New Techniques for Efficient Verification with Implicitly Conjoined BDDs *

Alan J. Hu
Department of Computer Science
Stanford University

Gary York
Cadence Labs
Cadence Design Systems, Inc.

David L. Dill
Department of Computer Science
Stanford University

Abstract — In previous work, Hu and Dill identified a common cause of BDD-size blowup in high-level design verification and proposed the method of implicitly conjoined invariants to address the problem. That work, however, had some limitations: the user had to supply the property being verified as an implicit conjunction of BDDs, the heuristic used to decide which conjunctions to evaluate was rather simple, and the termination test, though fast and effective on a set of examples, was not proven to be always correct. In this work, we address those problems by proposing a new, more sophisticated heuristic to simplify and evaluate lists of implicitly conjoined BDDs and an exact termination test. We demonstrate on examples that these more complex heuristics are reasonably efficient as well as allowing verification of examples that were previously intractable.

I. Introduction

Formal design verification is attracting increasing interest as a tool to deal with the ever increasing cost and complexity of hardware designs and protocols. Binary decision diagrams (BDDs) [3] have enabled much of the recent progress in this area, starting from the early work applying BDDs to verification [1, 6, 5, 11, 24] and continuing through the current work of many researchers.

Current research on automatic formal hardware verification has focussed mainly on gate and transistor-level design. We believe that automatic formal verification also has an important role at the very highest levels of design, for example, in checking communications and consistency-maintenance protocols in a very large system. Verification of high-level, abstract specifications can catch *conceptual errors* early in the design cycle, when they are easier and cheaper to correct.

The attractions of BDD-based approaches for high-level design verification are threefold. First, they can conceivably handle large, real-world designs. Second, most of the proposed algorithms provide counterexamples if the verification attempt fails. Third, these approaches can be highly automatic, requiring minimal user effort. Minimizing user effort, in particular, is crucial to the economic justification of formal verification; performing verification makes sense only if the cost and effort required to use the verifier is small compared to the cost and effort saved by catching errors during verification.

With a few notable exceptions (e.g., [21, 7, 9]), however, BDD-based approaches have largely failed to achieve these objectives for high-level design verification. The straightforward algorithms appear generally unable to handle designs much more complex than dining philosophers or rings of mutual exclusion elements, and behind the rubric of "automatic" formal verification lurks considerable human labor invested in the minutiæ of preventing BDD-size blowup. Indeed, in our own research on large, real examples (e.g., industrial directory-based cache-coherence and link-level protocols), a brute-force approach that stores states explicitly in a hash table [13] has generally out-performed BDD-based approaches. Clearly, new techniques are needed to realize the potential advantages of BDDs at this high level of verification.

Implicitly conjoined invariants [17] is a recently-introduced technique, designed specifically to address some commonly-occurring causes of BDD-size blowup in high-level verification. The basic idea is that in high-level verification, we frequently encounter functions for which the BDD is huge, but which we can represent as the conjunction of small BDDs. By using these implicitly conjoined lists of BDDs instead of building the huge BDD, we can expand the set of problems for which BDD-based automatic verification is feasible.

Some problems, however, remain to be solved. (For instance, the method is not proven to terminate.) In this paper, we start from that result and introduce two new techniques for manipulating implicitly conjoined lists of BDDs: a heuristic for shortening an implicitly conjoined list of BDDs by searching for good (i.e., reduces memory usage) conjunctions to evaluate explicitly, and an exact termination test based on an efficient heuristic for comparing two implicitly conjoined lists of BDDs. These new techniques further expand the range of problems that can be verified, which we demonstrate by verifying some examples that are intractable by other means.

II. Background

We consider a verification paradigm that, though simple, has proven adequate for debugging a wide range of high-level designs [13]. Intuitively, we are verifying that every reachable state of the system satisfies a specified property. (Equivalently, we are model-checking CTL formulas of the form AGp only.) Formally, we model the system being verified as a single non-deterministic finite-state machine. (Such a finite-state machine is easily generated from a higher-level description. We are using the Ever verifier, which supports higher-level constructs using BDDs [18].) Non-determinism is important for high-level verification both to model non-determinism in the environment and also to abstract away implementation details, allowing us to postpone making low-level decisions until we've finished high-level verification. Let the machine have state space Q, transition relation $\delta : Q \times Q \rightarrow \{0, 1\}$, and a set of start states $S \subseteq Q$. The verification task is, given the set of "good" states $G \subseteq Q$ that satisfies the property being verified, to determine if there exists a path starting from a state in S and leading to a state not in G, and, if such a path exists, to output it as a counterexample to the property being verified.

*This research was supported in part by the Stanford Center for Integrated Systems's Multi-Module Systems Thrust. Much of this work was done while the first author was a summer intern at Cadence Design Systems. Some of this work was done using equipment donated by Sun Microsystems.

A. Image Operators

Before we proceed, we need a bit of notation:

Definition 1 *Given a set $Z \subseteq Q$, define the following operators:*

$$\begin{aligned} \mathit{Image}(\delta, Z) &= \{v \mid \exists u[u \in Z \wedge \delta(u, v)]\} \\ \mathit{PreImage}(\delta, Z) &= \{u \mid \exists v[v \in Z \wedge \delta(u, v)]\} \\ \mathit{BackImage}(\delta, Z) &= \{u \mid \forall v[\delta(u, v) \Rightarrow v \in Z]\}. \end{aligned}$$

Intuitively, Image gives the set of states that can be reached in one transition from a state in Z, PreImage gives the set of states that in one transition *can* reach a state in Z, and BackImage gives the set of states that in one transition *must* end up in Z.

