Managing Space for Finite-State Verification

Jianbin Tan, George S Avrunin, Lori A Clarke
University of Massachusetts, Amherst
May 24, 2006
ICSE’06 - Shanghai
Finite-State Verification (FSV)

- Attempts to prove properties about a model of a system

A model of a system

property

Finite State Verifier

✓

?
Typically, a system is modeled by a set of processes where each process is modeled by a Finite State Automaton (FSA)
The overall system behavior is represented by the cross product FSA of all the process FSAs.
State Explosion Problem

- But the number of reachable states in the cross product FSA could be exponentially large w.r.t. the number of process FSA states.
Symbolic Approaches

- Use symbolic data structures to compactly represent subsets of reachable states
  - Shown to be useful for reducing the impact of the State Explosion problem in hardware verification

- But their value for software verification is less clear
Our Work

- Explored using two symbolic data structures
  - Binary Decision Diagrams (BDDs)
  - Zero-suppressed BDDs (ZDDs)

- in two finite state verifiers
  - LTSA
  - FLAVERS

- Developed a modified algorithm for computing the reachable states for ZDDs

- Developed a heuristic for variable ordering for BDDs/ZDDs

- Found an approach that usually *increases* the size of the systems that can be verified and *decreases* the time of the analysis
Outline

- Overview of BDDs and ZDDs
- Applying BDDs and ZDDs in FSV
- Experimental methodology and results
- Conclusions
Binary Decision Trees

- From which BDDs and ZDDs are derived
- Represent Boolean functions
Binary Decision Trees

Each path represents an assignment

- 0-edge: the variable is assigned to 0
- 1-edge: the variable is assigned to 1
- The label of the terminal vertex determines the function value

Exponentially Large!

\[ f(x, y, z) = (\neg x \land \neg z) \lor (x \land y \land z) \]

\[ f(x=0, y=1, z=1) = 0 \]
Deriving BDDs and ZDDs

- BDDs [Bryant] and ZDDs [Minato] are reduced Binary Decision Trees
  - Both diagrams are derived by applying 3 reduction rules
  - The first 2 rules are the same for both diagrams
  - Only the third rule is different
Deriving BDDs and ZDDs

- Rule 1 merges terminal vertexes

```
x y y z z z z 1 0 1 0 0 0 0 1
```
Deriving BDDs and ZDDs

- Rule 1 merges terminal vertexes

![BDD and ZDD diagrams](image-url)
Deriving BDDs and ZDDs

- Rule 1 merges terminal vertexes
- Rule 2 reuses common subgraphs
Deriving BDDs and ZDDs

- Rule 1 merges terminal vertexes
- Rule 2 reuses common sub-graphs
Rule 3 for Deriving BDDs

- Rule 3: Remove “don’t care” nonterminal vertexes (whose children are the same)
Rule 3 for Deriving BDDs

- Rule 3: Remove “don’t care” nonterminal vertexes (whose children are the same)
Rule 3 for Deriving BDDs

- Rule 3: Remove “don’t care” nonterminal vertexes (whose children are the same)
Rule 3 for Deriving ZDDs

- Rule 3: Remove nonterminal vertexes whose 1-edge points to the terminal vertex labeled by 0
Rule 3 for Deriving ZDDs

- Rule 3: Remove nonterminal vertexes whose 1-edge points to the terminal vertex labeled by 0
Rule 3 for Deriving ZDDs

- Rule 3: Remove nonterminal vertexes whose 1-edge points to the terminal vertex labeled by 0
Along a path, missing variables may be assigned to either 1 or 0.

Along a path, missing variables are assigned to 0.
Ordered BDDs/ZDDs

- Boolean variables are ordered
- For any vertex $p$ and either of its nonterminal children $q$: $\text{var}(p) > \text{var}(q)$ holds
- Given an order, BDDs/ZDDs for Boolean functions are canonical
- Ordering can have a significant impact on efficiency
  - Will describe an ordering heuristic later
To Apply BDDs/ZDDs in FSV

- Encode FSAs as Boolean functions represented as BDDs/ZDDs
- Compute reachable states with BDDs/ZDDs
  - BDD-based: standard algorithm
  - ZDD-based: modified algorithm
A process FSA with $n$ states can be encoded with $\lceil \log_2 n \rceil$ Boolean variables.

Each process FSA uses a different set of variables.

State 1 : ($\neg x_1 \land \neg x_2$)
State 2 : ($x_1 \land \neg x_2$)
State 3 : ($\neg x_1 \land x_2$)

State 1 : ($\neg x_3 \land \neg x_4$)
State 2 : ($x_3 \land \neg x_4$)
State 3 : ($\neg x_3 \land x_4$)
State 4 : ($\neg x_3 \land \neg x_4$)
Encoding Transitions

- Uses two sets of paired variables
  - Source variables: $x_1, x_2, \ldots$
  - Destination variables: $x_1', x_2', \ldots$

- Transitions labeled by the same event are encoded by a single Boolean function

- The whole transition relation is a disjunction of these functions

Transition 1$\rightarrow$2

$$\left( \neg x_1 \land \neg x_2 \right) \land \left( x_1' \land \neg x_2' \right)$$
Encoding Cross Product FSA

- Each state of the cross product FSA is a conjunction of process states

State \(<1,1>\):
\[\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4\]

State \(<2,1>\):
\[x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4\]
Each state of the cross product FSA is a conjunction of process states.

