Efficient Verification with BDDs using Implicitly Conjoined Invariants *

Alan J. Hu and David L. Dill

Department of Computer Science, Stanford University

Abstract. Many researchers have reported that using Boolean decision diagrams (BDDs) greatly increases the size of hardware designs that can be formally verified automatically. Our own experience with automatic verification of high-level aspects of hardware design, such as protocols for cache coherence and communications, contradicts previous results; in fact, BDDs have been substantially inferior to brute-force algorithms that store states explicitly in a table.
We believe that new techniques are needed to realize the potential advantages of BDD-based verification at the protocol level. Here, we isolate several common causes of BDD-size blowup and show how these problems can be alleviated by a new verification approach based on partially evaluating the invariant being checked into an implicit conjunction of small BDDs. We describe the new method and give several examples of its application.

1 Introduction

With the increasing cost and complexity of hardware designs and protocols, formal verification techniques become ever more attractive. Boolean decision diagrams (BDDs) [3] have enabled much progress in this area, from the early work applying BDDs to verification [1, 6, 5, 8, 19] through the current work of numerous researchers.

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

We have developed the Murφ verification system specifically for this application domain, refining our design by verifying large, real examples (e.g. industrial multiprocessor cache coherence and link-level protocols) in addition to the usual academic examples (dining philosophers, alternating bit protocol, etc.) [10]. Murφ encompasses both a C++-based verifier and a high-level BDD-based verifier, called Ever, which supports integer and enumeration types, arrays, and records, a wide range of arithmetic, logical, and relational operators, and high-level imperative control structures (sequence, if-then-else) as well as non-determinism directly using standard BDDs [13]. (Although similar in spirit and motivation, Ever differs in this respect from other approaches to

* This research was supported by the Defense Advanced Research Projects Agency (contract number N00014-87-K-0828) and by a gift from Mitsubishi Electronics. The first author was supported by an ONR Graduate Fellowship. Most of this work was done using equipment generously donated by Sun Microsystems.

higher-level BDD-based verification, like MDDs [18, 15] or EVBDDs [16], which are extensions to the basic BDD data structure.)

To our initial surprise, the BDD-based verification method has been disappointing — a method which stores all of the reachable states explicitly in a hash-table substantially outperforms the BDD methods on our real examples. Moreover, existing techniques for efficient verification with BDDs hold no particular promise for alleviating these problems.

There are several sources of BDD-size blowup in our examples, each of which must be addressed if we are to have efficient verification of invariants. We have found a unified view of many sources of BDD-size blowup — that a single BDD must represent the conjunction of many small BDDs in such a way that the BDD for the conjunction is huge, regardless of variable-ordering — and propose a unified framework that eliminates BDD-size blowup in these cases by only partially evaluating the property being verified into a list of small BDDs that is *implicitly* conjoined, rather than explicitly building the large BDD for the conjunction, and maintaining these implicitly conjoined lists of small BDDs throughout the verification process.

2 Theoretical Basis

In modeling the system being verified, we generally assume an interleaving model of concurrency, both because such a model translates easily and efficiently into our underlying representation and also because that is the most intuitive model for the high-level distributed protocols we want to verify. The high-level Ever description is translated automatically into the underlying model of computation [13], which is a single non-deterministic finite-state machine with set of states Q, transition relation $\delta : Q \times Q \rightarrow \{0, 1\}$, and set of start states $S \subseteq Q$. The verification task is, given the set of states $I \subseteq Q$ that satisfies the invariant property being verified, to determine if there exists a path starting from a state in S and leading to a state not in I, and, if there is such a path, to output that path as a counterexample to the property being verified.

The usual approach to such a verification task is to compute the set of states reachable from S and to check that the set of reachable states is a subset of I (e.g. [8, 5, 7, 19, 4]). This approach entails computing the set of reachable states as the fixed-point $\mu Z.\lambda u.S(u) \vee \exists v[Z(v) \wedge \delta(v, u)]$, which is the smallest set Z such that $S \subseteq Z$ and any state that is a successor under δ of a state in Z is also in Z [6]. We will call this approach "forward traversal."

