

# Managing Space for Finite-State Verification\*

Jianbin Tan, George S. Avrunin, Lori A. Clarke  
Department of Computer Science  
University of Massachusetts  
Amherst, Massachusetts 01003, USA  
{tjb, avrunin, clarke}@cs.umass.edu

## ABSTRACT

Finite-state verification (FSV) techniques attempt to prove properties about a model of a system by examining all possible behaviors of that model. This approach suffers from the state-explosion problem, where the size of the model or the analysis costs may be exponentially large with respect to the size of the system. Approaches that use symbolic data structures to represent the examined state space appear to provide an important optimization. In this paper, we investigate applying two symbolic data structures, Binary Decision Diagrams (BDDs) and Zero-suppressed Binary Decision Diagrams (ZDDs), in two FSV tools, LTSA and FLAVERS. We describe an experiment showing that these two symbolic approaches can improve the performance of both FSV tools and are more efficient than two other algorithms that store the state space explicitly. Moreover, the ZDD-based approach often runs faster and can handle larger systems than the BDD-based approach.

## Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*formal methods, model checking, validation*

## General Terms

Verification

---

\*This research was partially supported by the National Science Foundation under grant CCF-0427071 and grant CCR-0205575, by the U.S. Army Research Laboratory and the U.S. Army Research Office under agreement DAAD190110564, and by the U.S. Department of Defense/Army Research Office under agreement DAAD190310133. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation, the U.S. Army Research Office or the U.S. Department of Defense/Army Research Office.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE-28, May 20-28, 2006, Shanghai, China.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

## Keywords

Finite-state verification, BDD, ZDD, LTSA, FLAVERS

## 1. INTRODUCTION

Finite-state verification (FSV) approaches attempt to prove properties about a model of a system. These approaches are not as general as theorem-proving, but they are usually easier to use. They do suffer, however, from the so-called state-explosion problem, where the size of the model or the analysis costs may be exponentially large with respect to the size of the system being analyzed.

There are a number of different techniques for dealing with this state-explosion problem. Symbolic approaches that use certain symbolic data structures such as Binary Decision Diagrams (BDDs) to represent the state space of a system seem to be promising and have been demonstrated to often be effective in hardware verification [8, 15]. A symbolic approach encodes the verification problem as Boolean functions and then uses symbolic data structures to represent those Boolean functions.

In this paper we report on our exploration of two symbolic data structures, BDDs and Zero-suppressed Binary Decision Diagrams (ZDDs), applied to two different FSV tools, LTSA [14] and FLAVERS [12]. We describe how the systems to be verified are encoded as Boolean functions and how the verification problem is solved via the Boolean operations. Moreover, we present a BDD-based algorithm and a novel ZDD-based algorithm for implementing a key Boolean operation used in the verification. We demonstrate that these two symbolic approaches can improve the performance of both FSV tools and are more efficient than the tools' native algorithms that store the state space explicitly. In addition, we find that on average, the ZDD-based approach uses only 60% as much time as the BDD-based approach in LTSA and 71% in FLAVERS, and can often handle larger systems than the BDD-based approach. Based on these experimental results, we believe that it would be worthwhile to explore applying this ZDD approach to other FSV techniques.

In the next two sections of this paper, we provide an overview of LTSA and FLAVERS so that the reader can understand how the verification problem is encoded using Boolean functions and solved by Boolean operations. Sections 4 and 5 describe how BDDs and ZDDs, respectively, are applied in these two FSV tools. Section 6 discusses our experimental methodology and Section 7 presents and analyzes the experimental results. Section 8 describes related work, and we conclude in Section 9 with a summary of the results and a discussion of future work.

## 2. LTSA OVERVIEW

The Labeled Transition System Analyser (LTSA) [14] is an FSV tool for modeling and analyzing the behavior of systems represented by labeled transition systems. In LTSA, a system is modeled by a set of interacting processes, where each process is described in the Finite Process Language. This description is then translated to a Finite State Automaton (FSA) representation. Formally, an FSA is a five-tuple,  $F = (S, \Sigma, \delta, s^0, s^\epsilon)$ , where  $S$  is a finite set of states,  $\Sigma$ , the alphabet, is a finite set of events,  $\delta : S \times \Sigma \rightarrow S$  is a total transition function,  $s^0$  is a unique start state, and  $s^\epsilon$  is a unique ERROR state such that any transition from this state is a self-loop.

An FSA basically models sequences of events that are allowed to happen in a process. These sequences of events include only those that start at the start state and do not reach the ERROR state of the FSA. Figure 1(a) and (b) show an example system composed of two processes, a client and a server, where the client process must get the lock before writing and must release the lock after writing. To simplify the illustration, the ERROR state of each FSA and all transitions ending with the ERROR state are not shown explicitly in the figures.

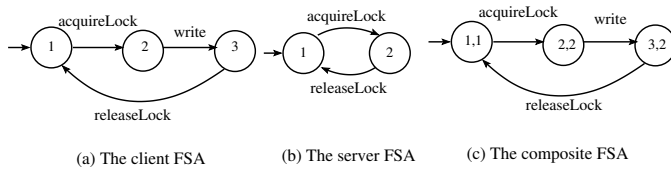


Figure 1: A simple labeled transition system

The behavior of a concurrent system is represented by the composition, or cross-product, of all process FSAs in the system. Formally, the composition of  $n$  FSAs,  $F_1, F_2, \dots, F_n$ , is also an FSA,  $\tilde{F} = (\tilde{S}, \tilde{\Sigma}, \tilde{\Delta}, \tilde{s}^0, \tilde{s}^\epsilon)$ , where  $\tilde{S} = S_1 \times S_2 \times \dots \times S_n$ ,  $\tilde{\Sigma} = \bigcup_{1 \leq i \leq n} \Sigma_i$ ,  $\tilde{s}^0 = \langle s_1^0, s_2^0, \dots, s_n^0 \rangle$ ,  $\tilde{s}^\epsilon$ , the ERROR state of the FSA, is formed by merging the state set  $\{\langle s_1, s_2, \dots, s_n \rangle \mid \exists i (1 \leq i \leq n) : s_i = s_i^\epsilon\}$ , and  $\tilde{\Delta} : \tilde{S} \times \tilde{\Sigma} \rightarrow \tilde{S}$  is defined as follows:  $\tilde{\Delta}(\langle s_1, s_2, \dots, s_n \rangle, e) = \langle \delta_1(s_1, e), \delta_2(s_2, e), \dots, \delta_n(s_n, e) \rangle$  where  $e \in \tilde{\Sigma}$ . Note that for FSA  $F_i$  and event  $e$ , if  $e \in \tilde{\Sigma} - \Sigma_i$ , then  $\delta_i(s_i, e) = s_i$ .

Each state in the composite FSA, called a composite state, is a tuple of states, with one state from each process FSA. The application of the transition function to event  $e$  and a composite state is equivalent to the application of the transition function to event  $e$  and the corresponding state in each process FSA. Therefore, the communication among processes can be viewed as synchronous since the transitions in each process fire atomically. Figure 1(c) shows the composite FSA of the two FSAs shown in (a) and (b).

