Probabilistic Design Verification*

Jawahar Jain Jim Bitner Donald S. Fussell Jacob A. Abraham

Computer Engineering Research Center
The University of Texas at Austin
Austin TX, 78712

Abstract

We present a novel method for verifying the equivalence of two Boolean functions. Each function is hashed to an integer code by assigning random integer values to the input variables and evaluating its integer-valued representation. The equivalence of two functions can be verified with a very low probability of error. The probability of error can be exponentially decreased by making multiple runs. Results indicate significant time and space advantages for this method over deterministic techniques. Some functions known to require space (and time) exponential in the number of input variables for deterministic verification require only polynomial resources using our technique.

1 Introduction

As the level of integration of digital systems increases, it is becoming prohibitively difficult to develop circuits with a high probability of correct operation without resorting to CAD tools for formal verification. The most popular approach for "solving" the verification problem is *simulation*. A large set of Boolean input patterns is selected or generated, and the circuit is tested against them. If the circuit passes this test, confidence in it is increased, but its complete correctness is not verified unless all possible input patterns are exhausted. This approach is unsatisfactory because: (1) for a large circuit, only a tiny fraction of its functionality can be tested and (2) errors seem to result from unexpected situations, which are unlikely to have been checked. Therefore, more exact verification methods are now being developed. The most successful employ ordered binary decision diagrams (OBDDs) as a canonical representation for both Boolean circuit specifications and logic designs. However, for many practical circuits, verification systems using OBDDs require an unacceptably large amount of time and memory.

Thus, each of these methods has significant limitations. To circumvent them, we have developed and implemented a new *probabilistic* method for verification, which provides nearly 100% confidence in correctness and has lower space and time overhead than OBDD-based approaches. To validate this claim, we have run a preliminary implementation on the ten ISCAS benchmark circuits. Except for some of the smallest ISCAS circuits, our method significantly outperformed deterministic methods in both time and memory requirements.

*This work was supported in part by the Texas Advanced Research Program and by Semiconductor Research Corporation Grant #90-MJ-136.

2 A New, Probabilistic Approach

We now give a brief description of our method. A more complete description is found in [5], [6].

2.1 Basic Strategy

Instead of evaluating the circuit and design on random Boolean input vectors, we will use random integer values, modulo p, for some prime p. To define the "output" of a circuit when *integer* values are assigned to its inputs, a *functional transformation*, the **A**-transform, is used to map F, the Boolean function realized by the circuit, into an arithmetic function $\mathbf{A}[F]$. We can then compare two functions probabilistically as follows: (1) Transform Boolean functions F_1 and F_2 into arithmetic functions $\mathbf{A}[F_1]$ and $\mathbf{A}[F_2]$. (2) Evaluate $\mathbf{A}[F_1]$ and $\mathbf{A}[F_2]$ on a randomly chosen integer vector to obtain integer codes H_1 and H_2, and (3) Compare H_1 and H_2. After making this comparison, there are two possibilities:

- If $H_1 \neq H_2$, we conclude that the functions are inequivalent *with certainty*.
- If $H_1 = H_2$, we conclude the functions are equivalent, with a *very small* probability of error.

This approach can essentially be considered a method of *hashing* functions to produce integer hash codes. Errors result from very unlikely collisions between inequivalent functions.

2.2 Properties of Hash Codes

The mathematical definition of the **A**-transform is made in terms of its 2^n truth values. We associate a "key polynomial" with each of the 2^n input assignments to the given function $\beta(x_1, \ldots, x_n)$. We then sum the polynomials associated with assignments producing the output value *true*, and interpret the result as an integer-valued function to obtain the required algebraic transform.

The key polynomial for a given row of the truth table is a product of terms, where each term is associated with a particular input variable x_i. If b_i represents the truth value assigned to x_i in the current row of the truth table, then the corresponding

term $\mathbf{w}(b_i, x_i)$ in the key polynomial is defined as $\mathbf{w}(b_i, x_i) = b_i x_i + (1-b_i)(1-x_i)$. Parameter b_i simply acts as a selector between x_i and $1 - x_i$.