These image operators form the basic operations of BDD-based verification algorithms. If Z and δ are both represented by small BDDs, these operations can be done directly using BDD operations [6, 5, 24]. If the BDD for δ is too large to build (a common problem), a number of techniques are available to compute these images without building the BDD for δ [4, 18]. Also, note that $\text{BackImage}(\delta, Z) = \neg\text{PreImage}(\delta, \neg Z)$, so if Z is represented by a small BDD, computing either of these two images is equally fast for an efficient BDD implementation (where negation is constant-time) [2].

What happens if the BDD for Z is also too large to build (also a common problem for high-level design verification)? The following theorem enables computing the BackImage of an implicit conjunction of a list of BDDs without building the BDD for the conjunction [17]. (Dually, we can compute the Image and PreImage of implicit disjunctions without building the BDD for the entire disjunction.)

Theorem 1

$$\mathit{BackImage}(\delta, Y \wedge Z) = \mathit{BackImage}(\delta, Y) \wedge \mathit{BackImage}(\delta, Z).$$

Proof: $\text{BackImage}(\delta, Y \wedge Z)$ is defined as $\forall v[\delta(u, v) \Rightarrow (Y(v) \wedge Z(v))]$, which equals $\forall v[(\delta(u, v) \Rightarrow Y(v)) \wedge (\delta(u, v) \Rightarrow Z(v))]$, which equals $\forall v[\delta(u, v) \Rightarrow Y(v)] \wedge \forall v[\delta(u, v) \Rightarrow Z(v)]$, which is defined as $\text{BackImage}(\delta, Y) \wedge \text{BackImage}(\delta, Z)$. □

Using these image operators, it's easy to explain the standard approaches to our verification problem.

B. Verification Approaches

One standard algorithm we call "forward traversal." The intuition is that we iteratively compute the set R_i of states that can be reached in i or fewer transitions from the start states. Mathematically, we initialize $R_0 = S$, and compute $R_{i+1} = R_0 \vee \text{Image}(\delta, R_i)$. If R_i ever goes outside the set of good states ($R_i \not\subseteq G$), then we have a violation, and it's easy to produce a counterexample trace. Otherwise, the sequence will eventually converge to the set of reachable states, meaning that the verification succeeds. Details of this approach are available elsewhere (e.g., [11, 5, 8, 24, 4]).

The other standard algorithm we call "backward traversal." The intuition here is that we iteratively compute the set G_i of states such that all paths of length i or less starting in G_i must remain within the set of good states G. Mathematically, we initialize $G_0 = G$, and compute $G_{i+1} = G_0 \wedge \text{BackImage}(\delta, G_i)$. If we reach a point where G_i does not contain all of the start states ($S \not\subseteq G_i$), then there exists a sequence of i transitions from a start state to a violating state. Otherwise, the sequence will converge, meaning the verification succeeds. Details for this approach are also available from several sources (e.g., [6, 23, 14]).

C. Implicitly Conjoined Invariants

The method of implicitly conjoined invariants [17] is built on the backward traversal. As mentioned already, this method is predicated on the observation that in high-level design verification, we frequently encounter the situation where the G_i's in the backward traversal require huge BDDs, but could be expressed as the implicit conjunction of small BDDs. Suppose the set G of states that satisfy the property we are verifying is actually specified as the implicit conjunction of small BDDs: $G = G[1] \wedge \cdots \wedge G[n]$. Can we perform the backward traversal algorithm keeping all the G_i's in this form, never building the huge BDD required to represent the conjunction?

In the original paper [17], the answer is a qualified yes. Clearly, the violation check $S \overset{?}{\not\subseteq} G_i$ can be broken down into individual checks $S \overset{?}{\not\subseteq} G_i[j]$ for each j. Similarly, Theorem 1 allows us break down the BackImage computation: $\text{BackImage}(\delta, G_i[1] \wedge \cdots \wedge G_i[n]) = \text{BackImage}(\delta, G_i[1]) \wedge \cdots \wedge \text{BackImage}(\delta, G_i[n])$. Thus, computing the BackImage of an implicitly conjoined list of n BDDs results in another implicitly conjoined list of n BDDs. This phenomenon leads to a problem: On each iteration, we AND the n BDDs representing G_0 into the current implicit conjunction; if we just add these n BDDs to the current list, the length of the implicitly conjoined list will grow on each iteration. Thus, we need some way of deciding which of the implied conjunctions to actually evaluate (i.e., actually build the BDD for the conjunction of two BDDs, reducing the length of the list by one). Furthermore, the BDD for $X \wedge Y$ can be smaller than the BDD for X plus the BDD for Y, so evaluating some conjunctions might even reduce the amount of memory used. The other crucial problem is how to determine termination. Since an implicitly conjoined list of BDDs is not a canonical form (unlike single BDDs), testing whether $G_i = G_{i+1}$ is not trivial.

Another important point is that an implicitly conjoined list of BDDs gives us an opportunity to perform don't-care optimizations. The correctness of the backward traversal algorithm relies on computing the G_i correctly, not on the particular representation used. Therefore, if we represent G_i as an implicitly conjoined list of BDDs, the specific BDDs in the list don't matter, as long as the implied conjunction represents the correct set. Thus, each conjunct defines a *care* set for the other conjuncts, since when any one conjunct is false, the entire conjunction is false. Suppose we have an operator $\text{BDDSimplify}(f, c)$ that takes BDDs f and c and performs care-set simplification (returning a smaller BDD that agrees with f whenever c is true). We can apply such an operator freely to any implicitly conjoined list of BDDs, reducing the sizes of the BDDs in the list. Much of the efficiency of implicitly conjoined invariants derives from these simplifications, but determining the best way to apply simplification is an open problem.

