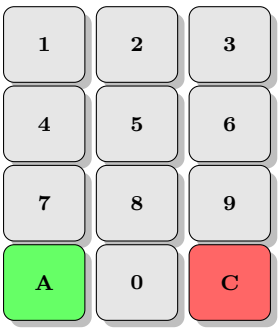
Model Checking WS 2022: Project

Institute for Symbolic Artificial Intelligence

Guidelines

- This is a team project. Each team must consist of **3 students** per team. Students may request to write their project alone, but approval by the lecturers is required. Please register your team before **November 8, 2022 at 23:59** in Moodle (`https://moodle.jku.at/jku/mod/grouptool/view.php?id=7571666`).
- Once all teams have been assigned, your team will receive an instance of the problem. The instance each team must solve is different, and techniques that may work for one team might be useless for another.
- A report of the solution for the problem, including all relevant code and sufficient explanation, must be submitted by **January 8, 2023 at 23:59** through Moodle (`https://moodle.jku.at/jku/mod/assign/view.php?id=7571673`). Please submit a single archive file (e.g. `.zip` or `.tar.gz`) with *all* the relevant files, including a PDF containing the report and all source files and code you wrote to solve the problem.
- Additionally, the solution and the process to find it must be presented by all team members in a short interviews on the week of **January 16–22, 2023**. Presentation slots will be published in Moodle (`https://moodle.jku.at/jku/mod/choice/view.php?id=7571674`) in December. The deadline to register for the presentation is the same as the report submission deadline.
- This project contributes up to **8 points** to the final UE course grade (of a total 32 points). Both the report and the presentation will be taken into account for the grade.
- A tutoring session on this project will be held on **December 15, 2022** during normal lecture hours. This is intended to provide some guidance with already started projects; do not wait until then to start working!



1 Problem Description

Your department has been commissioned to implement a keypad lock controlling the door to a restricted area for a client. The client has provided a specification that must be fulfilled by the implementation, including detailed instructions on how the keypad is operated. Your department has developed an implementation in C, which passed all unit tests. However, this is application is critical, and management wants to be sure that the implementation adheres strictly to the specification. You have been assigned by the project manager the task of verifying specification conformance for this implementation.

Intuitively, the keypad behaves in a similar way to an ATM. A four-digit PIN must be introduced, and there are 3 attempts to do so; if all attempts are exhausted, the lock becomes irreversibly blocked. If the correct PIN is introduced, the door becomes unlocked. Pressing `A` while unlocked opens the door, and then pressing afterwards `C` closes the door. Furthermore, if the door has been unlocked, the PIN can be changed by introducing another four-digit sequence, leaving the door locked afterwards. Any PIN input can be canceled prematurely by pressing `C`.

The specification of the keypad lock reads as follows:

- At any time, the keypad stores a 4-digit *stored PIN*, and the door is either **locked**, **unlocked**, **open**, or **blocked**. Furthermore, in the **locked** state, the keypad keeps track of a *counter* tallying the number of failed attempts (either 0, 1 or 2).
- The only inputs to the keypad are the twelve buttons in the keypad. This includes ten *digit keys*, an *accept key* (`A`) and a *cancel key* (`C`). The inputs are sequential, instant events, so simultaneous keypresses are not possible.
- While the keypad is **locked**, pressing the accept key has no effect.
- While the keypad is **unlocked**, pressing the accept key leaves the door **open**.
- While the keypad is **open**, pressing the cancel key leaves the door **locked** with the counter set to 0; any other input is ignored.
- While the keypad is **blocked**, no input has an effect.

- When the keypad is **locked**, any sequence of accept keys and at most 3 digit keys finished with a cancel key has no effect.
- When the keypad is **locked**, any sequence of accept keys and exactly 4 digit keys where the last key is a digit key is a *PIN attempt*. The *input PIN* is its subsequence of 4 digits.
  - If the input PIN equals the stored PIN, the door becomes **unlocked**.
  - If the input PIN differs from the stored PIN and the counter is either 0 or 1, the door remains **locked** and the counter is increased by one.
  - If the input PIN differs from the stored PIN and the counter is 2, the door becomes **blocked**.
- When the keypad is **unlocked**, any sequence of at most 3 digit keys finished with a cancel key has no effect.
- When the keypad is **unlocked**, any sequence of exactly 4 digit keys is a *new PIN*. In this case, the stored PIN is replaced by the new PIN, and the door becomes **locked** with the counter set to 0.