LTSA can check a system description for violation of safety properties and for deadlock. A safety property in LTSA is specified as an FSA in the same way as a process except for the ERROR state. In a process FSA, any sequence of events reaching the ERROR state does not belong to the behavior of the process. In a safety property FSA, however, a sequence of events that reaches the ERROR state represents a violation of the property. When checking whether a system is consistent with a safety property, the sequences of events that reach an ERROR state of any process FSA will be excluded from consideration. For all other sequences

of events, if any of them reaches the ERROR state of the property FSA, then a violation is reported. For the sake of clarity, the ERROR state in a property FSA is called the VIOLATION state. To check a safety property, the property FSA is incorporated into the composite FSA in the same way as process FSAs.

A deadlock happens in a system when there is a non-ERROR composite state from which all transitions go to the ERROR composite state. This check can be done by simply examining each non-ERROR composite state.

Checking for violation of safety properties or for deadlock can be done on-the-fly while computing the composite FSA of all process FSAs and the property FSA. A commonly used method for this check is to use a search algorithm. In such an algorithm, the states that have already been encountered are stored to avoid duplicate examinations. The problem here is that the number of composite states explored can be exponentially large with respect to size of the system. Hash tables have often been used for storing these states, but performance degrades when handling a large number of states.

## 3. FLAVERS OVERVIEW

FLAVERS/Ada<sup>1</sup> [12] is an FSV tool that uses data-flow analysis techniques to verify that all possible executions of a system are consistent with a property. The property, which represents desirable sequences of events that should occur on all of the executions of the system, is represented by an FSA.

The model of the system used in FLAVERS, called a *Trace Flow Graph* (TFG), is automatically derived from the system description (e.g., the source code). The TFG basically represents the control flow among events in the system. For a sequential system, this model would be an annotated control flow graph, where each node in the graph is labeled by at most one event. To model a concurrent system, each task is represented by an annotated control flow graph and then some modifications are made to model synchronization and the interleaving of events. In addition, a TFG has two special nodes, the initial node and the final node, which represent the entry of the program and the termination of the program, respectively.

The TFG model is imprecise since it over-approximates the event sequences allowed by the system. To improve precision, an analyst can add constraints to eliminate spurious event sequences. A constraint is also represented by an FSA and may be used, for example, to track the value of a variable or to model the program counter for a task.

Given the TFG, the property and the constraints, FLAVERS uses a fixed-point algorithm, called *state propagation* to determine what tuples should be associated with each node in the TFG, where a tuple is a vector of states with one state from each FSA. Each path through the TFG determines a sequence of events. A tuple  $t$  is associated with a node  $n$  if, for some path from the initial node to  $n$ , the corresponding sequence of events drives the property FSA and the constraint FSAs to the corresponding states that compose  $t$ . The algorithm is a typical forward-flow, any-path data-flow analysis problem. To determine whether the property holds, FLAVERS checks the set of tuples associated

<sup>1</sup>A version of FLAVERS that handles Java programs is under development.

with the final node. Tuples in this set that contain any non-accepting state of any constraint FSA are excluded from this check. After excluding these, if there is a tuple including a non-accepting property FSA state, then FLAVERS returns *inconclusive*, meaning that the property may be violated. (Since the model may overapproximate the behavior of the actual system, the property might be violated in the model, but not in the actual system.) Otherwise, FLAVERS returns *conclusive*. The current implementation of FLAVERS does not check for deadlock.

The state propagation algorithm constructs and stores all the tuples associated with each node of the TFG. As with LTSA, these tuples are stored in a hash table, but as mentioned above, this is not suitable for very large systems. Considerable research has been devoted to finding alternative data structures. One promising direction, as described in the following sections, is to use a symbolic data structure to replace the hash table.

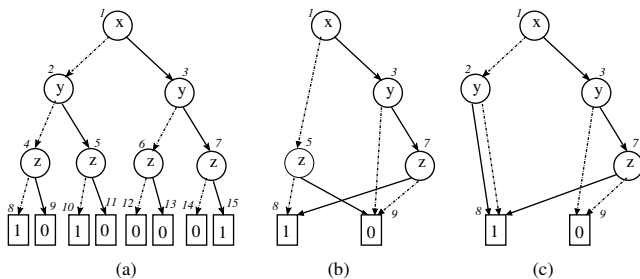
The state propagation algorithm can be thought of as exploring the cross-product of the TFG, the property FSA, and all constraint FSAs, where each state in the cross-product FSA is a *node-tuple* pair. From this point of view, the basic approach of FLAVERS is similar to that of LTSA, although the detailed implementation is quite different. We therefore focus on LTSA as we explain how the symbolic data structures are applied in the following sections.

## 4. APPLYING BDDS

In this section, we give a brief introduction to BDDs and then describe their application in FSV using LTSA.

### 4.1 Background on BDDs

A BDD [7] is a directed acyclic graph that can represent a Boolean function. BDD is derived from the Binary Decision Tree (BDT), which can also represent Boolean functions. Figure 2(a) shows an example of a BDT representation.



**Figure 2: A BDT (a), BDD (b), and ZDD (c) representation for the Boolean function  $f(x, y, z) = (\neg x \wedge \neg z) \vee (x \wedge y \wedge z)$**

A BDT representing a Boolean function comprises two kinds of vertexes, nonterminal vertexes and terminal vertexes, as denoted in Figure 2 (a) by the circular and rectangular nodes, respectively. Each nonterminal vertex  $p$  has a labeled Boolean variable,  $var(p)$ , and two children,  $low(p)$  and  $high(p)$ . Child  $low(p)$  is connected by the *low-edge* of  $p$ , shown as a dashed line in Figure 2 (a), and child  $high(p)$  is connected by the *high-edge* of  $p$ , shown as a solid line in Figure 2 (a). Each terminal vertex is labeled by either 0 or 1. Depending on its label, a terminal vertex is called a 0-terminal or 1-terminal vertex.

A Boolean function composed of  $n$  Boolean variables has  $2^n$  assignments. We call the assignments under which the function value is one 1-assignments, while assignments making the function zero are 0-assignments. A BDT representation for the function has  $2^n$  paths, where each path corresponds to an assignment of the function. In each of these paths, there are  $n$  nonterminal vertexes, each of which is labeled by a different Boolean variable. In the corresponding assignment of the path, the value of variable  $v$  is determined by which edge of the nonterminal vertex labeling  $v$  is in the path: the low-edge means that  $v = 0$  whereas the high-edge means  $v = 1$ . The label of the terminal vertex in a path indicates the value of the function under the corresponding assignment. In Figure 2 (a), for example, the path “1 → 2 → 5 → 11” represents the assignment  $f(x = 0, y = 1, z = 1) = 0$ .

Usually, a total ordering  $<$  is put over all Boolean variables in a BDT. This means that for any nonterminal vertex  $p$  and  $q$ , if  $q$  is a child of  $p$ , then  $var(p) < var(q)$  must hold. In Figure 2 (a), for example, the variables are ordered as  $x < y < z$ .

