EE382C-3: Verification and Validation of Software
Problem Set 1 – Modeling in Alloy*

Out: January 16, 2025; **Due: February 7, 2025 11:59pm**
Submission: *.zip via Canvas
Maximum points: 40

Instructions. Complete the Alloy models for the two questions in this problem set and submit your solutions as a tarball on Canvas. For each question, you are given a skeletal Alloy model, which you need to complete following the instructions given. You must use Alloy 4.2 (or newer), which you can download from the Alloy website: "`http://alloytools.org/`". **The code you write should only be inside the given predicate bodies as described in the comments and TODOs.**

Question 1: Doubly Linked List

Consider modeling a doubly linked list, where each list has a header node and each node has a previous node, a next node, and an integer element. The signature `DLL` declares a set of atoms, which represent the doubly linked lists. The signature `Node` declares a set of atoms, which represent the nodes in lists. The qualifier `one` declares the set to contain exactly one atom. The field `header` declares a partial function from lists to nodes. The fields `prev` and `link` introduce partial functions from nodes to nodes. The field `elem` maps each node to exactly one integer value.

The fact `Reachable` requires that all nodes are in the doubly linked list, which we write to simplify the model.

```
one sig DLL {
  header: lone Node
}

sig Node {
  prev, link: lone Node,
  elem: Int
}

// All nodes should be reachable from the header along the link.
fact Reachable {
  Node = DLL.header.*link
}
```

Complete the predicate/fact bodies in the file DLL.als as described in parts (a), (b), (c) and (d) below.

Part (a) Acyclicity

Implement the `Acyclic` fact below:

```
fact Acyclic {
  // The list has no directed cycle along link, i.e., no node is
  // reachable from itself following one or more traversals along link.
  -- TODO: Your code starts here.
}
```

*Many thanks to Kaiyuan Wang and Darko Marinov for their help in defining these models.

Part (b) Unique Element

Implement the `UniqueElem` predicate below:

```
pred UniqueElem() {
  // Unique nodes contain unique elements.
  -- TODO: Your code starts here.
}
```

Part (c) Sorted

Implement the `Sorted` predicate below:

```
pred Sorted() {
  // The list is sorted in ascending order (<=) along link.
  -- TODO: Your code starts here.
}
```

Part (d) Consistent Prev and Link

Implement the `ConsistentPrevAndLink` predicate below:

```
pred ConsistentLinkAndPrev() {
  // For any node n1 and n2, if n1.link = n2, then n2.prev = n1; and vice versa.
  -- TODO: Your code starts here.
}
```

Question 2: Finite State Machine

Consider modeling a finite state machine (FSM), where each FSM has a start state and a stop state, and each state has a set of subsequent states. The signature `FSM` declares a set of atoms, which represent the finite state machine. The signature `State` declares a set of atoms, which represent the FSM states. The qualifier `one` declares the set to contain exactly one atom. The field `start` declares a binary relation, and requires that each FSM has exactly one start state. The field `stop` declares a binary relation, and requires that each FSM has exactly one stop state. The field `transition` maps a state to a set of states.

```
one sig FSM {
  start: set State,
  stop: set State
}

sig State {
  transition: set State
}
```

Complete the predicate bodies in the file FSM.als as described in parts (a), (b) and (c) below.

Part (a) One Start State and One Stop State

Implement the `OneStartAndStop` predicate below:

```
pred OneStartAndStop {
  // FSM only has one start state.
  -- TODO: Your code starts here.

  // FSM only has one stop state.
  -- TODO: Your code starts here.
}
```

Part (b) Valid Start State and Stop State

Implement the `ValidStartAndStop` predicate below:

```
pred ValidStartAndStop() {
  // The start state is different from the stop state.
  -- TODO: Your code starts here.

  // No transition ends at the start state.
  -- TODO: Your code starts here.

  // No transition begins at the stop state.
  -- TODO: Your code starts here.
}
```

Part (c) Reachability

Implement the `Reachability` predicate below:

```
pred Reachability() {
  // All states are reachable from the start state.
  -- TODO: Your code starts here.

  // The stop state is reachable from any state.
  -- TODO: Your code starts here.
}
```