The expression $\exists v[Z(v) \wedge \delta(v, u)]$ is generally called the image of set Z under transition relation δ [9, 19], which we will denote by Image(δ, Z). Also commonly used is the image on the domain of δ of a subset of the codomain given by $\exists v[Z(v) \wedge \delta(u, v)]$, which we denote by PreImage(δ, Z). We will also be using the expression $\forall v[\delta(u, v) \Rightarrow Z(v)]$, which we will denote by BackImage(δ, Z). The BackImage operator behaves in some sense as the inverse of the Image operator because BackImage(δ, Z) is the largest set such that Image$(\delta,$ BackImage$(\delta, Z)) \subseteq Z$. (The PreImage is sometimes called the inverse image [19]. PreImage, however, has this "inverse of Image" behavior only when δ is a total function (deterministic), in which case PreImage and BackImage are the same.)

An important property of these image operators is that if δ is described in Ever (using high-level data structures, imperative semantics, and non-determinism) and the set Z is represented as a BDD (either built from an Ever description or computed by the verifier), we can compute the BDD for any of these images without building the BDD for δ, thereby avoiding a common cause of BDD-size blowup. For Image and PreImage, this computation is described elsewhere [13]. We easily compute BackImage(δ, Z) as $\neg$PreImage$(\delta, \neg Z)$ because negation of a BDD is constant-time for efficient BDD implementations [2].

An alternative to forward traversal is to note that we are essentially model-checking the CTL formula $S \Rightarrow \mathbf{AG}I$. Computing $\mathbf{AG}I$ is easily seen to entail computing the fixed-point $\nu Z.\lambda u.I(u) \wedge \forall v[\delta(u,v) \Rightarrow Z(v)]$, which is the largest set Z such that $Z \subseteq I$ and all successors of any state in Z are also in Z [6, 11]. The fixpoint exists by monotonicity of the predicate transformer. We will call this approach "backward traversal." Figure 1 gives pseudo-code for this computation, as well as for producing the counterexample if the verification fails.

```
Backward Traversal:
Let I_0 := I and i := 0.
Repeat
        Loop Invariant: For all i, set I_i is the set of all states x such that
                        any path from x to a state violating the invariant
                        must have length greater than i.
        If S ⇏ I_i, then print counterexample and exit. (Point A)
        I_{i+1} := I_0 ∧ BackImage(δ, I_i). (Point B)
        i := i + 1.
Until I_i = I_{i−1}. (Point C) Termination guaranteed by monotonicity.
Verification succeeds. Exit.

Counterexample:
Let P := S.
Loop
      Loop Invariant: There exists a path from P of length i leading to
                      an invariant violation, but no path of length i − 1.
      Choose a state x that satisfies P ∧ ¬I_i. (Point D)
      Print x.
      If i = 0, then x is a violating state. Exit.
      P := Image(δ, {x}).
      i := i − 1.
End Loop
```

Fig. 1. This algorithm checks that all computations from the start states S satisfy the invariant I. If the verification fails, the counterexample algorithm gives an execution that violates the invariant.

Clearly, the given backward traversal algorithm can be easily and efficiently implemented with BDDs, provided none of the BDDs becomes too large. As we've mentioned (and later as we'll demonstrate), an excessively large BDD is frequently needed to represent the conjunction of a set of properties, even if the BDD for each conjunct is, by itself, reasonably sized. We wish, therefore, to avoid evaluating the huge BDD needed to represent the conjunction. An obvious approach is to verify each conjunct independently as a separate invariant. This approach has very limited benefit, as we'll discuss later. Suppose, instead, that we represent the invariant being checked as the *implicit* conjunction of a set of small BDDs. Can we modify the backward traversal algorithm to avoid ever explicitly evaluating the BDD for the conjunction? Reviewing Figure 1, we have marked all points where evaluating the BDD for the conjunction might be necessary. Let us now consider these points one-by-one.

