, the depend operation, and the compose operation. The general application of different engines imposes two specific requirements on the functional extractor: First, in order to maintain the memory efficiently, it is crucial to control the lifetime of Boolean functions during the extraction process. This is done by reference counting, similar to the scheme used in BDD packages [5]. For BDDs the effective saving through releasing dead functions is often negligible since the corresponding BDD nodes might be reused in other functions. Contrarily, for a simulator the potential saving is significantly

higher since each unique function is represented by a pattern entry.

The second requirement is the ability to digest estimated return values from the comparison of Boolean functions. Depending on the underlying algorithm, a reasoning engine might return an exact or estimated value. Table 2 summarizes the possible comparison results for the BDD package and the simulator.

Result of Comparison	Interpretation by Extractor	Reasoning Engine
equal-exact	Proven to be equal	BDDs, Exhaustive Simulation
not-equal-exact	Proven to be not equal	BDDs Random Simulation
equal-estimate	Not proven to be not equal	Random Simulation

Table 2: Possible results of functional comparison for BDDs and simulation.

The exactness of the result depends not only on the reasoning package but also on the exactness of the functions being manipulated. A function might become marked as estimated if during its construction an estimated comparison result was used. For example, loop extraction based on non-exhaustive simulation might determine that a given structural loop is combinational. Since, any one of the missing pattern could cause the loop to be sequential, the loop classification is not exact and all outgoing functions must be marked as estimated.

4 Implementation Details

In this section we will discuss implementation details of the simulation engine. We will first focus on the general simulation mechanism and then explain the more advanced functions such as the compose operation. In the following we will use the term *node* to refer to a logic gate.

Our approach combines circuit construction and simulation as follows. Whenever a Boolean operation is invoked by the functional extractor, we first apply two optimizations, namely constant folding and inverter elimination. To eliminate inverters we either apply deMorgan's Law or the negation is folded into a node's Boolean function. For our functional extraction scheme inverter elimination typically reduces the circuit size by 20%. At the same time the reference counters of the node's immediate predecessors are decremented while the node itself is assigned a reference count of 1. If a node's reference count is 0, that node and its associated pattern memory is released and may be reused.

One can visualize the extraction process as moving a wavefront of active nodes from the primary inputs towards the primary outputs. As will be shown in section 5 the number of nodes at this frontier is usually much smaller than the total number of nodes. However, extreme care is needed when manipulating reference counters. As an example assume that node I in figure 1 is an active node with reference count 1. It

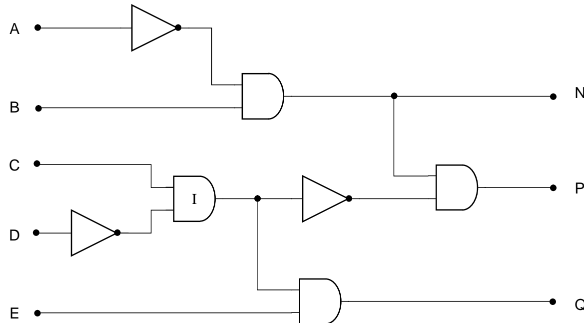


Figure 1: Sample circuit

is essential, that the functional extractor increments the reference counter for nets with multiple fanouts. Otherwise undefined results will occur.

4.1 Compose Operation

The compose operation is versatile and can be used for multiple purposes, for example cofactor computation and—in combination with simple Boolean operations—existential and universal quantification as well as the depend operation. As mentioned earlier the compose operation replaces every occurence of variable v in function F with some function G. With a simulator as reasoning engine, the compose operation can be implemented as follows:

1) Simulate function G.

2) Replace the original pattern of variable v with the pattern obtained in step 1. This includes saving v's original pattern for later retrieval.

3) Simulate F which yields the result of the compose operation.

4) Restore v's original pattern.

The problem with this approach is, that simulating a function requires the presence of the circuit structure implementing this function. However, this conflicts with what has been said in the previous section, namely, that the circuit structure is removed as soon as possible in order to save memory. Fortunately, it is known upfront that only loop variables are subject to the compose operation. All paths originating from a loop variable and all nodes feeding a gate on a compose-path are kept during forward composition. An example is shown in figure 2, where input C is a loop variable. Only the bold printed gates and paths are kept. Since all nodes of a compose-path are always simulated, the performance could potentially suffer. However, in practice the number of gates forming a structural loop is usually small, which results in a small number of compose-paths. Furthermore, since every loop variable is composed exactly once, each compose-path is resimulated only once, and the circuit structure can be removed. The resulting performance loss is negligible.

