Model Checking Practicals: Assignment 1 - Warmup v1.1

March 20, 2023

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 **Sunday 9th of April** 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 20th 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/mc23/mc23gXX"
URL2="https://extgit.iaik.tugraz.at/scos/scos.teaching/mc/mc2023.
    git"
git clone $URL1
cd mc23gXX
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"
```

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.

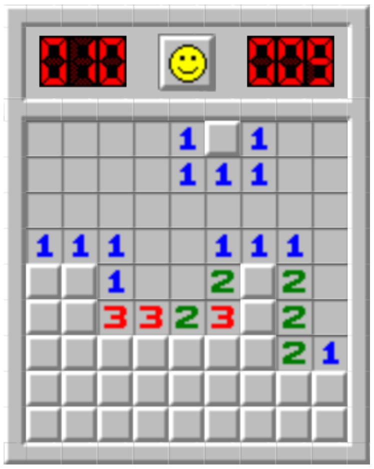

Figure 1: Example *Minesweeper* game state

4 Task 1: Minesweeper [10 Points]

In the first task, your goal is to get acquainted with Z3 and using it to solve a fun puzzle game. Minesweeper is a pre-installed game on many Windows operating systems and Linux desktop environments. The game is set up on a $n \times n$ grid, where initially all fields are hidden. After opening up a field, it can either be a *mine*, a *number*, or *empty*. Opening a mine means that the game is over and that the player lost. A filed with number m means that, out of the 8 neighboring fields there are m mines. If the player clears all non-mine fields, the player wins the game and can enter their name on the scoreboard. An example of a Minesweeper field is shown in Figure 1.

In this exercise, we will consider Minesweeper games that have already been started. Using the current game state, your task is to determine all fields are *safe* (guaranteed to not be a mine), as well as all fields that are *deadly* (guaranteed to contain a mine) using Z3. At the end, your implementation is supposed to output the state of the game, where all safe and deadly fields (which have not already been uncovered) are marked appropriately. Implement the functionality inside the file `mines.cpp`. Below, we discuss the details of the implementation, which should serve as a guide on how to solve the task.

4.1 Input and Output

Your program will receive input in the format shown in Listing 1. Each character represents a field in the Minesweeper game state. Numbers represent fields that are either empty or contain a number indicating neighboring mines, the character `?` represents an unopened field. The output of your implementation is going to label all safe unopened fields with `S` and all deadly fields with `D`, keeping the same format otherwise.

