Model Checking Project Report – 2023/01/08

Merel and Jack

1. Report

**Literature** used: The SMT-LIB Standard Version 2.5 (2015), Barret, Fontaine, Tinelli

Both teammembers were new to SMT (and CBMC) and took a while to re-do Assignment 1.2, the simple SMT verification, as preparation for this project. Still, it would not have been possible for us to arrive at a solution from scratch, so **we needed to draw from the tutorial session SMT code substantially to be able to make a hand-in** at all: this is commented in the code as well, as we absolutely do not intend to plagiarize without due attribution.

That said, with this basis, we did rework the code and restructured a bit, so as to best match our own understanding of how these model checking tools are used.

Our first step was to draw the state transitions and check manually (looking at the code) if the state numbering is coherent in buggy05.c. This involved figuring out how state is encoded, in our case:

// new\_keypad(pin) returns a keypad data structure in the Locked state with 0 attempts and stored PIN `pin`

keypad new\_keypad(uint32\_t pin) {

// creates a new keypad with the input PIN

    keypad kp = {pin, {0, 0, 0, 0, 0, 1}};

    return kp;

}

We were able to deduce from the code that the state is encoded in the last array item. The first array item holds the index of the current digit being read.

**Diagram

Description automatically generated**The numbering in the drawing is the state according to this array and matches the program specification as far as we can see without model checking.

Our **assumptions**: Character-reading (Character to PIN conversion) and ASCII-arithmetic work as expected. (We do not encode the specifics in SMT).

Our **basic approach**: we know from the project description that our code contains one of eight bugs (b). So we adapt the SMT to follow the implementation (I) and encode the bugs one at a time. This leads to eight SMT files.

Graphical user interface, text, application, email

Description automatically generated

Together with the specification, the implementation and the bug must hold true. Our **expectation** is therefore one sat execution and seven unsats.

After some discussion, we are thinking of the bug as a specification encoding, deciding to encode the positive of the bug as part of the specification points.

Some more tweaks to our SMT: the comment about 30 keypresses might imply 30 individual keys including digits, but we interpret a keypress as either A, C, a full (4 digit), a partial (less than 4) entry, to be safe (the number of actual presses will be larger, most likely).

Difficulties: Understanding this part

(assert (and

    (<= start failure)

    (<= failure end)

    (not (= (is\_open failure) (impl\_is\_open failure)))))

We realized we need this part because it encode the fundamental flaw of the program (f):

Graphical user interface, text, application, Word

Description automatically generated

In other words, this is our basic **countermodel** (failure (state!) is open/closed according to specification, but implementation is closed/open, i.e. the other one).

To keep track of smt correspondences in the code, here is a mapping of Implementation and Specification smt function mappings to C code. To be completely transparent, we note where we edited to the smt relative to the tutorial reference.

**Implementation:**

Impl\_partial\_pin -> cancel\_key() \***not** edited, i.e. same for buggy.c and buggy05.c

Impl\_correct\_pin -> digit\_key(), specifically

if (index == 4) {

if(kp->state[5] == 0) {

kp->pin = input;

**kp->state[5] = 1; 🡨**

} else if (kp->pin == input) {

kp->state[5] = 0;

} else {

kp->state[5] += 1;

}

kp->state[0] = 0;

}

Inside digit\_key() **\*edited**, the first two branches are encoded

Impl\_wrong\_pin -> digit\_key():

if(kp->state[5] == 0) {

kp->pin = input;

kp->state[5] = 1;

} else if (kp->pin == input) {

kp->state[5] = 0;

} else {

kp->state[5] += 1;

}

**\*edited:** the last branch triggers, but it is the same edit as above (first branch)

Impl\_accept 🡪accept\_key() \*not edited

Impl\_skip 🡪 digit\_key(), in the case if (kp->state <= 4) and not if (kp->nread == 4), so nothing happens.

**Specification** (mapping, SMT -> spec. point): Again, this part is largely taken from the tutorial smt reference code! We want to be completely transparent about this.