The original paper addressed these problems with some simple heuristics. The details of these heuristics do not concern us here, (See [17].) except to point out the key weaknesses. First, the evaluation policy is simple, making no effort to adapt to different problems or seek out particularly good conjunctions to evaluate. Second, the termination test given, while fast, simple and successful on several examples, isn't proven to be correct — it could conceivably fail to detect convergence. Most importantly, the heuristics used require the user to supply the property being verified as an implicit conjunction. Failure to do so reduces the algorithm to the ordinary backward traversal with BDDs. Since we seek to minimize user effort and sophistication required to use formal verification, such a requirement is undesirable.

III. New Techniques

The new techniques given here address the preceding problems. We present a new evaluation and simplification policy that attempts to find good conjunctions to evaluate and also attempts automatically to form implicitly conjoined lists of small BDDs, relieving the user of this burden. We also present an exact termination test that determines if two implicitly conjoined lists of BDDs are equal. While the exact test requires exponential time in theory, some examples will show that the exact test is frequently not too time-consuming in practice.

A. *Evaluation and Simplification Policy*

On each iteration of the backward traversal algorithm, we compute an implicitly conjoined list of BDDs for $G_{i+1} = G_0 \wedge \text{BackImage}(\delta, G_i)$. We seek to find an equivalent list of BDDs that is smaller overall. More abstractly, given function X expressed as implicit conjunction of BDDs $X_1 \wedge \cdots \wedge X_n$, we want to find an implicit conjunction with smaller overall size $Y = Y_1 \wedge \cdots \wedge Y_m$, such that $X = Y$.

We are using the BDD simplification operator proposed by Coudert, Berthet, and Madre [11], generally known as Restrict [10] or Reduce [20]. While this operator doesn't always reduce the size of the BDD it is applied to, it seems generally effective, so we first simplify each BDD X_i by every other BDD X_j that's smaller than it. (Simplifying a small BDD by a large BDD, in our experience, does little good.) The remaining problem is simply to decide which conjunctions to evaluate to minimize the total size of the implicitly conjoined list.

At first glance, this problem appears an ideal combinatorial optimization problem. For every subset of the BDDs in the list, we can replace that subset by the single BDD that's the conjunction of all the BDDs in the subset. Thus, we arrive at a set-covering problem:

> Let X be a set of n conjuncts $X = \{X_1, \ldots, X_n\}$. For every subset $s \subseteq X$, define the cost of that subset $c(s)$ to be the size of the (single) BDD that represents the conjunction of all conjuncts contained in s: $c(s) = \text{BDDSize}\left(\bigwedge_{X_i \in s} X_i\right)$. Find the minimum cost set S of subsets that covers all the conjuncts in X, i.e., find the S that minimizes $\sum_{s \in S} c(s)$ subject to $\forall i \exists s \in S[X_i \in s]$.

Unfortunately, this approach yields an instance of Minimum Weight Cover, and Minimum Weight Cover is clearly NP-hard by reduction from Minimum Cover [15], even if restricted to subsets of three or fewer conjuncts. (The constraints on the cost function imposed by BDD properties, however, might make this problem easier than Minimum Weight Cover in general.) If we restrict ourselves to pairwise subsets only, we can solve the problem in polynomial time:

Theorem 2 *Finding the min cost pairwise cover is polynomial time.*

Proof: Draw a complete graph with a vertex for each conjunct. Label each edge with the size of the BDD for the conjunction of the BDDs on the two incident vertices. Next, make a copy of each vertex. Connect each original vertex to its copy; label that edge with the minimum of the size of the BDD at that vertex and the labels of all other incident edges. (This edge indicates the cheapest way to include this conjunct ignoring all the other conjuncts.) Connect all the copy vertices to each other with weight 0 edges. Minimum weighted matching, which is polynomial time (e.g. [22]), on this graph gives the optimum cover. □

Even this result is of limited practical value because in reality, for efficient BDD implementations, BDD sizes do not add, since all BDDs in the system can share nodes with each other [2]. Using a complex "optimum" algorithm for a rough approximation to a problem makes little sense. Thus, we turn to a greedy heuristic.

```
Conjunction Evaluation:
Let GrowThreshold = 1.5.
Build a table P of all pairwise conjunctions: P_ij := X_i ∧ X_j.
Loop
    Find the i, j (with i ≠ j) that minimizes the ratio:
        r = BDDSize(P_ij)/BDDSize(X_i, X_j)
        Note: BDDSize(X_i, X_j) takes node-sharing into account.
    If r_min > GrowThreshold, then exit.
    Replace X_i and X_j with P_ij.
    Update P to reflect the modified conjunct list.
EndLoop
```

Figure 1: This algorithm is a greedy algorithm to find a good set of conjunctions to evaluate in an implicitly conjoined list of BDDs. We have arbitrarily set the GrowThreshold to 1.5 with satisfactory results. Additional tuning could improve results further: a smaller threshold holds BDD size down, but can get caught in a local minimum, whereas any threshold greater than 1 could theoretically allow us to build exponentially-sized BDDs.

