# Reducing BDD Size by Exploiting Functional Dependencies *

Alan J. Hu and David L. Dill
Department of Computer Science, Stanford University

*Abstract*—**Many researchers have reported that the use of 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 will be needed to realize the potential advantages of BDD verification at the protocol level. Here, we identify *functionally dependent variables* as a common cause of BDD-size blowup, and describe new techniques to avoid the problem. Using the improved algorithm, we reduce an exponentially-sized problem to a provably $O(n \log n)$-sized one, achieving several orders of magnitude reduction in BDD size.**

## I 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, starting from early work applying BDDs to verification [1, 6, 5, 9, 18] and continuing through the current work of numerous researchers.

Most of the current research on automatic formal hardware verification has focussed 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 Mur$\varphi$ 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.) [11]. Mur$\varphi$ encompasses both a C++-based verifier and a high-level BDD-based verifier, called Ever, which supports integer and enumeration types, arithmetic, arrays, records, programming-language-like control structures, and non-determinism directly using standard BDDs [12]. [1]

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, and 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. The focus of this paper is one of these, which is crucial to high-level descriptions of concurrent systems: *functionally dependent variables*, i.e., a variable which is always a function of other (independent) variables of the system.

To understand why dependent variables can cause problems, we first review the intuition behind size blowup in BDDs. The Boolean 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. (The BDD must have many copies of nodes to "remember" the value of the early variable in order to make sure that value is consistent with the value of the later variable.) Hence, achieving a good variable ordering, in which correlated variables are close together in the order, is essential for efficient use of BDDs [3, 13]. 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

*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.

[1] Although similar in spirit and motivation, Ever differs in this respect from other approaches to higher-level BDD-based verification, like MDDs [17, 14] or EVBDDs [15], which are extensions to the basic BDD data structure.

requesting a simple service from another process, and then enters a state waiting for an reply message. In this case, the process is in the "wait for reply" state exactly when the original 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. In this case, the state variable which stores the "wait for reply" state locally is functionally dependent on the state variables encoding the queues and the state of the server process.

The few case studies of successful protocol verification with BDDs also illustrate the effect of dependent variables, but by their relative absence. In verifying the cache consistency protocol of the Encore Gigamax, for example, McMillan and Schwalbe's careful choice of abstraction as well as the bus-snooping nature of the protocol combine to create a description of the system with few dependent variables [16]. Similarly, the Controller Area Network protocol verified by Chiodo *et al.* has almost no depedent variables because it relies on a broadcast bus with automatic collision detection, and the individual nodes have no information about the rest of the system [8]. Academic examples like dining philosophers also tend not to have many dependent variables because each process has very little state and behaves largely independently of the rest of the system. If BDD-based verification is to apply to a broader range of problems, dealing with functionally dependent variables is a must.

We present a new idea for reducing the explosive effects of dependent variables. The user declares the dependent variables as a function of the independent variables. During verification, the BDD representing the set of reachable states contains only the *independent* variables. (The values of dependent variables are implicit from the user-supplied function.) The fundamental difficulty of this approach is that, typically, the dependent variables are only really dependent when the protocol is error-free. If the protocol malfunctions due to a description or design error, the nature of the problem is often a discrepancy between the dependent and independent variables (e.g., the local image of the global state is not a true image). Hence, the method must check at each step of verification whether the functional dependency continues to hold. In other words, the method *assumes* that the functional dependency holds, thereby realizing a substantial BDD-size savings, while *simultaneously verifying that the assumption of functional dependency was correct.*

## II Theoretical Basis

We first need to formalize our notion of a functionally dependent variable.

**Definition 1 (Functionally Dependent Variable)** *A BDD variable $y$ is functionally dependent on a set of BDD variables $x_1, \ldots, x_n$ iff there exists a function $f$ such that $y = f(x_1, \ldots, x_n)$. For notational convenience, we extend this definition in the obvious way for a vector of BDD variables $\vec{y}$: $\vec{y}$ is functionally dependent on $\vec{x}$ iff there exists a vector-valued function $f$ such that $\vec{y} = f(\vec{x})$.*

