Formal Modeling (SS 2020)
Assignment 1 (May 20)

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at

The result is to be submitted by the deadline stated above ia the Moodle interface of the course as a single archive file in `.zip` or `.tgz` format which contains the following files:

1. A single PDF file with the following contents:
   - a cover page identifying the course, the assignment, and the submitter;
   - a section for each part of the assignment, which contains
   - the deliverables requested for this section with a snapshot of the listing of the corresponding RISCAL file that contains all additions/changes to the skeleton file handed out (nicely formatted and typeset in a fixed-width font of readable size with no line overflows), and
   - optionally any explanations or comments you would like to make.
2. All RISCAL files developed in the assignment.

All assignments only ask to complete the definitions of predicates by formulas in first order logic (operations $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\forall$, $\exists$). You may also use if-then-else and let-in expressions to make the formulas more readable.

Hint: you may annotate arbitrary formulas and terms by the `print` expression (see the RISCAL manual section B.5.15) to understand the derived results. For instance:

```
// result is p(e), prints first e and then p(e) in separate lines
print p(print e)

// result is f(x,y), prints x and y in one line, then f(x,y) in another
print "x:{1}, y:{2}", x, y in print f(x,y)
```

Assignment 1a (30 Points): Array Replacement

We consider the problem of replacing elements in an array. The attached RISCAL file `Replace.txt` gives an algorithm `replace` that returns a duplicate *b* of array *a* where every occurrence of an element in *from* has been replaced by the corresponding element in *to* (*from* must not have duplicate elements).

1. Complete the definition of the predicates `input` and `output` that define the precondition and the postcondition of the algorithm, respectively.
2. Define in the pop-up window "Other Values" suitable values for the model parameters N and M (e.g., $N = 4$, $M = 2$, $R = 3$). Press in the "Operation" panel the button "Show/Hide Tasks" to open the "Tasks" menu.
3. Validate your specification by running (with option "Nondeterminism" switched *on* and option "Silent" switched *off*) the task "Execute specification". Analyze the printed output to investigate which input/output pairs are allowed by your definition. Are those (and only those) pairs printed that you expect?
4. Further validate your specification by running (with option "Nondeterminism" switched *off* and option "Silent" switched *on*) the tasks "Is precondition satisfiable?", "Is precondition not trivial?", "Is postcondition always satisfiable?", "Is postcondition always not trivial?", "Is postcondition sometimes not trivial?", "Is result uniquely determined?". Are the results as you have expected?
5. Run task "Execute Operation" to check whether the algorithm indeed satisfies your specification (respectively, whether your specification matches the algorithm; the algorithm most likely is correct).

Demonstrate by (a reasonable selection of) the RISCAL output that you indeed have performed above tasks. Interpret the results and judge whether your specification is adequate.

Assignment 1b (35 Points): Ultimate Tic-Tac-Toe

We consider the game "Ultimate-Tic-Tac Toe"[1]. The RISCAL file `UltimateTicTacToe.txt` contains a procedure `play` (together with accompanying auxiliary definitions) that plays (depending on the execution option "Non-determinism") some/all possible instances of this game.

1. Complete the definition of the predicate `wins` that determines whether a particular player has won the game.
2. Complete the definition of the predicate `legal` that determines whether a particular move (a choice of a local board and a position in that board) is legal.

Validate your specification by running (with option "Nondeterminism" switched *on* and option "Silent" switched *on*) all possible games; the procedure prints out all games which was won by some players (the games resulting in a draw are not printed).

[1] https://en.wikipedia.org/wiki/Ultimate_tic-tac-toe

The procedure may have to execute some 10^5 non-deterministic execution branches (which may take a minute or so) to find a game that was won by some player. Once some such games have been found, you may interrupt the execution (press the "Stop Execution" button) and investigate some game(s) to determine whether it was indeed correctly played and the winner was correctly determined.

Explain in your submission one played game in detail and why it was correctly played.

Assignment 1c (35 Points): Elevator Control

Consider a building with F floors that are served by E elevators. On each floor, there are two "call" buttons "up" and "down" that the user may press to announce the intention of going up or down (for simplicity, we also let the bottom and the top floor have two buttons). Inside each elevator there are F "floor" buttons that the user may press to request to go to that particular floor (of course, this request need not match the originally announced intention). For serving floors, elevators apply the "elevator algorithm"[2] which is also applied in disk scheduling[3]: the basic idea is that an elevator continues to travel in its current direction (up or down) while there are remaining requests in that same direction; if there are no further requests in that direction, then it stops and becomes idle, or changes direction if there are requests in the opposite direction.