The heuristic we propose is fairly simple, yet accounts for some degree of node sharing among different BDDs. The intuition is to find the pair of BDDs for which evaluating the conjunction gives the greatest savings over not evaluating the conjunction. We replace the pair of BDDs with the single BDD for the conjunction and repeat the process. The process terminates when the best conjunction to evaluate doesn't give sufficient savings. The algorithm is given in Figure 1.

B. *Exact Termination Testing*

Deciding termination in the verification algorithms requires testing whether the iteration has converged — is $R_i = R_{i+1}$ or $G_i = G_{i+1}$? (Actually, checking implication suffices since these sequences are monotonic. The current implementation does not exploit this optimization.) The termination test proposed in the original paper relied on the structure of the original simple evaluation policy to provide a good chance of operating correctly [17]. Given that the evaluation and simplification technique proposed in the preceding section can extensively modify an implicitly conjoined list of BDDs, a method to compare two arbitrary implicitly conjoined lists of BDDs seems necessary for reliable termination testing. Furthermore, verification, by nature, should favor a method that is guaranteed correct, but possibly slow, over a method that is fast, but possibly wrong. Thus, we look for an exact test of equality for arbitrary implicitly conjoined BDDs.

Suppose we have two implicitly conjoined lists of BDDs $X = \{X_1 \wedge \cdots \wedge X_n\}$ and $Y = \{Y_1 \wedge \cdots \wedge Y_m\}$. Our task is to determine whether or not $X = Y$, without building the BDDs for X or for Y, since those BDDs are presumably too big to build. We proceed by decomposing the problem. First, note that $X = Y$ if and only if $X \Rightarrow Y$ and $Y \Rightarrow X$, so we can check each implication separately. For brevity, we'll describe the $X \Rightarrow Y$ case only. Next, note that $X \Rightarrow Y$ if and only if $(X \Rightarrow Y_1) \wedge \cdots \wedge (X \Rightarrow Y_m)$. Again, we can check each case separately. Again, all the cases are identical, so, for brevity, we'll describe only the $X \Rightarrow Y_1$ case. Checking whether $X \Rightarrow Y_1$ is true is equivalent to checking whether $\neg X \vee Y_1$ is a tautology, which is actually checking whether $\neg X_1 \vee \cdots \vee \neg X_n \vee Y_1$ is a tautology. Thus, we have reduced the problem of checking the equality of two implicitly conjoined lists of BDDs to the problem of checking whether the disjunction of a list of BDDs is a tautology.

We can't simply build the BDD for this disjunction, as that would still blow up, so we further decompose the problem into smaller, more manageable pieces. Our strategy here is to look for easy special cases first, and if that fails, to perform a Shannon expansion on the implicit disjunction. Specifically, we perform the following steps in sequence:

1. If any BDD in the list is the constant True, the whole disjunction is a tautology. If any BDD is the constant False, discard it.
2. If any two BDDs in the list are complements, the whole disjunction is a tautology. (Recall that negation is fast in efficient BDD implementations.) If any two BDDs are identical, discard one.
3. If the disjunction of any two BDDs is the constant True, the whole disjunction is a tautology.
4. If all else fails, choose a BDD variable from a BDD in the list, perform a Shannon expansion, and check tautology recursively on both cofactors. (The positive and negative cofactors will each be an implicitly conjoined list of BDDs.) For simplicity, we are currently selecting the top BDD variable of the first BDD in the list as the variable to cofactor on.

We further optimize Step 3 using the following theorem:

Theorem 3 *If the Restrict or the Constrain operator [10] is used for BDDSimplify, then for any Boolean functions a and b, the disjunction $a \vee b$ is a tautology if and only if BDDSimplify$(a, \neg b)$ is a tautology.*

Proof: We give the proof for Restrict; the proof for Constrain is almost identical. The proof is by induction on the size of the supports of the BDDs. The base case, when either a or b is either the constant True or the constant False, is easy to verify. The inductive step relies on the recursive definition of Restrict, which defines Restrict(f, c) in terms of the Shannon cofactors f_x, $f_{\bar{x}}$, c_x, and $c_{\bar{x}}$, where x is a BDD variable in f or in c: (The exact definition of x is irrelevant here.)

$$\text{Restrict}(f,c) = \begin{cases} \text{Restrict}(f, c_x \vee c_{\bar{x}}) & \text{if } f_x = f_{\bar{x}} \\ \text{Restrict}(f_x, c_x) & \text{if } c_{\bar{x}} = \text{False} \\ \text{Restrict}(f_{\bar{x}}, c_{\bar{x}}) & \text{if } c_x = \text{False} \\ (x \wedge \text{Restrict}(f_x, c_x)) \vee & \\ \quad (\bar{x} \wedge \text{Restrict}(f_{\bar{x}}, c_{\bar{x}})) & \text{otherwise} \end{cases}$$

In the first case, Restrict$(a, \neg b)$ is a tautology iff
Restrict$(a, (\neg b)_x \vee (\neg b)_{\bar{x}})$ is a tautology iff
$a \vee \neg((\neg b)_x \vee (\neg b)_{\bar{x}})$ is a tautology (inductive hypothesis) iff
$(a_x \vee b_x) \wedge (a_{\bar{x}} \vee b_{\bar{x}})$ is a tautology (in this case $a = a_x = a_{\bar{x}}$) iff
$(a \vee b)$ is a tautology.
The other cases are similar. □
Since we are using Restrict as our BDDSimplify function, this theorem means we get the effect of Step 3 automatically if we simplify each BDD in the list by all the other BDDs, and then repeat Step 1 to handle any resulting constant True's and False's in the list.

