Formal Hardware Verification with BDDs: An Introduction *

Alan J. Hu
Department of Computer Science
University of British Columbia

Abstract—**This paper is a brief introduction to the main paradigms for using BDDs in formal hardware verification. The paper addresses two audiences: for people doing theoretical BDD research, the paper gives a glimpse of the problems in the main application area, and for people building hardware, the paper gives a peek under the hood of the formal verification technologies that are rapidly gaining industrial importance. Topics described include combinational equivalence checking, symbolic simulation, sequential equivalence checking, model checking, and symbolic trajectory evaluation.**

I. Introduction

The current interest in BDDs from the theoretical computer science community has largely been motivated by the practical importance and success of BDDs in formal hardware verification. Conversely, the growing industrial interest in formal hardware verification has largely been inspired by the effectiveness of BDD-based techniques in finding real bugs in practical, large-scale designs. This paper attempts to address both of these communities: on one hand explaining to people building hardware what the main BDD-based formal verification techniques are and what kinds of problems they can solve, and on the other hand, explaining to people doing research on BDDs what the challenges are in one of the main application areas.

Most of this paper describes the main paradigms for using BDDs for formal hardware verification. We will briefly survey the basic algorithms for using BDDs in combinational equivalence, symbolic simulation, sequential equivalence, model checking, and symbolic trajectory evaluation. The remainder of the paper describes some issues and directions of current research to expand the size and complexity of designs that can be verified.

Before proceeding, let me address three basic questions: What is formal verification? Why has there been a surge of interest in formal hardware verification? And what is a BDD?

A. *What Is Formal Verification?*

Formal verification means **proving** that a **property** holds of a **model** of a design. The bold-faced words are the key ideas. The promise of verification is **proving** in the sense of mathematical proof, in contrast to conventional simulation and test, which can tell us only that nothing went wrong on the specific cases we tried. (Of course, exhaustively trying every possible execution of a system is a valid proof. We can view formal verification as giving the effect of exhaustive simulation.) However, specifying the **property** to prove and creating an accurate **model** of the design are difficult problems. One can ponder endlessly the philosophical impossibility of proving a system correct: Is the spec correct? Is the model accurate? Is the verifier correct? Is the computer we used to run the verifier correct? Etc. (Cohn [10] gives an insightful and readable analysis of the fundamental obstacles to proving a hardware design correct.) In practice, we can choose the property to be "conforms exactly to a golden reference model which we assume is correct," or else we can simply prove easy-to-state properties that matter to us (e.g., absence of deadlock, interface lines follow a handshake protocol, etc.). Ideally, the model we verify is as close to the actual hardware as possible (e.g., a circuit extracted from the VLSI layout), but for complicated designs, an abstracted model is usually needed to simplify the verification process.

Although formal verification of all kinds of properties — timing, performance, reliability, etc. — is conceivable, most formal hardware verification research has focused on verifying functional correctness, so we will restrict our attention to that here.

B. *Why Formal Hardware Verification?*

People have researched formal verification of computer hardware and software for decades. Traditionally, the emphasis had been on approaching the ideal of proving a system correct. The verification methods (typically a mathematically expressive but computationally undecidable logic with support from a semi-automated theorem prover) require considerable time and expertise to verify even fairly simple systems. As a result, practical application has been limited to a few domains, such as security and safety-critical systems, where ethical or legal requirements demand the highest assurance of correctness, regardless of cost. For normal hardware design projects, hiring or training formal verification experts, delaying a product launch to allow time for formal verification, and reducing product performance or features to simplify formal analysis are all economically unacceptable.

A major factor in the current industrial interest in formal verification is a different emphasis for formal verification that explicitly recognizes economic demands. (The other main factors are the new verification techniques that support this emphasis, and the high design complexity and short design cycles that are straining current validation methods.) Basically, bugs cost money — especially the hard-to-find bugs that surface late in the design cycle, that force an extra spin of silicon, that delay a product launch, or that require a massive product recall. Any technique that finds these bugs earlier is enormously valuable. So, instead of trying to certify correctness, formal verification is used as a powerful de-

*This work was supported in part by a UBC new faculty start-up grant and an NSERC Research Grant.