A BDD is derived from a BDT by applying the following three reduction rules: Rule 1 merges all 0-terminal vertexes and merges all 1-terminal vertexes. Rule 2 shares common sub-graphs. Specifically, when two nonterminal vertexes are the same in that they have the same labeled variable and the same children, then one of them can be eliminated and all its incoming edges redirected to the other one. Rule 3 removes “don’t care” nonterminal vertexes. A “don’t care” nonterminal vertex is one whose two children are the same. When such a vertex is eliminated, its incoming edges are redirected to its child. The basic idea of Rule 3 is that variables that do not affect the functional assignments do not need to be stored in the BDD representation. Given a BDT, these three rules are applied recursively from the bottom-up. The final resulting BDD is called an ordered reduced BDD. From now on, all BDDs we discuss will be ordered reduced BDDs. Figure 2 (b) shows an example of the BDD representation derived from the BDT shown in Figure 2 (a).

Note that the variable ordering selected for a BDD representation is very important to the efficiency of the representation [7]. To find the optimal ordering for a BDD representation, however, is an NP-complete problem [6]. We discuss the heuristic used in our work to find a good ordering in Section 6.

### 4.2 Using BDDs with LTSA

A BDD is just a data structure for representing Boolean functions. Thus, to use BDDs in FSV, the verification problem must be encoded using Boolean functions. More specifically, the following problems need to be addressed: how to encode FSA states and transitions; how to compute the transition functions; and how to check for safety property violation and deadlock, by using Boolean functions. We now describe the approaches we used for each of these.

- **Encoding the FSA States.** Given an FSA with  $n$  states (not counting the ERROR state in a process FSA but counting the VIOLATION state in the property FSA), we use  $\lceil \log_2 n \rceil$  Boolean variables to encode these states. Each state is represented by a Boolean function that has only one 1-assignment. For example, for the client FSA shown in Figure 1 (a), 2 Boolean variables,  $x_1$  and  $x_2$ , are enough to

encode the 3 states in the FSA. State 1 in this FSA can be represented by the Boolean function  $f_{c,1} = \neg x_1 \wedge \neg x_2$ , which can be 1 only when  $x_1 = 0$  and  $x_2 = 0$ .

After the states in each FSA are encoded, the composite state  $s = \langle s_1, s_2, \dots, s_n \rangle$  is encoded by  $f_s = \bigwedge_{1 \leq i \leq n} f_{F_i, s_i}$ , where  $f_{F_i, s_i}$  is the Boolean function for state  $s_i$  in  $F_i$ . For a set of FSA states, the Boolean function is simply the disjunction of Boolean functions for each state in the set.

- **Encoding the Transitions.** A transition has three components: a source state, a destination state, and an event. To encode a transition, two sets of Boolean variables are used in order to distinguish the source state and the destination state. The event is not encoded explicitly using Boolean variables. Instead, transitions triggered by different events are represented by different Boolean functions.

The two sets of variables used in encoding transitions are  $\mathcal{X}$ , the source variable set used for encoding source states, and  $\mathcal{X}'$ , the destination variable set used for encoding destination states. Each variable  $x \in \mathcal{X}$  has one and only one paired variable  $x' \in \mathcal{X}'$  and vice versa.

The Boolean function for a transition from  $s$  to  $s'$  is simply  $f_s \wedge f'_{s'}$ , where  $f_s$  is the Boolean function for state  $s$  using the source variables and  $f'_{s'}$  the Boolean function for state  $s'$  using the destination variables. For transition  $\hat{\Delta}(\langle s_1, s_2, \dots, s_n \rangle, e) = \langle s'_1, s'_2, \dots, s'_n \rangle$  in a composite FSA where  $s'_i = \delta_i(s_i, e)$ , its Boolean function is  $\mathcal{F}_{\langle s_1, s_2, \dots, s_n \rangle, e} = [\bigwedge_{1 \leq i \leq n} f_{F_i, s_i}] \wedge [\bigwedge_{1 \leq i \leq n} f'_{F_i, s'_i}] = \bigwedge_{1 \leq i \leq n} (f_{F_i, s_i} \wedge f'_{F_i, s'_i})$ , where  $f_{F_i, s_i}$  is the Boolean function encoding state  $s_i$  using source variables and  $f'_{F_i, s'_i}$  is the Boolean function encoding state  $s'_i$  using destination variables.

The Boolean function for all transitions in a composite FSA composed of  $n$  FSAs is built in three steps. First, a Boolean function for all transitions triggered by event  $e$  in each FSA  $F_i$  is constructed. This function is denoted by  $\mathcal{F}_i(e)$ . Second, the Boolean function  $\mathcal{F}(e) = \bigwedge_{1 \leq i \leq n} \mathcal{F}_i(e)$  is built to encode all transitions in the composite FSA that are triggered by event  $e$ . Third, the Boolean function for all transitions in the composite FSA is  $\mathcal{F} = \bigvee_{e \in \hat{\Sigma}} \mathcal{F}(e)$ . Note that which transitions are fired by which event cannot be distinguished using the function  $\mathcal{F}$ , but that information is not needed for checking for violations of safety properties and for deadlock, as shown below.

- **Computing the Transition Functions.** A key step in exploring reachable composite states is to compute all states reachable by transitions starting from a given source state. With states and transitions encoded in Boolean functions, this computation should be accomplished by operating on these functions. Indeed, three operations are used for this computation [15].

First, given a Boolean function  $f_s$  encoding a composite state  $s$  and the Boolean function  $\mathcal{F}$  encoding all transitions, the *and* operation is used to compute  $f_s^{[1]} = f_s \wedge \mathcal{F}$ . Second, the source variables need to be removed from the function  $f_s^{[1]}$ . The operation to achieve this is the *existential quantification* operation,  $\exists$ , which is defined as  $\exists_x f = f|_{x=0} \vee f|_{x=1}$ , where  $f|_{x=0}$  is the Boolean function derived from  $f$  by letting  $x = 0$  and  $f|_{x=1}$  is defined similarly. Applying this operation, we get the Boolean function  $f_s^{[2]} = \exists_{\forall x \in \mathcal{X}} f_s^{[1]}$ . The function  $f_s^{[2]}$  actually represents all destination states of  $s$  but uses destination variables. Finally, the *replace* operation is used to replace every destination variable in  $f_s^{[2]}$  with its corresponding source variable, yielding the function

```
//Given a BDD p encoding a set of source states and
//BDD t encoding all transitions of the system,
//return another BDD r encoding all destination states
//of source states represented by p
//A computed table is used to store results of
//previous computations
```

```
APPLYTRAN(p,t) {
1  if p is the 0-terminal vertex, return 0-terminal vertex
2  if t is the 0-terminal vertex, return 0-terminal vertex
3  if t is the 1-terminal vertex, return 1-terminal vertex
4  if the computed table contains the entry {p,t},
5     return the stored result
6  end if
7  if p is the 1-terminal vertex or var(p) > var(t)
8     let r0 = APPLYTRAN(p, low(t))
9     let r1 = APPLYTRAN(p, high(t))
10    let v = var(t)
11  else if var(p) < var(t)
12    let r0 = APPLYTRAN(low(p), t)
13    let r1 = APPLYTRAN(high(p), t)
14    let v = var(p)
15  else // var(p) = var(t)
16    let r0 = APPLYTRAN(low(p), low(t))
17    let r1 = APPLYTRAN(high(p), high(t))
18    let v = var(p)
19  end if
20  if v ∈ X'
21    let v2 ∈ X be the paired variable of v
22    find or make a BDD node r such that:
23    var(r) = v2 and low(r) = r0 and high(r) = r1
24  else
25    let r = or(r0, r1)
26  end if
27  put ({p,t}, r) into the computed table
28  return r
}
```

**Figure 3: An algorithm that computes transition functions using BDDs**

$f_s^{[3]} = \text{replace}(f_s^{[2]}, \mathcal{X}' \leftarrow \mathcal{X})$ .

