Abstract

1 Introduction

- Importance of autonomous systems and their verification
- The state explosion problem, summary of different approaches
- Our idea and contributions
- Project objectives

2 Background

2.1 Logics for Multi-Agent Systems

- Multi-Agent Systems and Interpreted Systems
- Temporal Epistemic Logics: (LTL,) CTL, CTLK

2.2 Model Checking

- Explicit Approach to Model Checking
- Symbolic Model Checking and state-space representations

2.3 Symbolic Representations of State Spaces

- BDDs: BDTs, reduction, canonicity up to reordering
- *A Knowledge Compilation Map*
- Negation Normal Forms and choice of SDDs for the project

2.4 SDDs

- Definitions and Construction
- Syntax and Semantics
- Canonicity
- OBDDs are SDDs

3 Technical Preliminaries

3.1 MCMAS Specifics

- *Some* implementation details (enough to understand the steps needed for replacing BDDs with SDDs).
- Variable allocation and how we keep track of it
- Description of the 4 variable orderings available
- ADDs to represent algebraic expressions

3.2 The SDD Package

- Development, summary of features
- Dynamic minimization and automatic garbage collection
- Algorithm for vtree search (and operations for navigating the space of vtrees)

4 Contributions

4.1 Implementation of a model checker based on SDDs

- Functionality and limitations

4.2 Some heuristics

(This is one of the most important sections - but heavily dependent on the next couple of weeks)

5 Evaluation

5.1 Models

- Description of models used for quantitative analysis

5.2 Comparison without dynamic variable reordering

5.2.1 Comparison of various variable orderings with corresponding right-linear vtrees

- Built-in orderings

  Results in Table 1. Only need to do time comparisons since structures have the same size

- Orderings resulting from dynamic vtree search (TODO)

  Results should be in Table 2.

5.2.2 Observations

- The environment generally contains the highest number of variables (and hence the largest evolution SDD).
- Applying the transition relation in the state space generation represents a significant amount (think more than 80%) of the overhead. In some cases `sdd_exists()` is also very slow, but only in some cases, which can lead to thinking that significant improvements are possible.

5.2.3 Comparison of various variable orderings with equivalent non-right-linear *dissections* of these orderings

- Built-in orderings with standard vtrees:

  Left-linear and vertical vtrees proved to be very bad, so we will focus on balanced vtrees.

- 'Clever' vtrees

  We aim to find an initial vtree leading to faster computations. We notice that whatever the initial vtree, dynamic reduction algorithms always result in a particular type of vtree, which we call *pseudo-right-linear*. All our experiments are with pseudo-right-linear vtrees.

  - vtree experiment 1 (option 5): one balanced subtree per agent.
    This does *not* work well.
  - vtree experiment 2 (option 6): We set a maximum size for agent subtrees. An upper bound of $\log_2(n^2)$ (where n is the number of vars) has proved relatively efficient. If an agent has more variables then we create more subtrees for it. Variables of a subtree come from the same agent. Subtrees are balanced, and we experiment with different orderings for them. We observe that the best results are obtained when two principles are followed: action variables are close to each other in the subtree (i.e they alone form a balanced vtree of size `action_count`) and states variables are paired with their primed counterpart. (TODO: confirm this!)
  - vtree experiment 3 (option 7): Each subtree contains one variable for each agent, as well as its corresponding primed variable. Action variables for each agent form their own subtree. We order state subtrees 'largest to smallest' and put action subtrees at the bottom. This seems to be the best ordering but TODO need to try: intercaler actions/states.
  - vtree experiment 4 (option 8): Think of something!

5.2.4 More suggestions for speed improvement

- Call `sdd_apply()` on a reduced vtree
- Existentially quantify out variables in a smart order.
  The CUDD algorithm for this is recursive - need to sort sth out

5.2.5 Analysis and conclusion

5.3 Comparison using dynamic minimization algorithms

- Experiments with different initial orderings and vtrees
- Analysis of heuristics proposed in previous section
- Comparison of different vtree minimization algorithms/settings (The SDD package allows the user to implement their own minimization function, or change the settings in the original functions. It would be interesting to find out what happens if we change the minimization thresholds for time and memory, or if we choose a different algorithm for searching the space of vtrees)

6 Conclusions and further work

6.1 Review

6.2 Future work

I will see what I don't have time to do before the deadline. Potential missing features will be counterexample/witness generation, and checking for deadlock or model overflow. Also being able to check ATL formulas. At some point I would like to have a go at implementing an ADD equivalent for SDDs.

7 Bibliography