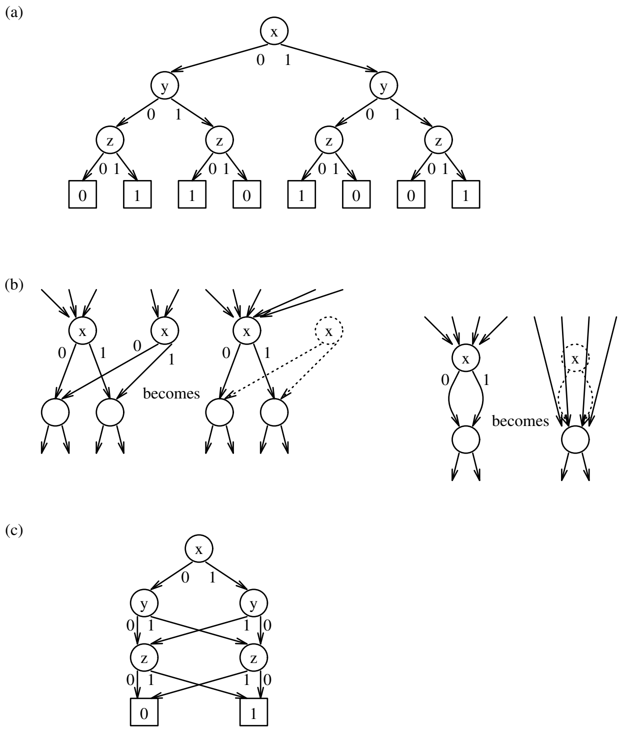

Figure 1: Creating the BDD for $(x \oplus y \oplus z)$

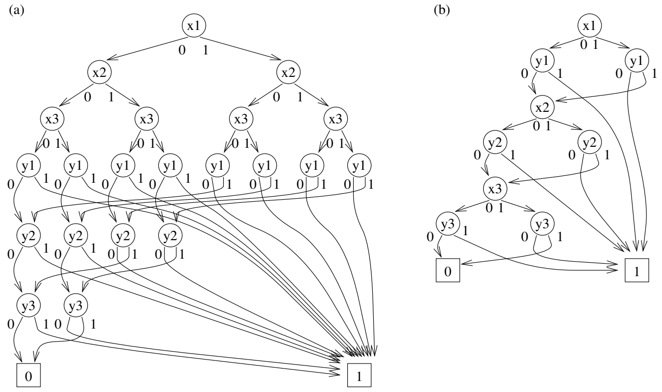

Figure 2: Two Different Variable Orders for the Same Function

bugging tool. If the time and effort invested in formal verification is less than the time and effort saved by uncovering difficult bugs earlier, then formal verification is a win, regardless of whether or not we can make any claims about proving the system correct.

This cost-benefit comparison favors formal verification techniques that are automatic and easy-to-use, even if they lack theoretical expressiveness. Not coincidentally, this practical, debugging emphasis for formal verification developed in parallel with new formal verification algorithms that offer far greater automation than had previously been possible. These algorithms are the BDD-based algorithms that we will explore in this paper.

C. *What's a BDD?*

"BDD" stands for "binary decision diagram." A BDD is just a data structure for representing a Boolean function. Bryant [3] introduced the BDD in its current form, although the general ideas have been around for quite some time (e.g., as branching programs in the theoretical computer science literature). Conceptually, we can construct the BDD for a Boolean function as follows. First, build a decision tree for the desired function, obeying the restrictions that along any path from root to leaf, no variable appears more than once, and that along every path from root to leaf, the variables always appear in the same order (Figure 1(a)). Next, apply the following two reduction rules as much as possible: (1) merge any duplicate (same label and same children) nodes, and (2) if both child pointers of a node point to the same child, delete the node because it is redundant (Figure 1(b)). The resulting directed, acyclic graph is the BDD for the function (Figure 1(c)). In practice, BDDs are generated and manipulated in the fully reduced form, without ever building the decision tree. In a typical implementation, all BDDs in use by an application are merged as much as possible to maximize node sharing, so a function is represented by a pointer to its root node. For example, in Figure 1(c), the function $(x \oplus y \oplus z)$ is represented by a pointer to the top node, whereas the function $(y \oplus z)$ is represented by just a pointer to the leftmost node labeled y, rather than by copies of the nodes.[1]

