JOURNAL OF THE SCHOOL OF ENGINEERING, THE UNIVERSITY OF TOKYO
Vol. XLVII (2000)

Formal Verification Based on Recurrence Equations and Equivalence Checking

By

Masahiro FUJITA*

(Received July 12, 2000)

Abstract

Recently there have been sigfinicant progress in technologies for formally comparing two structurally similar large logic circuits [2, 14, 13]. Circuits having more than 10,000 gates, whose BDD cannot be built, have been verified in several minutes. However, verification of arithmetic circuits with respect to their specification is still a hard problem. As shown in [16] some arithmetic circuits, such as multipliers, square function, cube functions, etc., must satisfy recurrence equations, such as f(x + 1; y) = f(x; y) + y where f(x; y) = xy. Those equations can be used for verification. In this paper, we use such recurrence equations in order to drive Boolean comparison problems of structurally similar circuits. That is, left hand sides and right hand sides of equations are realized as separated circuits and then compared. Using the recurrence equations properly, these circuits have many internal equivalent signals and many implications among signals, by which Boolean comparison programs, such as [2, 14, 13], can work very effectively. Using the proposed method, 16-bit multipliers, such as C6288 of ISCAS85 benchmark circuits, are verified within 12 minutes.

Also, we discuss about a way to use the proposed verification method based on recurrence equations for the verification of more general programs. That is, we deal with programs instead of arithmetic circuits as targets of verifications. Equations are driven from the properties to be satisfied by the programs, such as the semantics of programs. By utilizing these appropriate equations, we can check consitency on the programs and their processing. We pick up an HDL (hardware description language) as an example and shows a way to verify that processing of the HDL descriptions is consistent to their definitions.

1. Introduction

Formal verification techniques have been paid much attention recently. There have been lots of works on formal hardware verification [11, 10], and among them, Binary Decision Diagram (BDD) [3] based verification techniques, such as [5, 8, 17, 6, 15], have given successful results for practical designs.

However, BDD may not work well for arithmetic circuits, such as multipliers. Therefore, several extensions are made on BDD, such as, BMD [4], HDD [7], OKFDD [9], etc. Although originally word-level verification is necessary in order to use BMD, by using the technique shown in [12] which compute BMD from outputs to inputs instead of inputs to outputs, we can use BMD directly to verify arithmetic circuits, such as multipliers. However, if there are design errors (bugs) in the circuits, BMD can easily

* Department of Electronic Engineering

blow up and the verification program may not terminate, since those circuits represent different logic functions from multipliers which can have exponential sizes of BMD. This depends on each error but we actually observed this BMD explosion by randomly inserting logical errors to the multiplier circuits and generating BMDs.

In this paper, we show another approach to attack the verification of arithmetic circuits. Instead of directly generating BDD (or its extensions) from given circuits, we create circuits based on the recurrence equations that must be satisfied by the circuits. This idea was originally proposed by Ochi [16]. For example a recurrence equation for multipliers is:

$$f(x+1, y) = f(x, y) + y \text{ where } f(x, y) = xy \text{ and } x, y \text{ are inputs}$$

Any circuits which satisfy the above recurrence equation are multipliers*. He used recurrence equations to verify arithmetic circuits by first generating BDD from the circuits and then checked if that BDD satisfies the required recurrence equations. Clearly this method has a drawback that we have to build BDD for the circuits first, which is often impossible for large arithmetic circuits, especially for multipliers.

On the other hand, recurrence equations, such as shown above (for multipliers), may indicate a comparison problem of two circuits (or Boolean functions). That is, checking recurrence equation means comparing the left hand side and right hand side of the equation. Basically we can use Boolean comparison techniques for such equivalence checking.