When BDDs are used as the data structure for representing Boolean functions, these three operations can be implemented efficiently [7]. Furthermore, as mentioned in [15], the *and* operation and the *existential quantification* operation can be combined together into the *andExists* operation, which can often improve the performance substantially because it avoids creating the BDD representing  $f_s^{[1]}$ . In addition, we notice that the *replace* operation can also be combined into the *andExists* operations, further eliminating the temporary BDD for  $f_s^{[2]}$ . We give this BDD algorithm in Figure 3.

The algorithm in Figure 3 is a recursive procedure. Lines 1 to 3 compute the base cases. Lines 7 to 19 carry out the *and* operation. At line 20, when  $v$  is a destination variable, the *replace* operation is computed. In lines 24 and 25, when  $v$  is a source variable, the *existential quantification* operation is computed. Note that this algorithm computes destination states for a set of source states at the same time.

- **Checking for Violation of Safety Properties and for Deadlock.** To check for violation of safety properties, a Boolean function, denoted by  $\mathcal{V}$ , is created to represent all VIOLATION states from each property FSA. Note that variables used for encoding process FSAs are not included in  $\mathcal{V}$ .

To check whether a composite state  $s$  violates a property, one can compute  $f_s \wedge \mathcal{V}$  and then check whether the result is equal to  $\perp$ , the function with no 1-assignments. If  $f_s \wedge \mathcal{V} = \perp$ , it means that the state  $s$  contains no VIOLATION states and thus no violation will be reported. Otherwise a violation is found.

For the deadlock check, another Boolean function, denoted by  $\bar{\mathcal{D}}$ , is created to represent all composite states at which no deadlock happens. This function is built in a way similar to the function that encodes all transitions. For a system with  $n$  FSAs, one first identifies all states in each FSA  $F_i$  such that these states can go to a non-ERROR states by a transition triggered by event  $e$ . Let  $f_i(e)$  denote the function representing these states in  $F_i$ . Then, the Boolean function  $f(e) = \bigwedge_{1 \leq i \leq n} f_i(e)$  is built to encode all composite states that can go to a non-ERROR composite state by a transition fired by  $e$ . At last,  $\bar{\mathcal{D}} = \bigvee_{e \in \bar{\Sigma}} f(e)$ , which represents all composite states at which no deadlock happens, since they all can go to at least one non-ERROR composite state by some transition.

To check whether deadlock occurs at a composite state  $s$ , one can simply compute  $f_s \wedge \bar{\mathcal{D}}$ . If the result is equal to  $f_s$ , it means that state  $s$  is one of the states represented by  $\bar{\mathcal{D}}$  and thus no deadlock happens at state  $s$ . Otherwise a deadlock is reported.

There are two differences between FLAVERS and LTSA at this point. First, as mentioned above, FLAVERS does not check for deadlock. Second, when checking a property violation, FLAVERS considers every element included in a node-tuple. Indeed, the function  $\mathcal{V}$  in FLAVERS represents all node-tuples such that the node is the final node of the TFG, the property state in the tuple is non-accepting and all constraint states in the tuple are accepting.

The algorithm that checks for safety property violation and for deadlock in LTSA using BDDs is given in Figure 4. This algorithm keeps computing reachable composite states until no new states are encountered. The check for violation of safety properties or deadlock is executed whenever a new set of composite states is computed. A difference between this algorithm and the standard search algorithm is that this algorithm performs computations on a set of composite states, not just one state, at a time.

The performance of this algorithm, with BDDs as the underlying data structure, is highly dependent on the efficiency of the BDD representations. There are some variants of BDD that are more efficient than BDDs in certain situations. In the hope of finding a more compact data structure, we explored using ZDDs to replace BDDs in the algorithm in Figure 4.

## 5. APPLYING ZDDS

We now discuss how a ZDD [16] is used to represent a Boolean function, focusing on the differences between a ZDD and BDD representation. We also describe the application of ZDDs in the context of LTSA.

### 5.1 Background on ZDDs

A ZDD, like a BDD, can represent a Boolean function and is also derived from a BDT. The fundamental difference between a ZDD representation and a BDD representation lies in the difference in the third reduction rule. Rule 3' for deriving ZDD representations removes a nonterminal vertex

Build the following BDDs:

- $f_{s^0}$ : represents the composite start state
- $\mathcal{F}$ : represents all transitions in the system
- $\mathcal{V}$ : represents all VIOLATION states in property FSAs
- $\bar{\mathcal{D}}$ : represents all composite states with no deadlock

Let  $f_{visited}$  be the BDD representing all states that have been encountered

```

 $f_{visited} = f_{s^0}$ 
while true
   $f1 = APPLYTRAN(f_{visited}, \mathcal{F})$ 
  if  $f1 \wedge \mathcal{V} \neq \perp$ , return Property Violated
  if  $f1 \wedge \bar{\mathcal{D}} \neq f1$ , return Deadlock
  if  $f1 == f_{visited}$ , return No Violation and no Deadlock
   $f_{visited} = f1 \vee f_{visited}$ 
end while

```

**Figure 4: An algorithm that checks safety properties and deadlock using BDDs**

$p$  when  $high(p)$  is a 0-terminal vertex. After this vertex is removed, all its incoming edges are redirected to  $low(p)$ . Figure 2 (c) shows a ZDD representation derived from the BDT in Figure 2 (a) by applying Rule 1, 2 and 3'.

The rationale behind Rule 3' is that a Boolean function can be represented by storing its 1-assignments only. Moreover, for each assignment, only variables that are assigned to 1 need to be stored. Basically, each path in a ZDD represents a functional assignment in the same way as the paths in a BDT or BDD do. Not every functional assignment, however, has a corresponding path in its ZDD representation. Assignments with no corresponding paths in the ZDD must be 0-assignments. Moreover, not every variable is included in a path of a ZDD. Missing variables in a BDD representation, called “don’t care” variables, may have value 0 or 1 in the corresponding assignments. In contrast, missing variables in a ZDD representation must be assigned to 0 in the corresponding assignment under the ZDD semantics. In the example shown in Figure 2 (c), there is no path corresponding to the 0-assignment  $f(x = 1, y = 0, z = 1) = 0$ . For the path “1  $\rightarrow$  3  $\rightarrow$  9” in this figure, variable  $z$  is missing in the path, so the path actually represents the assignment  $f(x = 1, y = 0, z = 0) = 0$ .