Point A is easy. Suppose I_i is represented by an implicitly conjoined list of small BDDs $I_i[1] \wedge \cdots \wedge I_i[n]$. Then $S \not\Rightarrow I_i$ if and only if there exists an $I_i[j]$ such that $S \not\Rightarrow I_i[j]$. We can therefore check for a violation of the implicitly conjoined invariant by checking each conjunct independently from the other conjuncts.

Point B consists of two operations — one straightforward, the other interesting. The straightforward operation turns out to be the BackImage computation, thanks to the following lemma:

Lemma 1.

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

Proof:

$$\begin{aligned}\text{BackImage}(\delta, Y \wedge Z) &= \forall v[\delta(u,v) \Rightarrow (Y(v) \wedge Z(v))] \\ &= \forall v[(\delta(u,v) \Rightarrow Y(v)) \wedge (\delta(u,v) \Rightarrow Z(v))] \\ &= \forall v[\delta(u,v) \Rightarrow Y(v)] \wedge \forall v[\delta(u,v) \Rightarrow Z(v)] \\ &= \text{BackImage}(\delta, Y) \wedge \text{BackImage}(\delta, Z). \quad \Box\end{aligned}$$

We can therefore compute the BackImage on each conjunct independently. For convenience, we will denote BackImage$(\delta, I_i[j])$ by $H_i[j]$, with $H_i = H_i[1] \wedge \cdots \wedge H_i[n]$.

The interesting part of Point B turns out to be the conjunction of I_0 with H_i, because there are myriad possibilities, all of them mathematically correct, but with different efficiency implications. On one extreme, for example, we could take the implicitly conjoined lists of BDDs representing I_0 and H_i and attempt to build the single BDD explicitly representing the conjunction of all of these small BDDs. Although this approach maintains the correctness of the backward traversal algorithm, it defeats our intention of using lists of implicitly conjoined BDDs to avoid potential BDD-size blowup. On the other extreme, we could simply concatenate the list of conjuncts representing I_0 with the list representing H_i, again maintaining the correctness of the algorithm, but in this case never building the BDD for any of the implied conjunctions. This choice has the undesirable property that the length of the list of conjuncts will grow on each iteration. Another option keeps this length constant on all iterations by setting I_{i+1} to be the implied conjunction $I_{i+1}[1] \wedge \cdots \wedge I_{i+1}[n]$ where each $I_{i+1}[j]$ is the BDD for $I_i[j] \wedge H_i[j]$ (explicitly evaluated as a BDD AND), essentially verifying each conjunct $I[j]$ of the invariant I completely independently of the other conjuncts. This choice has the advantage that it trivializes proving termination for Point C but the disadvantage that each conjunct ignores information from the other conjuncts.

A key insight at this point exposes further options at Point B. The correctness of the backward traversal algorithm depends only on the value of the implied conjunction of an entire list of conjuncts, not on the specifics of each conjunct. Therefore, if one conjunct is false at some state x, the whole conjunction is false at state x, and the values of all the other conjuncts at state x don't matter at all. Accordingly, we can simplify the BDD representation for each conjunct based on the other conjuncts. For example, to simplify conjunct $I_i[j]$ by conjunct $I_i[k]$ ($j \neq k$), we can replace $I_i[j]$ by any other Boolean function $\hat{I}_i[j]$ so long as $I_i[j] = \hat{I}_i[j]$ whenever $I_i[k]$ is true, and hopefully the BDD for $\hat{I}_i[j]$ is smaller than the BDD for $I_i[j]$. ($\neg I_i[k]$ can be viewed as a don't-care set to simplify Boolean function $I_i[j]$.) These simplifications have no effect on the correctness of the backward traversal algorithm, because they do not change the value of the (implied) conjunctions, but they can have a significant effect on the efficiency of the backward traversal algorithm. Obviously, there are many possible policies for applying these simplifications. We have achieved reasonable results with a very simple policy: we use a BDD simplification algorithm proposed by Coudert, Berthet, and Madre [8] to simplify one BDD using another as a don't-care, (Another choice is the constrain [9] or generalized cofactor [19] operator.) we compute $I_{i+1}[j] = I_i[j] \wedge H_i[j]$ as above, but then simplify each $I_{i+1}[j]$ by all $I_{i+1}[k]$ for $k < j$. Further research into conjunction and simplification policies may uncover other successful policies.

