Model Checking Practicals: Assignment 1 - Warmup

March 17, 2022

1 Assignment Summary

The goal of the first exercise in the model checking practicals is to get familiar with the SMT solver Z3 and hardware circuits in Verilog. In this exercise, you will check properties of simple C programs, either finding and fixing bugs or proving they are correct, write small state machines in Verilog, and apply a half-automated BMC algorithm to a hardware implementation.

You should have already received GIT repositories in which you will implement all of the exercises. Submissions are done directly in the repository, by creating and pushing tags. The preliminary submission deadline is **Friday 8th of April** end-of-day. We provide question hours every **Wednesday from 10:00 to 11:00** on **Discord** where you can ask us implementation related questions. You will get the feedback and assignment points by **Friday the 15th of April**.

The rest of the document provides more details.

2 Setup

You should have received an email that grants you access to a GIT repository intended for the model checking exercises with some group number XX. The repository we provide you with is empty. Therefore, as a first step, you have to 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.

First, we suggest that you set up an SSH key to make everything easier. GitLab provides a good tutorial. First, clone your repository from our GIT server, declare the upstream remote and pull the framework. For group number XX, you should do something like this:

```
URL1="https://git.teaching.iaik.tugraz.at/mc22/mc22gXX"
URL2="https://extgit.iaik.tugraz.at/scos/scos.teaching/mc/mc2022.git"
git clone $URL1
cd mc22gXX
```

```
git remote add upstream $URL2
git pull upstream master
git push origin master
./mk_submodules.sh
```

After implementing everything, you submit the solution by running:

```
git tag "warmup"
git push origin "warmup"
```

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

3 Template

After setting up the repository and pulling from the upstream and building the submodules, you should have everything you need to implement the tasks. For this assingment, you will primarily work in the `warmup` directory. Inside, there is a `CMakeLists.txt` file which is used by CMake to generate the makefiles that will build your implementations. You should create a build directory here and configure it when you start working on the tasks.

```
mkdir build && cd build
cmake ..
```

Afterwards, you should be able to just call `make` inside the build directory to compile your implementation. Most importantly, all the tasks in this assignment have files and targets associated with them. Inside, there is usually a clearly marked part of the implementation you are supposed to complete. You should only edit those parts so as not to break unrelated parts of the code, or our semi-automated testing.

4 Task 1: Magic Square [6 Points]

In the first task, your goal is to get acquainted with Z3 and using it to solve a simple puzzle. The puzzle at hand is the existence of a magic square. Put simply, we want to write constraints describing the problem, so that the solver can automatically find a set of numbers which fulfill all of the given properties. In this exercise, you should learn more about the basic workflow of using a solver. All of your work should be constrained to the `square.cpp` file. At the end, your implementation must output a correct magic square found by Z3. Below, we discuss the details of the implementation, which should serve as a guide on how to solve the task.

First the variable context and solver are created with `z3::context` and `z3::solver`. You can think of a context as the variable storage, which tells the solver which variables exist, their name and types. The solver itself only contains constraints which we give it.

Because we are working with magic squares, we first create the number matrix of variables. This is the task of the context. Confusingly, Z3 variables of appropriate types are created with `z3::context::int_const`, `z3::context::bool_const`, or `z3::context::bv_const`. Real constants that are fixed and not decided by the solver are accordingly defined with `z3::context::int_val`, `z3::context::bool_val`, or `z3::context::bv_val`. We only use unbounded integers in this task, but other tasks and the upcoming assignments focus much more on Booleans and fixed-size bitvectors.

Adding the constraints for magic squares should be straightforward. Essentially, every row, column and diagonal must add up to the same number. With the Z3 API for C++ it is extremely easy to formulate such constraints, because each operator is overloaded for the `z3::expr` class which we use for variable and expression storage. Therefore, creating the sum of unbounded integers is as easy as writing `expr1 + expr2`. Same goes for the logical equivalence operator `==`. However, you have to be careful about the typing, because the type system of the expressions is dynamic, and you might get errors at runtime. For example adding a variable created with `z3::context::int_const` to another variable created with `z3::context::bv_const` would crash your program.

