Formal Verification using Probabilistic Techniques

René Krenz* Elena Dubrova
Department of Microelectronic and Information Technology
Royal Institute of Technology
Stockholm, Sweden
{rene,elena}@ele.kth.se

Abstract

Formal verification is of importance in many phases of the design of digital systems. In spite of the inherent complexity of the problem, the state-of-the-art in the area has been significantly advanced by BDD-based tools. However, the growing complexity of the verification instances keeps motivating the exploration of alternative approaches. This paper considers probabilistic equivalence checking techniques. Probabilistic methods decide equivalence of two circuits by hashing them and comparing the resulting hash codes. We design an algorithm which takes $O(n \cdot k)$ time to compute the hash code for an n-variable Boolean function given as a disjoint sum-of-products with k products.

1 Introduction

Combinational equivalence checking is one of the key techniques of the current verification methodology for digital systems. Given two combinational netlists of the same number of inputs and outputs, the goal is to determine whether for every possible input assignment, each pair of corresponding outputs evaluates to the same value. Although this problem is known to be coNP-hard, many practical instances are tractable.

Most of the current successful equivalence checkers use Binary Decision Diagrams (BDDs) [1] or their derivatives as a core of the equivalence deduction engine. The circuits to be verified are converted into BDDs which are then structurally compared. This type of equivalence checking is fast and independent of the actual circuit structure. However, if the BDD representation grows too large, its storage and manipulation becomes infeasible. Various techniques have been proposed to reduce the memory complexity of BDDs by exploiting the structural and functional similarities of the circuits [2]-[5]. Some alternative to BDD-based equivalence checkers use Boolean Satisfiability (SAT) [6], [7] or SAT-like methods (ATPG [8], recursive learning [9]) as a principal engine.

In spite of the considerable advances in the area, the growing complexity of the verification instances motivates exploring the alternative approaches. We study the properties of the transform which is used in the probabilistic method [11] for transforming Boolean functions into polynomials and derive several rules for speeding-up the computation of hash codes. The main goal is to avoid generating the entire polynomial for a function f, but instead to derive

*This work was supported in part by the Research Grant No 64700 from the SSF INTELECT program.

the polynomials only for very small subfunctions of f, hash them and combine the results to compute the hash code for f. We show that if the input function is given as a disjoint sum-of-products, our rules allow to shrink the size of the subfunctions for which the polynomial has to be generated to 1-variable subfunctions (x and x'). The polynomials for x and x' are x and $1-x$, correspondently, so on practice we do not generate polynomials *at all* but simply symbolically replace. Using these rules, we design an algorithm which takes $O(n \cdot k)$ time to compute the hash code for an n-variable function if the function is given as a disjoint sum-of-products with k products. Our experimental results show that it takes less than 0.1sec to compute and compare hash codes for disjoint sum-of-products of size $k < 5000$. Furthermore, this time is smaller than the time for building can comparing BDDs for the corresponding functions.

The paper is organized as follows. In Section 2 gives a brief overview of the arithmetic transform and describes the new algorithm for computing hash codes from disjoint sum-of-products. Section 3 shows the experimental results. Section 4 concludes the paper.

2 Probabilistic verification method

2.1 Notation

Throughout the paper, we denote by f_1, f_2 Boolean functions of type $\{0,1\}^n \rightarrow \{0,1\}$ and by $A[f_1]$, $A[f_2]$ the functions over the field Z_p, p - prime, of type $Z_p^n \rightarrow Z_p$. We use the symbols $\vee$, $\wedge$ and $'$ for Boolean operations AND, OR and NOT, correspondently. We denote by $\cdot$, $+$ and $-$ the arithmetic operation over the field Z_p (all modulo p). We use $x_1, x_2, \ldots, x_n$ for both, Boolean and field variables. For example, $x_1 \wedge x_2$ stands for a Boolean expression representing an AND of two Boolean variables; $x_1 \cdot x_2$ stands for a field expression (polynomial) representing a multiplication mod p of two field variables.

