Model Checking Practicals: Assignment 1 - Warmup

March 18, 2024

1 Assignment Summary

The goal of the first exercise in the model checking practicals is to get familiar with the SMT solver Z3 and symbolic representation. In this exercise, you will learn how to work with the Z3 C++ API and how to check properties of simple programs using Single Static Assignment.

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 21st of April** end-of-day. We provide question hours every **Monday from 11:00 to 12:00** during the practicals timeslot. You can also ask questions using at any time on **Discord**. This year there is an automated test system, which will give you feedback for your submission whenever you push a tag to your repository. 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="git@git.teaching.iaik.tugraz.at:mc24/mc24gXX.git"
URL2="https://extgit.iaik.tugraz.at/scos/scos.teaching/mc/mc2024-public
    .git"
git clone $URL1
cd mc24gXX
git remote add upstream $URL2
git pull upstream master
git push origin master
./mk_submodules.sh
```

Since we make heavy use of the Z3 solver library, please make sure that the version `4.8.12` is installed on your system, so that you can develop with a version that is compatible with the one we use. You can do this with most package managers on Linux distributions, e.g. for Ubuntu:

```
sudo apt install libz3-dev=4.8.12-1
```

Alternatively, we also provide you with a Docker image that is compatible with the Docker image used for the automated test system, which you can use to build and test your implementations. To build the docker image and run a shell in it interactively, you can do the following:

```
docker build -t mc_docker .
docker run -it -v $(pwd):/mnt/data mc_docker /bin/bash
```

After running this in your terminal, you should be inside the docker container, and your files should be mounted to `/mnt/data`. From here, you can change into the given directory and perform all of the building and testing steps you require.

When you are confident that your implementation is good and want to test it with the test system, create a tag and push it. Pushing without a tag assumes that you are not done and does not run the test system. The results of the test system will be visible in a CI pipeline in your repository, and pooled on the test system website[1] as soon as it becomes available.

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

After implementing everything, you should submit the solution by running:

```
git tag "warmup-final"
git push origin "warmup-final"
```

Your code only counts as submitted when you tag it with the `warmup-final` tag, and will not be graded otherwise. This means that you can still use the test system without getting a bad grade if you decide to stop participating in the practicals before the second assignment.

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 are two sub-directories containing the two warmup tasks. Inside each, there is a `CMakeLists.txt` file which is used by CMake to generate build scripts that compile your implementations. You can use the following to create a `build` directory with the build scripts, and then run them.

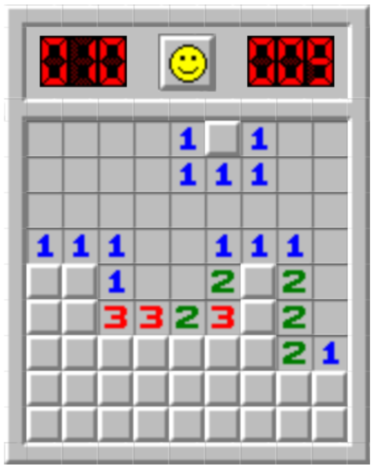
[1] https://mc.student.iaik.tugraz.at/

```
cmake -B "build" .
cmake --build "build"
```

You can use the second command whenever you want to re-build your implementation. For each of the tasks, there is usually a clearly marked part of the implementation you are supposed to complete. Moreover, there is extensive documentation for the provided framework code. You should only edit the parts labeled with `@todo` so as not to break unrelated parts of the code, or the automated testing.

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 within the 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 1a.



(a) Game state

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

(b) Text input

Figure 1: Example of a *Minesweeper* state and a corresponding text input

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 Figure 1b. 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.

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 the other task 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 logical 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