Testing for termination at Point C could conceivably be problematic, as implicitly conjoined lists of BDDs, unlike a single BDD, are not a canonical representation for Boolean functions, complicating testing for equality. Even proving convergence could be difficult, depending on how complex our choice of simplification policy at Point B is, since the don't-care simplification can introduce non-monotonicity in the sequence $I_0[j], I_1[j], \ldots$, for some values of j. (The entire conjunction, of course, forms a monotonic sequence $I_0, I_1, \ldots$, but we cannot build the BDD for the entire conjunction.) We have chosen a simple test for termination that checks that $I_i[j] = I_{i-1}[j]$ for all j, which is clearly a sufficient condition. In practice, this simple test has consistently performed correctly, even with very aggressive BDD-simplification policies.

Point D turns out to be straightforward. We know that there exists a path of length i from P that violates the invariant, but no path of any length less than i. Therefore, $P \wedge \neg I_i$ is non-empty, implying that there exists a j such that $P \wedge \neg I_i[j]$ is non-empty and that any $x \in P \wedge \neg I_i[j]$ is also in $P \wedge \neg I_i$. Again, we can perform this operation on each conjunct independently.

At this point, let's reconsider the obvious reduction of our original verification task $S \Rightarrow \mathbf{AG}(I[1] \wedge \cdots \wedge I[n])$ into the independent verification of $S \Rightarrow \mathbf{AG}I[j]$ for each j. This approach is essentially a special case of what we have just outlined, using the simple pairwise conjunction policy at Point B, and no BDD simplification whatsoever. At best, this simple approach gains only a factor of n in space at the expense of a factor of n in time, ignoring program overhead, over the method we propose, by serializing the independent verification tasks. Usually, however, the obvious method does worse, since the correctness of each invariant frequently relies on the correctness of the others. Therefore, without simplification, each independent verification must essentially derive the BDD for the entire conjunction, creating the BDD-size blowup we are attempting to avoid and also greatly increasing the run time. Of the examples we will consider later, the first and third exhibit nearly this worst-case behavior, giving all of the disadvantages of both the conventional forward and backward traversals with no advantages. Only the second example behaves reasonably using this obvious approach. We will not consider this approach further in this paper.

Putting all of the above together, we find that the backward traversal algorithm can indeed be converted to use implicitly conjoined lists of BDDs. The important questions, therefore, are how easy it is to write invariants in a form that takes advantage of this approach and whether this approach actually reduces BDD size. These questions we can answer only by testing examples.

3 Examples

In all of our examples, writing invariants that capitalize on this approach has been very easy and intuitive. Indeed, invariant-checking tends to require verifying that all of a set of invariants are correct, leading naturally to a list of properties whose conjunction can be left implicit. An optimization is to note that, as in the verification process earlier, only the value of the whole conjunction matters to the overall correctness, so we can use the conjuncts of the invariant to simplify each other. As a practical issue, we must carefully choose the order of simplifications to the original invariant because the counterexample facility tells the user which conjunct was violated, and a very aggressive simplification policy could result in the counterexample claiming that invariant j was violated, because the simplified $I[j]$ was violated, even though the original $I[j]$ was not. (This issue affects only the user-interface, not the correctness. The simplification never changes the value of the whole conjunction, so some other conjunct must have been violated.) We have chosen to simplify each $I[j]$ only by all $I[k]$ where $k < j$, precluding this user-interface confusion.

Another efficiency consideration is to leave the set of start states S as unconstrained as possible. Doing so makes S larger, reducing the length of the path required to establish that $S \not\Rightarrow I_i$. Again, this optimization tends to be easy and intuitive, as the user tends to have a feel for which state variables must be initialized (e.g. counters and status variables), and which can be allowed to start with an arbitrary value (e.g. data paths).

Let's look at some concrete examples.