A *product-term* is a Boolean product (AND) of one or more variables $x_1, \ldots, x_n$ or their complements. A *sum-of-products* is a Boolean sum (OR) of one or more product-terms. A sum-of-products $F = c_1 \vee c_2 \vee \ldots \vee c_k$ is *disjoint* if $c_i \wedge c_j = 0$ for any $i, j \in \{1, \ldots, k\}, i \neq j$.

By $sup(f)$ we denote the *support set* of a Boolean function f, which is the set of variables on which the function actually depends, i.e. $sup(f) = \{i \mid f|_{x_i=0} \neq f|_{x_i=1}\}, i \in \{1, \ldots, n\}$.

2.2 Basic idea of the probabilistic method

The probabilistic method based on arithmetic transform has been developed in [11] for Boolean functions $\{0,1\}^n \rightarrow \{0,1\}$ an extended in [12] to the multiple-valued case $\{0,1,\ldots,m-1\}^n \rightarrow \{0,1,\ldots,m-1\}$, m - positive integer greater than 2. In both cases, the combinational logic circuits f_1 and f_2 to be compared are transformed to the integer-valued polynomials $A[f_1]$ and $A[f_2]$. These polynomials are functions of type $Z_p^n \rightarrow Z_p$ over a finite field of integers Z_p with p being a prime. To compute hash codes, $A[f_1]$ and $A[f_2]$ are evaluated for the values of variables taken independently and uniformly at random from Z_p. The resulting hash codes, H_{f_1} and H_{f_2}, are compared and the following conclusions are drawn:

1. If $H_{f_1} \neq H_{f_2}$, then $f_1 \neq f_2$.
2. If $H_{f_1} = H_{f_2}$, then $f_1 = f_2$ with a probability of error ϵ.

Conclusion (1) can be made because a polynomial is unique for a given function. Note that in this case the correctness of the conclusion is 100% guaranteed. This makes this method particularly attractive for applications where the expectation to get the answer *no* is high, e.g. in detection of bugs. If $H_{f_1} = H_{f_2}$, then the decision is taken with a quantified probability, because an error may result from collision between non-equivalent functions. It was shown in [11] that, for any n-variable Boolean function, the error probability ϵ is at most $(p-1)^n/p^n \approx n/p$, for $p >> n$, where p is the size of the field Z_p. For example, if $n = 64$ and p is a 32-bit integer, then $\epsilon \approx 1.5 \cdot 10^{-8}$. The error probability can be made even smaller by using a larger p.

The formal definition of arithmetic transform $A : f \mapsto A[f]$ is made by associating a *key polynomial* with each of the 2^n input assignments of a function $f(x_1, \ldots, x_n)$. The key polynomials of assignments producing the non-zero output value of f are then summed up to producing the output value of f. This sum is interpreted as an integer-valued polynomial $A[f](x_1, \ldots, x_n)$ over Z_p.

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, $i \in \{1, \ldots, n\}$. If b_i represents the value of x_i in a given row of the truth table, then the corresponding term $w(b_i, x_i)$ in the key polynomial is defined as $w(b_i, x_i) = b_i \cdot x_i + (1 - b_i) \cdot (1 - x_i)$. Parameter b acts as a selector between x_i and x_i'.

For a more detailed description of arithmetic transform the reader is referred to [12].

2.3 An new algorithm for computing hash codes

Polynomials $A[f]$ are not efficient as a data structure. Often, it takes more memory to store $A[f]$ than to store f. Therefore, we would like to avoid computing and storing a complete representation of $A[f]$. Instead, we would rather derive the polynomial only for very small subfunctions of f, hash them and then combine to compute the hash code for f. This would be very easy to perform incrementally if the following property was satisfied:

$$A[f_1 \bullet f_2] = A[f_1] \bullet_p A[f_2] \tag{1}$$

where $\bullet$ is some Boolean operation and $\bullet_p$ is some operation over the field Z_p. If (1) holds, then, for any assignment $(a_1, \ldots, a_n) \in Z_p^n$, $A[f_1 \bullet f_2](a_1, \ldots, a_n) = A[f_1](a_1, \ldots, a_n) \bullet_p A[f_2](a_1, \ldots, a_n)$ and thus $H_{f_1 \bullet f_2} = H_{f_1} \bullet_p H_{f_2}$. So, we can first compute H_{f_1} and H_{f_2}, and then apply $\bullet_p$ to them to get the $H_{f_1 \bullet f_2}$.

