Model Checking Practicals: Assignment 3 - K-Induction

April 26, 2023

1 Assignment Summary

The goal of the third exercise in the model checking practicals is to implement the **k-induction (KIND)** method. The implementation is supposed to closely follow the *Model Checking* book. Your implementation **must** extend the provided framework to implement KIND using incremental solving with **Z3**, while also supporting BMC from the previous assignment. All work is done in the same `hwmc` directory as the second assignment. The preliminary submission deadline is **Sunday 21st of May** end-of-day. We provide question hours every **Thursday from 18:00 to 19:00** during the practicals timeslot. You can also ask questions using at any time on **Discord**. We will try to provide feedback on the assignment until **Thursday 1st of June**. The rest of the document provides more details.

2 Setup and Submission

You are working in the same repository as for the last exercise, and all of the setup is the same. Additionally, before you begin implementing this exercise, you have to pull from the upstream and update the repository so that you get the framework. You do this with:

```
git pull upstream master
git push origin master
```

After implementing everything, you submit the solution by running:

```
git tag "kind"
git push origin "kind"
```

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

3 Bounded Model Checking

This section briefly recounts the formalization of BMC you should use as a guide for the actual implementation tasks. BMC is an algorithm that unrolls the hardware up to a certain depth and checks whether any bad states can be reached. As such BMC maintains a trace of frames, where each frame corresponds to the state of a circuit in a given clock cycle. Each frame consists of several components.

State variables. Each frame has a set of state variables V. The variables $v \in V$ represent registers and circuit inputs. All of them together, that is V itself, fully determines the state of the circuit.

Formulas. Each frame is also associated with a set of wire formulas W. Each element $w \in W$ represents either a constant, a state variable or some expression combining state variables. Importantly, the next value of each state variable $v \in V$ is associated with some wire $w_v \in W$.

Environment constraints. Each frame also contains a set C of environmental constraints. These are *assumptions* about the inputs of the circuit.

Bad properties. Each frame also contains a set B of bad properties that should never occur. These are essentially negated *assertions* in the circuit design.

Transitions. For the transitions between the $(i-1)$-th and i-th frame, BMC constructs a set of equalities $T_i := \{v = w_v\}$ where $v \in V_i$ and $w \in W_{i-1}$. Using this notation, we can think of the initial state V_0 as being constrained with equalities T_0 where W_{-1} only contains constants, i.e., the initial state variables $v \in V_i$ are set to equal some constants through equalities T_0.

In each BMC step, the implementation tries to find a sequence of states such that the last state in the sequence satisfies a bad state property. In each time step i, the solver tries violate a bad state property, so it tries to solve Equation 1.

$$\left(\bigvee_{b \in B_k} b\right) \wedge \bigwedge_{i=0}^{k} \left(\left(\bigwedge_{t \in T_i} t \right) \wedge \left(\bigwedge_{c \in C_i} c \right) \right) \tag{1}$$

Because the BMC algorithm is iterative, and would have already proven that none of the bad state properties $b \in B_i$ are reachable in $i < k$ steps, we can add them to the problem we are trying to solve, in order to speed up the solving process, as shown in Equation 2.

$$\left(\bigvee_{b \in B_k} b\right) \wedge \bigwedge_{i=0}^{k} \left(\left(\bigwedge_{t \in T_i} t \right) \wedge \left(\bigwedge_{c \in C_i} c \right) \right) \wedge \bigwedge_{i=0}^{k-1} \bigwedge_{b \in B_i} \neg b \tag{2}$$

If any such states are found, BMC terminates and prints the counterexample as a simulation trace for the given circuit. In case none are found, BMC expands the trace by one frame and tries again. Note here, that the bad state property is only checked in the last frame, as the previous iteration show that no bad state is reachable in any of the previous frames.

4 K-Induction

This section briefly summarizes k-induction and you should use it as a guide for your implementation later on. The formulas required for checking KIND and BMC are very similar, so in your implementation, you will reuse the same solver for both.