BDDs have several useful properties. First, many common function have small BDDs. For example, generalizing the pattern in Figure 1(c), we see that the BDD for the parity of n variables requires $2n - 1$ nodes, whereas parity requires exponential size using, for instance, sum-of-products form. In addition, BDDs are easy to manipulate. Efficient algorithms exist for all the usual Boolean operations (AND, OR, NOT, etc.) as well as other useful operations. Finally, once we fix the order in which the variables appear, a BDD is a canonical representation for a Boolean function, i.e., every distinct Boolean function has exactly one unique BDD representation. Thus, comparing Boolean functions becomes just a pointer comparison.

Choosing a good variable order is important. For example, suppose we wish to build a BDD for the function $(x_1 \oplus y_1) \vee (x_2 \oplus y_2) \vee (x_3 \oplus y_3)$. Figure 2 shows two BDDs for this function using two different variable orders. In general, the choice of variable order can make the difference between a linear size BDD and an exponential one.

Bryant [5] provides a detailed exposition on BDDs and surveys some applications and variations.

II. Fundamental Algorithms

Let's now examine the basic algorithms for formal hardware verification using BDDs.

A. *Combinational Equivalence*

The most obvious application of BDDs is to check the equivalence of two combinational circuits.[2] For example, we may want to verify that optimization or logic synthesis was done correctly by comparing the circuit before and after. The basic algorithm is

[1] In this paper, I will use $\oplus$ for exclusive-OR, $\equiv$ for equivalence or exclusive-NOR, $\wedge$ for AND, $\vee$ for OR, and $\neg$ for NOT.

[2] A combinational circuit is a digital circuit without state-holding elements or feedback loops, so the output is a function of the current input. A circuit with state-holding elements is called a sequential circuit.

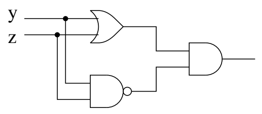

Figure 3: A Simple Example: Is This XOR?

for each circuit, to build the BDDs for the outputs in terms of the primary inputs. Since BDDs are a canonical representation, the two combinational circuits implement the same function if and only if they have the same BDD.

For example, let's consider verifying that the circuit in Figure 3 implements exclusive-OR. First, label the primary inputs with the BDDs for the variables y and z. Next, build the BDD for each gate output as a function of its inputs — labeling the OR gate with the BDD for $(y \vee z)$, the NAND gate with the BDD for $\neg(y \wedge z)$, and the AND gate with $(y \vee z) \wedge \neg(y \wedge z)$. For the specification circuit, we build the BDD for $(y \oplus z)$. Since these two expressions give the same Boolean function, they have the same BDD, which verifies that the circuit is indeed an exclusive-OR.

In practice, this approach is limited by the size of the BDDs generated, which is highly sensitive to the function being verified and the variable order used. For pathological examples like multipliers, even 16-bits is too big to handle. Typically, circuits with up to a few hundred primary inputs can often be verified. For larger circuits, more sophisticated methods are needed.

B. Symbolic Simulation

Symbolic simulation [4] is a combination of the preceding ideas with conventional logic simulation. The advantage of a conventional logic simulator is accuracy. Detailed timing models, hazards, and oscillatory behavior can all be simulated. The disadvantage of a conventional logic simulator is that only one simulation vector can be run at a time. In Figure 3, we would have had to run four simulations with the inputs equal to 00, 01, 10, and 11 to verify the circuit. A circuit with 20 inputs would have required over a million runs. Symbolic simulation adds two innovations to conventional logic simulation that give the effect of running large numbers of simulation vectors simultaneously.

The first innovation is a third logic value X that represents an unknown value. This value is propagated through the circuit just as the 0 and 1 logic values are, although the X is always treated conservatively. For example, $0 \vee$ X is X, but $1 \vee$ X is 1, since 1 is a controlling value for OR. Setting an input to X gives the effect of simulating the circuit for both the case where the input was 0 and the case where the input was 1, thereby cutting in half the number of simulation runs required. However, the X value loses information. In Figure 3, setting one or both inputs to X yields an X at the output, a useless result for verification.