The constraints are added into the solver with the function `z3::solver::add` and must be of Boolean type. The most complex constraint you require in this task is saying that all of the numbers the solver chooses must be different. Z3 provides the function `z3::distinct` for exactly this purpose. However, the expressions you pass into the function must first be stored in a `z3::expr_vector`, which can be created by specifying a context, and then filled with `z3::expr_vector::push_back`.

Actually solving the formula that we added into the solver is done with `z3::solver::check`. If the problem is satisfiable, the solver has internally created a solution in the form of a model. You can obtain the model with `z3::solver::get_model` and then evaluate variables or even expressions with `z3::model::eval`. However, when evaluating an unbounded integer, or any other Z3 type for that matter, the model will return an immutable value of the appropriate Z3 type. In the case of integers, they can be converted back into C++ integers with e.g., `z3::expr::get_numeral_int64`.

4.1 Task 2: Min-Max Search [10 Points]

In this task, we want to actually use the Z3 solver for something more useful than puzzles. More precisely, we want to actually verify that a function implemented in C is correct for any input we give it. For this purpose, you have to manually edit the code of the function so that the variables it uses are symbolic Z3 variables which the solver can then pick and try to break the guarantees provided in the program. That is, you must construct a problem for Z3 in such a way, that when it has a solution, we know that the implementation in C has a bug, and the solution is actually a counterexample. The greatest takeaway of this task should be the concept of single static assignments and property checking. At the end, the file `minmax-z3.cpp` should contain your implementation which performs the transformation, runs Z3 and prints out the verification result. Your implementation will be checked manually. Below, we have prepared a guide on how to properly transform the C code and perform the verification.

The function we want to verify is implemented in C and searches through a list to find the values of the largest and smallest element in an array.

```
unsigned max = arr[0];
unsigned min = arr[0];
for (int i = 0; i < ARRAY_LENGTH; ++i) {
    if( arr[i] < min ){ min = arr[i]; }
    if( arr[i] > max ){ max = arr[i]; }
}
```

At the end of the execution, we expect the found elements to fulfill the properties we were looking for. In the original C code, these were written using assertions.

```
assert(max >= min);
for (int i = 0; i < ARRAY_LENGTH; ++i)
    assert(max >= arr[i] && min <= arr[i]);
```

First, we must determine which variables in the function have fixed values and which ones can change in each execution of the function. When doing this, we see that the variable `int i` does not depend on the symbolic input. In contrast, the `unsigned max` and `unsigned min` variables do depend on the inputs, which can change every time.

Single static assignment is a way of writing programs so that each variable is only assigned one time and always to the same expression. This is something we have to do in order to transform the program into a formula for Z3.

We do this as follows. Every time a new variable is created in C, we create a new Z3 variable. Afterwards, every time the variable is assigned in C, we first create a new temporary Z3 variable, and turn the assignment into an equality that is given to Z3 as a constraint. This is illustrated in the following program excerpt. You should do the same for this task.

```
int x = 0;      // create z3 var. x_0, assert x_0 == 0
x += 3;         // create z3 var. x_1, assert x_1 == x_0 + 3
x = x * x - 5; // create z3 var. x_2, assert x_2 == x_1 * x_1 - 5
```

When doing verification, we have to model the input array using Z3 symbolic variables. In particular, since the `unsigned` type is 32-bit long in C, we have to create an appropriate amount of fixed-size bitvectors with length 32. You can do this `z3::context::bv_const`. Similarly, create symbolic variables that old the values of `max` and `min`.

Now, apply the single-static transformation, but only on the Z3 variables. Any time a symbolic variable would be overwritten, create a new one and constrain it appropriately with `==`. One problem you will encounter are `if` statements that take symbolic variables as input. Each time you encounter such a situation, you have to translate both branches just like before. Then, for each variable that is assigned in either branch, create copies that represent the value after the `if` executes. At that point, the value of the variable will be either the final value in the *then* branch or the final value in the *else* branch, depending on the condition. You can encode this using `z3::ite`. Here is an example that should make this clear.

```
            // create z3 var. x_0 and y_0
if (x < 0)  // create z3 var. cond, assert cond == x_0 < 0
{ y = x;  } // create z3 var. y_1, assert y_1 == x_0
else
{ x = -5; } // create z3 var. x_1, assert x_1 == -5
            // create z3 var. x_2 and y_2
            // assert x_2 == z3::ite(cond, x_0, x_1)
            // assert y_2 == z3::ite(cond, y_1, y_0)
```

