Model Checking Practicals:
Assignment 3 - K-Induction

May 6, 2024

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 2nd of June** end-of-day. We provide question hours during practicals sessions every Monday at 11:00. You can also ask questions over **Discord** in the dedicated channel. The rest of the document provides more details.

2 Setup and Submission

The tags for this assignment are `kind` and `kind-final`. Only commits with the `kind-final` tag that were submitted before the deadline will be graded. For more details on setting up your wor environment and managing your repository, please consult the guide in the warmup assignment handout.

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. The frame has a set of variables V_i for registers and inputs, and a set of formulas F_i for the intermediate computations of wires. For the transitions between the $(i-1)$-th and i-th frame, BMC constructs a set of equalities $T_i := \{v = w\}$ where $v \in V_i$ and $w \in V_{i-1} \cup F_{i-1} \cup L$ and L is a set of constants. Using this notation, we can think of the initial state V_0 as being constrained with equalities T_0 where $V_{-1} \cup F_{-1} = \emptyset$, i.e., the initial state variables $v \in V_i$ are set to equal some constants through equalities T_0. Additionally, the set of constraints C_i makes sure that the solver respects the assumptions about the circuit's environment.

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. If we call B_i the set of bad state properties in each frame, then the solver 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.

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

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, as it is, 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.

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

5 Task 1: Implement K-Induction [12+10 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.

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`.

For the second part of the implementation, implement the complete version of KIND by adding the constraints from Equation 4 into the solver. 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.

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.

6 Task 3: Testcases [8 Points]

For the last task, you are supposed to implement small hardware modules in (System)Verilog, translate them to BTOR using Yosys and use them to test your implementation of KIND.

```
VLOG_FILE="my_test.v" \
TOP_MODULE="my_test" \
BTOR_FILE="my_test.btor" \
yosys verilog_btor.tcl
```

You can also use the makefile we provide for SystemVerilog support. If you have installed the SV2v utility[1] and have a SystemVerilog file `my_test.sv` in your tests directory, just run:

```
make my_test.btor
```

The idea behind this task is to thoroughly test your implementation. These testcases are supposed to show different aspects of your KIND implementation, e.g., testcases that prove unreachability of bad states at different depths k, tests that would not terminate without you implementing the completeness constraints and so on. Since the tests are in the same directory as the BMC tests, we distinguish them by their reported results on the reference implementation. Tests with bugs are counted towards BMC, whereas tests without bugs are counted towards KIND. Points gained per testcase are exponentially decaying. The first four testcases each give 1 point, the next eight testcases each give 0.5 point for a total of 8 points. Finally, earning points for testcases is going go through randomized manual review, and e.g., submitting 12 testcases that check whether a counter ever reaches the numbers from 1 to 12 is not going to be considered a valid test suite. Moreover, a bad performance on private tests will scale the points you get from writing tests accordingly. For example, if your KIND implementation only correctly solves 50% of our private test suite, your test suite only receives 50% of the points you would have otherwise gotten. You should also document your testcases. If it is not obvious at a glance what you are actually doing in the testcase, it might lead to you not getting any points during the randomized manual review.

---

[1] https://github.com/zachjs/sv2v