```
int num = input();
assume(num >= 0);
int low = 0;
int high = num;
int sqrt = low;
repeat (17) {
    int mid = (low + high) / 2;
    print(mid);
    if (num < (mid*mid))
    {
        high = mid;
    } else if (num > (mid * mid))
    {
        low = mid;
    }
    sqrt = mid;
}
print(sqrt);
int num_l = (sqrt - 1) * sqrt;
int num_m = sqrt * sqrt;
int num_h = (sqrt + 1) * sqrt;
print(num_l); print(num_m); print(num_h);
assert((num_l <= num) && (num_h >= num));
```

Figure 2: Example program computing the square root of a number

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: Single Static Assignment [20 Points]

In this task, you will use the Z3 solver for something more useful than games. More precisely, we want to actually verify that simple programs are correct for any input we give them. For this purpose, you have to symbolically execute the program by replacing regular variables with Z3 variables and checking whether it is possible to find an assignment to these variables that reaches a false assertion. That is, you must construct a problem for Z3 in such a way, that when it has a solution, we know that the given program 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, your verifier should either print ”Correct!” if the program cannot reach a false assertion, and print ”Found bug:”, followed by the counterexample. Below, we have prepared a guide explaining the language in which the programs are written, as well as the concept of single static assignment.

5.1 The Simplelang Language

In this task, you will have to verify programs in the custom simplelang language. The language is implemented in a framework called PEGTL, and you are already given both an interpreter and a parse-tree visualization tool. Your task is to create a symbolic interpreter based on the SSA principle that is able to verify programs in this custom language. However, we have designed the language to be as simmilar to C as possible. For the details of the grammar, you can take a look at the code in `simplelang.h`. Instead of giving a dry list of grammar rules, we instead present a complex example in Figure 2 that showcases what the language capabilities. Here is a short list of features:

- Defining variables: `int low = 0;`
- Assigning variables: `high = mid;`
- Reading input from stdin: `int num = input();`
- Printing values to stdout: `print(sqrt);`
- Code blocks are statements: `{ high = mid; }`
- Repeating statemets: `repeat (10)i = i + 1;`
- If-then-else statements: `if (cond) i = i + 1; else { low = mid; }`
- Assumptions: `assume(num >= 0);`
- Assertions: `assert(no_bug_pls);`

Here is a short list of language level constraints:

- There is only one the signed 32-bit integer type: `int`
- There is no operator precedence, sub-expressions must be in parentheses: `num > (mid * mid)` is ok while `num > mid * mid` is not.
- Repeat loops are always finite, you can only provide an integer: `repeat (10){ i = i + 1; }` is fine while `repeat (n){ i = i + 1; }` produces a parsing error.

Here are also some of the interesting language semantics:

- The language supports scoping. If you have something like `int a = 5; { int a = a + a; }`, then you are defining a new variable `a` in the inner scope that shadows the outer one. After the block is executed, a will still have the value 5.
- `input()` is a language builtin that asks the user for input while providing the exact position in code where it is called. The same goes for `print(expr );`, which tells you the exact code position of the call, as well as the value of the provided expression `expr`.
- When an `assume(expr);` is evaluated, it checks the condition `expr` and prints a warning of violation if it is 0. However, when an assertion with `assert( expr);` evaluates `expr` to 0, it triggers a runtime error.

```c
1 int x = 0;
2 x = x + 3;
3 x = (x * 4) - 5;
```

(a) Code

```c
1 int x_1 = 0;               // x_1 == 0
2 int x_2 = x_1 + 3;         // x_2 == x_1 + 3
3 int x_3 = (x_2 * 4) - 5; // x_3 == (x_2 * 4) - 5;
```

(b) Equivalent SSA program and solver constraints

Figure 3: Example showing a program, its SSA equivalent and solver constraints

5.2 Single Static Assignment

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 (either implicitly or explicitly) whenever we want to verify a program by turning it into a formula and checking for assertion violation with a solver. In the exercise, the Symbolic SSA checker you will write interprets a program using this principle. Whenever a variable is assigned, it treats it as the definition a new variable with a different name. Any further references to the same variable are adapted to reference the new one.

We illustrate this in Figure 3. In the exercise, we provide you with a function to generate unique variable names for each of the assignments to the given variable. Here, for brevity, we show off the same concept by rewriting a program using the SSA principle, and use only the line numbers to get unique variable names in all examples.

As can be seen in Figure 3, any time a variable would be reassigned, the SSA equivalent program instead creates a new variable with a unique name. In the example, any references to the variable `x` after line 2 do not use `x_1`, but instead the new symbol `x_2`. The comments describe what an SSA-based checker does implicitly. When evaluating the statement in line 1, it first creates a new symbolic variable `x_1` that is to be used whenever the program references the variable `x`. Furthermore, the checker adds the constraint `x_1 == 0` to the solver's formula. In line 2, the SSA checker sees that the variable `x` is reassigned, so it creates a symbolic variable `x_2` and adds the constraint `x_2 == x_1 + 3`. Moreover it replaces the value of `x` with `x_2` in all further references. The same thing happens in line 3, where `x_3` is created, the constraint `x_3 == (x_2 * 4)- 5` is added to the formula, and `x` is afterwards `x_3`.

One problem you will encounter when applying the SSA transformation are `if-then-else` statements. In general, these can have an expression as the branching condition, and you cannot know ahead of time which branch is executed in a real execution. Therefore, SSA performs both branches and then merges the changes to the program state afterwards using the symbolic branching condition. Figure 4 shows an example of how this works.

Here, everything works as usual, until the program reaches an `if`. In line 4, the SSA checker creates a new variable for the condition called `c_4` and asserts that its value is the value of the branching condition. Afterwards, both the *then* and *else* blocks/statements are evaluated. Finally, in line 8, the two possible states of the program are merged. We have represented this in C code