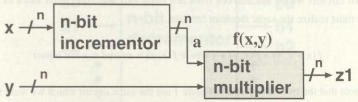
Recently there have been much progress in technologies for Boolean comparison of similar circuits. Here "similar" means that we can find many logical relationships, such as equivalences or implications, among the internal signals in the two circuits to be compared. In a practical design environment, designers want to check the equivalence of the two circuits which are very structurally similar. For example, it is often the case to check the equivalence between unoptimized circuits and manually optimized circuits. For such cases, there are lots of relationships among the internal signals in the two circuits. By utilizing these relationships, methods like [2, 14, 13] can verify much larger circuits than the circuits which can be verified directly by BDDs. 100,000 gates or larger circuits can be verified within practical time.

Basically recurrence equations suggest two structurally similar circuits (left hand side and right hand side). Here we propose a new verification method for arithmetic circuits based on recurrence equations. We first generate two circuits which correspond to left hand side and right hand side of the recurrence equations. Then Boolean comparison program for similar circuits is applied to those circuits. Please note that we need only one circuit which should be verified. The two circuits to be compared are generated from that circuit by adding appropriate extra circuits, such as adders, incrementors, etc. Also note that we do not need specification in Boolean functions. Specification is fully described in the recurrence equations that we are using to generate two circuits.

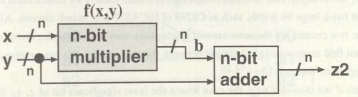
By appropriately using case splitting on values of input variables, we can verify 16-bit multipliers, such as C6288 of ISCAS benchmark circuits, in less than 12 minutes on a SUN Sparc20 workstation. Moreover, different from the method in [12], the proposed method can finish verification in similar

* Circuits must also satisfy boundary conditions, such as, $f(0, y) = 0$, which can be checked rather easily.

time, even if the circuits are not correct as shown in section 3.



(a) Circuit corresponding to f(x+1,y)



(b) Circuit correponding to f(x,y)+y

Figure 1: Circuit realization of the recurrence equation for multipliers

In the next section, we introduce our verification method. Then section 3 gives experimental results. Although we discuss only about multipliers in this paper, clearly the proposed method can be applied to many arithmetic functions which have proper recurrence equations, such as, square functions, cube functions, etc.

In section 4, we discuss about a way to use the proposed verification method based on recurrence equations on the verification of more general programs. That is, we deal with programs instead of arithmetic circuits as targets of verifications. By using appropriate recurrence equations driven from the programs, we can check important propoerties that must be satisfied by the programs in a similar way to the verification of the arithmetic circuits. Finally section 5 gives our concluding remarks.

2. Verification algorithm

In this section we introduce our verification method. For simplicity, we use multipliers as examples all the time in this paper, although we can verify many other arithmetic circuits which have proper recurrence equations, such as, square functions, cube functions, etc. As long as there are proper recurrence equations, we can verify any circuits, including random circuits (assuming such recurrence equations are given)*.

* In some sense, we can consider the proposed method is a kind of self-checking methods proposed in [1]. What we are doing in this paper can be described in the following way: by appropriatly using recurrence equations (self checking properties), we are reducing verification problems on arithmetic circuits into Boolean comparison problem for similar circuits. Of course, if the reduced Boolean comparison problems are too large, we can use random simulation based checking just like in [1].

The basic idea for multipliers is illustrated in Figure 1. Since multipliers satisfy the following equation, two circuits which are deriven from left hand side and right hand sides of the equation respectively must realize the same Boolean function.

$$f(x+1, y) = f(x, y) + y \text{ where } f(x, y) = xy \text{ and } x, y \text{ are inputs}$$

Please note that the two multipliers in Figure 1 are the same circuit which we want to verify. We are assuming here that an incrementor and an adder are given and they are guaranteed to be correct.

Please also note that the above equation together with boundary condition for $x = 0$ completely specify the function and that must be a multiplier.