While this is a fine definition mathematically speaking, it is not at all effective computationally. Therefore, we have developed a set of theorems that provides a calculus for manipulating **A**-transforms computationally without resorting to this cumbersome definition. One of these theorems allows the replacement of Boolean functions by corresponding arithmetic functions. For example, $\neg\beta_1$, $\beta_1 \wedge \beta_2$ and $\beta_1 \vee \beta_2$ can be replaced by $\neg_{\mathcal{F}}(\beta_1) = 1 - \beta_1$, $\beta_1 \wedge_{\mathcal{F}} \beta_2 = \beta_1 * \beta_2$ and $\beta_1 \vee_{\mathcal{F}} \beta_2 = \beta_1 + \beta_2 - \beta_1 * \beta_2$, respectively. All these arithmetic operations are conducted modulo p. More generally, these operations can be carried out in any *field*, $\mathcal{F}$. A second theorem allows the removal of exponents greater than 1. This enforces the idempotence of Boolean multiplication. Using such theorems, we can calculate the **A**-transform of a Boolean expression. For example,

$$\mathbf{A}[x \wedge y] = \mathbf{A}[x * y] = x * y$$

$$\mathbf{A}[(x \vee x) \wedge y] = \mathbf{A}[(x + x - x^2) * y] = \mathbf{A}[(2x - x) * y]$$
$$= \mathbf{A}[x * y] = x * y$$

$$\mathbf{A}[x \wedge y \wedge \overline{z}] = \mathbf{A}[x * y * (1 - z)] = x * y * (1 - z)$$

Note that the first two Boolean expressions are different, but *equivalent*. For equivalent functions, identical **A**-transforms result. Thus, identical hash codes will be generated from equivalent functions when values are assigned to the variables.

These examples illustrate only a few of the important properties of the **A**-transform. For example, we can show that **A**-transforms can be expanded in a manner analogous to Shannon's expansion of Boolean functions [6]. Using the notation $f_{x=R}$ to denote the function which is the restriction of f where x is replaced by R, $(R \in \mathcal{F})$, we have

Linear Expansion Theorem: Let variable x occur in the parameter list of $\mathbf{A}[f]$, i.e., $\mathbf{A}[f](\ldots, x, \ldots)$, then $\mathbf{A}[f] = (1 - x) \cdot \mathbf{A}[f_{x=0}] + x \cdot \mathbf{A}[f_{x=1}]$.

3 Bounds on the Error Probability

If we are asked to determine if the second and third Boolean expressions given above are equivalent (using say, prime $p = 46,337$), we choose random values for x, y and z, say, 18370, 38945, and 16372, and evaluate the two **A**-transforms. The hash codes 22707 and 25454 result, and the functions are proved inequivalent.

If we had chosen $x = 0$ instead, identical hash codes of 0 would have resulted, and we would have failed to distinguish the two functions. However, *only* integer vectors which have x, y or z equal to 0 fail to distinguish these two functions. Thus, $(p-1)^3$ of the p^3 possible integer vectors distinguish the functions, and the probability of failing to distinguish the two functions, and thereby concluding that they are equivalent, is approximately $3/46337 = 6.5 \times 10^{-5}$.

A general bound on the probability that this conclusion is erroneous, i.e, the functions are really *inequivalent*, relies on a theorem in [6]: For *any pair* of inequivalent functions, at least $(p-1)^n$ of the p^n possible integer vectors will succeed in distinguishing the functions. Thus, a randomly chosen vector distinguishes the functions with probability at least $(p-1)^n/p^n \approx 1 - n/p$. For a 64-input circuit, if p is a 32 bit prime, the probability of failing to distinguish two inequivalent functions is $\epsilon \approx n/p \approx 1.5 \times 10^{-8}$. The probability of error is 15 in a billion and can be made even smaller by using a larger prime.

