Model Checking Practicals: Assignment 1 - K-Induction

March 26, 2021

1 Assignment Summary

The goal of the first exercise in the model checking practicals is to implement the **k-induction** method. The implementation is supposed to closely follow the *Model Checking* book. Your implementation **must** extend the provided BMC implementation using incremental solving with **Z3**. You will receive repositories inside which you implement this task in teams of up to **three people**. Submissions are done directly in the repository, by creating and pushing the tag "**kind**". The preliminary submission deadline is the **Thursday 22nd of April** end-of-day. We provide question hours every **Tuesday from 10:00 to 11:00** where you can ask us implementation related questions. If there is enough interest, we can also give you a short demo of the C++ API you will use. The rest of the document provides more details.

2 Overview

Model Checking is a new course, first introduced this year. While the overall course covers all topics related to model checking, the practicals only look at SAT-based methods. In particular, throughout the practicals, you will implement a full-fledged hardware model checker that could appear at the annual competition [BvH17]. At the end of the practicals, your implementation will support *bounded model checking* (BMC), *k-induction* (KIND), and *property directed reachability* (PDR). You and your teammates are not doing this from scratch of course, and we provide you with a minimal implementation that parses the input format, program options, and tries to check whether a property holds.

3 Setup

As already mentioned in the lecture, you can do this exercise in groups of up to three students. Working in a team is not strictly necessary, you can do the

exercise on your own, but we strongly encourage collaboration. On one hand you have to do less work in principle, and on the other hand you have someone you can bounce ideas at and get a better understanding of the algorithms.

Each student will receive a repository for this exercise. In case you collaborate in a team, you have to choose one of the repositories we provided and do all of your work there. As soon as you create a team, please contact us and we can give you the appropriate access rights.

The repository we provide you with is empty. As soon as the teams are decided, you can declare our template repository as your upstream, and pull the framework we provide from there. Any improvements or fixes will be published in that repository and we will notify you as soon as possible.

```
URL="https://extgit.iaik.tugraz.at/scos/scos.teaching/mc2021.git"
git remote add upstream $URL
git pull upstream master
git push origin master
./mk_submodules.sh
```

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

```
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.

4 Template

This section is meant as a lightweight documentation of the framework you will use in the exercise. It is meant to provide you with details about the features that are already implemented. In principle, you can change pretty much all parts of the program, as long as you do not break it. While this section provides you with an overview, Section 5 contains all the details of your actual task.

4.1 Input Format

The template implementation already includes a lot of things needed for a hardware model checker. Since the benchmarks used at the official competition use the BTOR2 format [NPWB], we include a BTOR2 parser that extracts a circuit from the input file. As a bit-vector format, BTOR2 has its own type system, where each file declares its *sorts* as bit-vectors of a certain length. Furthermore, the format specifies *state* variables which are actually just registers in the hardware design, that include an *init* for reset values, and *next* for flip-flop inputs triggered with a clock. Similarly, circuit each *input* corresponds to one of the signals provided from outside the circuit, and might change in each clock cycle. Assumptions about these inputs are defined using *constraint* properties, which

model the environments interaction with the system. Other than that, all the wires are represented as gate outputs. In contrast to real RTL, BTOR2 includes a few special declarations related to model checking. Out of those, the only interesting one for this exercise is the *bad* property, which defines one condition which makes a state bad. There can be multiple bad state conditions, and if any of them is satisfied, the state is undesirable. Essentially, these are negations of invariant properties which we want to prove using model checking.

4.2 Modeling with Z3

Since we are looking at satisfiability-based model checking methods, we need to perform a translation of the input file into something a solver could understand. In this exercise, we use the Z3 SMT solver [dMB, BdMNW] which already provides a way to represent bit-vectors, and apply bit-vector operations. This makes the translation from BTOR2 rather straight-forward. When representing the state of a circuit in one clock cycle, we use a set of appropriate bit-vector variables together with their names in the BTOR2 input file. For each gate in the design, we recursively construct a formula representing it. Each such formula only uses state variables, inputs in the given clock cycle and constants. For state variables, we tell the SMT solver that their value should be either defined by initial values in the first cycle, or by the computation from the previous cycle. This is achieved by adding equality constraints into the solver which force any satisfiable answer to obey the next state logic and reset logic from the BTOR2 file.

4.3 Bounded Model Checking

The template contains an implementation of BMC. The implementation, as expected, does an unrolling of the circuit. The implementation maintains a trace of BMC frames, where each frame corresponds to the state of a circuit in a given clock cycle. Each frame is constructed as described in the previous section, and 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. 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, the initial transition state of variables V_0, we have a transition T_0 where $V_{-1} \cup F_{-1} = \emptyset$. 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, meaning that it violates an invariant. If we call B_i the set of bad state properties in each frame, then the solver tries to solve Equation 1.

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

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.

5 Implementing K-Induction

Your first task for the practicals is to extend the provided template with your own implementation of k-induction. Your implementation is supposed to reuse the whole infrastructure we provided, including the BMC solver `exp_solver` and trace `exp_trace`. This is something done by all modern model checkers that support both BMC and k-induction. They reuse the same solver, and use assumptions to determine which algorithm they are executing.

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 2 summarizes the consecution phase.

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

Looking at Equations 1 and 2 more closely, we see that they share everything except the conditions of the initial state, and the unreachability of bad states in the first k states.

$$\left(\bigwedge_{t \in T_0} t\right) \wedge \bigwedge_{i=0}^{k-1} \left(\bigwedge_{b \in B_i} \neg b\right),$$

You are allowed to change the implementation of BMC so that you can reuse the solver properly. The mentioned differences already give you a hint of what you need to change and turn on or off depending on the algorithm you are executing. Consider using the `z3::solver::push` and `z3::solver::pop` functions to dynamically create backtracking points in the solver so that you can add and remove parts of the formula at will. An example of this is already used in the BMC implementation.

References

[BdMNW] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph M. Wintersteiger. Programming Z3. In Jonathan P. Bowen, Zhiming Liu, and Zili Zhang, editors, *Engineering Trustworthy Software Systems - 4th International School, SETSS 2018, Chongqing, China, April 7-12, 2018, Tutorial Lectures*.

[BvH17] A. Biere, T. van Dijk, and K. Heljanko. Hardware model checking competition 2017. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 9–9, 2017.

[dMB] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*.

[NPWB] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , btormc and boolector 3.0. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*.