Verifying a 3-bit Combination Lock

You are given a device and told it is a combination lock. It looks like this:

![3-bit Combination Lock Diagram]

**Inputs:**
- A RESET button. Whenever RESET is pushed, the device always goes to a predetermined state.
- A single toggle switch labeled C (COMBINATION) for entering the bits of the combination.
- A pushbutton E (ENTER) telling the device when it to to read the next combination bit.
- A pushbutton TRY to be pushed when trying to open the lock.

**Outputs:** An OPEN signal that releases the physical lock. Think of it as a light that is on when the lock is open.

**Specification:** Here are the steps one must take to open the lock. Assume that the combination bits are: $c_1, c_2, c_3$.

1. Push RESET
2a. Set C to $c_1$
2b. Push Enter
3a. Set C to $c_2$
3b. Push Enter
4a. Set C to $c_3$
4b. Push Enter
5. Push Try

- You may wait an arbitrarily long time between any two of these steps.
- The OPEN signal remains on until you hit RESET again.

**Other:** The device contains no more than six bits of memory.

**Demonstration by simulation**

Suppose you have a simulation model with the inputs and outputs of the lock. Consider the problem of demonstrating that the design it models is a correct 3-bit combination lock with combination 010. It is reasonable to assume that
the RESET button works properly and that the device is a proper sequential system (or finite state machine).

QUESTIONS:

1. What properties must be demonstrated to establish that the device works correctly?

2. In the worst case, how many such steps will it take to verify all the necessary properties, assuming it contains exactly six bits of memory?

3. Are there features of the design behavior that you can exploit to improve on the worst case?

More specifically, assume your simulation procedure is a sequence of 3-part steps in which

- (a) you present the inputs;
- (b) the system takes a step; and
- (c) you observe the outputs, perhaps comparing them to an expected list of values or the outputs of a “specification model.”

<table>
<thead>
<tr>
<th>Question</th>
<th>How many such steps would it take to demonstrate the correctness of the lock?</th>
</tr>
</thead>
</table>

Think carefully about your answer to this question before we discuss it in the next class.

Two Simulation Models

The files mentioned below are located in directory

```
/l/www/classes/p415/post/
```

Two Scheme simulation models are provided in files `lock1.so` and `lock2.so`. At least one of these models is a faulty design. Each model is represented as a compiled scheme function, named `lock1` and `lock2`, respectively, expecting four arguments, numbers 1 or 0, for inputs RESET, COMBINATION, ENTER, and TRY. The function returns the value of the OPEN light.

The simple simulator shown in Figure 1 is located in file `lock.ss` takes a model model and a script `(((r t e c) o) ...)`. Each element of script contains a 4-tuple of inputs and the expected value of output OPEN. The simulator applies model to the inputs, in turn, and displays the outcome.

For example, with the simulation sequence `fig:script1`, shown in Figure 2, is used to simulate model `lock2` in Figure 3 here is a transcript showing the outcome for model `lock1`: This simulation shows that model `lock2` correctly opens if the combination is entered according to one instance of the specified protocol.
(define simulate (lambda (model script)
  (if (null? script)
     '(done)
     (let*
        ((ins (caar script)) ; Next simulation step
         (vr (car ins)) ; RESET
         (vt (cadr ins)) ; TRY
         (ve (caddr ins)) ; ENTER
         (vc (cadddr ins)) ; COMBINATION BIT
         (outs (cadar script)) ; Expected output value
         (sim (model vx vt vr vc))) ; The model's output
       (cons
        (list vx vt vr vc '--> outs '=?= sim)
        (simulate model (cdr script))))))

Figure 1: A basic simulator for lock models

(define script1
  ; (x t r c) o) ; (inputs expected-output)
  ; """"
  ; '( ((1 0 0 0) 0) ; reset
    ((0 0 0 0) 0) ; set the first bit to 0
    ((0 0 1 0) 0) ; assert "read"
    ((0 0 0 1) 0) ; set the second bit to 1
    ((0 0 1 1) 0) ; assert "read"
    ((0 0 0 1) 0) ; set the third bit to 0
    ((0 0 1 1) 0) ; assert "read"
    ((0 1 0 0) 0) ; try to open the lock
    ((0 0 0 0) 1) ; it should be open
    ((0 0 0 1) 1) ; and remain open
    ((0 0 1 0) 1) ; and remain open
    ((0 0 1 1) 1) ; and remain open
    ((1 0 0 0) 0) ) ; until we reset

Figure 2: An example simulation script script1, defined in source file lock.ss
%scheme lock.ss
Chez Scheme Version 6.7
Copyright (c) 1985-2001 Cadence Research Systems

> (load "lock2.so")
> (simulate lock2 script1)
((1 0 0 0 --> 0 =?= 0)
 (0 0 0 0 --> 0 =?= 0)
 (0 0 1 0 --> 0 =?= 0)
 (0 0 0 1 --> 0 =?= 0)
 (0 0 1 1 --> 0 =?= 0)
 (0 0 0 1 --> 0 =?= 0)
 (0 0 1 1 --> 0 =?= 0)
 (0 1 0 0 --> 0 =?= 0)
 (0 0 0 0 --> 1 =?= 1)
 (0 0 0 1 --> 1 =?= 1)
 (0 0 1 0 --> 1 =?= 1)
 (0 0 1 1 --> 1 =?= 1)
 (1 0 0 0 --> 0 =?= 0)

done)
>

Figure 3: Trace script showing simulation of model lock2 using simulation script script1