We will also need a brief overview of the Ever verifier. Details are available elsewhere [12]. Ever models the system being verified as a non-deterministic state machine. The user defines the state variables of the machine using typical data-structuring constructs like integers, enumerations, records, and arrays, which Ever converts to BDDs. (Intuitively, Ever allocates BDD variables the way a normal compiler allocates bits of memory.) Ever supports a large set of arithmetic, logical, and relational operators on these data structures, essentially as a normal programming language would. For describing the next-state relation, Ever provides normal imperative semantics; the user defines the next-state relation of the machine by writing ordinary-looking code: assignment statements, compound statements, if-then-else statements, and non-deterministic choice. Ever can either build the BDD for the next-state relation described in this manner, or can also leave the description in its high-level form, performing the verification without ever building the BDD explicitly (a common source of BDD-size blowup).

The verification task is, given the high-level description of a possibly non-deterministic state machine, to traverse all the reachable states checking for any violations of the verification conditions. If a violation is found, print an execution trace that leads from a start state to the violating state. Algorithms for this task are standard and can be found in several sources [9, 1, 5, 6, 18]. For concreteness, let us define the state variables to be $\vec{x}$ and $\vec{y}$ (each a vector of BDD variables), the set of start states to be $S(\vec{x}, \vec{y})$, the possibly non-deterministic next-state relation to be $N(\vec{x}, \vec{y}, \vec{x'}, \vec{y'})$, and the set of states violating the verification condition to be $V(\vec{x}, \vec{y})$. In this notation, our goal becomes, given a functional relationship $\vec{y} = f(\vec{x})$, to eliminate the BDD variables for $\vec{y}$, thereby reducing BDD size, while simultaneously proving that $\vec{y} = f(\vec{x})$ for all reachable states.

At first glance, this task appears impossible: to reduce BDD size, we must eliminate the BDD variables for $\vec{y}$ by assuming $\vec{y} = f(\vec{x})$, but if we assume $\vec{y} = f(\vec{x})$, how can we possibly check that the assumption is correct? Furthermore, even if the functional dependency is true, at intermediate points in the next-state relation, $\vec{y}$ need not be a function of $\vec{x}$ at all, frustrating attempts to check the functional dependency step-by-step through the description of the next-state relation. For example, suppose the functional dependence is that $\vec{y} = \vec{x}$, and the next-state relation includes the following code:

```
x := 0; -- Assignment 1
-- Additional code could go here.
y := 0; -- Assignment 2
```

After the first assignment, $\vec{y}$ no longer necessarily equals $\vec{x}$. Indeed, at that point, $\vec{y}$ may not even be any function of $\vec{x}$, since $\vec{x}$ must be 0, whereas $\vec{y}$ can still have any value it used to have. After the second assignment, however, the functional dependence holds once again.

The solution comes through the use of Boolean functional vectors [10]. A Boolean functional vector represents a function from vectors of Boolean values to vectors of Boolean values. Each bit of the output vector of the function is represented by a BDD whose variables represent elements of the input vector of the function.

Suppose we are verifying a deterministic automaton with next-state function $N$. In that case, we could theoretically compute a Boolean functional vector for the next state as a function of the current state: $\vec{x'} = \delta_x(\vec{x}, \vec{y})$ and $\vec{y'} = \delta_y(\vec{x}, \vec{y})$. We can therefore both check and assume the functional dependency as follows:

**Theorem 1** *Suppose $R_i(\vec{x})$ is the set of states reachable in $i$ or fewer steps from the start states, and that the deterministic next-state relation has next-state Boolean functional vectors $\delta_x(\vec{x}, \vec{y})$ and $\delta_y(\vec{x}, \vec{y})$. If the functional dependency $\vec{y} = f(\vec{x})$ is true for $R_i(\vec{x})$, then the functional dependency is also true for $R_{i+1}(\vec{x})$ iff*

$$R_i(\vec{x}) \Rightarrow \left[\delta_y(\vec{x}, f(\vec{x})) = f(\delta_x(\vec{x}, f(\vec{x})))\right] .$$

**Proof:** Since the functional dependency holds for $R_i(\vec{x})$, we can substitute $f(\vec{x})$ for $\vec{y}$ when computing the values of the next-state functions. The result follows trivially by substitution into the expression $R_i(\vec{x}) \Rightarrow [\vec{y'} = f(\vec{x'})]$, where the primed variables denote the value of the variable in the next state. □

The next-state relation $N$ is not, however, necessarily deterministic. We need, therefore, to extend the concept of Boolean functional vectors to handle non-determinism.