To see why a ZDD represents the Boolean function represented by the BDT from which the ZDD is derived, we need only look at Rule 3', since Rule 1 and Rule 2 do not change any path in the graph and thus do not change the functional assignments. Consider all paths in the graph that pass through nonterminal vertex  $p$  where  $high(p)$  is the 0-terminal vertex. These paths include either the high-edge of  $p$  or the low-edge of  $p$ . In the former case, these paths must reach the 0-terminal vertex, which means that they represent 0-assignments. Therefore, the high-edge of  $p$  can be removed according to the rationale described above. In the latter case, no matter which terminal vertex these paths end with, the variable  $var(p)$  must be assigned to 0 in the corresponding assignments. Therefore, the low-edge of node  $p$  can also be removed. As a result, vertex  $p$  can be eliminated from the graph, as defined in Rule 3'.

Like BDDs, the ZDDs we consider are also ordered and reduced. Again, the variable ordering still plays a critical role in affecting the efficiency of the ZDD representation.

### 5.2 Using ZDDs with LTSA

To use ZDDs to check for safety property violations and

for deadlock, one needs to solve the same problems as for BDDs. Some of the solutions to these problems for ZDDs, including how to encode states and transitions and how to check for deadlock, are the same as those for BDDs and are not repeated here. We describe below the methods for ZDDs that are different from those for BDDs, namely, how to compute transition functions and how to check for safety property violations.

- **Computing the Transition Functions.** The algorithm described in Figure 3 that uses Boolean functions and BDDs to compute transition functions is certainly applicable for ZDDs, although the underlying data structure is changed from BDDs to ZDDs. To use that algorithm with ZDDs, however, one must take into account the different semantics of the BDD and ZDD representations. Specifically, as shown in Figure 3, when computing the *and* operation between  $p$  and  $t$  using ZDD representations, note that  $p$  contains variables from  $\mathcal{X}$  only, whereas  $t$  contains variables from both  $\mathcal{X}$  and  $\mathcal{X}'$ . Under the semantics of ZDDs, missing variables in a ZDD mean that those variables are considered to have value 0 in their functional assignments. As a result, simply computing  $p \wedge t$  using their ZDDs does not get the right answer. A naive approach for solving this problem is to add variables from  $\mathcal{X}'$  to the ZDD  $p$ . This approach is plausible but not efficient, because it increases the size of the ZDDs and thus degrades performance. A better solution is to design a new operation that treats the ZDD  $p$  differently. The idea is that the set of variables defining a Boolean function forms the domain set of the function. In a ZDD representation for a Boolean function, those missing variables are considered to have value 0 only when they belong to the domain of the Boolean function. Variables not in this domain set, although they are definitely missing in the ZDD, should be considered as “don’t care” variables. In our case, the set  $\mathcal{X}$  is the domain of  $p$  and variables from  $\mathcal{X}'$  are treated as “don’t care” variables.

Accordingly, we can now change the algorithm shown in Figure 3 so that it can be used with ZDDs. Figure 5 gives the adapted algorithm. Lines 1 to 5 of this figure compute the base cases of the recursive procedure. In lines 9 to 21, the procedure handles the case where  $t$  is the 1-terminal vertex or  $\text{var}(t) \in \mathcal{X}$ , whereas the case where  $\text{var}(t) \in \mathcal{X}'$  is handled by lines 22 to 29.

For the case where  $t$  is the 1-terminal vertex or  $\text{var}(t) \in \mathcal{X}$ , the procedure computes the *and* operation between  $p$  and  $t$ , and then computes the *existential quantification* operation. There are three sub-cases. First, when  $t$  is the 1-terminal vertex or when  $\text{var}(p) < \text{var}(t)$  (lines 10 to 12), only the low-edge of  $p$  needs to be considered. This is because for all paths including the high-edge of  $p$ , their corresponding assignments must assign  $\text{var}(p)$  to 1. On the other hand,  $\text{var}(p)$  does not appear in  $t$ , which means that  $\text{var}(p)$  is always 0 in all assignments represented by  $t$ . Therefore, the intersection between assignments represented by  $t$  and those represented by  $p$  including the high-edge of  $p$  is simply empty. Also because of this, the *existential quantification* operation ignores the high-edge of  $p$ . The second sub-case (lines 13 to 15) is symmetric to the first one. In the third sub-case (lines 16 to 21), both children of  $p$  should be considered.

When  $\text{var}(t) \in \mathcal{X}'$ , the procedure first computes the *and* operation (lines 24 and 25) and then the *replace* operation (lines 26 and 27). Note that  $\text{var}(t)$  in this case is a variable

from  $\mathcal{X}'$  and is a “don’t care” variable in  $p$ . Therefore, both the high-edge and the low-edge of  $t$  need to be considered.

```

//Given a ZDD p encoding a set of source states
//and ZDD t encoding all transitions of the system,
//return another ZDD r encoding all destination states
//of source states represented by p
//A computed table is used to store results of
//previous computations

APPLYTRAN(p,t) {
1  if p is the 0-terminal vertex, return 0-terminal vertex
2  if t is the 0-terminal vertex, return 0-terminal vertex
3  if both p and t are the 1-terminal vertex
4  return 1-terminal vertex
5  end if
6  if the computed table contains the entry {p,t}
7  return the stored result
8  end if
9  if t is the 1-terminal vertex or var(t) ∈ X
10 if t is the 1-terminal vertex or var(p) < var(t)
11 return APPLYTRAN(low(p),t)
12 end if
13 if p is the 1-terminal vertex or var(p) > var(t)
14 return APPLYTRAN(p,low(t))
15 end if
16 if var(p) = var(t)
17 let r0 = APPLYTRAN(low(p),low(t))
18 let r1 = APPLYTRAN(high(p),high(t))
19 let r = or(r0,r1)
20 put ({p,t},r) into the computed table
21 end if
22 else // var(t) ∈ X'
23 let v ∈ X be the paired variable of var(t)
24 let r0 = APPLYTRAN(p,low(t))
25 let r1 = APPLYTRAN(p,high(t))
26 find or make a ZDD node r such that:
27 var(r) = v and low(r) = r0 and high(r) = r1
28 put ({p,t},r) into the computed table
29 end if
30 return r
}

```

**Figure 5: An algorithm that computes transition functions using ZDDs**

- **Checking for Safety Property Violations.** The approach to checking for a safety property violation with ZDDs is essentially similar to the one used with BDDs. The difference here again involves the missing variables. Like the method used with BDD, the function  $\mathcal{V}$  is first built to encode all VIOLATION states. This ZDD representation for this function contains only variables used for encoding property FSAs. Then,  $f_s \wedge \mathcal{V}$  is computed with ZDDs. Here, variables used for encoding process FSAs should be treated as “don’t care” variables with respect to the ZDD representing  $\mathcal{V}$ , just like how destination variables are handled when computing the transition functions.