```
1  int x = 7;
2  int y = 8;
3  int z = y;
4  if (x < 7){
5      y = x;
6  } else {
7      x = -5;
8  }
```

(a) Code

```
1  int x_1 = input();          // x_1 == input_1
2  int y_2 = input();          // y_2 == input_2
3  int z_3 = y_2;              // z_3 == y_2
4  int c_4 = x_1 < 7;          // c_4 == (x_1 < 7)
5  int y_5 = x_1;              // y_5 == x_1
6  //                          //
7  int x_7 = -5;               // x_7 == -5
8  int x_8 = c_3 ? x_1 : x_7;  // x_8 == ite(c_4, x_1, x_7)
9  int y_8 = c_3 ? y_5 : y_2;  // y_8 == ite(c_4, y_5, y_2)
```

(b) Equivalent SSA program and solver constraints

Figure 4: Example showing a program, its SSA equivalent and solver constraints

```
int x = 5;
repeat (3) {
    x = x * x;
}
```

(a) Code

```
int x_1 = 5;                 // x_1 == 5
//
int x_3_0 = x_1 * x_1;       // x_3_0 == x_1 * x_1
int x_3_1 = x_3_0 * x_3_0;   // x_3_1 == x_3_0 * x_3_0
int x_3_2 = x_3_1 * x_3_1;   // x_3_2 == x_3_1 * x_3_1
```

(b) Equivalent SSA program and solver constraints

Figure 5: Example showing a program, its SSA equivalent and solver constraints

in Figure 4b (the simplelang language has no ternary operators for simplicity). The conditions added to the solver are also shown, where we use the `z3::ite` construct that represents C-like ternary operator (we ignore type-casting here). If the *then* branch would have been executed, the variable `x` would have the symbolic value `x_1` in line 8. Otherwise if the *else* branch was taken it would have the value `x_7`. The same principle is used to merge the values of variable `y`. Since `z` was not modified in either branch, it does not need to be merged in line 8 of Figure 4a.

Repeat statements behave as if the code statement/block they reference is just copy-pasted the appropriate number of times. In order to give each variable assignment a unique name, the names are appended with the current loop iteration in which they are set. This also works recursively with multiple nested `repeat` statements. Figure 5 illustrates this.

Assumptions and assertions are very special language features. With `assume`, the programmer puts forward assumptions about the program's state that must hold in order for the guarantees defined with `assert` to hold. Unlike other statements, assumptions and assertions depend on the path the program took to reach them. In other words, since both the *then* and *else* blocks are always executed, assumptions and assertions are ammended to only trigger on the program paths that reach them. When we want to verify that a program is correct, we want to prove that there are no program executions where all assumptions are fulfilled, but at least one assertion is not. Figure 6 shows an example of checking a program that has assumptions and assertions. Whenever an assumption is

```
int x = input();    // x_1 == input_1
int y = input();    // y_2 == input_2
assume(y != 0);     // c_3 == implies(1, y_2 != 0)
if (x > y) {        // c_4 == x_1 > y_2
    x = x - y;      // x_5 == x_1 - y_2
   assert(x > 0);   // c_6 == implies(c_4, x_5 > 0)
}                   // x_7 == ite(c_4, x_5, x_1)
                    // For assumes: c_3
                    // For asserts: !c_6
```

Figure 6: Example showing a program with assumptions and asserts

evaluated, a condition variable is created that contains an implication between the conjunction of the current path conditions and the expression provided to `assume`. The same thing also happens for assertions. Finally after the whole program is evaluated, a logical conjunction of all assumption conditions is added to the formula, as well as a negated logical conjunction of all assertions. If we ask a solver to check the formula for satisfiability, it will only re The `SymbolicInterpreter`

6 Implementation

In the upstream repository, there is already a concrete interpreter, a parse-tree visualisation tool and a hollow symbolic interpreter. Your task is to complete the symbolic interpreter with all the SSA and verification capabilities mentioned above. Again, take note of the code labeled as `@todo` and try to keep your changes confined to that part of code. Also, pay attention to the exact requirements for variable names and program output, as well as the types of expressions, because a wrong naming or gratuitous output can result in the test system rejecting your implementation, and typing errors can lead to unexpected runtime exceptions being thrown. For hints on how to implement certain functionality, please also consult the comments labeled with `@details` in the codebase.
Regarding the concrete interpreter, you can run it with:

```
"./build/simplelang-concrete" my-program.cpp
```

Similarly, if at any point you are unsure of a program's parse tree structure given to the `SymbolicInterpreter`, there is an utility to view and debug them:

```
"./build/simplelang-dot" my-program.cpp my-program.dot
xdot my-program.dot
```