**Definition 2 (MOCB)** *A* **multi-set of conditional Boolean functional vectors** *(MOCB) is a multi-set of pairs $(c_i, a_i)$, where the condition $c_i$ is a Boolean-valued expression, and the action $a_i$ is a Boolean functional vector. The value of a MOCB is defined to be the non-deterministic choice of one action $a$ from the set of all actions $a_i$ for which the corresponding condition $c_i$ is true.*

**Definition 3 (MOB)** *A* **multi-set of Boolean functional vectors** *(MOB) is a MOCB in which all conditions $c_i$ are identically equal to true.*

Note that a next-state relation represented as a MOB must be total on its domain, since Boolean functional vectors are total functions. To handle a general, non-deterministic next-state relation, which can be partial, we need to use the more cumbersome MOCB. For example, in Ever, it is easy to write a next-state relation for which some states have no successors. On the other hand, for any description language based on guarded commands or asynchronous interleaving, such as UNITY [7] or Mur$\varphi$ [11], provided we use self-loops instead of successorless states for null actions, we can use the simpler and more efficient MOB, since the underlying model of computation becomes the non-deterministic choice of deterministic actions. We will not consider MOCBs in the remainder of this paper.

With this extension of Boolean functional vectors, we can now present the improved verification algorithm:

**Algorithm 1 (Invariant Checking with Functionally Dependent Variables)** *Let the state variables be $\vec{x}$, with functionally dependent variables $\vec{y}$ defined as $\vec{y} = f(\vec{x})$. Define the set of start states to be $S(\vec{x})$ and the possibly non-deterministic next-state relation to be $N(\vec{x}, \vec{y}, \vec{x'}, \vec{y'})$. Let the set of states violating the verification condition be $V(\vec{x}, \vec{y})$.*

1. *Compute the MOB for $N(\vec{x}, \vec{y}, \vec{x'}, \vec{y'})$.*
2. *Let $R_0(\vec{x}) = S(\vec{x})$.*
3. *Loop*
   (a) *Check $R_i(\vec{x}) \Rightarrow \neg V(\vec{x}, f(\vec{x}))$. If not, generate counter-example trace.*
   (b) *For each Boolean functional vector $(\delta_x, \delta_y)$ in the MOB, check that Theorem 1 holds:*

   $$R_i(\vec{x}) \Rightarrow \left[\delta_y(\vec{x}, f(\vec{x})) = f(\delta_x(\vec{x}, f(\vec{x})))\right] .$$

   *If not, generate counter-example trace.*
   (c) *$R_{i+1}(\vec{x}) = R_i(\vec{x}) \vee Image(R_i, N)(\vec{x})$.*
   (d) *If $R_{i+1}(\vec{x}) = R_i(\vec{x})$, terminate successfully.*

As can be seen, this algorithm is essentially standard and is open to the standard optimizations. The only differences are the generation of the MOB, and the step to check that the functional dependency will be preserved on each iteration.

## III IMPLEMENTATION

We have implemented the above algorithm as an extension to the Ever high-level BDD-based verifier. Ever's techniques for handling high-level data and control structures are described elsewhere [12]. We are using Brace *et al.*'s BDD implementation [2].

To use functionally dependent variables, the user simply declares a variable to be functionally dependent, giving the function that specifies the dependency. From that point on, the verifier handles everything automatically. The interesting implementation issues are how to build the MOB and how to build the Boolean functional vectors within the MOB.

We must first ensure that the next-state relation corresponds to a small MOB, not requiring the more general MOCB. To do so, we restrict the Ever expressions used to define the next-state relation as follows:

- On any branch of the parse tree for the next-state relation, all non-deterministic choice expressed as a disjunction between sub-relations must occur first. This restriction keeps the degree of non-deterministic choice reasonably small.

- A (sub-)next-state relation must not evaluate to false. To express a null action, use a self-loop, denoted by the keyword `CurNextEq`. This restriction, along with the next one, precludes writing partial next-state relations.
- All assignments must be deterministic. Eliminating non-deterministic assignment both discourages descriptions resulting in very large MOBs and also closes a loop-hole that would have allowed writing a partial next-state relation.