By comparing the two circuits shown in Figure 1, we can formally verify multipliers*. Unfortunately this is not an easy Boolean comparison problem. Clearly we cannot build BDD for each circuit, if that has a large bit width, such as C6288 of ISCAS85 benchmark circuits. Although large portion of the two circuits are the same circuits (multiplier), they are not similar circuits in the sense that we cannot find many equivalent signals between the two circuits. We cannot directly apply the Boolean comparison methods like [2, 14, 13].

However, if we consider only the case where the least significant bit of x, x_0, is 0, then the incrementor becomes just an inverter as shown in Figure 2**. Thus the two circuits become like the ones shown in Figures 3. Now there are many equivalent signals between the two circuits, since most inputs are common and large portion of the circuits are the same. In fact there are a lot of functional relationships among internal signals of the two circuits.

By this case splitting, we can verify multipliers when $x_0 = 0$. Then how about the cases when $x_0 = 1$? We can proceed with the same idea: further case splitting with $x_1, x_2, \ldots$ For example, if $x_0 = 1$ *and* $x_1 = 0$, then the incrementor becomes just two inverters as shown in Figure 4. Again the two circuits generated are similar as shown in Figure 5.

The next splitting case is $x_0 = 1$, $x_1 = 1$, and $x_2 = 0$ which needs three inverters for x_0, x_1, and x_2. This case splitting process can be continued until we reach the case where $x_0 = 1, x_1 = 1, x_2 = 1, \ldots, x_{n-2} = 1$, and $x_{n-1} = 0$.

What we are doing here is just to check the equation:

$$f(x+1, y) = f(x, y) + y \text{ where } f(x, y) = xy \text{ and } x, y \text{ are inputs}$$

by case splitting the values of x_i.

As we have larger number of inverters, the two circuits become less similar. However, as we have larger number of inverters, we can fix the values of x_i more. That is, in the case of Figure 4, since this is the case where $x_0 = 1$ and $x_1 = 0$, we can fix the values of x_0 and x_1. There are trade-offs in terms of difficulty and the most difficult case happens when there are two inverters as shown in Figure 4 according to our experiments for C6288 in the next section.

* In this paper, we assume that extra circuits, such as incrementor, adder, subtracter, etc., are guaranteed to be correct. Or those should be verified first.

** If $x_0 = 0$, then increment does not affect the values of $x_1, x_2, \ldots, x_{n-1}$.

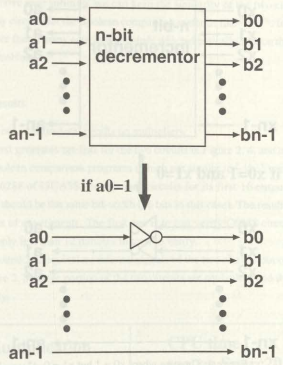


Figure 2: If the least significant bit is 0, incrementor becomes just an inverter

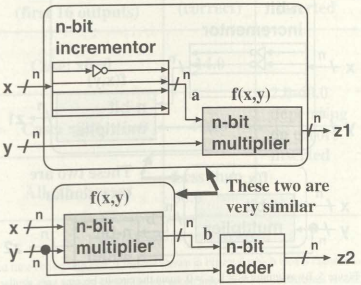


Figure 3: By assuming x0 = 0, the circuits become very similar

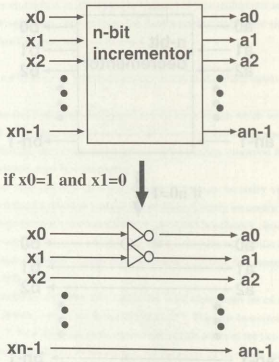


Figure 4: The case where x0 = 1 but x1 = 0

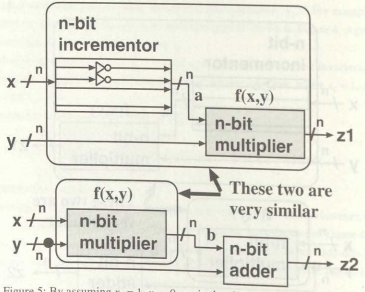


Figure 5: By assuming $x_0 = 1$, $x_1 = 0$, again the circuits become very similar

By using the above case splitting, we can keep the similarity of the two circuits. These circuits should be rather easy circuits for the Boolean comparison methods like [2, 14, 13]. In fact, as shown in the next section, we have found many equivalent signals which drastically reduce the verification time or complexity of the problem.

