Bounded model checking by SAT-based temporal induction

Sami Liedes

November 13, 2007

Problem statement

- Intuitively, we want to verify, or find a counterexample (trace) disproving the hypothesis, that our model Behaves Correctly^{TM}.

Problem statement

- Intuitively, we want to verify, or find a counterexample (trace) disproving the hypothesis, that our model Behaves Correctly^{TM}.
- In practice we are limited to requiring that some mathematical quality (invariant) holds for all execution paths.

Problem statement

- Intuitively, we want to verify, or find a counterexample (trace) disproving the hypothesis, that our model Behaves Correctly^{TM}.
- In practice we are limited to requiring that some mathematical quality (invariant) holds for all execution paths.
- For the purposes of symbolic model checking (or at least SAT-based temporal induction), we only consider *finite models*, i.e. models with finite state space.

Prerequisites

Some things we assume the audience has a good understanding of:

Prerequisites

Some things we assume the audience has a good understanding of:

- The boolean satisfiability (SAT) problem
- Mathematical induction
- Finite state machines (FSM)

Problem statement, part 2: Formalism

- Since we have restricted ourselves to finite problems, the models being verified can be specified as finite state machines.

Problem statement, part 2: Formalism

- Since we have restricted ourselves to finite problems, the models being verified can be specified as finite state machines.
- But with FSMs we can trivially find all reachable states. Problem solved?

Problem statement, part 2: Formalism

- Since we have restricted ourselves to finite problems, the models being verified can be specified as finite state machines.
- But with FSMs we can trivially find all reachable states. Problem solved?

THE END

Problem statement, part 2: Formalism

- Since we have restricted ourselves to finite problems, the models being verified can be specified as finite state machines.
- But with FSMs we can trivially find all reachable states. Problem solved?

THE END

- Not so. With n state bits, we have 2^n states, which is way too much to go through.

Symbolic model checking

Symbolic model checking: good when it works. But how to make it fast?

Symbolic model checking

Symbolic model checking: good when it works. But how to make it fast?

- No one-size-fits-all solution
- Some approaches already presented on this course
- Perhaps most approaches based on BDDs (Binary Decision Diagrams)
- One more approach: SAT-based temporal induction

Some definitions

Let

- M be the finite state machine implementation of the model being examined

Some definitions

Let

- M be the finite state machine implementation of the model being examined
- I be a predicate satisfied by exactly all the possible initial states of the program (automaton M)

Some definitions

Let

- M be the finite state machine implementation of the model being examined
- I be a predicate satisfied by exactly all the possible initial states of the program (automaton M)
- P be a property being checked against, which we would like to hold for all reachable states, in which case we call the program *P-safe*;

Some definitions

Let

- M be the finite state machine implementation of the model being examined
- I be a predicate satisfied by exactly all the possible initial states of the program (automaton M)
- P be a property being checked against, which we would like to hold for all reachable states, in which case we call the program *P-safe*;
- T be a (binary) transition relation on the states S of the automaton M

Some definitions

Let

- M be the finite state machine implementation of the model being examined
- I be a predicate satisfied by exactly all the possible initial states of the program (automaton M)
- P be a property being checked against, which we would like to hold for all reachable states, in which case we call the program *P-safe*;
- T be a (binary) transition relation on the states S of the automaton M
- $$\text{path}(s_0..s_n) \equiv \bigvee_{0 \leq i < n} T(s_i, s_{i+1})$$

Some approaches to solving the problem

- Check that

$$\forall s_0..s_i \in S : I(s_0) \wedge \text{path}(s_0..s_i) \implies P(s_i),$$

  first for $i = 0$, then for $i = 1$, $i = 2$ and so on. If P holds for all reachable states, this formula is always true.
- Problem: This is not a *complete* algorithm: It will eventually find all counterexamples, but otherwise it never terminates.

Some approaches to solving the problem, continued

- Due to the finiteness of M, at some point it will be safe to stop incrementing i and declare our program P-safe. But how do we know when?

Some approaches to solving the problem, continued

- Due to the finiteness of M, at some point it will be safe to stop incrementing i and declare our program P-safe. But how do we know when?
- Solution: We only need to consider *loop-free* paths, i.e. paths where the same state never repeats.
- Clearly there is a longest loop-free execution path in any model. Therefore if we incorporate this into our algorithm, it will become complete. We call the length of this path the *diameter* of the state transition graph, $\mathrm{diam}(M)$.

SAT solving

- Now our algorithm can be expressed as boolean SAT if we know the length of the longest loop-free path.
- Loop-freeness for paths of length n can be represented by n^2 inequality constraints.
- So, if we just find the length of the longest loop-free path, represent our problem as SAT and give it to a solver, it will finally tell us what we want to know.
- Problem: SAT solvers are good and quite magical, but their magic is not (yet) always powerful enough to consider the still exponential number of states in reasonable time.