3.1 Type Invariants

In all high-level BDD-based verifiers, integer types are encoded as bit vectors. [18, 15, 13] The most natural encoding scheme is simply to use the binary representation of the integer. In many instances, however, especially when performing high-level verification, the number of possible values is not exactly a power of 2. For example, in a communication protocol, a variable might indicate which one of 17 different message types is being sent, or another variable might contain parity-encoded data. Therefore, all legal states of the system will obey a set of properties like "$x < 17$" or "y has odd parity". We call these properties "type invariants".

By itself, each of these type invariants typically has a small BDD. The system as a whole, however, must satisfy all of the type invariants, leading to a BDD for the conjunction of all of them that may be very large, depending on the variable-ordering used. Consider, for example, a FIFO buffer of d words, each of which is k bits wide. (Such a buffer occurs frequently in verification models, both to model actual queues in a system and also to time-delay computed results to check an implementation against a differently timed specification.) Each word of the buffer must satisfy some property P that depend on most of the k bits. (A typical example is a subrange constraint.) Let w be the width of any cut of the BDD for P; for example, if P says that the word must be less than a constant, then $w = 2$. Furthermore, suppose we have interleaved the bits for the FIFO buffer, putting the high-order bit of all words first, followed by the next most significant bit of all words, etc. This variable ordering is generally necessary in datapaths to minimize BDD size for comparisons and arithmetic. [18, 15, 14] In this case, the BDD for the conjunction of all the type invariants is of size $O(kw^d)$, since in the BDD, after each bitslice, we must encode all w^d possible intermediate states of each type invariant, and there are k bitslices in all. If we attempt to perform a forward traversal on this system, we will be forced to build this exponentially-sized BDD, as any reachable state must satisfy all type invariants. Performing a conventional backward traversal requires building this exponentially-sized BDD just to start the verification process. Leaving the conjunction implicit, however, keeps the various type invariants separate throughout the verification process, thereby avoiding the BDD-size blowup.

A different approach to this same problem is to change the encoding scheme so that unused binary combinations become don't-cares by assigning multiple encodings to some values. [18, 15] This encoding eliminates the type invariants altogether, as every possible bit pattern encodes a legal value. This approach seems promising for small, low-level systems (for which it was designed) where we might be performing state encoding for a small controller. For larger systems and high-level verification, however, this approach has two major drawbacks. First, by complicating the encoding scheme, designing general algorithms for high-level operations like arithmetic, array-indexing, or comparison becomes impossible. Indeed, when using this don't-care-encoding approach, one must define specific operators (like addition) for each combination of different types. The second major drawback is that *checking* that the type invariants are satisfied becomes impossible, since we've already encoded them out of existence. Again, at the low-level this isn't an issue, but for high-level verification, checking that type properties hold (e.g. a counter doesn't go out-of-range) is essential.

Figure 2 summarizes results for this example. Clearly, the implicitly conjoined invariant greatly reduces both the memory and the time required for the verification.

3.2 Functionally Dependent Variables

In earlier work, [12] we isolated functionally dependent variables — variables that are always a function of other variables of the system, provided the system is operating correctly — as a common source of BDD-size blowup when verifying high-level concurrent systems. To understand why dependent variables cause problems, we must review the intuition behind size blowup in BDDs. The variables in a BDD must be totally ordered. Intuitively, blowup occurs when variables whose values are highly correlated are widely separated in this total order, since the BDD must make copies of nodes to "remember" the value of the earlier variable to guarantee that the later variable is consistent. Hence, achieving a good variable ordering, in which correlated variables are close together in the order, is essential for efficient use of BDDs. [3, 14] However, in some cases there is no good variable order, because placing two correlated variables close together may widely separate other correlated variables. Dependent variables have a high degree of correlation with the variables on which they depend (and possibly on other dependent variables). In a concurrent system with many dependent variables, it becomes difficult or impossible to order the variables so that the dependent variables are close to the variables upon which they depend, so the BDDs are large.