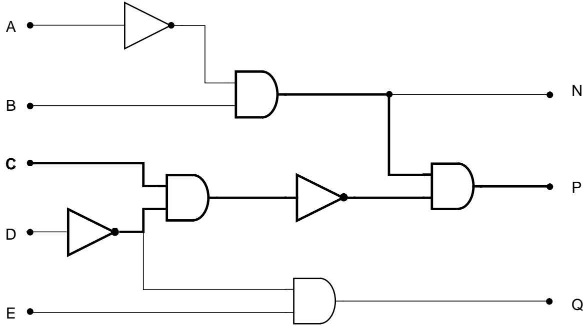
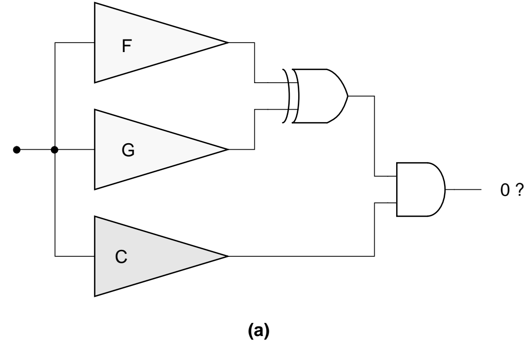


Figure 2: Structural preservation of circuit parts for the compose operation



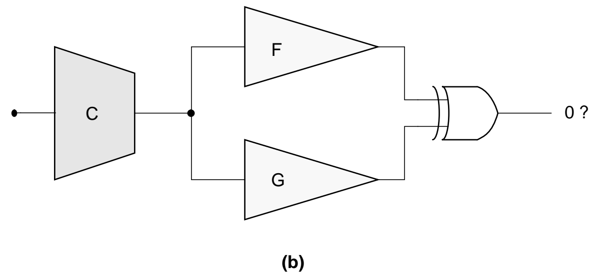


Figure 3: Methods for testing functional equivalence in presence of input constraints

4.2 Input Constraints

An important aspect of practical verification tasks are boundary conditions at the primary inputs of a design. They imply, that the equivalence proof has to hold only for a specific care-set. As an example consider a decomposition of a design into two parts A and B with outputs of A being inputs of B. Furthermore assume, that the outputs of A are one-hot-encoded. If the implementation of B relies on a one-hot-encoding of its inputs, this encoding forms an input constraint for B. The verification of B must take this constraint into account, otherwise a false negative will occur. Furthermore, the verification of A now includes the proof that its outputs actually have the required encoding.

In this section we will focus on the problem of proving the equivalence of two functions F and G with respect to some constraint C. There are basically two ways to approach this problem. The method shown in figure 3.a is normally chosen for BDD-based verification approaches. Empirical evidence shows that constraining the BDD as late as possible helps keeping its size within reasonable limits during the functional extraction process. In case of random simulation however, this method is not appropriate due to the nature of constraints. It is our experience, that constraints often reduce the care-set dramatically, e.g. by specifying one-hot-encodings for busses. Therefore it is very unlikely, that any randomly generated pattern will satisfy the constraint. As a consequence a possible miscompare of F and G would never be detected, which greatly undermines our goals.

For simulation purposes the method of figure 3.b is better suited. Instead of filtering the pattern universe, we take a constructive approach and *generate* only those pattern which satisfy constraint C. The constraint expression is converted into a DAG, with leaf nodes and nonterminal nodes representing primary inputs and Boolean operations, respectively. Complex operators such as the orthogonality operator are preserved during tree construction. A recursive function traverses this tree and computes satisfying pattern, which is theoretically a NP-complete problem. In practice, however, this is not an issue because constraints are provided by the user and are typically simple expressions.

Note, that there is a subtle difference between the proposed methods. In case of an empty care-set, method (a) delivers a 0 while method (b) fails to generate any pattern leaving the output undefined. Although empty care-sets do not make sense, a robust verification method must be able to handle them by enforcing a time limit for the pattern generation process.

4.3 Pattern Generation and Quality

Apart from the simulation process itself, the quality of the generated pattern is important for a good discrimination performance of the simulation engine. We measure pattern quality informally as follows:

First of all, it is important that all pattern satisfy a given input constraint. Otherwise the design is simulated with illegal pattern. In case of a miscompare it cannot be decided whether this is due to a false negative or the functions are in fact different.