(define-fun partial\_pin\_locked\_or\_unlocked ((i Int)) Bool (=>

    (and

        ((\_ is partialpin) (keypresses i))

        (or ((\_ is locked) (keypadstate i)) ((\_ is unlocked) (keypadstate i))))

    (= (keypadstate (+ i 1)) (keypadstate i))))





(define-fun correct\_pin\_locked ((i Int)) Bool (=>

    (and

        ((\_ is correctpin) (keypresses i))

        ((\_ is locked) (keypadstate i)))

    (= (keypadstate (+ i 1)) unlocked)))

Text

Description automatically generated

(define-fun wrong\_pin\_locked ((i Int)) Bool (=>

    (and

        ((\_ is wrongpin) (keypresses i))

        ((\_ is locked) (keypadstate i)))

    (and

        (=> (< (attempts (keypadstate i)) 2)

            (= (keypadstate (+ i 1)) (locked (+ (attempts (keypadstate i)) 1))))

        (=> (>= (attempts (keypadstate i)) 2)

            (= (keypadstate (+ i 1)) blocked)))))

Table

Description automatically generated

(define-fun complete\_pin\_unlocked ((i Int)) Bool (=>

    (and

        (or ((\_ is correctpin) (keypresses i)) ((\_ is wrongpin) (keypresses i)))

        ((\_ is unlocked) (keypadstate i)))

    (= (keypadstate (+ i 1)) (locked 0))))

Text

Description automatically generated

Note: a new pin is either the old pin or another pin (incorrect relative to the previously correct pin.

(define-fun open\_unlocked ((i Int)) Bool (=>

    (and

        ((\_ is accept) (keypresses i))

        ((\_ is unlocked) (keypadstate i)))

    (= (keypadstate (+ i 1)) open)))



(define-fun keypress\_open ((i Int)) Bool (=>

    ((\_ is open) (keypadstate i))

    (and

        (=> ((\_ is partialpin) (keypresses i))

            (= (keypadstate (+ i 1)) (locked 0)))

        (=> (not ((\_ is partialpin) (keypresses i)))

            (= (keypadstate (+ i 1)) (keypadstate i))))))



(define-fun keypress\_blocked ((i Int)) Bool (=>

    ((\_ is blocked) (keypadstate i))

    (= (keypadstate (+ i 1)) blocked)))



**This part ist edited -** Reference Code:

(define-fun ignore\_accept ((i Int)) Bool (=>

    (and

        ((\_ is accept) (keypresses i))

        (not ((\_ is open) (keypadstate i))))

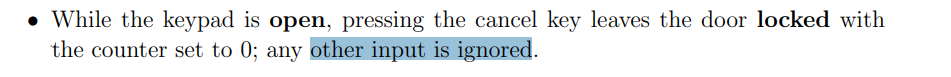
    (= (keypadstate (+ i 1)) (keypadstate i))))

I.e. accept is ignored unless in state open. But we saw: while for blocked and locked this is true, in unlocked, we also have an effect of accept. See the third specification point that follows:









The code becomes:

(define-fun ignore\_accept ((i Int)) Bool (=>

    (and

        ((\_ is accept) (keypresses i))

        (not ((\_ is unlocked) (keypadstate i)))) ; edited: now unlocked, was open

    (= (keypadstate (+ i 1)) (keypadstate i))))

Finally, for the specification:

(define-fun ignore\_skip ((i Int)) Bool (=>

    ((\_ is skip) (keypresses i))

    (= (keypadstate (+ i 1)) (keypadstate i))))

We were a bit unsure, actually, which part is implemented, but left it untouched.

1. Results

3)

Potential improvements: encoding attempts as states, rewriting specification and implementation with regards to attempts as well. Attempts are currently checked with valid\_state (function), but maybe attempts-manipulation (transition) needs to be covered in the encoding as well, not just the snapshot of the validity.