With these restrictions, the next-state relation is essentially a tree with deterministic next-state relations at the leaves. Hence, building the MOB consists simply of traversing the disjunctions at the top levels of the parse tree, collecting the Boolean functional vectors built for each deterministic next-state sub-relation.

Building a Boolean functional vector for a deterministic next-state relation is a more complex computation. Permissible next-state relations are recursively built from self-loops, assignment statements, sequentiality, and conditionals. The basic idea is to start with an initial Boolean functional vector for the current state and to push it through the description of the next-state relation. More formally, let $N$ be the description of the next-state relation. Let $\delta_0$ be the Boolean functional vector for the current state in terms of the current state variables, i.e. $\delta_0 = (\vec{x}, f(\vec{x}))$. The Boolean functional vector for next-state relation $N$ is given by BFV$(\delta_0, N)$, where the function BFV is defined as follows:

**Self-Loop:** If $N$ is "`CurNextEq`", then BFV$(\delta, N) = \delta$.

**Assignment:** If $N$ is an assignment statement, then BFV$(\delta, N)$ = ASSIGN$(\delta, N)$, where ASSIGN computes the correct Boolean functional vector for an assignment statement (described below).

**Sequence:** If $N$ is a sequence of statements "$(N_1; \ldots; N_n)$", then BFV$(\delta, N)$ = BFV(BFV$(\delta, N_1), (N_2; \ldots; N_n))$.

**Conditional:** If $N$ is a conditional statement "if $c$ then $N_1$ else $N_2$", let $c_\delta$ be the BDD for the condition computed using $\delta$ to give the current value of all variables, let $\delta_1$ = BFV$(\delta, N_1)$, and let $\delta_2$ = BFV$(\delta, N_2)$. In this case, BFV$(\delta, N)$ = if $c_\delta$ then $\delta_1$ else $\delta_2$, where the if-then-else applies bitwise to each function in the Boolean functional vectors.

The ASSIGN function is similar to the BECOMES function used in Ever to compute the next-state relation for an assignment [12], so we will not describe here the details of evaluating the right and left-hand sides of the assignment. First the function computes the value of the right-hand side using the given Boolean functional vector to supply the current value of all variables. Call this value $r$. (Value $r$ can be arbitrary-sized, expressed as a vector of BDD functions.) For each variable $v$, compute condition $c_v$ that variable $v$ will be assigned to. For example, if we have the assignment statement "`a[i].field1[j] := 42`", then the condition for the bits representing `a[3].field1[5]` will be $(i = 3) \wedge (j = 5)$, whereas the condition for the bits representing `a[3].field2[5]` will simply be `false`. If we denote the input Boolean functional vector for a variable $v$ as $\delta_{\text{in}}[v]$ and the resulting Boolean functional vector for $v$ as $\delta_{\text{out}}[v]$, the resulting Boolean functional vector is:

$$\delta_{\text{out}}[v] = \text{if } c_v \text{ then } r \text{ else } \delta_{\text{in}}[v]$$

for all variables $v$.

A final implementation issue is the computation of the image of a set of states through the next-state relation. One possibility, since we've already computed Boolean functional vectors for the next-state relation, is the straightforward extension of the Boolean functional vector image computation techniques [9, 18] to handle the non-determinism of MOBs. In our examples, however, we have identified a special case that allows a simpler solution. If the description of the next-state relation of the system does not rely on the updated value of functionally dependent variables, assignments to functionally dependent variables are no-ops from this perspective. Hence, we can use the existing next-state calculation (which allows image computation without explicitly building the BDD for the next-state relation) by simply ignoring those assignments. (Of course, the assignments are *not* ignored when computing the MOB for the next-state relation.)

## IV BENCHMARK EXAMPLE

In many of our industrial examples, we need to model a communication network that doesn't preserve message order. In addition, individual processors keep track of what messages they have outstanding in the network. Clearly, each processor's message status is a function of the network. Furthermore, the correctness of each processor's status is a crucial component of the overall correctness of the system. Thus, this situation perfectly illustrates functionally dependent variables arising in practice.