3. Experimental results

Here we show our experimental results on multipliers.

Our program first generates net-lists for the two circuits in Figure 2, 4, and others* from the given multipliers. Our Boolean comparison programs (which are similar to [2, 14, 13]) are applied to them.

We verified C6288 of ISCAS85 benchmark circuits for its first 16 outputs, since as shown in Figure 1, all values should be the same bit-width (16 bits in this case). The results are shown in Figure 6. We did two types of experiments. The first one is to just verify C6288 circuit, which is a correct multiplier. It took only less than 12 minutes in total to verify.

The program found 342 equivalent internal signals of the two circuits out of 360 internal signals for the case of Figure 2. So large portion of the two circuits are equivalent and that is why verification can finish so quickly.

Multiplication Circuit	CPU time sec. on Sparc20	
C6288 (first 16 outputs)	Original (correct)	Error inserted
Case: x0=0	14.0	2.0-60.0 depending on errors inserted
Case: x0=1, x1=0	496.0	
All other cases	Less than 60.0	

Figure 6: Results for 16-bit multipliers

The most time consuming case is the one shown in Figure 4 which took 8 minutes to finish. This is the case where the two circuits are similar but not so much and their circuit sizes are still large (only small number of x_i have fixed value). All the other cases are less than one minute.

* In the case of 16-bit multipliers, there are 16 cases in total. But some of them are trivial, since most of x_i are constants.

Second experiment we did is to try to verify incorrect multipliers (verification fails) by intentionally inserting errors into C6288 (changing function of a gate, etc.*). Depending on changes, it took less than one minute (sometime in a couple of seconds) to prove that the circuit is not a multiplier. Depending errors, the cases where verification fails are different, but mostly verification fails in multiple cases. Again this is extremely fast. Please note that the method in [12] may not work well for incorrect circuits.

4. Application of verification method based on recurrence equations to the verification of programs

In this section, we show how to check the consistency on the translation process for each statement of an HDL (hardware description language). That is, we deal with programs instead of arithmetic circuits as targets of verifications. Equations are deriven from the properties to be satisfied by the programs, such as the semantics of programs. By utilizing these appropriate equations, we can check consitency on the programs and their processing. We pick up an HDL (hardware description language) as an example and shows a way to verify that processing of the HDL descriptions is consistent to their definitions.

Arithmetic operations, sub-module instantiation statements, and conditional statements, like if-then-else and case statements, are most common statements in HDLs. There can be bugs in the programs that translate HDL descriptions and generate state transition relations. The generated transition relations are processed by verification procedures based on, for example, model checking [5], and bread-first traversal of finite state machines [8]. Here assuming the correctness of such verification procedures, we discuss a way to check the correctness of translation procedures from HDL descriptions into state transition relations. This is a common verification problem in programs that translate from one form to another and has many applications.

As for the translation of arithmetic operations, we can directly use the verification method introduced in the previous sections. Therefore, here we concentrate on sub-module instantiation statements and conditional statements.

HDL translation programs accept an HDL description and generate its corresponding state transition relations, $R(x, y)$. There can be multiple finite state machines which are working in parallel. We assume that each finite state machines, FSM_i, is independently translated into its corresponding state transition relation, $R_i(x, y)$, and the state transition relation for the entire system is given as

$$R(x, y) \equiv \bigwedge_i R_i(x, y)$$

In the following, we show our idea by showing translation examples from a couple of HDL statements into state transition relations. Other HDL statements can also be similarly processed.