Concurrent systems typically have many dependent variables: individual processes in a concurrent system must maintain a "local image" of certain aspects of the global state of the system. For example, a cache entry is a local copy of a remotely-stored value. The cache entry is then functionally dependent on the remote value. For another example, suppose a process sends a message to a server and then enters a "wait for reply" state. In this case, the process is in the "wait for reply" state exactly when the original

	Forward			Backward			Backward with Impl. Conj.		
n	t	Mem	BDD	t	Mem	BDD	t	Mem	BDD
2	0:00	516K	31	0:00	484K	31	0:00	496K	9
3	0:00	644K	87	0:00	548K	87	0:00	560K	9
4	0:01	968K	223	0:00	680K	223	0:00	628K	9
5	0:02	1544K	543	0:01	1004K	543	0:00	692K	9
6	0:07	1996K	1279	0:03	1712K	1279	0:01	792K	9
7	0:20	1972K	2943	0:07	1968K	2943	0:01	1016K	9
8	0:51	3636K	6655	0:17	3672K	6655	0:02	1148K	9
9	2:10	6840K	14847	0:44	7000K	14847	0:03	1568K	9
10	5:04	13340K	32767	1:46	13468K	32767	0:03	1696K	9

Fig. 2. FIFO Buffer Example: Forward Traversal vs. Backward Traversal vs. Backward Traversal with Implicitly Conjoined Invariant. Each buffer word is 8-bits wide. Buffer depth ranges from 2 to 10 words. The type invariant is that each word is less than or equal to 128. Column label "n" indicates buffer depth, "t" gives run times in minutes and seconds, "Mem" is the total memory used by the entire program, and "BDD" is the number of nodes in the largest single BDD used to represent the set of states reachable in i steps or the set of states which cannot violate the invariant in i steps. (The total number of BDD nodes used by an implicit conjunction is at most a factor of n greater than the number shown.) The forward traversals required $n + 1$ iterations to terminate; the backward traversals, 1 iteration. All examples were run on a SUN 4/75. We are using a BDD package developed by David Long at CMU. [17]

message is in some transmission queue, a remote server process is in an appropriate state processing the request, or the reply message is in the server's transmission queue. The state of the process, therefore, is dependent on the states of the transmission queues and the server.

Under the restrictions that the transition relation δ can be represented as the non-deterministic choice between total transition functions and that the BDDs for these total transition functions are not too large, our previous work permits completely eliminating the functionally dependent variables from all BDDs, while checking to ensure that the system never violates the functional dependency. [12]

Alternatively, we can apply implicitly conjoined invariants to this problem. The BDD that specifies a functional dependency (e.g. that $y = f(x)$) by itself is usually quite small. The size blowup occurs because we must conjoin this functional dependency BDD with many other functional dependency BDDs as well as with any other properties of the system, thereby widely separating the closely correlated variables. If we leave this conjunction implicit, however, we can keep all of these small functional dependency BDDs separate, never explicitly building the huge BDD for the conjunction.

We test this approach on a simple network example. [12] Suppose we have a set of processors that non-deterministically issue requests into the network. Each request contains the processor's ID as a return address. A server non-deterministically pulls messages out of the network (modeling a non-order-preserving network), and places an acknowledgement back into the network. Eventually, the processor will receive the acknowledgement. When a processor sends a request, it increments a counter. When the processor receives an acknowledgement, it decrements the counter.

For simplicity, we will assume that there are n processors and that the network can hold n messages. Integers will be k bits, with $k > \log n$. Thus, we declare an array of processors numbered 0 through $n-1$. Each processor has a k-bit integer counter. We model the network as an n-element array (numbered 0 through $n-1$) of messages. Each message is a record with a valid bit, a request/acknowledge bit, and a k-bit integer return address. At each time step, any one processor can choose to send a request, or any one valid request in the network can be served (becoming an acknowledge), or any one acknowledgement in the network can be delivered back to the sender. If the system is correct, each processor's message counter will be equal to the number of messages from or to that processor in the network, making each processor's message counter a functionally dependent variable. The invariant to be checked is that these counts are correct at all times.

Results from this example are summarized in Figure 3. Both forward and backward traversal clearly result in exponential BDD-size blowup. The functionally dependent variable algorithm [12] keeps the maximum BDD size very small, but at the cost of very long run times. For this example, the implicitly conjoined invariant approach gives acceptably small BDDs and excellent speed, emerging as the clear winner over the other methods.