Our benchmark example distills the essential properties from the example above. 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 there are $n$ processors and that the network can hold $n$ messages. Integers will be $k$ bits, with

$k > \log_2 n$. 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.[2] 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 we don't exploit the functional dependency, the BDD for the set of reachable states will clearly be exponential for any straightforward variable ordering. For instance, if the processor counts come first, any truth assignment such that the sum of the counts is less than or equal to $n$ is potentially possible, giving rise to a fooling set of size $\binom{2n}{n}$, which is exponential. [4] We conjecture that the BDD will be exponentially sized for any variable ordering.

In contrast, if we declare the processor counts as functionally dependent variables, we can prove low-order polynomial bounds on the sizes the BDDs for the set of reachable states.

**Theorem 2** *The size of the BDD for the set of all reachable states ($R_\infty$ in Algorithm 1) is $n(c(n)+1)$, where $c(n)$ is the number of BDD nodes needed to show that an integer is less than $n$. Since the BDD size for integer comparison is at most the number of bits in the integer, $c(n) \leq \lceil \log_2 n \rceil$. In practice, $c(n)$ is bounded above by a constant, the logarithm of the largest possible value of $n$.*

**Proof:** Any state is reachable provided that for each element in the network array, valid $\Rightarrow$ (src $< n$), which is clearly size $c(n)+1$. Multiplying by $n$ for the $n$ array elements gives the result. □

**Theorem 3** *The size of the largest BDD for an intermediate reachable set (any $R_i$ in Algorithm 1) is $O(n^2 \log n)$. In practice, we can again consider the $\log n$ factor to be a constant.*

**Proof:** After $i$ iterations, at most $i$ "events" could have happened and still be visible, where an event is either the creation of a new request or the servicing of a request (making it an acknowledge). At iteration $i$, any state is reachable as long as it obeys the bound given in the preceding theorem, and as long as the state doesn't require more than $i$ events to reach. Any state is reachable in $2n$ events, so $R_i$ can always be represented by a BDD smaller than $n$ copies of the the $n \log n$-sized BDD given in the preceding theorem. (There is one copy for each possible number of events that has been seen in the BDD so far. If $i > n$, there is one copy for each possible number of events that haven't been seen in the BDD so far.) Thus, this bound is a factor of $n$ larger than the preceding bound. □

[2] The specific choice of data structure can greatly affect BDD size, an interesting result of the increased expressive power of a higher-level verifier. Instead of having a valid bit for each entry, a separate integer count of valid entries in the array would use fewer BDD variables. Doing so, however, results in exponential blowup in BDD size, regardless of whether or not the array entries are interleaved.

| | Conventional Traversal | | | Algorithm 1 | | |
|---|---|---|---|---|---|---|
| $n$ | Time | max $R_i$ | $R_\infty$ | Time | max $R_i$ | $R_\infty$ |
| 1 | 0:00 | 11 | 10 | 0:00 | 6 | 5 |
| 2 | 0:02 | 64 | 52 | 0:01 | 15 | 8 |
| 3 | 0:08 | 286 | 207 | 0:05 | 36 | 15 |
| 4 | 0:32 | 1197 | 787 | 0:11 | 40 | 12 |
| 5 | 2:36 | 5187 | 3103 | 0:27 | 90 | 25 |
| 6 | 12:01 | 21578 | 12176 | 0:51 | 105 | 24 |
| 7 | | | | 1:34 | 168 | 35 |
| 8 | | | | 2:37 | 108 | 16 |
| 9 | | | | 4:15 | 270 | 45 |
| 10 | | | | 8:30 | 275 | 40 |
| 11 | | | | 13:07 | 396 | 55 |
| 12 | | | | 21:37 | 312 | 36 |
| 13 | | | | 43:46 | 546 | 65 |
| 14 | | | | 128:39 | 525 | 56 |

Table 1: The "Conventional Traversal" columns are runs in which the processor counts were not declared to be functionally dependent. Times are given in minutes and seconds. Times given for Algorithm 1 take advantage of processor symmetry. Without using symmetry, times are less than a factor of $n$ slower. (BDD sizes are not affected.) Columns labeled max $R_i$ and $R_\infty$ are the sizes in BDD nodes of largest $R_i$ encountered and the final reachable $R_i$. For $n = 7$, the conventional traversal could not complete; the last $R_i$ it generated had $79,568$ nodes. All results are from a Sun 4/75.