The more important innovation is the introduction of symbolic values, which avoids the information loss from using X values. The basic idea is to set an input to a symbolic value that can be either 0 or 1, rather than to a constant like 0, 1, or X. Alternatively, we can think of the symbolic value as remembering whether we assigned a 0 or 1 to a given input. Returning to Figure 3, suppose we set primary input y to 1 and primary input z to the symbolic value a. The symbolic simulator would then calculate that the OR gate will settle to 1 (since 1 OR anything is 1), that the NAND gate will settle to $\neg a$, and that the AND gate will settle to $\neg a$. Thus, we've effectively run two simulation vectors (yz equal to 10 and 11) at once, computing the output as a function of the symbolic values. To implement this idea, a conventional logic simulator is modified to use BDDs to represent the values on wires as a function of the symbolic values.

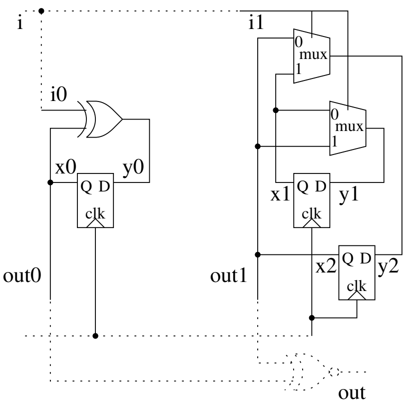

Figure 4: Comparing Two State Machines

In practice, the user must trade off using explicit 0s and 1s, the X value, and symbolic values. Setting an input to an explicit value gives conventional logic simulation. Setting an input to X halves the required number of simulation runs, but loses information so the simulation result might not be useful. Setting an input to a symbolic value halves the required number of simulation runs and does not lose information, but makes the BDDs representing the values on the wires larger. Too many symbolic values will make these BDDs too large to build.

C. Sequential Equivalence

Although symbolic simulation can be applied to sequential circuits as well as to combinational circuits, we would often like to reason about sequential circuits as finite state machines, rather than as just a bunch of gates. (This is analogous to the difference between cycle-based and event-driven logic simulation.) A typical application would be comparing that two state machines have identical behavior, in order to verify the correctness of logic optimization, register retiming, state re-encoding, etc.

The problem of comparing two state machines can be converted into the problem of finding all of the reachable states of a state machine. Given two state machines to compare, tie the input lines together, send the outputs to a comparator, and clock the two machines together in lockstep. This combination is just another, bigger state machine. The original two machines have identical behavior if and only if the new machine indicates the outputs are equal for all reachable states. For example, consider the simple circuits in Figure 4. We have two small state machines: one with input i_0, latch x_0, and output out_0; the other with input i_1, latches x_1 and x_2, and output out_1. To compare the two machines, we add the dotted lines, creating a new machine with input i, latches x_0, x_1, and x_2, and output *out*.

Computing the set of reachable states using BDDs requires three basic ideas: representing sets of states using BDDs, computing images, and the reachability iteration.

The first idea is to represent sets of states using BDDs. So far, we've been using BDDs to represent the logic function computed by a circuit. Now, we're going to use BDDs in a different manner. Basically, we can think of a BDD as representing a set of truth assignments: if the function the BDD represents is true for a given truth assignment, that assignment is in the set; if the function is false, that assignment is not in the set. For example, if we consider three Boolean variables x_0, x_1, and x_2, the BDD for the function $x_0 \wedge x_1 \wedge \neg x_2$ represents the set containing only one truth assignment $\{110\}$; the BDD for $x_0 \vee x_1$ represents the set of six truth assignments $\{100, 101, 110, 111, 010, 011\}$, and the BDD for 1 (the Boolean value True) represents the set of all eight truth assignments. If we associate a Boolean variable with each latch in a circuit, then these BDDs can be viewed as representing sets of states of the state machine.