Note that this problem does not exist in FLAVERS. As explained before, the function  $\mathcal{V}$  in FLAVERS includes information for every element of a node-tuple. Thus, the domain set of function  $\mathcal{V}$  includes all source variables.

To check for violation of safety properties or for deadlock with ZDDs, we can simply reuse the algorithm shown in Figure 4. The performance difference between the BDD-based algorithm and the ZDD-based one comes mainly from the compactness difference between these two representations.

## 6. EXPERIMENTAL METHODOLOGY

We evaluated six different algorithms in our experiment. For LTSA, we compared the native search algorithm of the tool and the BDD-based and ZDD-based algorithms. For FLAVERS, we compared the native state propagation algorithm and the BDD-based and ZDD-based algorithms. We implemented the BDD-based and ZDD-based algorithms using the JavaBDD [13] package. We built a lightweight BDD/ZDD package from the JavaBDD package, dropping some features such as variable reordering. In addition, we implemented this package in a consistent way for BDDs and ZDDs in that both diagrams use the same hash function, the same cache mechanism, the same memory management strategy and the same data structure for representing BDD/ZDD nodes.

As mentioned before, the variable ordering is critical to the efficiency of both representations. Roughly speaking, there are two approaches to finding a good ordering. A static approach uses heuristics to choose an ordering that is maintained throughout the analysis. Dynamic approaches, on the other hand, modify the ordering as the various BDDs or ZDDs are constructed. We use a static approach, based on the heuristic called FORCE [1]. The idea of this heuristic is to try to put variables that are related to each other as close together as possible. This heuristic is domain-independent, meaning that one must give a measure for evaluating how closely two variables are related to each other. In our case, before we use this heuristic, we first put two restrictions on the ordering. First, we order every destination variable right after its paired source variable. This restriction is usually effective and is often used in other BDD-based FSV tools. Second, we group together all variables encoding one process or property FSA. As explained in [3], this restriction is suggested by the fact that any state in an individual FSA depends more on states from the same FSA than states from other FSAs. With these two restrictions, we now only need to find a good ordering for all FSAs in a given system. Variable ordering within each FSA group is not considered here, because we think that that ordering is not so important given that variables in a single FSA depend heavily on each other. To apply FORCE in this situation, we define a metric to evaluate how closely one FSA is related to another. This metric considers the number of events shared by two FSAs. The more shared events between two FSAs, the more closely related these two FSAs are considered. The FORCE heuristic is applied in the BDD-based and ZDD-based algorithms in the same way.

We use 9 systems for LTSA and 9 systems for FLAVERS. Table 1 gives the complete list of these systems. All systems are scalable, allowing an evaluation of performance as the size of the system increases. In addition, the systems we examined are free of deadlock and all the properties we checked hold. We chose to examine only properties that hold because the checking algorithms stop exploration once a deadlock or violation is found. Therefore, the number of states explored in such systems depends on the algorithm used. When the property holds, however, all the algorithms need to consider all reachable states of the system and, thus, the number of such states does not depend on the algorithm. This makes our comparison among algorithms easier.

To evaluate these algorithms on each system, we first set the memory used to be 512MB, and then ran each algorithm on the given system, repeatedly increasing the size

of the system. When the algorithm ran out of memory at a specific size of the system, or when the algorithm had run for 24 hours, we stopped running the algorithm on this system. As a result, we use two metrics to evaluate these algorithms. The first one is the runtime, which measures the performance of these algorithms straightforwardly. The second one is the size of the largest system that an algorithm can handle with a memory limit (512MB) and a time limit (1 day). This metric can be used to evaluate the memory used by an algorithm.

There are several threats to the validity of our results. First, the selection of example systems may bias the results. Most of our systems represent somewhat unrealistic programs that have been constructed to illustrate issues in the design of concurrent systems. These examples may not adequately represent the range of systems to which the FSV tools might be applied. A second threat arises from our restriction to systems for which the properties hold. As a consequence, the results reported in the paper may not reflect the performance of these algorithms in cases where the properties are violated. Finally, we only considered one approach to variable ordering for the BDDs and ZDDs. Therefore, it is possible that results inconsistent with ours may be found for some systems with other variable orderings.

## 7. EXPERIMENTAL RESULTS

All the inputs and results from our experiment are available at <http://laser.cs.umass.edu/symbolic>.

Table 1 shows the largest size each algorithm can handle for each system under the specified time and memory limits. Also shown in this table is the number of reachable states for the largest size of each system. From this table, we can see that symbolic algorithms can handle much larger systems than non-symbolic algorithms in all except one case. This shows that both symbolic data structures are more efficient than the hash table in both LTSA and FLAVERS. In addition, it can be seen that the ZDD-based algorithm is better than the BDD-based algorithm in terms of the largest size an algorithm can handle. In 8 out of 9 systems in LTSA and 5 out of 9 systems in FLAVERS, as shown in Table 1, the ZDD-based algorithm can handle larger sizes than the BDD-based algorithm under the given resource limits.

The graphs in Figure 6 show the ratio of runtime for the ZDD-based algorithm to the runtime of the BDD-based algorithm for those cases where both algorithms can finish within our time and memory limits. Each point in the graphs gives the ratio for a single *subject*, a pair consisting of a system at a particular size and a property. The graph on the left shows the ratios for LTSA and the one on the right shows the ratios for FLAVERS. We see that the ZDD-based algorithm almost always runs faster than the BDD-based algorithm. On average, the ZDD-based algorithm uses only 60% as much time as the BDD-based algorithm in LTSA and 71% in FLAVERS.

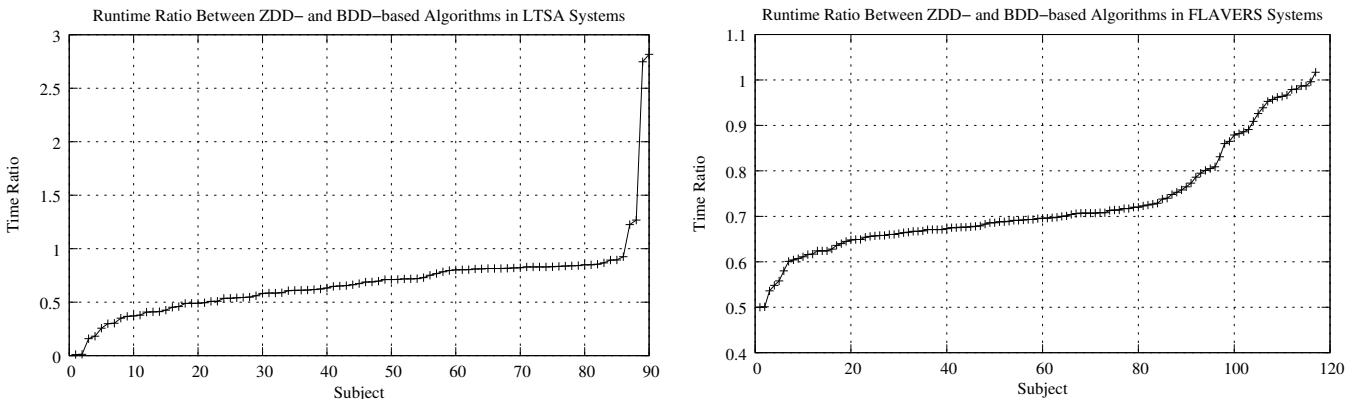
Figure 7 shows the runtimes for each algorithm used with LTSA for two examples. In Figure 7, the left vertical axis in each graph indicates the runtime information while the right vertical axis indicates the number of states explored. The horizontal axis indicates the size of the given system. The crosses show the number of reachable states at each size of the system, which is independent of the algorithm. Each curve in a graph shows the runtime of an algorithm as the size of the system increases. These curves stop at the