After transforming the implementation, you have to transform the guarantees as well. In particular, every `assert` in the program describes the negation of a bad property. That is, if all asserts succeed, and none of them crash the program, everything is fine. On the contrary, if even one assert does not hold, the program crashes, and we have found a bug. The same is true when verifying. You essentially collect all of the negated assert statements, and then add their logical disjunction into the solver. In Z3, you can negate expressions with the `!` operator, and create disjunctions with `z3::mk_or`. The argument to this function is a `z3::expr_vector` you have to prepare beforehand.

Finally, if you run your symbolic version of the algorithm and give it to the solver for checking, it should say that the formula you gave it is unsatisfiable. If that happens, it means that the solver was not able to find a violation of the assertions, no matter what the inputs are.

4.2 Task 3: Hardware Modeling [14 Points]

The last task of this assignment concerns hardware, and in particular, modeling of hardware with Z3. The goal of this task is to familiarize you with the hardware execution model, the way we can symbolically represent hardware in order to check if certain properties are fulfilled. Most importantly, however, you should get a feeling for unrolling hardware over its transition function.

More concretely, in this task, you are given a hardware module, and you have to model the functions of every single wire in the hardware module. Afterwards, you should *unroll* the hardware and check whether it satisfies safety constraints within the first five cycles. All of the work you do for this exercise will be inside `counter-z3.cpp`. Additionally, you have to submit `counter-protocol.md` where you describe what you implemented, as well as the responses Z3 were when checking the satisfiability of the generated formulas. In the following, we provide a guide on how to tackle this exercise.

The hardware we are looking at in this example is a simple counter module implemented in `counter.v`. The counter has an internal 4-bit wide register `cnt` which it uses as an accumulator. Furthermore, the module has a 4-bit input `add` which represents by how much the counter is supposed to increase in each clock cycle. Finally, the `rst` signal tells the module to clear the contents of cnt instead of updating them. An excerpt of the corresponding Verilog code is shown below.

```
initial cnt = 0;
always @(posedge clk) begin
    cnt <= cnt + add;
    if (rst) cnt <= 0;
end
```

For safety reasons, we want to make sure that cnt never reaches the value 4 at which point the system crashes. Similarly, we know that the input add is always driven in a such a way that the inner 2 bits are always 0. This is written in Verilog as:

```
assert property (cnt != 4'b0100);
assume property (add[2:1] == 2'b00);
```

The shown hardware implementation can be broken down into several pieces. Every hardware implementation operates on some kind of state. For our purposes, the state of a circuit is defined by its inputs and register values in any given clock cycle. This is something we have to model symbolically. For this purpose, we have provided you with the state_store data structure, which the name of a state component onto its symbolic value. In the implementation, you will have to manipulate this data structure in various ways. In particular, you will have to implement the function create_state, which constructs the state using std::map::emplace in a given clock cycle. The symbolic values of state elements can similarly be retrieved with std::map::at.

Another piece of the implementation is the next-state function. That is, the function which describes how states are updated whenever the clock has a positive edge. In this example, this only concerns the cnt register. You have to implement this functionality inside the given get_trans function. Finally, there are all the assumptions and assertions that must be modeled. You will implement them as part of init_state, get_asserts and get_assumes.

For the unrolling part of the implementation, you must create a new state, constrain it with the transition function and the environmental assumptions. Checking whether the assertion can be violated involves adding the negated assertion into the solver and checking satisfiability. If Z3 says the problem is satisfiable, it will provide a counterexample. Otherwise, assuming we unrolled n times, we know that there cannot be a violation within the first n cycles.

However, after adding the negated assertion and getting an unsatisfiable result, the solver is essentially poluted, and no matter what you do, it will always return UNSAT. This issue can be mitigated by using incremental solving. In Z3, anything added to the solver after calling a z3::solver::push will be reverted when calling z3::solver::pop. You should use this to your advantage for the unrolling portion of the task.