Second, for the purpose of error detection random pattern are needed. From a user's perspective a random pattern is considered "good", if it detects his or her error in the design. While a specific pattern might detect error A it might not be able to detect error B. This classification makes the quality of a pattern dependent on the actual design errors which are not known upfront. Therefore, we use a linear feedback shift register for generation of random pattern. Generally, the probability that an error gets detected depends on the number of pattern and also on the number of minterms that exercise the error. It is our observation, that about 80% of practical design errors can

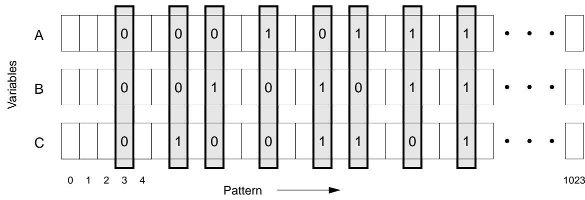


Figure 4: Detection of exhaustive pattern

be detected with only 1024 random pattern.

Third, to proof functional equivalence, exhaustive pattern are needed. Exhaustive simulation can only be done for problems with a small set of inputs feeding them. Instead of generating exhaustive pattern for selected problems, we generate random pattern and hope that these pattern contain exhaustive pattern for the problem under consideration.

As an example consider a comparison of two functions F and G. Suppose that both functions depend on the primary inputs A, B and C. In order to proof the equivalence of F and G, the pattern must contain all minterms $000, 001, 010, \ldots, 111$ (see figure 4). Clearly, the larger the number of pattern the higher the probability that those eight minterms are found among them. Obviously, there are different ways to test the existence of the required minterms. The most costly way in terms of runtime is to loop over the pattern and test the existence of every individual minterm. A faster way is to exploit the fact that a pattern is stored as an array of `int` and to assume that *all* minterms are present in *one* word. This allows to check the existence of all minterms in parallel with a single expression. However, if the minterms are spread over several words, the fast algorithm will fail to detect them.

The process of pattern generation can be summarized as follows.

1) For all primary inputs contributing to input constraints generate pattern, which satisfy the constraint.

2) For all primary inputs, which do not contribute to any input constraint, generate random pattern.

3) To test a set of pattern for its exhaustiveness with respect to a subset of primary inputs, use the fast algorithm as described above.

4.4 Signature Computation

Cut-point guessing [3] is a general technique which exploits structural similarities of the two designs to be compared by finding functionally equivalent nets. The cut-point identification is typically done in two steps, where first a list of candidates is determined and then their equivalence is verified.

The resulting pattern of random simulation can be used as signatures to functionally classify the internal nets and to select good candidates. In order to identify nets with inverted functions, the simulation pattern need to be converted into unique polarities. Depending on the simulation value for the first pattern,

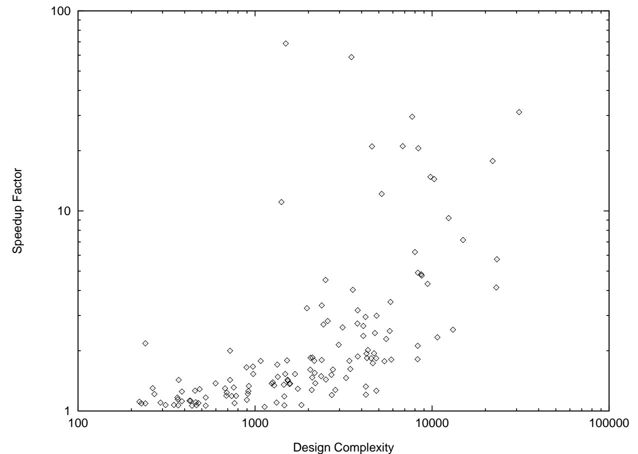


Figure 5: Performance gain by using exhaustive simulation

net entries are inverted or not. The resulting signature is then hashed to determine nets with identical signatures. The actual cut-point selection is based on a variety of heuristics which are not discussed in this paper.

5 Results

In this section we will present experimental results to furnish our claims.

To show that exhaustive simulation outperforms BDD-based reasoning we conducted the following experiment. Verity was applied to approximately 150 circuits which were taken from a microprocessor design within IBM. In a first run Verity used the simulator to identify the problems which can be solved by exhaustive simulation. We then reran Verity on exactly those problems with the simulator and the BDD package as reasoning engine and measured the corresponding CPU time. The resulting runtime ratio (BDD / simulator) is plotted in figure 5. It is evident, that there is no runtime penalty for invoking the simulator. However, since designs with up to 1000 verification problems usually run within a few seconds, the speedup for small circuits is not significant. As expected, there is a significant performance gain for larger designs which often exceeds an order of magnitude.