IV. Experimental Results

The preceding algorithms clearly require non-trivial computations. Whether they are justified in practice can be demonstrated only by experimentation. We must establish that the performance cost of these algorithms is reasonable, or that these algorithms are more powerful, verifying examples that were previously intractable, or, ideally, both.

Comparisons of different algorithms for verification with BDDs can be problematic. Runtime comparisons are meaningful provided the different methods run on the same machine type with the same BDD implementation. Memory requirements are harder to quantify. The total memory used during verification is probably the most realistic measure of how memory-efficient a particular algorithm is, but is highly sensitive to details of the BDD implementation used. Vagaries of garbage collection and caching policies, for example, can easily account for a factor of two difference in reported memory usage. On the other hand, reporting the number of BDD nodes required to represent specific portions of the verification process is implementation independent, but fails to account for the various intermediate data structures and BDDs that can consume substantial memory. In this paper, we report the runtime on a Sun 4/75, the total memory used during the verification process, as well as the largest number of BDD nodes required to represent any of the R_i (for forward traversals) or G_i (for backward traversals). We are using David Long's BDD package [20].

A. *Performance Penalty vs. Previous Results*

First, let's tackle the question of performance. How much more time and memory do the algorithms we propose require over earlier and simpler methods? We use three examples, all previously used to demonstrate the efficacy of the original implicitly conjoined invariants approach and described thoroughly in that paper [17].

The first example is a typed FIFO queue. These data structures frequently occur in high-level design verification either to represent actual queues in the system or to delay data going from one part of the model to another. The specific example is 8 bits wide, with the bitslices interleaved (a standard variable-ordering heuristic for datapaths [19]). The data going into the queue obeys a type constraint: each item must be between 0 and 128 *inclusive*. We verify for various queue depths that all items in the queue always obey the type constraint.

The second example is an abstraction of a group of processors communicating via a shared network, again, a common occurrence in high-level design verification. We have a set of processors that non-deterministically issue requests into a non-message-order-preserving network. Each request carries only the requester's ID as a return address. A server non-deterministically pulls requests out of the network and sends acknowledgments back to the originating processor. When a processor issues a request, it increments a local counter of outstanding requests. When it receives an acknowledgment, it decrements the counter. We verify, for various numbers of processors, that each processor's counter correctly indicates the number of message it has outstanding in the network. (We assume that $n < 16$, so IDs are 4 bits each. The network is modeled as an n-element array of messages, each of which carries a valid bit, a req/ack flag, and a return address.)

The third example is a moving-average filter, a common DSP algorithm. We compare an implementation using a pipelined tree of adders against a combinational specification. See Figure 2. The samples being averaged are always 8 bits. We verify filters of depth 4, 8, and 16. In the original paper, no method could verify the larger filters. Carefully adding user-defined "assisting invariants" (essentially spelling out lemmas to help the verifier prove the property being verified), however, allowed successful verification. In this section, we'll verify with the assisting invariants (specifying that the average of each layer of the adder tree must equal the corresponding entry in the delay FIFO of the specification); in the next section, we'll reconsider this example without the user-defined assistance.

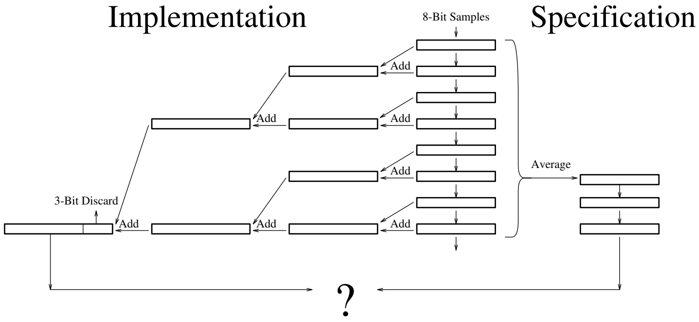

Figure 2: Diagram of Size 8 Moving Average Filter. The verification task is to prove that the implementation using a pipelined tree of adders gives the same result as the specification, which is the average computed directly and then delayed in a FIFO to match the pipeline depth of the implementation.

Table 1 summarizes results on these examples. Clearly, the additional time and space overhead of the more complex algorithms proposed in this paper is minimal. Also, both methods using implicit conjunctions of BDDs manage to avoid the exponential BDD-size blowups afflicting the conventional forward and backward traversals.

B. New Examples

Now, let's look at the question of power. Do the more sophisticated methods proposed in this paper enable verification of examples that are otherwise intractable? We consider two examples: a continuation of the preceding moving-average filter example, and a simple model of the register-bypass and branch-stall aspects of a pipelined processor.

The previous results for the moving-average filter required user-supplied assisting invariants to complete the verification. Since our goal is *automatic* formal verification, what happens if we attempt the example without the user-supplied help, if we simply give the description of the filter and ask the verifier to prove that the implementation and specification agree? Results for this case, summarized in Table 2, show that only the algorithms proposed in this paper can handle the larger filters. Interestingly, comparing the results here to the results in Table 1 that rely on user assistance, we find that the new evaluation and simplification algorithm is actually deriving the assisting invariants, fully automatically, at minimal cost in memory and runtime.