The transition is encoded as before.

State <1,1>: \((\neg x_1 \land \neg x_2 \land \neg x_3 \land \neg x_4)\)

State <2,1>: \((x_1 \land \neg x_2 \land \neg x_3 \land \neg x_4)\)

Transition <1,1> \(\rightarrow\) <2,1>: \((\neg x_1 \land \neg x_2 \land \neg x_3 \land \neg x_4) \land (x_1' \land \neg x_2' \land \neg x_3' \land \neg x_4')\)
Computing Reachable States

- Standard fixpoint algorithm: computes *images* until no new states are seen
  - Image: all direct successors for a set of states
- All operations used in the computation can be implemented with BDDs/ZDDs
- Developed a more efficient approach for computing images with ZDDs
Choose a Variable Ordering

- Ordering of variables impacts size of BDDs/ZDDs
- But finding an optimal ordering is NP-hard
- Typical two general approaches
  - Static heuristics to choose a fixed ordering before analysis
  - Dynamic reordering during analysis
- Developed a static ordering heuristic
General Ordering Constraints

- Put each destination variable right after its paired source variable
  - Effective and commonly-used
- Group together Boolean variables encoding each single process FSA
  - States in the same FSA depend more on each other than states from other FSAs
Adapted the FORCE Heuristic

- The FORCE heuristic [Aloul, Markov, Sakallah]: puts variables from FSAs that are *closely related* to each other as close as possible

- We used the number of events shared by two FSAs to measure how closely two FSAs are related
  - The more events shared, the closer they are
  - Within each FSA, variables are ordered arbitrarily
Evaluated Impact of BDDs/ZDDs on FSV

- BDDs/ZDDs implementation
  - Based on the JavaBDD [Whaley] library
  - BDDs and ZDDs are implemented in a consistent way

- Applied to two finite state verifiers:
  - LTSA [Magee, Kramer]:
    Labeled Transition System Analyzer
  - FLAVERS [Dwyer, Clarke, Cobleigh, Naumovich]:
    FLow Analysis for VERification of Systems
Algorithms Evaluated

- **LTSA**
  - The native search algorithm, which constructs a version of the reachability graph and uses a hashtable to store states
  - BDD-based
  - ZDD-based

- **FLAVERS**
  - The native search algorithm, which uses data flow analysis and a hashtable to store states
  - BDD-based
  - ZDD-based
13 systems were used
- Scalable, concurrent
- 4 systems modeled in LTSA only
- 4 systems modeled in FLAVERS only
- 5 systems modeled in both verifiers

18 properties in all

Only considered properties that hold for these systems
- Ensures that the number of reachable states explored by different algorithms is the same
Ran each algorithm on each system/property, scaling up the size until the algorithm:
• Ran out of memory, or
• Ran for more than 24 hours

Compared
• **Runtime**
• **The Largest size** each algorithm could handle for each system
Gas Station - LTSA

Typical profile for LTSA and FLAVERS
Experimental Results

- BDD-based and ZDD-based algorithms could handle much larger systems than both native algorithms in most cases
  - In only one LTSA system, the native algorithm outperformed BDD and ZDD based algorithms

- ZDD-based algorithm could handle larger systems and ran faster than BDD-based algorithm in most cases
  - In only 5 out of 200 subjects, BDD-based outperformed ZDD-based algorithm
Results for FLAVERS are similar.
Related Work

- **BDD-based tools**
  - NuSMV [Cimatti, Clarke, Giunchiglia, Roveri]
  - Rulebase [Beer, Ben-David, Eisner, Geist, Gluhovsky, et al]
  - ...

- **Usage of ZDDs**
  - Petri net verification [Yoneda, Hatori, Takahara, Minato]
  - ...

- **Comparisons between symbolic approach and non-symbolic approach in FSV**
  - [Avrunin, Corbett, Dwyer, Pasareanu, Siegel]
  - [Dong, Xu, Ramakrishna, Ramakrishnana, et al]
  - ...

Conclusion

- The ZDD-based algorithm was almost always better than the BDD-based and native algorithms
  - Reachable state space is usually sparse
  - ZDDs are good at representing Boolean functions with sparse *supports* (assignments for which the function value is 1)
- More evaluation is needed
  - ZDDs might be useful for other finite state verifiers
The computation needs two ZDDs

- $S$: represents a set of states using only source variables
- $T$: represents the whole transition relationship using both the source variables and destination variables

Missing destination variables in ZDD

$S$ are assigned to 0, but they should be considered as “don’t care”

- Treat destination variables in a BDD way
- Treat source variables in a ZDD way