The next concept is image computation. Basically, if we have a BDD that represents a set of states of a state machine, the image of that BDD is a new BDD that represents the set of all possible states that the machine could be in exactly one clock tick later. For example, return to the state machine in Figure 4. The BDD for $\neg x_0 \wedge x_1 \wedge \neg x_2$ represents the single state where latches x_0, x_1, and x_2 are outputting 0, 1, and 0. Depending on the value of the input, the machine has two possible states at the next clock tick, so the image of this BDD is the BDD for $(\neg x_0 \wedge x_1 \wedge \neg x_2) \vee (x_0 \wedge \neg x_1 \wedge x_2)$. The simplest way to compute images is as follows: First, build a BDD that represents the relationship between the present and next values of the latches. This BDD is called the transition relation. In our example, it would be the BDD for $(y_0 \equiv (x_0 \oplus i)) \wedge (y_1 \equiv (\neg i \wedge x_1) \vee (i \wedge x_2)) \wedge (y_2 \equiv (\neg i \wedge x_2) \vee (i \wedge x_1))$. Next, AND the transition relation with the BDD whose image you are computing. Then, existentially quantify[3] out the variables for the present state and the primary inputs.

The final idea is an iteration using images to compute all reachable states. Basically, we start with the reset state and compute the image to get the set of states reachable in one more clock tick, i.e.,

$$\begin{aligned} R_0 &:= \text{BDD for reset state} \\ R_1 &:= R_0 \vee \text{Image}(R_0) \\ &\ \ \vdots \\ R_{i+1} &:= R_i \vee \text{Image}(R_i) \end{aligned}$$

Intuitively, R_i is the set of all states reachable in i or fewer clock ticks from the reset state. This sequence will converge eventually, when $R_{i+1} = R_i$ (which is easy to test, since BDDs are canonical). In our example, the reset state $R_0 = \neg x_0 \wedge x_1 \wedge \neg x_2$, after one iteration $R_1 = (\neg x_0 \wedge x_1 \wedge \neg x_2) \vee (x_0 \wedge \neg x_1 \wedge x_2)$, and after two iterations $R_2 = R_1$, so we're done.

As with combinational verification, this approach is limited by the size of the BDDs generated, which is highly sensitive to the function being verified and the variable order used. Performance on any given circuit is extremely hard to predict. Nevertheless, as a very rough rule of thumb, the method described in this subsection can usually handle circuits with up to around one hundred latches. With more sophisticated enhancements, circuits with a few hundred latches are routinely verified, and occasionally practical circuits with thousands of latches can be verified.

The original papers on using BDDs for sequential verification (e.g., [11, 28]) are excellent references for the basic algorithms, including image computation and the reachability iteration.

D. *Model Checking*

Instead of just computing reachable states or comparing state machines, sometimes we'd like to check that a state machine obeys certain properties, e.g., that a one-hot encoded state machine is indeed one-hot encoded, that the machine is always resettable, that every request is eventually acknowledged, etc. Model checking lets us verify that a state machine obeys a property we specify using temporal logic.

Temporal logic is just a formal way of expressing properties that change over time. There are many different kinds of temporal logic; for brevity, we will only consider a few examples taken from one temporal logic — called CTL (Computation Tree Logic) — which is the most popular for formal hardware verification with model checking. The basic idea is that we start with ordinary Boolean logic, and then add special temporal operators for describing future events. For example, in CTL, the operator AX means "for all possible input values, in the next clock cycle,...", the operator EX means "there exists an input such that in the next clock cycle,...", the operator AG means "for all possible input values, it will always be true that,...", the operator EF means "there exists a sequence of input values such that eventually...", and so forth.[4] The temporal operators can nest, so for example, AGEF(*reset*) says that it is always possible to find a path back to *reset*, and AG(*req* $\Rightarrow$ AF*ack*) says that every request is always eventually followed by an acknowledgment. Returning to the simple circuit in Figure 4, let's consider what CTL formulas are true at the reset state ($x_0x_1x_2 = 010$). The formula x_1 is obviously true in the reset state, but the formula AXx_1 is false in the reset state because it is not the case that x_1 will be true for all input values at the next clock cycle (in particular, if $i = 1$). On the other hand, the formula EXx_1 is true in the reset state, because there exists an input value ($i = 0$) such that x_1 will be true at the next clock cycle. Similarly, AGx_1 is false, but EGx_1 is true (if the input stays at 0 forever).