Algorithms	LTSA Search		BDD		ZDD	
LTSA Systems	LS	Reachable States	LS	Reachable States	LS	Reachable States
Token ring	11	$6.78 \times 10^6$	33	$4.58 \times 10^{21}$	34	$2.18 \times 10^{22}$
Gas station	11	$7.15 \times 10^6$	24	$5.45 \times 10^8$	29	$1.49 \times 10^9$
Thinkteam	7	$3.65 \times 10^6$	12	$9.68 \times 10^9$	13	$4.48 \times 10^{10}$
Santa Claus	16	$9.12 \times 10^6$	86	$2.90 \times 10^{28}$	104	$8.65 \times 10^{33}$
Dining philosophers	14	$4.78 \times 10^6$	266	$8.21 \times 10^{126}$	272	$5.98 \times 10^{129}$
Cyclic scheduler	18	$7.08 \times 10^6$	100	$1.90 \times 10^{32}$	106	$1.29 \times 10^{34}$
Chiron (original version)	42	$6.78 \times 10^6$	69	$4.64 \times 10^7$	73	$5.77 \times 10^7$
Chiron (decomposed version 1)	32	$5.30 \times 10^6$	21	$1.20 \times 10^6$	23	$1.66 \times 10^6$
Chiron (decomposed version 2)	5	$3.39 \times 10^6$	19	$2.31 \times 10^{20}$	19	$2.31 \times 10^{20}$

Algorithms	State Prop.		BDD		ZDD	
FLAVERS Systems	LS	Reachable States	LS	Reachable States	LS	Reachable States
Token Ring	6	$3.04 \times 10^6$	12	$3.05 \times 10^{11}$	13	$1.95 \times 10^{12}$
Gas Station	5	$1.26 \times 10^7$	8	$4.04 \times 10^{10}$	8	$4.04 \times 10^{10}$
Smoker	5	$3.08 \times 10^6$	7	$1.23 \times 10^9$	8	$3.02 \times 10^{10}$
Pipeline computation	7	$2.46 \times 10^6$	21	$3.42 \times 10^{19}$	22	$2.88 \times 10^{20}$
Relay	4	$3.08 \times 10^5$	10	$1.97 \times 10^{11}$	11	$3.19 \times 10^{12}$
Cyclic scheduler	7	$3.38 \times 10^6$	21	$1.98 \times 10^{17}$	23	$5.95 \times 10^{18}$
Memory management	6	$3.01 \times 10^6$	11	$5.97 \times 10^{10}$	11	$5.97 \times 10^{10}$
Chiron (original version)	9	$4.35 \times 10^6$	21	$5.35 \times 10^7$	21	$5.35 \times 10^7$
Chiron (decomposed version 1)	8	$4.79 \times 10^6$	16	$3.79 \times 10^7$	16	$3.79 \times 10^7$

**Table 1: The largest size (denoted as “LS”) each algorithm can handle and the number of reachable states at the largest size.**



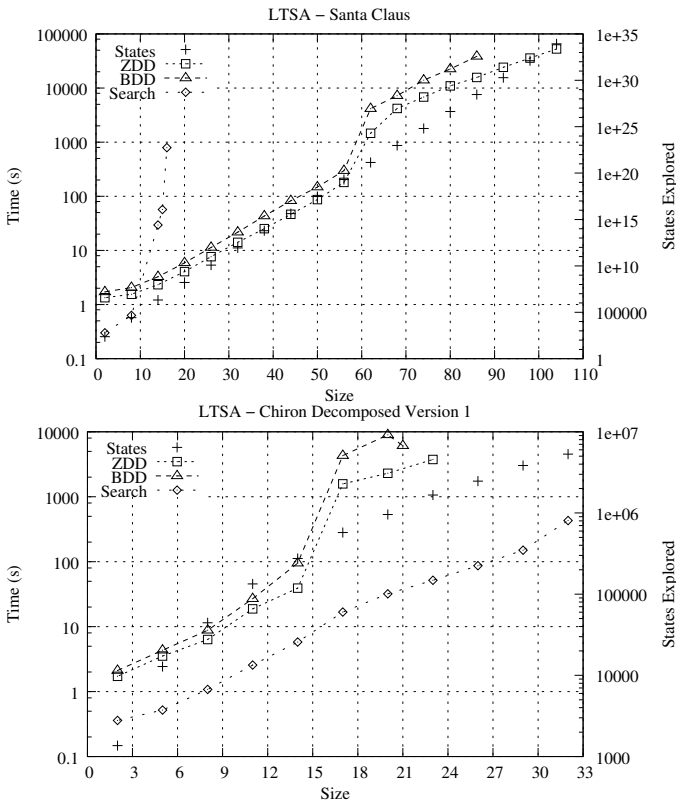
**Figure 6: Runtime ratios between ZDD- and BDD-based algorithms.**

largest size the algorithm can handle with the given time and memory limits. The first graph in the figure shows the results for the Santa Claus system [5], which are typical of what we saw with both LTSA and FLAVERS. The native search algorithm is superior for small sizes, but as the size increases, both symbolic algorithms surpass the native search algorithm. The ZDD-based algorithm runs somewhat faster than the BDD-based algorithm and can scale to larger sizes within the time and memory limits we imposed.

The second graph in Figure 7 show the results for version one of the decomposed Chiron system with LTSA, one case in our experiment in which an algorithm that stores individual states was superior to the symbolic algorithm. (This did not occur with FLAVERS.) Examination of this system reveals that it has a relatively small number of reachable states because the processes in the system are highly synchronized.

To see why the ZDD-based algorithm outperformed the BDD-based algorithm, we examined the sizes of the repre-

sentations in these algorithms. Not surprisingly, the ZDD representations are almost always more compact than the BDD representations. As shown in [16], a ZDD representation tends to be more efficient than a BDD representation for a sparse combination set, where a combination for a given set  $S$  is simply a subset of  $S$ . By letting  $S$  be the domain set of a Boolean function, a functional assignment can be represented by a combination of  $S$  that includes Boolean variables whose values are 1 under that assignment. Thus, a Boolean function can be represented by a combination set that represents all 1-assignments of the function. Based on this, we may expect ZDDs to be more efficient than BDDs when representing Boolean functions with sparse 1-assignments. In our experiments, the Boolean functions representing reachable composite states usually have a small number of 1-assignments, compared with the number of all possible assignments those functions can have. Whether the distribution of these 1-assignments is actually sparse, however, is not clear. More work regarding the distribution of the 1-



**Figure 7: Experimental results using LTSA on two systems. “ZDD” and “BDD” represent the ZDD-based and BDD-based algorithm, respectively, “Search” represents the search algorithm used in LTSA.**

assignments of a Boolean function needs to be done.