Representing the problem using induction

- Crazy idea: Try to help the SAT solver by using induction and using the solver just to prove the induction base and induction step. But how do we induct on this problem?

Representing the problem using induction

- Crazy idea: Try to help the SAT solver by using induction and using the solver just to prove the induction base and induction step. But how do we induct on this problem?

1. Induction base hypothesis: There are no paths of length n from any of the initial states that contain *bad states* (ones for which P does not hold).
2. Induction step hypothesis: After n valid execution steps, the next step is guaranteed to land us in a *valid state* (one for which P holds).

Representing the problem using induction

- Crazy idea: Try to help the SAT solver by using induction and using the solver just to prove the induction base and induction step. But how do we induct on this problem?

1. Induction base hypothesis: There are no paths of length n from any of the initial states that contain *bad states* (ones for which P does not hold).
2. Induction step hypothesis: After n valid execution steps, the next step is guaranteed to land us in a *valid state* (one for which P holds).

- Obviously, these constraints will hold for some $n \leq \text{diam}(M)$. Turns out they are provable for perhaps surprisingly small n for at least some real-world models.

Algorithm 1

1 i=0
2 **while** *true* **do**
3 **if** *not* Sat($I(s_0) \wedge \text{loopFree}(s_0..s_i)$) *or not* Sat($(\text{loopFree}(s_0..s_i) \wedge \neg P(s_i))$) **then**
4 **return** *true*
5 **end**
6 **if** Sat($I(s_0) \wedge \text{path}(s_0..s_i)) \wedge \neg P(s_i)$ **then**
7 **return** *Trace* $c_0..c_i$
8 **end**
9 $i = i + 1$
10 **end**

Improving on this solution - Tighter termination conditions

The termination conditions of Algorithm 1 can be made tighter:

- Consider only paths from an initial state through good non-initial states to a bad non-initial state.
- viz.

$$\underset{s_0}{IP} \rightarrow \underset{s_1}{\bar{I}P} \rightarrow \underset{s_2}{\bar{I}P} \rightarrow \cdots \rightarrow \underset{s_{n-1}}{\bar{I}P} \rightarrow \underset{s_n}{\overline{IP}}$$

- Intuitively, we do not want to go back to an initial state, since then there would be a shorter path to be found. The same holds for a bad state in the middle.
- Formally, we replace $I(s_0) \wedge \text{loopFree}(s_0..s_i)$ by

$$I(s_0) \wedge all.\neg I(s_1..s_i) \wedge \text{loopFree}(s_0..s_i).$$

Algorithm 2: Improved termination conditions

```
1  i=0
2  while true do
3      if not Sat(I(s_0) ∧ all.¬I(s_1..s_i) ∧ loopFree(s_0..s_i)) or not
       Sat((loopFree(s_0..s_i) ∧ all.P(s_0..s_{i-1}) ∧ ¬P(s_i))) then
4          return true
5      end
6      if Sat(I(s_0) ∧ path(s_0..s_i)) ∧ ¬P(s_i) then
7          return Trace c_0..c_i
8      end
9      i = i + 1
10 end
```

Removing need to start from zero depth

- When a model requires a high induction depth, it quickly gets expensive to iterate from zero depth.
- It turns out we can modify the algorithm to start from any depth.
- We modify the algorithm to check for bad paths of length *up to* i and not just exactly i.
- It also turns out to be convenient to switch the order of the two checks.

Algorithm 3: An algorithm that need not iterate from 0

1 i=some constant which can be greater than 0
2 **while** *true* **do**
3 **if** $\mathrm{Sat}(I(s_0) \wedge \mathrm{path}(s_0..s_i)) \wedge \neg all.P(s_0..s_i)$ **then**
4 **return** *Trace* $c_0..c_i$
5 **end**
6 **if** *not* $\mathrm{Sat}(I(s_0) \wedge all.\neg I(s_1..s_{i+1}) \wedge \mathrm{loopFree}(s_0..s_{i+1}))$ *or not* $\mathrm{Sat}((\mathrm{loopFree}(s_0..s_{i+1}) \wedge all.P(s_0..s_i) \wedge \neg P(s_{i+1})))$ **then**
7 **return** *true*
8 **end**
9 $i = i + 1$
10 **end**

Algorithm 4: Rewritten to look more like an induction proof

```
1  i=some constant which can be greater than 0
2  while true do
3      if Sat(¬(I(s_0) ∧ path(s_0..s_i) → all.P(s_0..s_i))) then
4          return Trace c_0..c_i
5      end
6      if Taut(¬I(s_0) ← all.¬I(s_1..s_{i+1}) ∧ loopFree(s_0..s_{i+1})) or
       Taut((loopFree(s_0..s_{i+1}) ∧ all.P(s_0..s_i) → P(s_{i+1}))) then
7          return true
8      end
9      i = i + 1
10 end
```

Doing away with some loop-freeness constraints

One limiting factor appears to be adding the n^2 loop-freeness constraints. Two things can be done to help this:

1. It is possible to analyze the FSM to prune some of the constraints, i.e. it is only necessary to consider the beginnings of *basic blocks*.
2. Examine the models returned by the solver in the induction step and add inequality constraints only as needed. This way we need to rerun the solver after adding the constraints. Turns out this method performs better in general anyway.