A second technique to further reduce the error bound is to make multiple *runs*. On each run, an independent set of input variable assignments is randomly chosen, and the two function values are computed. If the values differ, we are assured that the two functions are not the same. If they are equal, we choose a new set of input assignments and reevaluate. The probability of erroneously deciding the functions are equal decreases *exponentially* with the number of runs: after k runs, the error probability is ϵ^k.

Interestingly, if 0 and 1 are ruled out as choices for the random values, then *any* pair of functions that differ in only one minterm can be distinguished with probability 1. In contrast, this is the most difficult case to be distinguished by a Boolean simulation.

We can also identify many new scenarios for functional simulation. One is *partial simulation*, which assigns random values to v of the variables and keeps the remainder symbolic. Two functions are equivalent only if identical semi-numeric expressions are obtained for each. This technique reduces ϵ from n/p to v/p. A second technique is *mixed-mode simulation*. In this case, v variables receive random integer values and the remaining $n - v$ variables receive random Boolean values, and $\epsilon \leq 1 - (1/2^{n-v}) * (1 - v/p)$. It can be seen to represent a spectrum of strategies ranging from completely arithmetic simulation $(v = n)$ as discussed in this paper, to random Boolean simulation $(v = 0)$.

Any of these probabilistic verification techniques can be used to not only detect whether two functions are different, but also to produce an input Boolean vector on which they differ [5], [6].

4 Data Structures and Algorithms

Polynomials do not provide an efficient computational representation for the (transformed) arithmetic function. Instead we can employ BDDs as an efficient intermediate representation for Boolean functions.

We have developed an algorithm from the properties of the **A**-transform that efficiently calculates the hash code for a Boolean function directly from its BDD [6]. Thus, a simple way to compute the hash code would be to generate a BDD for the Boolean function and then hash it to an integer. However, the performance of such a strategy is limited by the need to generate the BDD. To achieve a computational advantage, we must use an *incremental hashing strategy*, i.e., (1) decompose the function, (2) hash its subfunctions, and then (3) *combine* these more compact intermediate forms to obtain the hash code for the entire function.

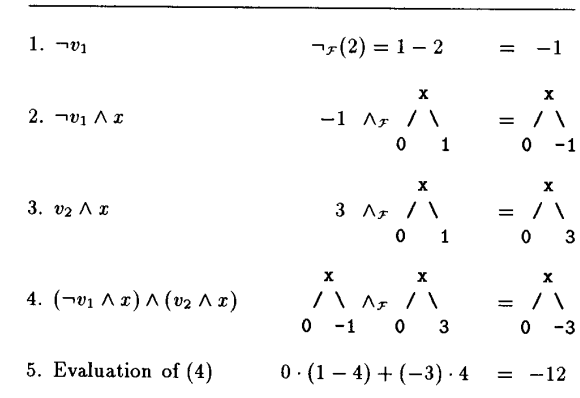

Figure 1: Incremental Evaluation of $f = (\neg v_1 \wedge x) \wedge (v_2 \wedge x)$ with assignments $v_1 = 2$, $v_2 = 3$ and $x = 4$.

One method of effecting step (3) is to combine the representations of the subfunctions pair-wise. Unfortunately, we can rarely combine hash codes for the subfunctions and get the correct hash code for the entire function. Therefore, we have developed several theorems about the **A**-transform that allow the combination of *semi-numeric* representations, where *some* of the variables are represented numerically. This necessitates a new data structure, the *semi-numeric decision diagram,* (snDD) for representing such semi-numeric (partially hashed) functions [6].

4.1 Semi-Numeric Decision Diagrams

Figure 1 shows an example for the sequence of operations on snDDs that would be done to compute a numeric value for f. We have implemented operations similar to *apply* and *reduce* of [2] to manipulate snDDs, and we can evaluate them in a bottom up fashion to obtain numeric representations of functions [6].

This example illustrates a situation in which the numeric values obtained during the computation are maintained at the terminal nodes of the graph. We can maintain weights at other places besides the terminals of an snDD. For example, weights may be placed at a given node and the resulting **A**-transform interpreted as an arithmetic sum of the **A**-transforms of each subgraph that it roots (a *summation* node). Similarly multiplication and subtraction nodes are defined.