Unfortunately, the equation (1) does not hold for all the operations and all the functions. We can identify the following special cases for which (1) is satisfied. Lemma 1 is a special case of Theorem 9 [11, p. 72] and Lemma 2 is a special case of Theorem 3 [11, p. 69]. Let f_1 and f_2 be Boolean functions.

Lemma 1 *If* $f_1 \wedge f_2 = 0$, *then* $A[f_1 \vee f_2] = A[f_1] + A[f_2]$.

Lemma 2 *If* $sup(f_1) \cap sup(f_2) = \emptyset$, *then* $A[f_1 \wedge f_2] = A[f_1] \cdot A[f_2]$.

Lemmata 1 and 2 allow us to perform the computation of hash codes incrementally when the conditions $f_1 \wedge f_2 = 0$ and $sup(f_1) \cap sup(f_2) = \emptyset$ are satisfied. For example,

$$\begin{aligned} A[x_1 \vee (x_1' \wedge x_2)] &= A[x_1] + A[x_1' \wedge x_2] && \text{since } x_1 \wedge (x_1' \wedge x_2) = 0 \\ &= A[x_1] + A[x_1'] \cdot A[x_2] && \text{since } sup(x_1) \cap sup(x_2) = \emptyset \\ &= x_1 + (1 - x_1) \cdot x_2 && \text{since } A[x] = x \text{ and } A[x'] = 1 - x \end{aligned}$$

Suppose we evaluate the polynomial $A[x_1 \vee (x_1' \wedge x_2)]$ for the values $(a_1, a_2) \in Z_p^2$, then $H_{x_1 \vee (x_1' \wedge x_2)} = A[x_1 \vee (x_1' \wedge x_2)](a_1, a_2) = a_1 + (1 - a_1) \cdot a_2$.

```
**HashFromDisjoint**$(F)$
**input**: a disjoint sum-of-products $F$ for $f(x_1, \ldots, x_n)$
**output**: the hash code for $f$

Generate random values $a_1, \ldots, a_n \in Z_p$ for variables $x_1, \ldots, x_n$;
Compute $a'_i = 1 - a_1$ for all $a_i$, $i \in \{1, \ldots, n\}$;
hash = 0;
for $(i = 0; i < |F|; i++)$
    hash_tmp = 1;
    for $(j = 0; j < n; j++)$
        if($x_j$ is complemented)
            hash_tmp = hash_tmp $\cdot$ $a'_j$;
        else
            if($x_j$ is not complemented)
                hash_tmp = hash_tmp $\cdot$ $a_j$;
    hash = hash + hash_tmp;
return(hash);
```

Figure 1: Pseudocode of the algorithm for computing the hash code.

Next we show that if a Boolean function f is given by a disjoint sum-of-products, then Lemmata 1 and 2 can be subsequently applied to break down the $A[f]$ to the polynomials of 1-variable subfunctions (x_i and x'_i, $i \in \{1, \ldots, n\}$). Since the polynomials for x_i and x'_i are $A[x_i] = x_i$ and $A[x'_i] = 1 - x_i$, correspondently, to compute the hash code H_f we can simply replace x_i by the value $a_i \in Z_p$ and x'_i by $1 - a_i$. This technique is implemented in the algorithm for computing hash codes **HashFromDisjoint**() shown in Figure 1.

Theorem 1 *The algorithm* **HashFromDisjoint**() *computes the hash code for a Boolean function* $f(x_1, \ldots, x_n)$ *given by a disjoint sum-of-products* F *in* $O(n \cdot |F|)$ *steps.*

The proof is available from the authors.