The invention of model checking [12] was a theoretical breakthrough in the use of temporal logic for formal hardware verification. Roughly speaking, the idea is to systematically explore the state space of a finite state machine in order to check that the given temporal logic formula holds of the machine. Symbolic model checking [7] means using BDDs in the model checking algorithm. The algorithms used in symbolic model checking are a generalization of the reachability algorithm in the preceding subsection. For example, in addition to image, symbolic model checking uses op-

[3] Existential quantification, written $\exists x.f$, gives us a function that is true when there is a value of x that makes f true. We can compute $\exists x.f = f_x \vee f_{\neg x}$, where f_x means f with x set to 1 and $f_{\neg x}$ means f with x set to 0.

[4] Note that although these examples are phrased in terms of sequential circuits, model checking is often used at other levels of abstraction. More generally, AX would mean "for all possible next events...", EX would mean "there exists a possible next event...", etc.

erators such as preimage, which computes the set of all possible states the machine could have been in during the preceding clock cycle. Computing $\mathrm{EX}x_1$ is just a single preimage computation, and computing $\mathrm{EF}x_1$ is just like the reachability iteration, except that we start with x_1 and iterate with preimage instead of image. The other CTL operators are computed similarly.

In practice, model checking has similar limits to the reachability computation — the BDDs become too big. Also, the more expressive the temporal logic used, and the more complicated the properties specified, the greater computational complexity becomes. Generally, one should use the simplest model checker that can express the desired verification properties.

E. Symbolic Trajectory Evaluation

Symbolic trajectory evaluation [25] is an attempt to combine the efficiency of symbolic simulation with a bit of the temporal expressiveness of model checking. The basic idea is that if we severely restrict the temporal logic used for specifying properties, we can verify the properties using symbolic simulation.

In symbolic trajectory evaluation, the property to be checked is written in the form $A \Rightarrow C$, which means that whenever the circuit behavior matches the pattern specified by A, it must also satisfy the pattern specified by C. The formulas A and C are written in a special form called "trajectory formulas". A trajectory formula only allows specifying the values of circuit nodes for a bounded number of events into the future (in contrast to CTL operators like EF that specify behavior arbitrarily far in the future). Furthermore, trajectory formulas cannot express negation of a trajectory ("Match any pattern that doesn't look like...") or the OR of trajectories ("Match any pattern that looks like this or that."). In practice, the specfication language typically provides many features to ease writing trajectory formulas, but fundamentally, many properties that could be expressed in, say, CTL, simply cannot be expressed with trajectory formulas.

In exchange for this loss of expressiveness, though, comes a crucial property for efficiency: for any trajectory formula, there is a unique symbolic simulation vector (assignment of 0s, 1s, Xs, and symbolic values to the circuit inputs) that captures all behaviors that satisfy the trajectory formula. Verification, therefore, can be done with a single run of symbolic simulation — we symbolically simulate the vector for A, and after each simulation event, we check that the circuit state is consistent with the corresponding part of C. This algorithms is usually much faster than the iterations required for reachability and model checking.

In practice, the main obstacle to symbolic trajectory evaluation is figuring out how to express the desired property using trajectory formulas that can be symbolically simulated efficiently. If the simulation vector has too many symbolic variables, the BDDs will become too big, just as in symbolic simulation.

III. Enhancements and Research Directions

For space reasons, this paper can only give a brief taste of an extensive and varied research area. The fundamental problem with all of the methods described in the preceding section is that the BDDs can become too large to build within the limits of available memory. The aim of most research, therefore, is how to make BDDs smaller. I will briefly describe three general directions of research: improving BDDs, improving combinational equivalence checking, and improving sequential verification.

A. Better BDDs

The most obvious approach for making BDDs smaller is to try to find a better version of BDDs. Numerous researchers have investigated countless variants on BDDs, generally producing design trade-offs that are useful in some cases and useless or even counterproductive in others. For example, zero-suppressed BDDs [20] have a slightly different reduction rule in order to represent sparse sets efficiently. Several BDD variants have edge weights and multiple terminal nodes in order to represent numerical functions. Many representations are more general than BDDs, and can provably represent more functions more compactly, but lose canonicity, thereby severely reducing their usefulness for formal verification. Bryant [6] surveys many of these variants.