4.2 Decomposition and Evaluation

An alternate method for effecting step (3) is to use an algorithmic equivalent of the *compose* algorithm for OBDDs [2], called *collapse-with-compose,* to recompose the subfunctions and introduce numeric values during the process. We begin by decomposing a circuit or function into component subfunctions. Each subfunction represents the output of some gate (subfunction). Then we construct a representation for each subfunction, and a pseudovariable graph to describe the (decomposed) function. Instead of labeling a node in the pseudovariable graph with a primary input, it has pointer to a substructure representing the function of the input variables corresponding to that pseudovariable.

Input variables which occur in more than one subfunction cannot be immediately replaced by their integer value. The numeric evaluation of this decomposed representation can be effected by sequencing through the input variables and applying the Linear Expansion Theorem (Section 2) for each. This involves computing the two restrictions of each subfunction and forming an snDD structure to represent the algebraic representation given in the theorem. Once a variable is "extracted" from the subfunctions in this manner, it then can be replaced by its numeric value.

The above technique can be significantly enriched by employing various properties of **A**-transforms [5]. Consider again the Linear Expansion Theorem. We say that each summand in its expression is functionally *orthogonal.* If C is the circuit for f, we can evaluate it by evaluating the two *reduced circuits,* $C_{x=0}$ and $C_{x=1}$, where x has been replaced by 0 and 1, respectively. To compute the hash code for f, we simply hash each reduced circuit separately and compute a weighted sum of the results. By using Boolean simulation on a small set of variables, several reduced copies of circuits are obtained, each of which can be now hashed separately. We have employed orthogonal circuit partitioning for two of the difficult benchmark circuits (c3540 and c6288).

Orthogonal partitioning also allows one to compute in $\Theta(n^3)$ space and time the hash code for the circuit realization of an n input function [3] which has no polynomial size OBDD. It can be also shown to reduce the space required to verify a multiplier circuit, such as c6288, by as much as 3 orders of magnitude [5]. Such techniques can also be useful in deterministic verification.

5 Experimental Results

In this section, we discuss practical results obtained with our implementation of the *collapse-with-compose* algorithm. Table 1 shows a comparison of the space-time characteristics of this algorithm as measured against our implementation of standard OBDD-based deterministic methods for the verification of the 10 ISCAS circuits. Both programs were written in C++ and run on a Sun-4/280 with 32 MBytes of memory. All runs on the same circuit were supplied with the same orderings on the OBDD variables, selected by a procedure similar to [7]. These orderings are known to be efficient for OBDDs. The decompositions were selected by a procedure similar to one outlined in [1], and the pseudovariables generated were ordered using approaches outlined in [4], [7].

The comparison of our probabilistic and deterministic implementations was made for the last output function of each ISCAS circuit. This appears to be the worst-case output function, except for c6288, a 16-bit multiplier, where the 16th output is the hardest output function and was selected instead. As our output representation is simply an integer, not a OBDD, a meaningful space comparison could only be established by comparing the intermediate memory require-

ments. Since most published works on OBDDs do not report intermediate memory requirements (which can be often 10 to 15 times the final OBDD size), our comparisons of the probabilistic technique are made against our own implementation of the deterministic approach; we could measure the peak space requirements of *both* methods.

To check the fairness of our deterministic implementation, we compared it with some other published implementations [4], [7] that employ the same basic BDD features (not shown, see [6]). Performance of such methods crucially depend on the orderings selected. Comparing on circuits where the orderings obtained by each method were of comparable quality, we found that our deterministic implementation was at least as efficient as these deterministic implementations. Thus our own deterministic implementation is a reasonable basis for comparison with our probabilistic method.