2 Project Assignment

You are given an **incorrect** implementation of the keypad at the start of the project. The goal of your team is to find these bugs using model checkers and SMT solvers. In particular, a complete solution for the project includes:

- An explanation of how you have used model checkers and SMT solvers to find the bug.
- An input (i.e. an initial stored PIN and an input string representing keypresses) that results in the door being open when it should have been closed (according to the specification) or vice versa, together with an explanation of why this behavior violates the specification.
- A proposed change in the code to fix the bug.

In order to identify the bug, you can use any model checking or theorem prover you find suitable. In particular, you **do not** need to implement your own model checker. We recommend the following tools, and we guarantee that the project can be solved using them:

- CBMC (`https://www.cprover.org/cbmc/`) is a bounded model checker for C programs. You can find its documentation in `https://www.cprover.org/cprover-manual/`.
- Z3 (`https://github.com/Z3Prover/z3/releases`) is an SMT solver. You can find a tutorial in `http://theory.stanford.edu/~nikolaj/programmingz3.html`.

Given how difficult finding an error in an implementation can sometimes be, we understand that some groups might just be unlucky and not find the bug in their assignment. We will not penalize groups who do not find a bug; it is still possible to earn full points while not having solved the problem. However, in this case, we expect the report to explain in full detail the following issues:

- What methods have been used to try to detect the bug, with the implementations and commands given to the solver or model checker.
- An assessment of what incorrect behaviors the group can be fully confident to have ruled out.

We do not recommend trying to debug the program "manually". The purpose of this project is to use real model checking software to find inconsistencies between the implementation and the specification. Hence, some of the bugs are purposefully hard to find by humans, but simple to detect using model checkers.

Each group received a different buggy implementation; in fact, even the base implementations are generally different. Errors are located in different locations and involve different incorrect behaviors for each group. Learning about the techniques that other groups are using might give you ideas about how to find the bugs in your assignment, but the bugs other groups find will give you no information about the bug in your implementation.

We guarantee that the bugs you must find match one of the following descriptions:

- Pressing `C` does not close the door if it is open.
- A locked door can be unlocked without introducing the correct PIN.
- The stored PIN cannot be changed into any arbitrary PIN.
- The door is not closed and locked after changing the stored PIN.
- The door can be opened after introducing three incorrect PINs.
- The door is blocked before introducing three incorrect PINs.
- Unlocking the door does not reset the attempt counter.
- The attempt counter is reset by any action other than unlocking the door.

Note, however, that the bug might only trigger after some actions have been performed first, or only if the stored PIN satisfies some specific properties. We also guarantee that at least one incorrect behavior occurs with at most 30 keypresses.

3 Setup and Program Interface

You will be provided with a C source file that implements the keypad. The source file can be compiled with all common compilers, and only the standard C library is needed. E.g using the GNU C Compiler (GCC) you can compile the implementation with

```
$  gcc -o keypad.run keypad.c
```

where *keypad.c* is the name of the source file; this produces the binary `keypad.run`. If you are using Linux, remember that you must give execution permissions to `keypad.run`.

The implementation can now be used as a console application with two arguments. The first argument is the stored PIN, and the second argument is sequence of input keypresses; this is expressed as a string where each character is a digit (`0...9`) or the letters `A` or `C`. The keypad is initialized with the stored PIN given by the first argument, in locked state with zero attempts.

```
$ ./keypad.run 1234 1234A
door is open
```

The keypad is initialized with the stored PIN 1234. Then the sequence `1234` unlocks the door; pressing `A` then opens the door.

```
$ ./keypad.run 1234 1234
door is closed
```

The keypad is initialized with the stored PIN 1234. The sequence `1234` unlocks the door, but does *not* open it. Hence, the door is closed.

```
$ ./keypad.run 1234 3451A3321A1234A
door is open
```

The keypad is initialized with the stored PIN 1234. Introducing the PIN `3451` does not unlock the door, so pressing `A` afterwards does not open it. The same holds for the PIN `3321` introduced afterwards. Finally, introducing `1234` unlocks the door, and pressing `A` opens the door.

```
$ ./keypad.run 1234 345133215555A1234A
door is closed
```

The keypad is initialized with the stored PIN 1234. Then three incorrect PINs, namely `3451`, `3321`, `5555` are introduced. After doing this, the door is blocked. Hence, the door remains closed no matter what keys are pressed afterwards.

4 Final remarks

The keypad implementation contains a `keypad` data structure, some functions and a `main` function. The bug *is not* located in the `main` function or the `keypad` data structure.

Both CBMC and Z3 are tools that, under some conditions, will take very long to finish and use quite a bit of memory. In our tests, we were able to find solutions for each implementation in under 5 minutes, but it might take longer with different solutions or hardware. Try to keep your solutions as computationally simple as possible to help the model checker.

If you have any questions about using CBMC/Z3, or about the project itself, please reach to us through Moodle so that everyone can benefit from the answers!