The actual runs confirm these theoretical results. We used 4-bit integers, so the BDD-size upperbounds from Theorems 2 and 3 are $5n$ and $5n^2$. The difference between linear or quadratic and exponential size is quite striking. Interestingly, although Algorithm 1 appears to be trading off time for space (since it's processing each deterministic choice separately), it actually runs faster, since the BDDs in the conventional approach are so large and the time for a BDD operation is a function of the size of the BDD. The results are summarized in the Table 1 and in Figs. 1 and 2.

## V Future Work and Conclusion

We have identified a common cause of BDD-size blowup and implemented a solution. Preliminary experiments are promising, reducing exponentially-sized BDDs to linear or quadratic ones. Much work remains to be done, however. We must test the method on a wider range of real problems to assess how generally useful it really is. Many implementation optimizations could be done to improve performance. More fundamentally, further research is required on the restrictions on non-determinism imposed by using MOBs instead of MOCBs. We need to show either that MOBs are sufficiently expressive for a wide range of problems or that the techniques described here can be efficiently extended to handle MOCBs.

Just as BDDs are no panacea, functionally dependent variables are no panacea, either. There are many other reasons why a BDD might become excessively large. On the other

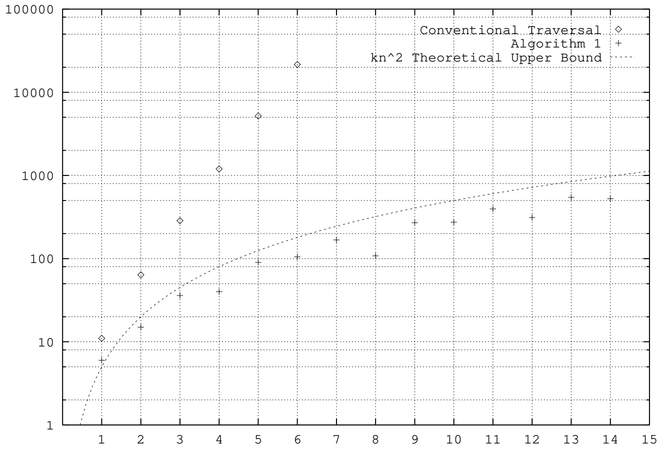


Figure 1: Plot of largest intermediate reachable BDD size on a log-scale. "Conventional Traversal" doesn't take advantage of functional dependencies and clearly shows exponential growth. The theoretical bound is from Theorem 3.

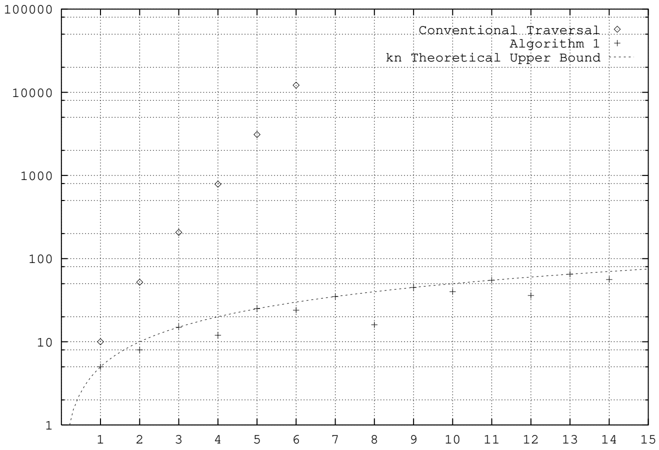


Figure 2: Plot of final reachable BDD size on a log-scale, again clearly showing the exponential growth of the conventional approach. The theoretical bound is from Theorem 2.

hand, functionally dependent variables are a common cause of BDD-size blowup, and the techniques presented here pay off dramatically, promising to extend the range of applicability of BDD-based verification. Although exploiting functional dependencies is not a cure-all, it is clearly a necessary step.

## 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] Randal E. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication," *IEEE Transactions on Computers*, Vol. 40, No. 2 (February 1991), pp. 205-213.

[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] K. Mani Chandy and Jayadev Misra, *Parallel Program Design,* Addison-Wesley, 1988.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Timothy Y. K. Kam and Robert K. Brayton, "Multi-Valued Decision Diagrams," UCB/ERL M90/125, December 1990.

[15] 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.

[16] 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.

[17] 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.

[18] 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.