# Model Checking Practicals: Assignment 2 - Bounded Model Checking

April 25, 2024

## 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 **Wednesday 22nd of May** end-of-day. We provide question hours on **Mondays from 11:00 to 12:00** after the lecture. Feel free to contact us via **Discord** if you have any additional assignment-related questions. Moreover, make use of the test system where you can see how well your implementation performs on our tests and how well it does compared to other students. The rest of the document provides more details.

## 2 Setup and Submission

The tags for this assignment are `bmc` and `bmc-final`. Only commits with the `bmc-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 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. 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.

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

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`.

According to the notation from before, you would store all variables $v \in V_i$ and expressions over the variables $f_i \in F_i$ 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 [8 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`. With this implementation, you have essentially implemented the checking as done in Equation 1. For those inclined to do more, you can think about and implement the second *optimized* Equation 2 for the solver call. Does this make the performance better? Test it!

## 7 Task 3: Testcases [12 Points] + [4 Bonus Points]

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

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

The idea behind this task is to thoroughly test your implementation. These testcases are supposed to show different aspects of your implementation. Points gained per testcase are exponentially decaying. The first two testcases each give 2 points, the next four testcases each give 1 point, and the next eight testcases each give 0.5 points, for a total of 12 points. If you *really* like testing, you can also get an additional 0.25 points for each of the next 16 testcases you create after that for a total of up to 4 bonus points. Finally, earning points for testcases is going go through randomized manual review, and e.g., submitting 30 testcases that check whether a counter ever reaches the numbers from 1 to 30 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 BMC 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.

Here are a few test ideas:

- Test all the operators that the format supports. Write a few tests, each of which peforms 3 or more different operations with the state and inputs and check for reaching a bad state.

- Create different state machines without a real data path. Examples can include simplified examples of traffic lights, arbiters, dishwashers, soda machines, a kettle, microwave, refrigerator light, a football game, the check-engine light in your car, the process of catching a Pokemon, winning in a fighting game, etc.
- Create modules that process a lot of data, but in reallity have very little control state. Examples can include modules for big arithmetic operations with accumulators, quirky computations exploiting overflow logic, toy CPU arightmetic-logic-units.
- Test what happens when your modules have multiple bad state properties, multiple environment assumptions, disjoint bad state sets, or modules where all states are bad due to specification issues.
- Think about modules which reach their bad state properties at a very late point in the computation, e.g., after 50 states. Con your implementation handle such cases?
- Also write modules that do not have any bugs in them, *i.e.*, although there are bad states, none of them are reachable because the implementation is correct. Every test module you write can be like this: have one implementation where the bad state property is reached due to an intentional "bug" you introduced, and have a fixed version that does not reach the bug in the first e.g., 100 states.

## 8 Analyzing the Output

As you may notice, we have already given you the function which prints the found bug in the BOTR2 witness format. Therefore, after running one of your tests, you can plug the output into the `btorsim` utility to see what the solver has found and analyze it further. Here is an example of how this might work:

```
./build/hwmc --btor2-file \
  tests/counter.btor > results/counter.btor
./btor2tools/build/bin/btorsim \
  -v tests/counter.btor results/counter.btor
```

The `btorsim` utility will then give you a detailed trace of what the hardware module was doing accoring to your model checker. Moreover, it will also tell you if there is a bug in your model checker and the result is in any way wrong!

```
[btorsim] checking mode: both model and witness specified
[btorsim] reading BTOR model from 'tests/counter.btor'
[btorsim] reading BTOR witness from 'results/counter.btor'
[btorsim] parsing unknown witness 1
[btorsim] initializing states at #0
#0
[btorsim] initializing inputs @0
@0
0 0001 add@0
```

```
1 0 clk@0
2 0 rst@0
[btorsim] simulating step 0
[btorsim] transition 1
[btorsim] initializing inputs @1
@1
0 0001 add@1
1 0 clk@1
2 0 rst@1
[btorsim] simulating step 1
[btorsim] transition 2
[btorsim] initializing inputs @2
@2
0 0001 add@2
1 0 clk@2
2 0 rst@2
[btorsim] simulating step 2
[btorsim] transition 3
[btorsim] initializing inputs @3
@3
0 0001 add@3
1 0 clk@3
2 0 rst@3
[btorsim] simulating step 3
[btorsim] transition 4
[btorsim] initializing inputs @4
@4
0 0000 add@4
1 0 clk@4
2 0 rst@4
[btorsim] simulating step 4
[btorsim] all 1 bad state properties reached
[btorsim] reached bad state properties { b0@4 }
[btorsim] constraints always satisfied
.
[btorsim] finished parsing k = 4 frames
[btorsim] finished parsing 1 witnesses after reading 226 bytes (0.0
     MB)
```

## 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.*