First of all sub-module instantiation statements can be checked in the following way. First we randomly generate two statements and use them in two different modules, like:

$FSM_1(x, y)$

$y \leftarrow \overline{x}$;

* Even if we make many changes in the circuit, situations are the same. The two circuits we generate according to Figure 1 are very similar.

$FSM_2(x, y)$

$y \leftarrow x + 1$;

Then we add a parent module which uses the above two modules as sub-modules, like:

$FSM_{12}(x_1, x_2, y_1, y_2)$

SUB MODULE M1: FSM1;

SUB MODULE M2: FSM2;

M1(x_1, y_1);

M2(x_2, y_2);

We prepare another module which does not use sub-module instantiation statements. Instead it expands descriptions for sub-modules, like:

$FSM'_{12}(x_1, x_2, y_1, y_2)$

$y_1 \leftarrow \overline{x}_1$;

$y_2 \leftarrow x_1 + 1$;

Then we check whether FSM_{12} and FSM'_{12} are completely equivalent or not by using formal verification procedures, such as the one in [5]. Assuming the correctness of the verification procedures, if FSM_{12} and FSM'_{12} are proved not to be equal, we can say that the translations are not correct, most possibly because of the translation of sub-module instantiate statements. This example is a very simple one but shows our basic ideas.

Now we show another verification, i.e., translations of conditional statements, particularly case statements. This has lots of practical value, since we really found misunderstandings by using the the proposed techniques.

An example of case statement looks like:

CASE

$case_1$: $statement_1$;

$case_2$: $statement_2$;

otherwise : $statement_0$;

END CASE

This is logically equivalent to:

$$(case_1 \rightarrow statement_1) \wedge (case_2 \rightarrow statement_2) \wedge (\overline{case_1} \wedge \overline{case_2} \rightarrow statement_0)$$

$statement_1$, $statement_2$, and $statement_0$ can be any statement including another conditional statement. Here we want to make sure that the verification system being tested translates case statements into state transition relations exactly in the above way. For that purpose, we restrict $statement_1$; $statement_1$, and $statement_0$ to be register transfer statements, such as,

$$y_i \leftarrow x_i$$

We randomly generate two conditions ($case_1$ and $case_2$ in the above) and use them in the following

two HDL descriptions. The first one uses two finite state machines which are working in parallel.

$FSM_1(x, y)$
CASE
$case_1$: $statement_1$;
otherwise : $statement_2$;
END CASE

$FSM_2(x, y)$
CASE
$case_2$: $statement_1$;
otherwise : $statement_2$;
END CASE

From this description we get the corresponding state transition relations in the following way.

$$\begin{aligned}
R_1(x, y) &= (case_1 \rightarrow statement_1) \wedge (\overline{case_1} \rightarrow statement_2) \\
&= (\overline{case_1} \vee statement_1) \wedge (case_1 \vee statement_2) \\
R_2(x, y) &= (case_2 \rightarrow statement_1) \wedge (\overline{case_2} \rightarrow statement_2) \\
&= (\overline{case_2} \vee statement_1) \wedge (case_2 \vee statement_2) \\
R_{12}(x, y) &= R_1(x, y) \wedge R_2(x, y) \\
&= (\overline{case_1} \vee statement_1) \wedge (case_1 \vee statement_2) \\
&\quad \wedge (\overline{case_2} \vee statement_1) \wedge (case_2 \vee statement_2) \\
&= ((\overline{case_1} \wedge case_2) \vee statement_1) \\
&\quad \wedge ((case_1 \wedge case_2) \vee statement_2) \\
&= ((case_1 \vee case_2) \rightarrow statement_1) \\
&\quad \wedge ((\overline{case_1} \vee \overline{case_2}) \rightarrow statement_2)
\end{aligned}$$

From the last equation, we get another way of describing the behavior, FSM'_{12}, which equals to FSM_{12} in the following way:

$FSM'_{12}(x, y)$
CASE
$case_1$. $case_2$: $statement_1$;
$\overline{case_1}$. $\overline{case_2}$: $statement_2$;
END CASE

When we applied the above proposed verification techniques to a verification program which accepts its own HDL, we have found that we misunderstood the semantics of case statement in a verification program. It happened in the following test cases, which is a little bit more complex than the above.

