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

Out: September 12, 2018; **Due: September 23, 2018 11:59pm**
Submission: *.zip via Canvas
Maximum points: 50

Instructions. Complete the Alloy models for **3 out of the 4** questions in this problem set and submit your solutions as a tarball on Canvas. You may choose to answer all 4, in which case the best 3 will count for the homework grade. 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://alloy.mit.edu`". **The code you write should only be inside the given predicate bodies as described in the comments and TODOs.**

Question 1: Array

Consider modeling an array using two relations: one that maps array indices to elements in the array, and the other that represents the array length. The signature (`sig`) `Element` declares a set of atoms, which represent the array elements. The signature `Array` declares a set of atoms, which represent the arrays. The qualifier `one` declares the set to contain exactly one atom, i.e., any instance created by the Alloy analyzer will contain exactly one array. The field `i2e` declares a ternary relation of type `Array -> Int -> Element`, which relates array atoms, integer indices, and array elements. The field `length` declares a binary relation of type `Array -> Int`, and maps array atoms to their lengths.

Fact `Reachable` requires that all elements are in the array, which we write to simplify our model. The expression `Array.i2e[Int]` is equivalent to `Int.(Array.i2e)`, which represents the set of all array elements.

```
sig Element {}

one sig Array {
  // Maps indexes to elements of Element.
  i2e: Int -> Element,
  // Represents the length of the array.
  length: Int
}

// Assume all elements are in the array.
fact Reachable {
  Element = Array.i2e[Int]
}
```

Complete the predicate/fact bodies in the file Array.als as described in parts (a) and (b) below.

Part (a) Bound Constraints

Complete the fact `InBound` such that it constrains `i2e` and `length` as specified in the comments.

```
fact InBound() {
  // All indexes should be greater than or equal to 0 and less than the array length.
  -- TODO: Your code starts here.
```

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

```
  // Array length should be greater than or equal to 0.
  -- TODO: Your code starts here.
}
```

Part (b) No Conflict Constraints

Complete the predicate `NoConflict` to constrain `i2e` such that any index in the array maps to at most one element.

```
pred NoConflict() {
  // Each index maps to at most one element.
  -- TODO: Your code starts here.
}
```

Question 2: Balanced Binary Search Tree

Consider modeling a binary tree, where each tree has a root node and each node has a left child, a right child, and an integer element. The signature `BinaryTree` declares a set of atoms, which represent the binary trees. The signature `Node` declares a set of atoms, which represent the nodes in trees. The qualifier `one` declares the set to contain exactly one atom. The field `root` declares a binary relation from trees to nodes; the qualifier `lone` restricts the relation to be a partial function, i.e., each tree has at most one root node. The fields `left` and `right` likewise 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 binary tree, which we write to simplify the model.

```
one sig BinaryTree {
  root: lone Node
}

sig Node {
  left, right: lone Node,
  elem: Int
}

// All nodes are in the tree.
fact Reachable {
  Node = BinaryTree.root.*(left + right)
}
```

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

Part (a) Acyclicity

Implement the following `Acyclic` fact for binary tree:

```
fact Acyclic {
  all n : Node {
    // There are no directed cycles, i.e., a node is not reachable
    // from itself along one or more traversals of left or right.
    -- TODO: Your code starts here.

    // A node cannot have more than one parent.
    -- TODO: Your code starts here.

    // A node cannot have another node as both its left child and
    // right child.
    -- TODO: Your code starts here.
  }
}
```

Part (b) Sorted

Implement the following `Sorted` predicate for binary search tree:

```
pred Sorted() {
  all n: Node {
    // All elements in the n's left subtree are smaller than the n's elem.
    -- TODO: Your code starts here.

    // All elements in the n's right subtree are bigger than the n's elem.
    -- TODO: Your code starts here.
  }
}
```

Part (c) Balanced

Implement the following helper predicate `HasAtMostOneChild` and function `Depth` as well as the `Balanced` predicate:

```
pred HasAtMostOneChild(n: Node) {
  // Node n has at most one child.
  -- TODO: Your code starts here.
}

fun Depth(n: Node): one Int {
  // The number of nodes from the tree's root to n.
  -- TODO: Your code starts here.
}

pred Balanced() {
  all n1, n2: Node {
    // If n1 has at most one child and n2 has at most one child,
    // then the depths of n1 and n2 differ by at most 1.
    -- TODO: Your code starts here.
  }
}
```

Question 3: 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.
}
```

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 4: 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.
}
```