Analysis

- All presented algorithms are suitable for SAT solvers.
- It is possible to constrain the algorithms to *shortest* paths between states. However this moves it to the QBF (Quantified Boolean Formula) domain, which is not so nice. As the peer papers did not show any results with this method, we will not discuss it further here.
- Middle ground can be found: Check that the paths are *locally shortest*, i.e. between no c non-neighbors in the path there is a transition ($c = 2$ is the simplest case). This makes the problem still stay in boolean clause satisfiability domain.

Incremental approach: Introduction

- One expensive thing in our algorithm is incrementing the depth always by 1 and redoing the computation.
- The induction depth can be incremented by a higher value than 1 at each step; however then the found counterexamples are not necessarily shortest ones.
- It would be beneficial to be able to *incrementally* solve the SAT instances of different depths.
- This turns out to be possible with a very modest modification to general SAT solvers (probably already integrated to the newest solvers).

Incremental SAT: Analysis

- It is fairly easy to *add* new clauses (constraints) incrementally to SAT instances being solved in a DPLL-based solver with conflict analysis and clause recording.
- However, removing clauses fundamentally conflicts with clause recording.
- For many interesting problems, removing clauses is simply necessary.

A solution for clause removal

- Idea: Permit the user to force some literals to true or false for the duration of a single search.
- This has the nice benefit of making *all* learnt clauses safe to keep, and thus there is no need for extra book-keeping.
- The authors of the paper implemented this in 5 lines of added code in some SAT solver.
- Using this approach, general clause deletion can be largely simulated.

Incremental induction

- Our BMC algorithm can be divided in two independent parts that can be run in parallel: The base case ("bug finder") and the induction step ("upper bound prover").
- These algorithms can be nicely implemented incrementally.
- We note that the problem is inherently symmetrical: We want to find a path from an initial state to a bad state through good non-initial states. That can be done forwards or backwards.
- Base case we want to search forwards, since that allows us to keep the often strong formula $I(s_0)$ in the solver. Analogously, the induction step should be done backwards for maximum efficiency.

Some more notation

We define some more notation for our incremental SAT solving. Let

- $[\varphi]^p$ be a set of clauses such that p is the literal representing the truth value of the whole formula. We call p the *definition literal* of φ.
- $[\varphi] \equiv [\varphi]^p \cup p$.
- $I_n \equiv I(s_n)$, $P_n \equiv P(s_n)$ and $T_0 \equiv T(s_n, s_{n+1})$.

Algorithms 5 and 6: Incremental base and induction step

```
1  addClauses([I_0])
2  for n ∈ 0..∞ do
3      addClauses([P_n]^{p_n})
4      solve({¬p_n})
5      if Satisfiable then
6          return Property fails
7      end
8      addClause({p_n})
9      addClauses([T_n])
10 end
```

```
1  addClauses([¬P_0])
2  for n ∈ −1..−∞ do
3      solve({})
4      if Unsatisfiable then
5          return Ind. step holds
6      end
7      addClauses([T_n])
8      addClauses([P_n])
9      for i ∈ 0..n + 1 do
10         addClauses([s_i ≠ s_n])
11     end
12 end
```

Further analysis

- These algorithms can be run in parallel.
- As soon as the induction step succeeds, an unsatisfiable base case of that length will constitute a proof of P.
- Could we still improve uniqueness constraint handling?
- We do not need to consider inputs: If two states are equal except for inputs, we could have fixed the inputs to be same.
- This way we only need to require all delayed elements in the model to be the same, which is a much stronger condition. This is important: each extra state bit can double the depth needed to prove the step.

Further analysis, continued

- On the other hand, sometimes we cannot separate inputs, perhaps we are just given two propositional formulas I and T.
- One solution:
  1. Include only variables occurring *both* in the current and the next state of T.
  2. Do not add uniqueness constraints including the first or the last state of the trace.
- If the correctness of this approach is not obvious, refer to the original paper.

Experimental results

- The ideas were implemented in a prototype tool called *Tip* which was integrated with the SAT solver *Satzoo*.
- Empirical results show that SAT-based induction can be a valuable complementary method to BDD-based methods.
- Tip was able to solve some instances where the BDD-based methods failed.
- On the other hand there were instances where the BDD-based methods were better.

Conclusion

All is well that ends in a cliché.