Our second example is to verify a simple pipelined processor against a non-pipelined specification. To reduce the size of the model, and since we're only concerned with the processor, we will abstract away the memory. Instead, both versions of the processor will execute the same non-deterministically-generated stream of instructions. Instructions are encoded as a 3-bit opcode, followed by fields specifying the source and destination registers, followed by a field for specifying immediate data values. There are eight instructions: NOP, BR, LD, ST, ADD, SUB, MOV, and SR. NOP performs no operation. BR models a branch instruction. Since the instruction stream is generated non-deterministically, we do not model a program counter, so the BR instruction essentially performs no operation. (It does, however, stall the pipeline, as we'll discuss later.) LD loads the specified destination register with the contents of the immediate field. ST is a no-op, since we aren't modeling memory. ADD adds the contents of the specified source register into the specified destination register. SUB subtracts the source register from the destination register. MOV copies the source register into the destination register. SR shifts the contents of the specified destination register right by one bit.

Size	Meth.	Time	Iter	Mem	BDD Nodes
Example: 8-Bit Wide Typed FIFO Buffer					
5	Fwd	0:03	6	936K	543
	Bkwd	0:01	1	936K	543
	ICI	0:00	1	552K	41 (5×9 nodes)
	XICI	0:00	1	556K	41 (5×9 nodes)
10	Fwd	5:37	11	13048K	32767
	Bkwd	1:56	1	10008K	32767
	ICI	0:03	1	1016K	81 (10×9 nodes)
	XICI	0:03	1	1020K	81 (10×9 nodes)
Example: Processors Sending Messages Through Network					
4	Fwd	0:04	9	1264K	1198
	Bkwd	0:02	1	1136K	994
	FD	0:13	9	1028K	41
	ICI	0:02	1	1008K	245 (4×62 nodes)
	XICI	0:02	1	1008K	245 (4×62 nodes)
7	Fwd	11:53	15	29324K	88647
	Bkwd	2:15	1	14412K	61861
	FD	3:20	15	2652K	169
	ICI	0:14	1	3152K	1086 (7×156 nodes)
	XICI	0:22	1	3660K	1086 (7×156 nodes)
Example: 8-Bit Wide Moving Average Filter					
4	Fwd	0:54	3	10976K	11267
	Bkwd	0:04	1	1248K	490
	ICI	0:03	1	832K	146 $(102, 45)$
	XICI	0:03	1	832K	146 $(102, 45)$
8	Fwd	Exceeded 60MB.			
	Bkwd	Exceeded 40 minutes.			
	ICI	0:25	1	3880K	638 $(390, 169, 81)$
	XICI	0:28	1	3880K	638 $(390, 169, 81)$
16	ICI	3:26	1	27416K	2558 $(1501, 629, 290, 141)$
	XICI	3:41	1	27416K	2558 $(1501, 629, 290, 141)$

Table 1: Performance vs. Previous Methods. See the text for descriptions of examples. "Meth." indicates the verification method used: "Fwd" is conventional forward traversal, "Bkwd" is conventional backward traversal, "FD" is forward traversal exploiting user-specified functional dependencies [16], "ICI" is backward traversal using the original implicitly conjoined invariants method [17], and "XICI" is the implicitly conjoined invariants method extended with the techniques in this paper. Time is in minutes and seconds. "Iter" gives the number of iterations before convergence. "Mem" shows total memory used by the verifier. "BDD Nodes" shows the largest number of BDD nodes used to represent the set of states computed at an iteration (R_i or G_i). The numbers in parentheses are the sizes of the individual BDDs in the implicit conjunction, with "($i \times j$ nodes)" indicating i BDDs of j nodes each. (Numbers don't always add up because of node sharing.) Examples were run on a SUN 4/75, using a BDD package developed at CMU by David Long [20].

The pipeline is three stages deep. The first stage fetches the next instruction from the non-deterministic instruction stream. The second stage decodes the instruction, fetches the appropriate values from the register file (or the immediate field for a LD) and computes the result. The last stage writes the result back into the register file. There are, of course, some complications. First, if one instruction relies on the result of the preceding instruction, the result won't be written back by the Writeback stage in time for the Execute stage to fetch the correct value, eg.:

```
; assume r0=0 and r1=0
LD r1, #1      ; make r1=1
ADD r0, r1     ; add r1 to r0
```

Size	Meth.	Time	Iter	Mem	BDD Nodes
4	Fwd	0:52	3	6880K	11267
	Bkwd	0:04	1	1248K	490
	ICI	0:04	1	1248K	490
	XICI	0:03	2	932K	146 (45, 102)
8	Fwd	Exceeded 60MB.			
	Bkwd	Exceeded 40 minutes.			
	ICI	Exceeded 40 minutes.			
	XICI	0:31	3	5676K	638 (61, 169, 390)
16	XICI	5:45	4	28544K	2558 (141, 290, 629, 1501)

Table 2: Moving Average Filter without Assisting Invariants. Previously, we verified the moving-average filter example with the aid of user-supplied assisting invariants — additional properties that guide the verifier in partitioning intermediate results into implicit conjunctions. In this experiment, we simply give the verifier the description of the filter and ask it to verify that the output of the implementation agrees with the output of the specification. Column labels and test conditions are the same as in the preceding table.