As we saw in Figure 2, choosing a good variable order can greatly affect BDD size. Many variable ordering heuristics have been developed for different domains, but much more work needs to be done. Currently, many people rely on dynamic variable reordering [23], a technique that periodically searches for better variable orders by exploiting the fact that small changes in the variable order are easy to make.

Finally, on a more mundane but very practical level is research aimed at efficient implementation of BDDs. Brace *et al.*'s paper [1] is the basis of most current implementations. Several papers have addressed making BDDs interact better with caches and virtual memory [21, 24, 18] and with parallel machines [27]. Sentovich [26] gives a comparison of several popular BDD packages.

B. Tricks for Combinational Equivalence

The key idea behind most research to improve combinational equivalence checking is to take advantage of structural similarities between the circuits. The circuits being compared in practice are usually quite similar, since the typical verification problem is to check that a small change didn't break the circuit.

Most approaches follows a framework proposed by Brand [2]. First, simulate the two circuits for a small number of random inputs. Points in the circuits that behaved identically during simulation are considered to be possibly equivalent. Then, try to prove that the possibly equivalent points are indeed equivalent, using any equivalences we've proven already to simplify the task.

Jain *et al.* [16] survey a wide variety of these algorithms. Here, let's consider a simple example of how such an algorithm might work. Suppose we are comparing two large circuits. In the first step, we run, say, 64 random simulation vectors. Points in the two circuits that behaved identically for all 64 simulation vectors are labeled as possibly equivalent. Next, we look for a possible equivalence between points in the two circuits that are close to the primary inputs. If we can prove these two points equivalent (by building the BDDs), we then delete the portions of the circuits that we have proven equivalent and introduce a new primary input at the equivalent point. If we can repeat this process all the way to the primary outputs, we have proven the two circuits equivalent, without ever building BDDs for the entire circuit (just BDDs for a small part at a time). Note that if this method fails to prove the two circuits equivalent, we cannot conclude that they are inequivalent without further computation, because the new primary inputs we introduced are not really primary inputs so we don't really have full controllability of them. This problem, called the false negative problem, is a serious obstacle for these algorithms in practice.

C. *Tricks for Sequential Reasoning*

The first problem encountered using the basic algorithms for sequential verification is that the BDD for the transition relation can be too large to build. Note that we were building the BDD for the transition relation only as a means to compute images. If we find an alternative way to compute images, we avoid this problem altogether. For example, good solutions have been found for synchronous circuits [11, 28], asynchronous circuits [7], and loop-free sequential programs [15].

The more serious problem is that the BDDs representing sets of states can become too large to build. Some attacks on this problem are to use multiple small BDDs (instead of one large BDD) to represent a set of states (e.g., [14, 19]), to perform a modified reachability iteration (e.g., [7, 8]), or to approximate the set of states with a smaller BDD (e.g., [22, 17]). This is an active research area, and much work remains to be done.

IV. Where to Learn More

For those interested in more details, Gupta [13] has written a much broader and deeper survey paper, which, although somewhat dated, is still an excellent source. More recently, Clarke and Kurshan [9] have written a very accessible introductory article that gives insight into the history and motivation behind formal hardware verification as well as the methods. The article also includes several sidebars authored by industrial researchers on practical experiences using formal hardware verification.

Exploring the research literature is more daunting because formal verification research is published in a wide variety of venues. Application and methodology papers tend to appear in conferences related to the specific application area. Research on improving verification algorithms often appear in VLSI CAD conferences and journals. Fundamental theoretical results generally appear in the theoretical computer science literature. In the past several years, a few publications have emerged that devote substantial attention to BDD-based formal verification, such as the conferences *Computer-Aided Verification* and *Formal Methods in Computer-Aided Design*, and the journal *Formal Methods in System Design*. Several survey papers and textbooks are due to appear shortly.

References

[1] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD Package," *DAC*, 1990, pp. 40–45.