3 Experimental results

The purpose of experiments was to compare the time it takes **HashFromDisjoint**() to compute and compare hash codes for two functions with the time it takes to build and compare BDDs for these functions. In both cases the input functions are given by disjoint sum-of-products. Our program has been implemented in C and uses the CUDD package [13] the for the BDD manipulations. All results are reported on a Sun Ultra 60 workstation operating with a 440 MHz CPU and with 128 MB RAM main memory. Time was measured using the Unix command *time*.

Table 1 shows the results for the IWLS93 benchmark set. The original functions in this benchmark set are given by regular sum-or-products in *.pla* format. For our experiments, we first transformed all the benchmarks to disjoint sum-of-products in *.pla* format. The disjoint sum-of-products were generated using Espresso with *-Ddisjont* option [14]. Then, we verified every function against its copy with a random number of introduced faults. The faults were introduced by flipping a bit to a complemented value. Columns 2 and 3 give the number of inputs n and the number of outputs m of the benchmarks functions. Columns 4 and 5 show the number of product-terms in the original and disjoint sum-of-products (SOP), correspondently. Column 6 gives the time it took **HashFromDisjoint**() to compute and compare hash codes starting from the disjoint sum-of-products. 0.00 stands for the times smaller that 0.01 sec. Note, that the numbers shown do not include the time it took Espresso to compute the disjoint SOPs. Columns 7 shows the time for building and comparing two BDDs starting from the disjoint SOPs using CUDD BDD package [13]. For a comparison, we also show in column 8 the time for building and comparing two BDDs starting from the original SOPs. The sign ”—” stands for the cases when the BDD package aborted with a ”problem during garbage collection” error message.

Our experimental results show that, if a disjoint SOP is given on the input, then the run-time of **HashFrom Disjoint**() is always faster than the BDD time. If the size of a disjoint SOP does not exceed 5000 product-terms, then the run-time of **HashFrom Disjoint**() is even faster than the BDD-building time from the original, non-disjoint SOP. However, representing functions by disjoint sum-of-products in infeasible for larger input instances. In the next section discuss our current work towards overcoming this problem.

Table 1: Hash code computation versus BDD building time.

name	n	m	number of products		time to compute 2 hash codes from disjoint covers and compare (sec)	time to build 2 BDD from disjoint covers and compare (sec)	time to build 2 BDD from original covers and compare (sec)
			original cover	disjoint cover			
Con1	7	2	9	11	0.01	0.15	0.13
Xor5	5	1	16	16	0.00	0.09	0.12
Misex2	25	18	29	29	0.01	0.17	0.11
Squar5	5	8	32	30	0.00	0.13	0.13
Misex1	8	7	32	32	0.01	0.14	0.17
Rd53	5	3	32	32	0.00	0.12	0.16
Inc	7	9	34	34	0.00	0.17	0.15
Sqrt8	8	4	40	44	0.00	0.13	0.12
E64	65	65	65	65	0.00	0.20	0.14
5xp1	7	10	75	75	0.08	0.13	0.14
Bw	5	28	87	96	0.00	0.11	0.14
Rd73	7	3	141	141	0.01	0.19	0.13
Sao2	10	4	58	152	0.00	0.11	0.1
Clip	9	5	167	185	0.00	0.19	0.11
9sym	9	1	87	189	0.00	0.14	0.14
Duke2	22	29	87	226	0.01	0.19	0.13
Table3	14	14	175	232	0.01	0.18	0.17
Rd84	8	4	256	255	0.00	0.14	0.16
Ex5p	8	63	256	256	0.01	0.17	0.19
B12	15	9	431	302	0.01	0.12	0.15
Table5	17	15	158	315	0.01	0.19	0.15
Apex4	9	19	438	537	0.01	0.17	0.20
Ex1010	10	10	1024	810	0.01	0.16	0.14
Cps	24	109	654	945	0.08	0.28	0.20
Vg2	25	8	958	958	0.03	0.34	0.14
Ex4p	128	28	621	1004	0.09	0.33	0.24
T481	16	1	481	1547	0.02	0.21	0.12
Pdc	16	40	276	1800	0.07	0.43	0.41
Spla	16	46	2307	2194	0.09	—	—
Seq	41	35	1459	2274	0.12	0.36	0.26
Apex3	54	50	280	2331	0.15	0.48	0.12
Misex3	14	14	1848	2349	0.05	0.27	0.25
Misex3c	14	14	305	2401	0.08	0.31	0.13
Alu4	14	8	1028	3605	0.06	0.44	0.19
Apex5	117	88	1227	5029	0.63	1.10	0.33
Apex1	45	45	206	12307	0.70	—	0.28
Cordic	23	2	1206	22228	0.53	2.19	0.27