```
00001?100
000011100
000000000
111001110
??1002?20
??3323?20
???????21
?????????
?????????
```

Listing 1: Example input corresponding to Figure 1

4.2 Modeling

In order to solve this problem with Z3, you will need to model the state of the game. As a first step, before doing anything else, you have to create a variable context and solver 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 the constraints you provide.

Since the game state is organized as a two-dimensional array of fields, you will need to do something similar in the modelling with Z3. Here, we suggest that you create a two-dimensional array of Z3 variables (*unknowns*), each representing whether the corresponding field is a mine or not.

This is the task of the context. Confusingly, Z3 variables of *integer*, *Boolean* and *bit vector* types are created with the functions `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 defined with `z3::context::int_val`, `z3::context::bool_val`, or `z3::context::bv_val`. The context will return a `z3::expr` expression representing your variable. In general, working with the solver will involve creating and manipulating `z3::expr` objects.

In this task, it is easiest for you to use integer variables, whereas other tasks and the upcoming assignments focus much more on Booleans and bit vectors.

4.3 Constraints

After creating all the variables needed to represent the state of the game, it is necessary to constrain them to reasonable values. In this case, each field can either contain a mine or not contain a mine. Therefore, constrain each variable so it can only be set to the values 1 (mine) and 0 (no mine).

With the Z3 API for C++ it is extremely easy to formulate all kinds of constraints. All C++ operators are overloaded in various ways to enable easier manipulation of expressions stored in `z3::expr` objects. This includes arithmetic operators like `+` and `-`, as well as logical operators like `!`, `==`, `||` and `&&`. 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. Same goes for addition of *Boolean* expressions, or negation of *integer* expressions.

In order to constrain your variables to be either 0 or 1, you should use the equivalence operator `==` and logical or operator `||` to create corresponding expressions. For already open fields, we know that they do not contain a mine, so create expressions that force the variable to equal 0 instead. To actually tell the solver that it needs to satisfy the constraints, you have to call the function `z3::solver::add` with the Boolean expressions you just created.

Finally, the most complex rule in Minesweeper concerns open fields that contain a number. The number in the field represents the number of surrounding fields that contain mines. In order to encode this for the solver, create a sum of all variables surrounding an opened number, and tell the solver that the sum must equal the desired number of mines. After adding the expression into the solver with `z3::solver::add`, you have finished encoding the rules of the game.

4.4 Iterative Solving

After creating the variables and constraints of the game, it is time to let Z3 solve the problem we are interested in. As stated previously, a field is *safe* if it is guaranteed to not contain a mine. This means that there is no possible assignment the solver could come up with, that places a 1 (mine) into the given variable (field). That is, you have to create a constraint saying the variable is equal to 1, and then check if the solver is able to satisfy all constraints. If it is not, the field is *safe*. A similar argument can be made for fields that are *deadly*, so guaranteed to be a mine.

Since there are many unopened fields we are interested in, we want to only temporarily add such assumption constraints into the solver. The Z3 API enables this with the functions `z3::solver::push` and `z3::solver::pop`. Calling push, creates a new *solving frame* and any constraints you add into the solver later, are only temporary. All constraints added after the last push are removed when calling pop. Therefore, whenever you want to check an unopened field, first do a push, add your constraints, let the solver check them, and then pop them again.

Actually solving the formula that we added into the solver is done with `z3::solver::check`. If the problem is satisfiable, the solver will return the value `z3::check_result::sat` and `z3::check_result::unsat` otherwise.

4.5 Looking at Solutions

If you are interested in the solution Z3 came up with if the problem is satisfiable, it provides a model assigning values to all variables. 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`.

5 Task 2: Absolute [10 Points]

In this task, you will use the Z3 solver for something more useful than games. 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. In this task, you have to do the modelling once with Z3 *integers* and once with Z3 *bit vectors*. Most of the implementation is common, whether the variables are integers or bit vectors. The shared part of your implementation should be implemented in `absolute-shared.cpp`, while the type specific parts are to be implemented in `absolute-bv.cpp` and `absolute-int.cpp`. Your implementation should only print the verification result. If the program is correct, print "correct", otherwise print "bug: arr = {*numbers...*}". Your implementation will be checked automatically and manually. Below, we have prepared a guide on how to properly transform the C code and perform the verification. In any case, we highly recommend that you use the provided SSA datastructure `ssa_info` and keep the bulk of your changes inside the `absolute_sum` and `test` functions in `absolute-shared.cpp`.

5.1 Target

The function we want to verify is implemented in C and computes the sum of absolute values in an array.

```
int sum = 0;
for (int i = 0; i < ARRAY_LENGTH; ++i) {
    if( arr[i] < 0 ){ sum -= arr[i]; }
    else            { sum += arr[i]; }
}
```

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

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

5.2 Single Static Assignment

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 `int sum` depends 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 var. x_0, assert x_0 == 0
x += 3;         // create var. x_1, assert x_1 == x_0 + 3
x = x * x - 5; // create 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. For the modeling part of this task, you should use Z3 integers when implementing the task in `absolute-int.cpp` and bit vectors when implementing the task in `absolute-bv.cpp`.

Create a variable of appropriate type for all elements of `arr`, as well as `sum`. 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 var. x_0 and y_0
if (x < 7)     // create var. cond, assert cond == x_0 < 7
{ y = x;   }   // create var. y_1, assert y_1 == x_0
else
{ x = -5; }    // create var. x_1, assert x_1 == -5
               // create 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)
```

5.3 Guarantees

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. If the formula you gave it is unsatisfiable, it means that the solver was not able to find a violation of the assertions, no matter what the inputs are. Therefore, print ”correct” to the output. Otherwise, if the solver finds a solution, it has found a bug. In that case, get the satisfying model and print the original contents of arr as described before.

6 Task 3: Hardware Modeling [10 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.cpp`. Additionally, you have to submit `counter-protocol.md` where you describe what you implemented, as well as the responses of Z3 when checking the satisfiability of the generated formulas. In the following, we provide a guide on how to tackle this exercise.

6.1 Hardware

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 8 at which point the system crashes. Similarly, we know that the input `add` is always driven in a such a way that the upper 2 bits are always 0. This is written in Verilog as:

```
    assert property (cnt != 4'b1000);
    assume property (add[3:2] == 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`.

6.2 Unrolling

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.

7 Errata

7.1 Version v1

Initial release of the assignment

7.2 Version v1.1

Formatting changes, added clarifications, fixed typos

- cleared up that deadline is on a Sunday
- added highlighting to bash, C and Verilog syntax
- added clarification about implementation of absolute
- changed typo in Task 3, from **[2:1]** to **[3:2]**.