The RISCAL file `Elevator.txt` contains the skeleton for the simulation an elevator system whose state s consists of the following components:

- $s.call(d)$ denotes, for direction $d \in \{up, down\}$, the set of floors in which a call button for direction d has been pressed.
- $s.button(e)$ denotes the set of floor buttons that have been pressed in elevator e.
- $s.door(e)$ indicates whether the door of elevator e is open (true) or not (false).
- $s.floor(e)$ denotes the floor in which the elevator currently is.
- $s.direction(e)$ denotes the direction d for which the elevator is currently serving requests ($d = none$ indicates that the elevator is currently not serving any requests).

The goal is to define the core of an elevator controller by a predicate $admissible(a, s)$ (apart from this definition, you need not change anything in the RISCAL file) that states whether in a state s of the system an action a of the following kind is admissible:

- $Call(f, d)$: on floor f the not yet pressed (i.e., unlit) button for direction $d \in \{up, down\}$ is pressed (and thus becomes lit).
- $Button(e, f)$: in elevator e the not yet pressed (i.e, unlit) button for floor f is pressed (and thus becomes lit).
- $Door(e, door)$: the door of elevator e opens ($door$=true) or closes ($door$=false). If the door opens, in the elevator the button for the current floor is unlit and on the current floor the button

[2] https://www.popularmechanics.com/technology/infrastructure/a20986/the-hidden-science-of-elevators/

[3] https://en.wikipedia.org/wiki/Elevator_algorithm

for the direction in which the elevator is heading is unlit; if the elevator is not heading any more in a particular direction (because all requests in that direction have been served), an arbitrary lit button may be unlit.

- $\mathit{Move}(e, d)$: elevator e moves into direction d to serve a request ($d \in \{\mathit{up}, \mathit{down}\}$) (thus changing the floor) or it stops serving requests ($d = \mathit{none}$); in any case the direction d is recorded as the "current" direction of the elevator.

Please see in the RISCAL file the definition of function $\mathit{execute}(a, s)$ that describes these actions by computing the state that results from state s by performing action a.

The execution of the system is described by a sequence of admissible actions as allowed by the predicate $\mathit{admissible}(a, s)$. For instance, for a building with $F = 3$ floors and $E = 1$ elevator, such as sequence might be

$$\begin{aligned}
&\mathit{Call}(1, \mathit{up}) \to \mathit{Move}(0, \mathit{up}) \to \mathit{Door}(0, \text{true}) \to \mathit{Button}(0, 2) \to \mathit{Door}(0, \text{false}) \\
&\to \mathit{Move}(0, \mathit{up}) \to \mathit{Call}(1, \mathit{down}) \to \mathit{Door}(0, \text{true}) \to \mathit{Door}(0, \text{false}) \to \mathit{Move}(0, \mathit{down}) \\
&\to \mathit{Door}(0, \text{true}) \to \mathit{Door}(0, \text{false}) \to \mathit{Button}(0, 0) \to \mathit{Move}(0, \mathit{down}) \\
&\to \mathit{Door}(0, \text{true}) \to \mathit{Door}(0, \text{false}) \to \mathit{Move}(0, \mathit{none}) \to \mathit{Call}(2, \mathit{down}) \to \cdots
\end{aligned}$$

Please apply above description and your common understanding of how elevators typically work to define the predicate $\mathit{admissible}(a, s)$.

As a *minimum* requirement, your predicate shall ensure:

- *safety:* an elevator never moves with an open door.

- *progress:* an elevator only moves to serve some request.

Validate your model by choosing appropriate (small) values for F, E, and N and executing (with options "Nondeterminism" and "Silent" switched *on*) the function `run()` that non-deterministically chooses and executes all action sequences of length N. For this, first uncomment the line that displays the chosen action sequence and demonstrate that *some* of the computed action sequences contain *Move* actions and generally look as "as expected". Then comment this line out again and have all execution sequences generated; this automatically checks that the sequences satisfy the safety and progress property indicated above.