4 Conclusion

This paper considers the application of probabilistic techniques to equivalence checking. We design an algorithm which takes $O(n \cdot k)$ time to compute the hash code for an n-variable function if the function is given as a disjoint SOP with k product-terms.

A clear shortcoming of our current algorithm is the requirement that the input functions are available in the disjoint SOP format. This make it infeasible for large input instances. To avoid this problem, we are presently working on the development of a pre-processing algorithm, which takes a netlist description of the circuit to be verified and partitions it into a network of nodes, with each node representing a disjoint SOP of a reasonable size.

References

[1] R. E. Bryant, ”Graph-based algorithm for Boolean function manipulation”, *IEEE Transactions on Computers* **C-35**, pp. 677-691, 1986.

[2] Matsunaga Y., ”An efficient equivalence checker for combinational circuits” *Proc. of Design Automation Conference*, 1996.

[3] Kuehlmann A., Krohm F., ”Equivalence checking using cuts and heaps” *Proc. of Design Automation Conference*, pp. 263-268, 1997.

[4] Burch J. R., Singhal V., ”Tight integration of combinational verification methods” *Proc. of IEEE/ACM International Conference on Computer-Aided Design*, pp. 570-576, 1998.

[5] Paruthi V., Kuehlmann A., ”Equivalence checking using cuts a structural SAT-solver, BDDs and simulation” *Proc. of International Conference on Computer Design*, 2000.

[6] E. Goldberg, M. R. Parasad, R. K. Brayton, ”Using SAT for Combinational Equivalence Checking”, *IEEE/ACM Design, Automation and Test in Europe, Conference and Exhibition 2001*, pp. 114-121.

[7] J.Marques-Silva, T.Glass, ”Combinational Equivalence Checking Using Satisfiability and Recursive Learning”, *IEEE/ACM Design, Automation and Test in Europe*, pp. 145-149, 1999.

[8] D. Brand, ”Verification of large synthesized designs”, *IEEE/ACM International Conference on Computer-Aided Design*, pp. 534-537, 1993.

[9] W.Kunz, ”HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning”, *IEEE/ACM International Conference on Computer-Aided Design*, pp. 538-543, November 1993.

[10] P.Tafertshofer, A.Ganz, M.Henftling, ”A SAT-Based Implication Engine for Efficient ATPG, Equivalence Checking and Optimization of Netlists”, *IEEE/ACM International Conference on Computer-Aided Design*, pp. 648-657, 1997.

[11] J. Jain, J. Bitner, D. S. Fussell, J. A. Abraham, ”Probabilistic verification of Boolean functions”, *Formal Methods in System Design*, Kluwer Academic Publishers, **1**, pp. 63-117, 1992.

[12] E. Dubrova, H.Sack, ”Probabilistic Verification of Multiple-Valued Functions”, *30th International Symposium on Multiple-Valued Logic*, pp. 461-466, 2000.

[13] Somenzi F. *CUDD: CU Decision Diagram Package, Release 2.3.0* University of Colorado at Boulder, 1998.

[14] R.K. Brayton, G. Hachtel, C. McMullen, A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publisher, 1984.