Our results suggest that the ZDD-based algorithm is the best overall among the algorithms we evaluated in terms of both time and space and it should be the first algorithm to consider when conclusive results are expected.

## 8. RELATED WORK

There are many BDD-based FSV tools, such as [4, 10]. The main difference between our work and these FSV tools is the use of ZDDs.

ZDDs have been used in many domains, such as SAT solvers [9], and arithmetic polynomial manipulation [16]. ZDDs have also been used in verification [17]. In that work, both BDD and ZDD representations are used and compared in model checking CTL properties of Petri nets. Several symbolic approaches using BDDs and ZDDs to manipulate Petri nets are proposed in that paper, and the experimental results suggest that ZDDs are more suitable.

A number of other investigators have compared the performance of different FSV tools, including those that use symbolic algorithms and non-symbolic algorithms [2, 11]. Since different tools use different modeling languages to describe the system, there is no guarantee that the models used with different tools are really equivalent. This factor may significantly affect the comparison of different algorithms. In our work, since the algorithms compared are implemented in the same FSV tool, the systems used for comparison are

modeled in exactly the same way.

## 9. CONCLUSION

FSV techniques can check whether a property holds for a given system. Once the system model and the property are specified, this check is done automatically by exhaustively exploring all possible states of the system. FSV techniques are especially useful for concurrent systems, where nondeterministic behavior usually makes testing problematic. FSV methods, however, are limited by the state-explosion problem. Much research has been done to ameliorate this problem but few comparative studies have been done using the same approach with different symbolic algorithms.

In this paper, we investigate the use of two symbolic data structures, BDDs and ZDDs, in two FSV tools, LTSA and FLAVERS. To do this, we developed an efficient algorithm to compute transition functions with ZDDs. This algorithm treats missing Boolean variables in a ZDD representation in different ways depending on whether the variable is from the domain set of the Boolean function being represented or not. In our experiments, both symbolic algorithms perform much better than the native, non-symbolic algorithms used by LTSA and FLAVERS, in terms of *both* runtime and the size of the system that the algorithm can handle under the given resource limits. Moreover, the ZDD-based algorithm is almost always better than the BDD-based one. Our results suggest that the ZDD-based approaches should be more widely investigated.

We intend to explore several additional directions. There are several additional variations of the BDD and ZDD based algorithms that appear promising. One variation, for example, is to change the algorithm so that each composite state visited during the check is used in the computation for the transition functions only once. To implement this, besides the function  $f_{visited}$ , another function  $f_{new}$ , which represents the composite states at the current exploration frontier only, is needed. Then only states represented by  $f_{new}$  would be used in the computation for the next frontier. We did a preliminary experiment with this variation and found that sometimes it improves performance, but sometimes it degrades performance.

Another variation we want to pursue is to change the encoding scheme for the ZDD representations. The encoding strategy described in this paper uses  $\lceil \log_2 n \rceil$  Boolean variables to encode  $n$  FSA states. We call this encoding a binary encoding. An alternative, unary encoding would use one variable for each FSA state. A benefit of the unary encoding is that it enables us to use an algorithm for computing transition functions without encoding all transitions in Boolean functions. We also did a preliminary experiment with this approach and found that it also sometimes helps improve performance but sometimes degrades performance. We intend to do more experiments to explore the system features that make one approach more effective than the other.

Besides these variations, we also want to investigate how the BDD and ZDD based algorithms work when the property does not hold (at least for the system model). In that situation, FSV tools can generate a counterexample, a trace through the model showing how the property is violated, that can help the analyst find out what went wrong. For the LTSA search algorithm, it is easy to generate counterexamples once a violation or deadlock is found. For the state

propagation algorithm and the symbolic algorithms, however, more work needs to be done on efficiently generating counterexamples.

Finally, as discussed above, we intend to explore the distribution of 1-assignments for a Boolean function. We believe that this will not only help understand why the ZDD representations of the functions we encounter are more compact than the BDD representations, but also may suggest further optimizations.

## 10. ACKNOWLEDGMENTS

We thank Jeff Magee and Jeff Kramer for kindly providing us with the LTSA source code and example systems.

## 11. REFERENCES

- [1] F. A. Aloul, I. L. Markov, and K. A. Sakallah. FORCE: A fast and easy-to-implement variable ordering heuristic. In *Proceedings of the 13th ACM Great Lakes Symposium on VLSI*, pages 116–119, Apr. 2003.
- [2] G. S. Avrunin, J. C. Corbett, M. B. Dwyer, C. S. Păsăreanu, and S. F. Siegel. Comparing finite-state verification techniques for concurrent software. TR UM-CS-1999-069, Department of Computer Science, U. of Massachusetts Amherst, Nov. 1999.
- [3] A. Aziz, S. Tasiran, and R. Brayton. BDD variable ordering for interacting finite state machines. In *Proceedings of the 31st ACM/IEEE Design Automation Conference*, pages 283–288, San Diego, CA, USA, Jun. 1994.
- [4] I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Y. Rodeh, G. Ronin, and Y. Wolfsthal. Rulebase: Model checking at IBM. In *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 480–483, Jun. 1997.
- [5] M. Ben-Ari. How to solve the Santa Claus problem. *Concurrency: Practice and Experience*, 10(6):485–496, 1998.
- [6] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-Complete. *IEEE Transactions on Computers*, 45(9):993–1002, Sep. 1996.
- [7] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sep. 1992.
- [8] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking:  $10^{20}$  states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33. IEEE Computer Society Press, Jun. 1990.
- [9] P. Chatalic and L. Simon. ZRES: The old davis-putman procedure meets ZBDD. In *Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *LNCS*, pages 449–454, Jun. 2000.
- [10] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *LNCS*, pages 495–499, Jul. 1999.
- [11] Y. Dong, X. Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, O. Sokolsky, E. W. Stark, and D. S. Warren. Fighting livelock in the i-protocol: A comparative study of verification tools. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 74–88, Mar. 1999.
- [12] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Transactions on Software Engineering and Methodology*, 13(4):359–430, 2004.
- [13] <http://javabdd.sourceforge.net/>. JavaBDD - Java Binary Decision Diagram library.
- [14] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, NY, USA, 1999.
- [15] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, MA, USA, 1993.
- [16] S. Minato. *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer Academic Publishers, MA, USA, 1996.
- [17] T. Yoneda, H. Hatori, A. Takahara, and S. Minato. BDDs vs. zero-suppressed BDDs: for CTL symbolic model checking of Petri Nets. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, volume 1166 of *LNCS*, pages 435–449, Nov. 1996.