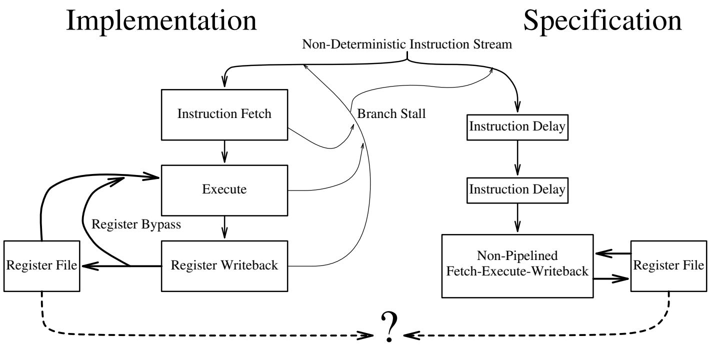

Figure 3: Diagram of Pipelined Processor Example. A pipelined and a non-pipelined version of a processor execute the same non-deterministically generated stream of instructions. The verification task is to prove that the register files of the two versions always agree.

After executing this code fragment, `r0` should be equal to 1. As described, however, the pipelined processor would not have updated `r1` to be 1 in time for the ADD instruction. The standard solution to this problem is to add a "register bypass path" to the pipeline: If the Execute unit detects that the current instruction needs the result of the previous instruction, it bypasses the register file and gets the value needed directly from the Writeback unit. Our example includes such a register bypass path. Another complication occurs because of branches. In a real machine, the Instruction Fetch unit does not know where the next instruction will be until the Writeback unit updates the program counter. For our example, we adopt a standard solution — the branch stall. If any stage in the pipeline contains a BR instruction, the pipeline is forced to stall. (We implement the stall by forcing NOP instructions into the Fetch unit until the BR clears the Writeback unit.)

The verification task is to show that the register files of the pipelined and non-pipelined processors always agree when executing the same sequence of instructions. In order to keep the two descriptions synchronized, the non-pipelined processor buffers incoming instructions for two cycles to match the pipelined processor. Also, a branch stall in the pipeline will also stall the non-pipelined processor. This example is summarized in Figure 3.

Table 3 summarizes the results for this verification example for various numbers of registers and datapath sizes (the bit-width of the registers and the immediate field). Clearly, the techniques of this paper expand the range of problems that can be verified automatically. It's worth noting that, as in the moving-average filter example, carefully hand-constructed assisting invariants give better results than the automatic techniques presented here. (For example, we can verify the 2-register, 3-bit datapath example in only 2 iterations, 6:19, using 25592K of memory and with the largest G_i using only 6602 BDD nodes.) That clever human intervention can improve the efficiency of a verification problem is not, however, surprising, and does not diminish the importance of minimizing the amount of human effort and sophistication required to use formal verification.

Size	Meth.	Time	Iter	Mem	BDD Nodes
2 R, 1 B	Fwd	5:11	4	49644K	284745
	Bkwd	0:27	4	4080K	10745
	ICI	0:27	4	4080K	10745
	XICI	0:31	4	4084K	10745
2 R, 2 B	Fwd	Exceeded 60MB.			
	Bkwd	Exceeded 60MB.			
	ICI	Exceeded 60MB.			
	XICI	1:48	4	7316K	8485 (45, 441, 1345, 6657)
2 R, 3 B	XICI	13:35	4	59480K	57510 (189, 2503, 9591, 45230)
4 R, 1 B	Fwd	Exceeded 60MB.			
	Bkwd	Exceeded 60MB.			
	ICI	Exceeded 60MB.			
	XICI	7:06	4	24156K	12947 (45, 849, 1290, 10767)

Table 3: Results for Pipelined Processor Example. We are verifying that the register files of a pipelined and a non-pipelined version of a simple processor always agree. The pipelined version includes a register bypass path as well as a stall for branch instructions. "R" indicates the number of registers in each processor. "B" indicates the width of the datapath. Column labels and test conditions are the same as in the preceding table.

V. Future Research and Conclusions

While the techniques presented are certainly not the last word in formal verification — it's quite easy to find examples that still cannot be verified automatically — they are definitely a step forward, increasing the applicability of automatic verification with BDDs. However, as always, progress seems to raise at least as many questions as it answers, and there are many avenues for further research.

Obviously, considerable performance and heuristic tuning can still be done. We have not, for example, investigated finding the best GrowThreshold in the evaluation algorithm, or experimented with choosing the best variable to use for cofactoring in the termination test. Another obvious direction for improvement is in algorithms for BDD simplification. Methods using implicitly conjoined BDDs rely on simplification for their efficiency, so any improvement in simplification will pay-off immediately.

Another BDD-simplification effect is more subtle. In our experiments, we frequently encounter a situation where we wish to simplify a BDD f by two other BDDs c_1 and c_2. Simplifying f by either c_1 or c_2, however, results in a several-fold increase in the size of f, and then simplifying the large resulting BDD by the other c shrinks the final result to something much smaller than the original f. Why not discard the result of the first simplify, since it's bigger than f? Unfortunately, this approach results in no simplification at all: the first simplify blows up the BDD and is discarded, and the second simplify also blows up the BDD and is discarded. We really wish to simplify by $c_1 \wedge c_2$, which gives a smaller care-set, but we can't afford to build the BDD for $c_1 \wedge c_2$. What's needed, therefore, is a routine that simplifies using multiple BDDs simultaneously.

Another issue also involves BDD-size blowups in intermediate computations. In our algorithm for choosing conjunctions to evaluate, we are building all pairwise conjunctions even though we'll be using only a few of them. Furthermore, before we build the BDD for any conjunction, we already have a limit on how large it can be and still be useful: if the conjunction becomes significantly larger than the size of the two conjuncts, we know we won't use that conjunction. Hence, it would be useful to have the capability to compute the size of a result without actually building the BDD for that result, and to abort any of these operations if the size exceeds a specified bound.