	Forward			Backward			Fwd. w/ Func. Dep.			Back. w/ Impl. Conj.		
n	t	Mem	BDD	t	Mem	BDD	t	Mem	BDD	t	Mem	BDD
1	0:00	504K	11	0:00	512K	10	0:00	472K	7	0:00	512K	10
2	0:00	656K	60	0:00	624K	51	0:00	556K	16	0:00	632K	21
3	0:01	1072K	268	0:01	976K	217	0:04	748K	37	0:01	984K	36
4	0:04	1904K	1129	0:03	1776K	925	0:14	1140K	41	0:02	1528K	58
5	0:17	3680K	4937	0:09	2272K	3671	0:39	1868K	91	0:06	2152K	83
6	1:22	11604K	20656	0:29	6612K	14627	1:34	2024K	106	0:10	2140K	114
7	12:32	29520K	85216	2:31	15280K	58430	3:18	3444K	169	0:18	3896K	149
8				>20min. and >50MB			7:31	3528K	109	0:28	4212K	191
9							12:56	6596K	271	0:42	7712K	236
10							24:19	6324K	276	0:58	8080K	287
11							33:31	12588K	397	1:23	13768K	342

Fig. 3. Network Example: Forward Traversal vs. Backward Traversal vs. Forward Traversal Exploiting Functional Dependencies [12] vs. Backward Traversal with Implicitly Conjoined Invariant. Column n indicates the size of the network; other columns are labeled as in Figure 2. The forward traversals required $2n+1$ iterations to terminate; the backward traversals, 1 iteration.

3.3 Assisted Verification

A moving average filter is a common building block in digital signal-processing algorithms. The filter continuously computes the moving average of the last n samples seen. For this example, we compare an implementation using a pipelined tree of adders (used in practice to trade longer latency for higher throughput) to compute the moving average of the samples against a specification that computes the average combinationally in a single clock cycle and then buffers the result to match the pipeline depth of the

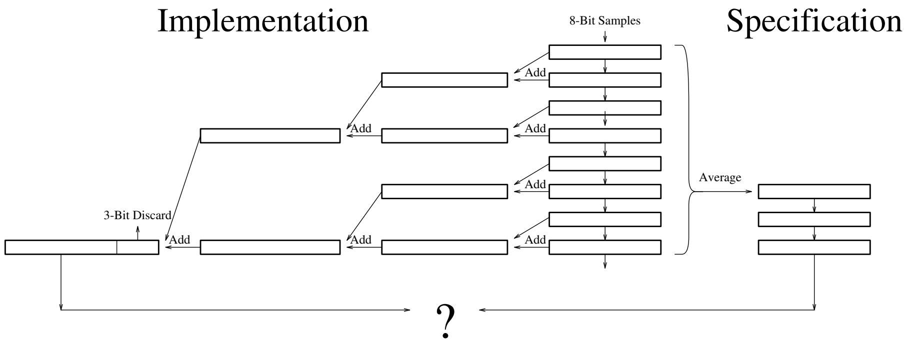

Fig. 4. Diagram of Size 8 Moving Average Filter. The verification task is to prove that the implementation using a pipelined tree of adders gives the same result as the specification, which is the average computed directly and then delayed in a FIFO to match the pipeline depth of the implementation.

implementation. See Figure 4. The verification task is to prove that the outputs of the implementation and of the specification always agree. We will assume 8-bit samples and will attempt to verify filters of size 4, 8, and 16.