$FSM_1(x, y)$
CASE
$case_1$: $statement_1$;

$case_0$: $statement_2$;
END CASE

$FSM_2(xy)$
CASE
$case_2$: $statement_1$;
$case_0$: $statement_2$;
END CASE

From this description we get the corresponding state transition relations in the same way as above:

$$\begin{aligned}
R_1(x, y) &= (case_1 \rightarrow statement_1) \wedge (case_0 \rightarrow statement_2) \\
&= (\overline{case_1} \vee statement_1) \wedge (\overline{case_0} \vee statement_2) \\
R_2(x, y) &= (case_2 \rightarrow statement_1) \wedge (case_0 \rightarrow statement_2) \\
&= (\overline{case_2} \vee statement_1) \wedge (\overline{case_0} \vee statement_2) \\
R_{12}(x, y) &= R_1(x, y) \wedge R_2(x, y) \\
&= (\overline{case_1} \vee statement_1) \wedge (\overline{case_0} \vee statement_2) \\
&\quad \wedge (\overline{case_2} \vee statement_1) \wedge (\overline{case_0} \vee statement_2) \\
&= ((\overline{case_1} \wedge case_2) \,.\, statement_1) \\
&\quad \wedge (\overline{case_0} \vee statement_2) \\
&= ((case_1 \vee case_2) \rightarrow statement_1) \\
&\quad \wedge (case_0 \rightarrow statement_2)
\end{aligned}$$

From the last equation, we get another way of describing the behavior, FSM'_{12}, which equals to FSM_{12} in the same way as the previous example:

$FSM'_{12}(x, y)$
CASE
$case_1 \vee case_2$: $statement_1$;
$case_0$: $statement_2$;
END CASE

However, initially we misunderstood the semantics of case statements in the following way:

$$(case_1 \rightarrow statement_1) \wedge (\overline{case_1} \wedge case_2 \rightarrow statement_2) \wedge (\overline{case_1} \wedge \overline{case_2} \rightarrow statement_0)$$

Here the conditions of case statements are automatically made disjoint by adding extra conditions for the complements of previous conditions, like $(\overline{case_1} \wedge case_2 \rightarrow statement_2)$. Although this is a typical meanings for case statements in sequential programs, this is not the same for the verification systems we tested. Since it must be able to handle non-deterministic finite state machines, some conditions for case statements can be overlapped. So, the above understanding of the case statement is incorrect. This has been revealed by the above verification.

Also, we have found another misunderstanding (in this case, we may be able to call it a kind of

bug). That is related to otherwise statement in case statements, and it was found when by chance *otherwise* statements are generated. In a formal verification program which uses BDD as an internal representation of transition relations, states are encoded using binary variables. For example, if there are three states, s_1; s_2, and s_3, two binary variables, x1 and x_2, are used to encode the states, e.g., $s_1 = \overline{x_1} \wedge \overline{x_2}$, $s_2 = \overline{x_1} \wedge x_2$, $s_3 = x_1 \wedge \overline{x_2}$. This means that the pattern, $x_1 \wedge x_2$, is not used. This causes a trouble. Our verification shows that the conditions for *otherwise* statements do not take this fact into count. The unused pattern, $x_1 \wedge x_2$ seems to appear in the conditions which are generated from otherwise statements.

These experiences clearly show that the proposed verification techniques have lots of practical values.

5. Conclusions

We have shown a verification method for arithmetic circuits. We also demonstrated that C6288 can be verified in less than 12 minutes. Even if the circuits are not correct (there is a bug in the circuits), verification time remains similar or less. Also, different from BMD or HDD based methods, we do not need another BDD package, such as, BMD package. We can use existing BDD packages or Boolean comparison programs to verify arithmetic circuits.