To rate the simulator's capability of providing counter examples we estimated the probability that a given error will be detected. For this purpose we recorded the number of inputs each verification problem depends on (*support*). Figure 6 shows that 95% of all problems depend on 15 or less primary inputs. 85% of all problems have a support size of 5 or less. This large number of small supports is due to the hierarchical verification approach and the applied blackboxing scheme. Since the simulator will find an error in these problems with absolute certainty (due to exhaustive pattern) the overall probability of error detection is 85% which matches our empirical observations.

To underline the importance of the reference

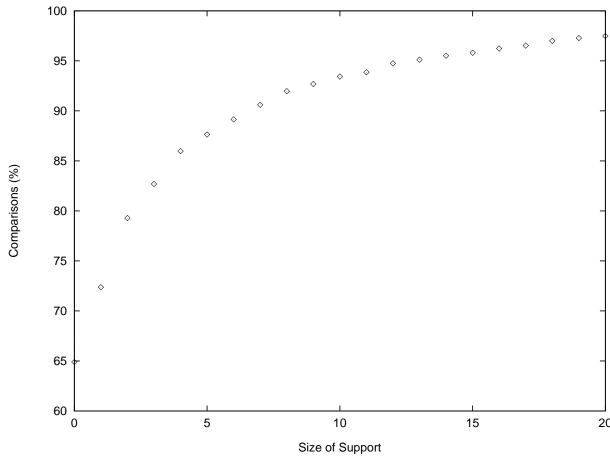


Figure 6: Distribution of verification problems with support size of x or less

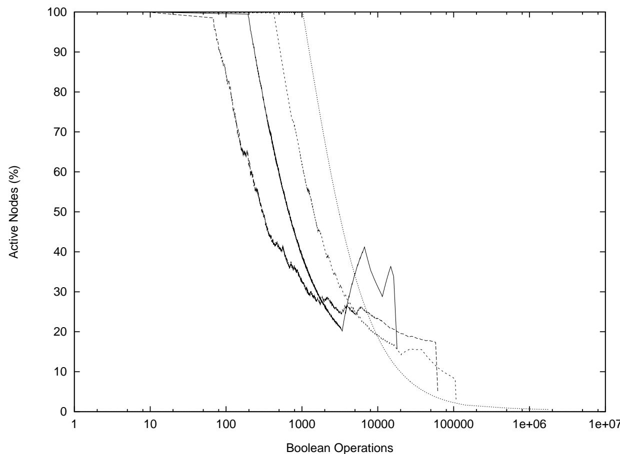


Figure 7: Profile of active nodes during verification process

counter scheme as introduced in section 4 we recorded the number of active net entries during a verification process and compared those numbers with a verification run where all nodes were kept in a unique table. The resulting ratio is plotted in figure 7 for designs of various complexity. The horizontal axis shows the number of net entries created as runtime proceeds. At the beginning of a verification process entries representing primary inputs are created. As soon as Boolean operations are performed there is a steep decrease in the number of active net entries due to releasing unused entries. After 10000 operations only 20% of all nodes are active. This allows to run the simulator with a large number of pattern to increase discrimination performance.

6 Conclusion

In this paper we presented the integration of a random simulation algorithm into the verification tool Verity. The simulator was implemented as one of many Boolean reasoning engines which communicates through a well defined interface with the functional extractor.

The simulator complements the BDD-based verification approach in Verity in mainly three areas: First, in case of functional miscompares, random simulation typically calculates a counter example pattern more efficiently than BDDs. Second, for a large number of functional comparisons of small logic expressions, exhaustive simulation can outperform BDDs by more than a magnitude. Third, random simulation pattern can be used as signatures for internal nets to partitioning the verification problem by cut-points.

In order to achieve efficient memory and runtime performance, the simulation algorithm is applied on the fly during the functional extraction step. In conjunction with constant folding, inverter elimination, and maintaining the lifetime of functions, this approach dynamically minimizes the memory resources and avoids redundant simulation effort.

References

[1] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, pp. 677–691, August 1986.

[2] A. Kuehlmann, A. Srinivasan, and D. P. LaPotin, "Verity - a formal verification program for custom CMOS circuits," *IBM Journal of Research and Development*, vol. 39, pp. 149–165, January/March 1995.

[3] C. L. Berman and L. H. Trevillyan, "Functional comparison of logic designs for VLSI circuits," in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 456–459, IEEE, November 1989.

[4] C. A. J. van Eijk and J. A. G. Jess, "Detection of equivalent state variables in finite state machine verification," in *1995 ACM/IEEE International Workshop on Logic Synthesis*, (Tahoe City, CA), pp. 3–35 – 3–44, May 1995.

[5] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, (Orlando), pp. 40–45, ACM/IEEE, June 1990.