Finally, a comparison of the moving-average filter and pipelined processor examples suggests that there is still considerable room for improving the simplify-and-evaluate algorithm. For the moving-average filter, the new algorithm was able to derive the same assisting invariants as were human-generated. For the pipelined processor example, however, the derived BDDs were clearly inferior to the human-generated assisting invariants (e.g., 57510 vs. 6602 BDD nodes for 2 registers, 3-bit datapath). How can we close this gap?

Additional research along these lines should further expand the envelope of problems that can be verified automatically. As the human time and expertise required to successfully use formal verification decreases, verification will become increasingly practical and valuable. For the long term, we need to make automatic formal verification truly automatic, thereby maximizing its beneficial impact.

Acknowledgments

We would like to thank Eric Torng for his construction in Theorem 2 and David Long for fast and responsive BDD package support.

References

[1] S. Bose and A. Fisher, "Automatic Verification of Synchronous Circuits Using Symbolic Logic Simulation and Temporal Logic," *IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design,* Luc J.M. Claesen, ed., North Holland, 1989.

[2] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant, "Efficient Implementation of a BDD Package," *27th ACM/IEEE Design Automation Conference,* 1990, pp. 40–45.

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

[4] J.R. Burch, E.M. Clarke, and D.E. Long, "Symbolic Model Checking with Partitioned Transition Relations," *VLSI '91: Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration,* Edinburgh, Great Britain, 1991.

[5] J.R. Burch, E.M. Clarke, K.L. McMillan, and David L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," *27th ACM/IEEE Design Automation Conference,* 1990, pp. 46-51.

[6] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," *Proceedings of the Conference on Logic in Computer Science*, 1990, pp. 428–439.

[7] Massimiliano Chiodo, Thomas R. Shiple, Alberto Sangiovanni-Vincentelli, Robert K. Brayton, "Automatic Compositional Minimization in CTL Model Checking," *IEEE International Conference on Computer-Aided Design,* 1992, pp. 172–178.

[8] Hyunwoo Cho, Gary Hachtel, Seh-Woong Jeong, Bernard Plessier, Eric Schwarz, and Fabio Somenzi, "ATPG Aspects of FSM Verification," *IEEE International Conference on Computer-Aided Design,* 1990, pp. 134–137.

[9] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness, "Verification of the Futurebus+ Cache Coherence Protocol," in L. Claesen, ed., *11th International Symposium on Computer Hardware Description Languages and their Applications*, North-Holland, 1993.

[10] Olivier Coudert and Jean Christophe Madre, "A Unified Framework for the Formal Verification of Sequential Circuits," *IEEE International Conference on Computer-Aided Design,* 1990, pp. 126–129.

[11] Olivier Coudert, Christian Berthet, and Jean Christophe Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution," *Automatic Verification Methods for Finite State Systems*, J. Sifakis, ed., Lecture Notes in Computer Science Vol. 407, Springer-Verlag, 1989.

[12] Olivier Coudert, Christian Berthet, and Jean Christophe Madre, "Verification of Sequential Machines Using Boolean Functional Vectors," *IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design,* Luc J.M. Claesen, ed., North Holland, 1989.

[13] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang, "Protocol Verification as a Hardware Design Aid," *IEEE International Conference on Computer Design,* October 1992.

[14] Thomas Filkorn, "Functional Extension of Symbolic Model Checking," *Computer-Aided Verification: Third International Workshop,* July 1–4, 1991, K.G. Larsen and A. Skou, eds., Lecture Notes in Computer Science Vol. 575, Springer-Verlag, published 1992.

[15] Michael R. Garey and David S. Johnson, *Computers and Intractability*, W.H. Freeman and Company, 1979, p. 222.

[16] Alan J. Hu and David L. Dill, "Reducing BDD Size by Exploiting Functional Dependencies," *30th Design Automation Conference*, 1993.

[17] Alan J. Hu and David L. Dill, "Efficient Verification with BDDs using Implicitly Conjoined Invariants," *Computer Aided Verification: Fifth International Conference,* 1993, published in Lecture Notes in Computer Science Vol. 697, Springer-Verlag, 1993.

[18] Alan J. Hu, David L. Dill, Andreas J. Drexler, and C. Han Yang, "Higher-Level Specification and Verification with BDDs," *Computer-Aided Verification: Fourth International Workshop*, July 1992, reprinted in Lecture Notes in Computer Science Vol. 663, Springer-Verlag, published 1993.

[19] S.-W. Jeong, B. Plessier, G.D. Hachtel, and F. Somenzi, "Variable Ordering for FSM Traversal," *Proceedings of the International Workshop on Logic Synthesis*, MCNC, Research Triangle Park, NC, May 1991.

[20] David E. Long, personal correspondence.

[21] K. L. McMillan and J. Schwalbe, "Formal Verification of the Gigamax Cache-Consistency Protocol," *Proceedings of the International Symposium on Shared Memory Multiprocessing,* Information Processing Society of Japan, 1991, pp. 242–251.

[22] Christos H. Papadimitriou and Kenneth Steiglitz, *Combinatorial Optimization*, Prentice-Hall, 1982, p. 262.

[23] Carl Pixley, "A Computational Theory and Implementation of Sequential Hardware Equivalence," *Computer-Aided Verification: Second International Workshop*, 1990, published in Vol. 3 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1991, pp. 293–320.

[24] Herve J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines using BDD's" *IEEE International Conference on Computer-Aided Design,* 1990, pp. 130–133.