K-induction, as used in model checking has two phases. The *initiation* phase is the same as BMC and checks whether a bad state is reachable in k transitions. If this phase fails, the algorithm aborts and reports the BMC counterexample. The *consecution* phase, commonly referred to as inductive step, checks whether, given that no bad state is reached in k-1 transitions, a bad state can be reached in the k-th transition. In the case a bad state is not reachable, k-induction has proven that a bad state is never reachable. Otherwise, if the k-th transition reaches a bad state, then k is incremented and the whole process repeats. Equation 3 summarizes the consecution phase.

$$\left(\bigvee_{b\in B_k} b\right) \wedge \bigwedge_{i=1}^{k}\left(\bigwedge_{t\in T_i} t\right) \wedge \bigwedge_{i=0}^{k}\left(\bigwedge_{c\in C_i} c\right) \wedge \bigwedge_{i=0}^{k-1}\left(\bigwedge_{b\in B_i} \neg b\right) \tag{3}$$

Looking at Equations 2 and 3 more closely, we see that they share everything except the transition conditions of the initial state, *i.e.* $\bigwedge_{t\in T_0} t$. This already gives you an idea of how you should implement this.

Furthermore, the K-induction from Equation 3 is not complete. This is because there could be a reachable loop of good states from which a bad state is reachable. The KIND routine would then just repeat these states indefinitely. Therefore, we add a constraint that all of the reached states are different, shown in Equation 4. Here v and v' refer to the same register or input instantiated in different frames. For example, this could be `var42@1` and `var42@2`.

$$\bigwedge_{i=0}^{k}\bigwedge_{j=0}^{i-1}\left(\bigvee_{v\in V_i, v'\in V_j} v \neq v'\right) \tag{4}$$

5 Task 1: Implement K-Induction [8+6 Points]

The forwarding functions were implemented in the previous assignment. For KIND, you will at most need to adapt them slightly. Like before, model checker keeps everything required for BMC and KIND ready.

The difference between BMC and KIND is the initial transition that defines the constraints for the first frame. Until now, this was always pushed into the solver just like every other transition. Now, you need to adapt this so that the constraints are saved by the `Checker` class. Then, in case you are doing BMC, you add them temporarily before calling `z3::solver::check`. For KIND you do not add the initial transition constraints into the solver.

Standard k-Induction [8 Points]. You have to implement the KIND method inside the `check_kind` function. It is triggered by passing the `-kind` command line option. Your implementation follows the same principle as BMC, and almost every part of Equation 3 should already be inside solver after calling `Checker::forward`.

The only missing part of Equation 3 should be $\bigvee_{b \in B_k} b$. Just like for BMC, you should break down the checking for bad states so that every $b \in B_k$ is checked separately. That is, iterate through all bad state properties, add the current one into the solver, and check for satisfiability. If the solver says UNSAT, you are done with this bad property and have proven that it is not reachable. Add it to a list of *impossible bad properties* and skip them when checking higher k later on. Otherwise, if the solver says SAT the bad state property is still reachable and you have to check it for a higher k. In any case, undo the addition into the solver and continue with the next bad property. After checking the bad state properties, return the number of still reachable bad state properties from `check_kind`. If no bad state properties are reachable anymore, print `"unsat"` and terminate.

Complete k-Induction [6 Points]. For the second part of the implementation, implement the complete version of KIND by adding the constraints from Equation 4 into the solver. This should be controlled with, and only enabled when, the `-kcomplete` option is provided. The easiest way of implementing this is modifying `Checker::forward_state` and storing a large concatenation of the state variables for each frame inside the `Checker` class. Then, you can use `z3::distinct` to say that each of the frames is different.

6 Task 2: Adapt BMC [6 Points]

Because of KIND, there might be bad state properties that you have already proven impossible. If there are any, do not check their reachability again with BMC. Configure this to be optionally enabled with the option `-prevopt`.

7 Task 3: Testcases [5×2 Points] + [4 Bonus]

For the last task, you are supposed to implement small hardware modules in BTOR and use them to test your implementation. In particular, you should create small modules with Verilog and compile them into BTOR with the synthesis tool Yosys. These testcases are supposed to show different aspects of your implementation. For each of the following specifications, you should have at least one testcase that produces the desired behavior of your KIND implementation:

1. [**2 Points**] A module with at least **three** bad state properties each of which is proven unreachable at **different bounds**
2. [**2 Points**] A module with **one bad state property** that is proven unreachable at bound **k=5**
3. [**2 Points**] A module where BMC without `-prevopt` is signifficantly slower
4. [**2 Points**] A module that is actually correct with respect to **all** bad properties, but KIND reports reachable violations in the **first 10** cycles
5. [**2 Points**] A module where KIND would **not terminate** without the completeness constraints Equation 4, *i.e.*, without `-kcomplete`

You can implement up to **two** additional testcases and get bonus points for those, as long as they show some interesting behavior. For each of the testcases, you should also create a `protocol-kind-`*number-testcasename*`-.md` file that describes what the testcase does, when the bad state properties are shown unreachable and give the output of your implementation as well.