As shown in the first two columns of Figure 5, neither a direct forward or backward traversal can handle the larger instances of the problem. Indeed, the forward traversal

	Forward			Backward			Backward with Impl. Conj.		
n	t	Mem	BDD	t	Mem	BDD	t	Mem	BDD
4	0:55	7424K	9453	0:06	1824K	373	0:05	1760K	82
8	>80MB			>30min.			0:47	6212K	351
16	>80MB			>30min.			6:35	27284K	1433

Fig. 5. Moving Average Filter Example: Forward Traversal vs. Backward Traversal vs. Backward Traversal with Implicitly Conjoined “Assisting” Invariants. Column n indicates the size of the filter (number of samples being averaged); other columns are labeled as in Figure 2. For $n = 4$, the forward traversal required 3 iterations to terminate; the backward traversal, 2 iterations. The backward traversal with assisting invariants required only 1 iteration for all filter sizes tested.

was not able to complete even the size 4 filter unless the set of start states was relaxed to allow arbitrary data to start in the sample buffer. All results in Figure 5 reflect the relaxed starting condition, although doing so only slightly affected the backward traversals.

In this example, the specified invariant does not obviously form itself into an implicit conjunction that reduces verification complexity. Nothing prohibits us, however, from *adding* additional invariants, which we will implicitly conjoin with the specified invariant. Note that not only must the outputs of the implementation and specification match, but each entry of the FIFO buffer in the specification must equal the average of the corresponding layer of the adder tree. If we explicitly conjoin these additional constraints into the invariant, the BDD for the invariant becomes too large. But, if we combine these additional constraints with the original invariant as an implicit conjunction, the verification completes quickly and efficiently, because the conjunct that gives the invariant for one layer of the filter greatly simplifies the conjunct for the next layer. The third column of Figure 5 shows the enormous improvement in time and space efficiency. We have, therefore, a natural mechanism for the user to assist the verifier by introducing additional information.

Furthermore, we believe we should be able to generate these "assisting" conjuncts automatically. For a pipelined system like this example, the conjunct generated by the BackImage (at Point B of Figure 1) of a property at one level of the pipeline is essentially a property at the previous level of the pipeline. Therefore, the verification algorithm is automatically generating these assisting conjuncts during each iteration. In the current implementation, this information is lost because of the policy of not increasing the size of the set of conjuncts on each iteration, but perhaps a clever choice of when to simplify one conjunct against another, when to evaluate a conjunction, and when to allow the set of conjuncts to grow could capitalize on this situation automatically.

4 Conclusion and Future Work

We have isolated several causes of BDD-size blowup and have developed implicitly conjoined invariants as an efficient new framework for invariant-checking to address these causes. Using this method, we have been able to verify designs that are intractable under other BDD-based approaches.

Many lines of further research suggest themselves. Clearly, the strategy used to decide which conjunctions to evaluate and which conjunctions to leave unevaluated and the strategy of which conjuncts to simplify are obvious areas for study. An intelligent heuristic could potentially generate assisting invariants automatically, greatly improving the efficiency of the verification process. Even in the absence of such automated breakthroughs, a deeper understanding of when implicit conjunctions pay off should greatly aid the user of the verification tool.

The problem of simplifying one BDD using another BDD as a don't-care may still have room for further results. While we have been happy with the operator proposed by Coudert, Berthet, and Madre [8], a better simplification operator would immediately improve the performance of the verification algorithm described here.

On a more theoretical level, proving termination for a given strategy of conjoining and simplifying is an open problem, as is the alternative of finding other efficient tests for termination. In practice, termination has not been an issue.

In sum, our preliminary results are very encouraging. We hope that further research along these lines will enable the potential advantages of BDD-based verification to apply to a much wider range of problems.

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. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. Alan J. Hu and David L. Dill, "Reducing BDD Size by Exploiting Functional Dependencies," *30th ACM/IEEE Design Automation Conference*, 1993, to appear.
13. 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, to be reprinted in Springer-Verlag's LNCS Series.
14. 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.
15. Timothy Y. K. Kam and Robert K. Brayton, "Multi-Valued Decision Diagrams," UCB/ERL M90/125, December 1990.
16. Yung-Te Lai and Sarma Sastry, "Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification," *29th ACM/IEEE Design Automation Conference*, 1992, pp. 608–613.
17. David E. Long, personal correspondence.
18. Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton, "Algorithms for Discrete Function Manipulation," *IEEE International Conference on Computer-Aided Design,* 1990, pp. 92–95.
19. 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.