[2] D. Brand, "Verification of Large Synthesized Designs," *ICCAD*, 1993, pp. 534–537.

[3] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Computers*, Vol. C-35, No. 8 (Aug. 1986), pp. 677–691.

[4] R. E. Bryant, "A Methodology for Hardware Verification Based on Logic Simulation," *J. of the ACM*, Vol. 38, No. 2, Apr. 1991, pp. 299–328.

[5] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams," *ACM Computing Surveys*, Vol. 24, No. 3, Sep. 1992, pp. 293–318.

[6] R. E. Bryant, "Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification," *ICCAD*, 1995, pp. 236–243.

[7] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill, "Symbolic Model Checking for Sequential Circuit Verification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. 13, No. 4 (Apr. 1994), pp. 401–424.

[8] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi, "Algorithms for Approximate FSM Traversal," *DAC*, 1993, pp. 25–30.

[9] E. M. Clarke and R. P. Kurshan, "Computer-Aided Verification," *IEEE Spectrum*, Jun. 1996, pp. 61–67.

[10] A. Cohn, "The Notion of Proof in Hardware Verification," *J. of Automated Reasoning*, Vol. 5, No. 2, 1989, pp. 127–139.

[11] O. Coudert and J. C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits", *ICCAD*, 1990, pp. 126–129.

[12] E. M. Clarke, E. A. Emerson, and A.P. Sistla, "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach," *Symp. on Princ. of Prog. Lang.*, 1983, pp. 117–126.

[13] A. Gupta, "Formal Hardware Verification Methods: A Survey," *Formal Methods in System Design*, Vol. 1, No. 2/3, 1992, pp. 151–238.

[14] A. J. Hu and D. L. Dill, "Efficient Verification with BDDs Using Implicitly Conjoined Invariants," *Computer-Aided Verification: 5th Int'l Conf.*, 1993, Springer LNCS 697, pp. 3–14.

[15] A. J. Hu, D. L. Dill, A. J. Drexler, and C. H. Yang, "Higher-Level Specification and Verification with BDDs," *Computer-Aided Verification: 4th Int'l Wkshp*, 1992, Springer LNCS 663.

[16] J. Jain, A. Narayan, M. Fujita, and A. Sangiovanni-Vincentelli, "Formal Verification of Combinational Circuits," *VLSI Design*, 1997.

[17] W. Lee, A. Pardo, J.-Y. Jang, G. Hachtel, and F. Somenzi, "Tearing Based Automatic Abstraction for CTL Model Checking," *ICCAD*, 1996, pp. 76–81.

[18] S. Manne, D. Grunwald, and F. Somenzi, "Remembrance of Things Past: Locality and Memory in BDDs," *DAC*, 1997, pp. 196–201.

[19] K. L. McMillan, "A Conjunctively Decomposed Boolean Representation for Symbolic Model Checking," *Computer-Aided Verification: 8th Int'l Conf.*, 1996, Springer LNCS 1102, pp. 13–25.

[20] S.-i. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems," *DAC*, 1993, pp. 272–277.

[21] H. Ochi, K. Yasuoka, and S. Yajima, "Breadth-First Manipulation of Very Large Binary-Decision Diagrams," *ICCAD*, 1993, pp. 48–55.

[22] K. Ravi and F. Somenzi, "High-Density Reachability Analysis," *ICCAD*, 1995, pp. 154–158.

[23] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *ICCAD*, 1993, pp. 42–47.

[24] J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "High Performance BDD Package By Exploiting Memory Hierarchy," *DAC*, 1996, pp. 635–640.

[25] C.-J. H. Seger and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, Vol. 6, 1995, pp. 147–189.

[26] E. M. Sentovich, "A Brief Study of BDD Package Performance," *Formal Methods in CAD: 1st Int'l Conf.*, 1996, Springer LNCS 1166, pp. 389–403.

[27] T. Stornetta and F. Brewer, "Implementation of an Efficient Parallel BDD Package," *DAC*, 1996, pp. 641–644.

[28] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines using BDD's" *ICCAD,* 1990, pp. 130–133.

DAC is the *Design Automation Conference*. ICCAD is the *International Conference on Computer-Aided Design*.