Model Checking Practicals: Assignment 2 - PDR

April 22, 2021

1 Assignment Summary

The goal of the second exercise in the model checking practicals is to implement the **property-directed reachability** method. The implementation is supposed to rely on the *Model Checking* book, and especially the method presented in the lecture slides. Unlike K-induction, PDR is significantly different from BMC, and requires several custom data structures. In particular, you will even need a second Z3 solver for PDR. Your work for this assignment continues in the same repository with the same team. At the end, both K-induction and BMC have to work as well, so do not break them and test regularly. As before, submissions are done directly in the repository, by creating and pushing the tag "**pdr**". The preliminary submission deadline is the **Thursday 27th of May** end-of-day. The question hour times stay the same, and we provide question hours every **Tuesday from 10:00 to 11:00** where you can ask us implementation related questions. The rest of the document provides more details.

2 Setup

You should already have a repository which you used in the first assignment with your teammates. There are minor changes in the upstream repository to make your implementation efforts easier. You can obtain these changes by pulling from the `upstream` remote and merging the changes. Any further improvements or fixes will be published in that repository and we will notify you as soon as possible.

```
git pull upstream master
# merge here if necessary
git push origin master
```

After implementing everything, you submit the solution by running the following commands.

```
git tag "pdr"
git push origin "pdr"
```

In case you need to fix something after tagging the submission commit, you just update the tag to the new commit.

3 Implementing Property-directed Reachability

In this assignment, your task is to extend your previous work with an implementation of PDR as discussed in the lecture. Since this model-checking method is significantly different from the other two, you will need several custom data structures (e.g. the trace, frames) and members of the `Checker` class (e.g. a separate Z3 solver, the two required states, proof obligation tracking, etc.).

As discussed in the lecture, works with one state transition. This means that you need to store two sets of state variables. Just like in the last assignment, we call these sets V_0 and V_1. In addition, there are two sets of transition equalities T_0, which represents the initial state, and T_1, which represents the transition equalities between V_0 and V_1. Just like with BMC and K-induction from the last assignment, we call C_i and B_i the constraints and bad state properties for the $i-th$ state variables V_i respectively.

First, PDR checks whether the initial state fulfills any bad state properties. If it does, the there is an early termination. Your implementation will check the formula in Equation 1.

$$\left(\bigvee_{b \in B_0} b\right) \wedge \left(\bigwedge_{t \in T_0} t\right) \wedge \left(\bigwedge_{c \in C_0} c\right) \tag{1}$$

Next, for each bound value k, we create a set of clauses F_k, which represents all states reachable in k steps. Importantly, this is a formula over the variables in V_0, as we will pretend that this over-approximation of the post-image is our initial state in calls to the solver.

When checking whether a bad state is reachable in $k+1$ steps, we start from all good states reachable in k steps, and ask the SAT solver whether any bad state property is satisfiable after a state transition. This satisfiability call is shown in Equation 2. Here, the variables V_0 are asserted to be in a good state that is part of F_k, all input constraints hold, and that a variables V_1 reach a bad state, while respecting the state transition.

$$\left(\bigwedge_{b \in B_0} \neg b\right) \wedge \left(\bigwedge_{f \in F_k} f\right) \wedge \left(\bigwedge_{t \in T_1} t\right) \wedge \left(\bigwedge_{c \in C_0 \cup C_1} c\right) \wedge \left(\bigvee_{b \in B_1} b\right) \tag{2}$$

If this check is unsatisfiable, then no bad state is reachable in $k+1$ steps. The algorithm increases k, prepares a new frame and continues with the search.

However, if the formula is satisfiable, then there is a reachable bad state. The solver returns assignments $s_0 : V_0 \to \{\bot, \top\}$ and $s_1 : V_1 \to \{\bot, \top\}$ to the state variables in V_0 and V_1, where s_1 is a bad state, and s_0 is its predecessor.

For example, if we have $V_0 = \{a_0, b_0\}$ and $V_1 = \{a_1, b_1\}$, $s_0(a_0) = \top$, $s_0(b_0) = \bot$, then s_0 can be turned into a cube $p := (a_0 = \top) \wedge (b_0 = \bot)$, which can be rewritten as $p := a_0 \wedge \neg b_0$.

Since F_k is an over-approximation of the reachable states, PDR needs to check whether this is a real problem or just an error caused by the over-approximation. Therefore, PDR attempts to forbid the predecessor of the bad state and therefore make the over-approximation tight again. This is what is done in the `removeBad` function you have seen in the lecture.

Given a variable assignment s_0, PDR replaces the variables in V_0 with variables from V_1. In the example from before, this would yield a cube $p[V_1/V_0] := a_1 \wedge \neg b_1$. This allows PDR to say that if wants find a state at in frame F_{k-1} which reaches the bad state in frame F_k within one transition. This loop is repeated recursively until either, we show that the predecessor of the original bad state is not reachable, or we find a chain of states between an initial state and the bad state we started `removeBad` with initially. The formula in Equation 3 shows an encoding of the satisfiability call you will use in your implementation. The formula $p[V_1/V_0]$ is again a conjunction over literals from V_1.

$$\left(\bigwedge_{b \in B_0} \neg b\right) \wedge \left(\bigwedge_{f \in F_{i-1}} f\right) \wedge \left(\bigwedge_{t \in T_1} t\right) \wedge \left(\bigwedge_{c \in C_0 \cup C_1} c\right) \wedge p[V_1/V_0] \tag{3}$$

PDR will then again use the value assignment to V_0 to call itself again for the previous frame. If however, the formula was not satisfiable, then the function extends all frames $j \leq i$ with the clause $\neg p$. Additionally, the implementation attempts to propagate the learned clause forward to frames $j > i$ as well, so long as it holds there as well.

Generalization. After finding that a state is not reachable in a given frame, we can attempt to generalize it to a set of states by shortening the cube that represents it. In general, this is done so that the new cube $\hat{p}$ contains less literals than p, and still makes the formula unsatisfiable, and does not intersect the initial states. This generalization step can be accelerated by looking at the unsatisfiable core of the assumptions given to the Z3 solver. This can be achieved with `z3::solver::unsat_core`, which gives you all assumptions that Z3 used when proving the formula unsatisfiable. The assignments present in the unsatisfiable core must be part of $\hat{p}$, and for those that are not, you just need to check whether they intersect with the initial states as dictated by T_0.

Counterexamples. Your implementation has to produce counterexample traces, just like BMC. The functions for printing the counter example is already in the template repository, you just need to reconstruct it. To be able to print the counter examples, you just need to keep track of the argumets that `removeBad` pseudocode is called with. In particular, we suggest that you keep a stack of these cubes, also called *proof obligations* in PDR, and instead do implement `removeBad` iteratively.