Although in this paper we only discussed about combinational circuits, the proposed techniques can be applied to sequential circuits by deriving appropriate recurrence equations. Surely this is one of our future research topics.

Also, the proposed method can be considered to be a kind of self-checking methods proposed in [1]. What we are doing here can be described in the following way: by appropriately using recurrence equations (self checking properties), we are reducing verification problems into Boolean comparison problem for similar circuits. Of course, if the reduced Boolean comparison problems are too large, we can use random simulation based checking just like in [1]. A preliminary idea on this direcdtion, which is a way to use the proposed verification method based on recurrence equations on the verification of general programs, is also discussed. Currently we are planning to explore this area and study on extensions of the proposed method.

References

1) M. Blum, M. Luby, and R. Rubinfeld. "self-testing/correctig with application to numerical problems". In*Proc. of 22nd ACM Theory of Computing*, pages 73-83, 1990.

2) D. Brand. "verification of large synthesized designs". In *Proc. of ICCAD*, pages 534-537, Nov. 1993.

3) R.E. Bryant. "graph-based algorithms for boolean function manipulation". *IEEE Trans. Computer*, C35(8):667-691, Aug. 1986.

4) R.E. Bryant and Y.-A. Chen. "verification of arithmetic functions with binary moment diagrams". In *Proc. of 32nd DAC*, Jun. 1995.

5) J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. "symbolic model checking: 10^{20} statesand beyond". In *Proc. of the Fifth Anual IEEE Symposium on Logic in Computer Science*, Jun. 1990.

6) H. Cho, G. Hachtel, S-W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi. "atpg aspects of fsm verification". In *Proc. IEEE Int. Conf. on Computer-Aided Design (ICCAD-90)*, pages 134-137, Nov. 1990.

7) E.M. Clarke, M. Fujita, and Z. Zhao. "hybrid decision diagrams - overcoming the limitations pf mtbdds andbmds". In *Proc. of ICCAD*, pages 159-163, Nov. 1995.

8) O. Coudert and J.C. Madre. "a unified framework for the formal verification of sequential circuits". In *Proc. IEEE Int. Conf. on Computer-Aided Design (ICCAD-90)*, pages 126-129, Nov. 1990.

9) R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. Perkowski. "efficient representation and manipulation of switching functions based on ordered kronecker functional decision diagrams". In *Proc. of 31st DAC*, Jun. 1994.

10) M. Fujita. "rtl design verification by making use of datapath information". In *Proc. of ICCD-92*, pages592-597, Oct. 1992.

11) A. Gupta. "formal hardware verification methods: a survey". *Formal Methods in System Design*, Vol. 1(2/3), Oct. 1992.

12) K. Hamaguchi, A. Morita, and S. Yajima. "efficient construction of binary moment diagrams for verifyingarithmetic circuits". In *Proc. of ICCAD*, pages 78-82, Nov. 1995.

13) J. Jain, R. Mukherjee, and M. Fujita. "advanced verification techniques based on learning". In *Proc. of 32ndDAC*, pages 420-426, Jun. 1995.

14) W. Kunz. "hannibal: An efficient tool for logic verification based on recursive learning". In *Proc. of ICCAD*, pages 538-543, Nov. 1993.

15) K.L. McMillan. "symbolic model checking: An approach to the state explosion problem". Technical Report CMU-CS-92-131, Carnegie Mellon University, May 1992.

16) H. Ochi and S. Yajima. "formal design verification of combinational circuits specified by recurrence equations". In *Proc. of SASIMI '95*, pages 101-105, Aug. 1995.

17) H. Touati, H. Savoj, B. Lin, R.K. Brayton, and A. Sangiovanni-Vincentelli. "implicit state enumeration offinite state machines using bdds". In *Proc. IEEE Int. Conf. on Computer-Aided Design (ICCAD-90)*, pages 130-133, Nov. 1990.