As can be seen from Table 1, except for some of the smallest ISCAS circuits, our probabilistic method significantly outperformed deterministic methods in both time and memory requirements. Besides the overall improvement, two outstanding points are also worth noting. First, we were able to compute an integer hash code for c3540, in spite of constructing our BDDs according to an ordering on the variables which was poor enough to cause our implementation of deterministic methods to run out of memory. Second, we were able to generate the hash code for the 16th output c6288, a 16×16-bit multiplier, allowing its probabilistic verification. We know of no technique using OBDDs which has successfully been used to verify this circuit deterministically.

	Probabilistic Method		Deterministic (compose)		Deterministic (apply)	
Ckt.	Time	Nodes	Time	Nodes	Time	Nodes
c432	9.4	4,021	10.7	6,886	9.8	4,817
c499	6.2	2,911	5.5	12,866	8.1	11,214
c880	5.2	2,329	5.3	8,330	4.3	4,137
c1355	9.4	2,911	9.2	12,848	14.5	15,752
c1908	15.2	5,107	21.9	21,749	24.8	14,309
c5315	21.2	3,197	21.5	7,641	20.8	5,921
c7552	4.8	737	4.6	1,228	4.8	1,219
c2670	43.2	30,329	1500	1,254,170	617.2	478,021
c3540	9:40 hr.	227,347	unable (1)		unable (1)	
c6288	5:56 hr.	288,260	unable (2)		unable (2)	

Table 1: Probabilistic–Deterministic Comparison

Unless otherwise noted, times are in seconds.

(1) For c3540 our ordering was not effective. Both deterministic methods ran out of memory in 18 to 30 minutes. This illustrates that though our method benefits from a good ordering, its performance degrades only slowly with a bad ordering. Sensitivity to ordering of input variables has been a major drawback of deterministic methods.

(2) Empirical results indicate that OBDD-based deterministic methods may require 10–15 million nodes to verify this circuit.

6 Conclusion

We have developed new techniques for design verification using probabilistic methods which can provide significant time and space advantages over deterministic techniques for verifying Boolean functions. Through these methods, problems requiring similar methods such as network reliability analysis and ATPG measures can also be efficiently solved.

An integer-valued algebraic representation of Boolean functions is the key to our method. We have developed theorems for manipulating these algebraic representations, as well as data structures and algorithms for efficiently realizing operations on them.

Exploiting the properties of algebraic transforms opens various options in limiting the cost of the verification. The *collapse-with-compose* algorithm and the Orthogonal Partitioning technique described here are only two of many such options. We are implementing more sophisticated versions of the above algorithms as well as new algorithms based on other properties of algebraic transforms.

If limited resources make even verification by completely arithmetic simulation infeasible, mixed mode simulation can allow verification to be tailored according to the space and time available on the given machine. In comparable time, it can likely provide an error coverage much superior to non-exhaustive random Boolean simulation.

Our preliminary implementation results indicate that probabilistic verification provides a very attractive alternative to current deterministic methods of verification when the functions to be verified are too large or complex to be handled by these techniques.

References

[1] C. L. Berman. Circuit width, register allocation, and reduced function graphs. *IBM Research Report, RC 14129*, October 1988.

[2] R. E. Bryant. Graph based algorithms for Boolean function representation. *IEEE Transactions on Computers*, C-35:677–690, August 1986.

[3] L. Fortune, J. Hopcroft, and E. M. Schmidt. The complexity of equivalence and containment for free single variable program scheme. *Goos, Hartmanis, Ausiello and Bohm, Eds., Lecture Notes in Computer Science 62, Springer-Verlag*, pages 227–240, 1978.

[4] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of Boolean comparison method based on binary decision diagrams. *ICCAD*, pages 2–5, 1988.

[5] J. Jain, J. Bitner, J. Abraham, and D. Fussell. A scheme for probabilistic design verification. *Submitted for publication*, 1991.

[6] J. Jain, J. Bitner, D. Fussell, and J. Abraham. Probabilistic design verification. Technical Report UT-CERC-TR-JAA91-01, University of Texas, 1991.

[7] S. Malik, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. *ICCAD*, pages 6–9, 1988.