Model Checking Practicals:
Assignment 2 - Bounded Model Checking

April 6, 2023

1 Assignment Summary

The goal of the second exercise in the model checking practicals is to implement the **bounded model checking (BMC)** method. The implementation is supposed to closely follow the *Model Checking* book. Your implementation **must** extend the provided framework to implement BMC using incremental solving with **Z3**. All work is done in the same repository as the last exercise, only in the `hwmc` directory instead of `warmup`. The preliminary submission deadline is **Sunday 7th 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 18th of May**. 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 "bmc"
git push origin "bmc"
```

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

3 Input Format

The framework 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 parts of 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, 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 the properties you want to reach when executing your BMC routine.

4 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 F_{-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.

5 Task 1: State Forwarding [20 Points]

In the framework we provide to you, the state of a circuit is stored as a map between BTOR indices and Z3 expressions in the `ExprMap` data structure. Similarly, the datatypes from the BTOR file are also stored in a similar map `SortMap`.

```
typedef std::map<int64_t, const z3::sort> SortMap;
typedef std::map<int64_t, const z3::expr> ExprMap;
```

According to the notation from before, you would store all variables $v \in V_i$ and expressions over the variables $w \in W_i$ inside such a `ExprMap` data structure. The *trace* is then just a vector containing such *frames*.

Forwarding is then just creating a new full frame, based on the previous frame. In the framework, you have to implement the following functions:

```
void forward_wires(Btor2Parser* parser, ExprMap& curr_state);
void forward_state(Btor2Parser* parser, ExprMap& curr_state,
       const ExprMap& prev_state, z3::expr_vector& eqs, uint32_t
           step);
static void forward_cons(Btor2Parser* parser, ExprMap& curr_state,
              z3::expr_vector& cons);
void forward(Btor2Parser* parser, const Options& opt, uint32_t step
    );
```

The function `forward` is main forwarding function that is called later by your BMC algorithm implementation to create a new frame. Internally it calls the other forwarding functions. It creates the state variables and inputs in the new frame with `forward_state` and constrains them with the transition equalieies (T_i from before). The transition equalities are determined based on the declared `next` statements from the BTOR file. Afterwards, `forward` calls `forward_wires` to determine the Z3 expressions representing all the wire values and storing them in the current frame. Finally, `forward` calls `forward_cons` to generate the environmental constraints and add them to the solver. After `forward` finishes, the current frame is completely generated and constrained properly, so that the caller can perform checks. Importantly, you should implement these functions as generally as possible, so that you can also use them with K-induction in the next exercise.

6 Task 2: Implement BMC [7+3 Points]

Standard BMC [7 Points]. After finishing forwarding functions, you are ready to implement the actual BMC routine. The model checker keeps everything required for BMC ready, meaning that at the point at which the `check_bmc` function is called, you just need to add the bad properties into the solver and perform the actual sat solver call. In other words, you can assume that the solver already contains

$$\bigwedge_{i=0}^{k} \left(\left(\bigwedge_{t \in T_i} t \right) \wedge \left(\bigwedge_{c \in C_i} c \right) \right). \tag{3}$$

Here, you should break down the $\bigvee_{b \in B_k} b$ expression into multiple solver calls. That is, iterate through all bad state properties, add the current one into the solver, and check for satisfiability. If the solver says SAT, you are done and return the index of the bad state property. Otherwise, you undo the addition of the bad property into the solver and check the next one. In case there are no bad state properties that are satisfiable, return -1 from `check_bmc`. The return value is handled in the `check` function, and prints the counterexample you found if there is one.

Improved BMC [3 Points]. With the implementation as described above, you have essentially implemented the checking as done in Equation 1. For the second part of this task, think about and implement the second *optimized* Equation 2 for the solver call. Keep in mind that you are only doing this optionally in the solver option `-prevopt` is provided through the command line.

7 Task 3: Testcases [5×2 Points] + [6 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 cre-

ate 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 BMC implementation:

1. [**2 Points**] A module with a state machine of at least **eight** unique states, one of which is associated with a bad property
2. [**2 Points**] A module that has at least **two** environmental constraints and **three** bad properties that are violated at **different** BMC depths
3. [**2 Points**] A module with **at least 32** state bits, that reaches a bad state at **k=7**
4. [**2 Points**] A module that is actually correct with respect to **at least one** bad proeprty, and BMC reports no violations in the **first 16** cycles
5. [**2 Points**] Implement a module that does something interesting when analyzed with BMC (take inspiration from the other tests)

For each of the testcases, you should also create a `protocol-`*number-testcasename*`-.md` file that describes what the testcase does, what the expected output is and in case it is satisfiable at some bound, give the output of your implementation as well.

Bonus [6 Points]. You can implement up to **three** additional testcases and get **2** bonus points for each, as long as they show some interesting behavior.

References

[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*.