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

Out: Jan 16, 2016
Due: Feb 10, 2016 11:59pm

Instructions. Submit one Alloy model each for the questions in this problem set as a tarball on Canvas. You must use Alloy 4.2, which you can download from the Alloy website: "`http://alloy.mit.edu`".

1 Singly-linked lists [20 points; 5 points/part]

Call a singly-linked list a *loop-list* if it is empty or if it is non-empty and its last node has a pointer back to itself. Consider modeling the following Java program that implements loop-lists with *sorted* elements:

```
public class List {
    // invariant: loop-list with sorted elements (ascending order ‘<=’)

    Node header;

    static class Node {
        Node link;
        int elem;
    }

    public int count(int x) {
        // ...
    }

    public boolean contains(int x) {
        // ...
    }
}
```

The following Alloy code gives a skeletal model for such lists:

```
module list

sig List {
  header: set Node
}

sig Node {
  link: set Node,
  elem: set Int
}
```

(a) Cardinality constraints

Implement the following `fact` such that it states the cardinality constraints on each of the fields `header`, `link`, and `elem` as described in the comments:

```
fact CardinalityConstraints {
  // each list has at most one header node



  // each node has at most one link



  // each node has exactly one elem



}
```

(b) Class invariant

Implement the following predicates `Loop` and `Sorted`, and the `run` command as described in the comments:

```
pred Loop(This: List) {
  // <This> is a valid loop-list



}

pred Sorted(This: List) {
  // <This> has elements in sorted order ('<=')



}

pred RepOk(This: List) { // class invariant for List
  Loop[This]
  Sorted[This]
}

// scope: #List <= 1, #Node <= 3, ints = { -2, -1, 0, 1 }
run RepOk for ...
```

(c) Specifying the `count` method

Implement the following predicate `Count` and the `run` command as described below:

```
pred Count(This: List, x: Int, result: Int) {
  // count correctly returns the number of occurences of <x> in <This>
  // <result> reprsents the return value of count

  RepOk[This] // assume This is a valid list




}
```

```
// scope: #List <= 1, #Node <= 3, ints = { -2, -1, 0, 1 }
run Count for ...
```

(d) Specifying the `contains` method

Implement the following predicate `Contains` and the `run` command as described below:

```
abstract sig Boolean {}
one sig True, False extends Boolean {}

pred Contains(This: List, x: Int, result: Boolean) {
  // contains returns true if and only if <x> is in <This>
  // <result> represents the return value of contains

  RepOk[This] // assume This is a valid list




}

// scope: #List <= 1, #Node <= 3, ints = { -2, -1, 0, 1 }
run Contains for ...
```

2 Binary trees [20 points; 5 points/part]

Consider the following model for binary trees:

```
module binarytree

one sig BinaryTree {
  root: lone Node
}

abstract sig Node {
  left, right: lone Node
}

pred Acyclic(t: BinaryTree) {
  all n: t.root.*(left + right) {
    n !in n.^(left + right)
    no n.(left) & n.(right)
    lone n.~(left + right)
  }
}
```

(a) Connectivity

Implement the following `fact` as described in the comments:

```
fact DisconnectedNodesHaveSelfLoops {
  // the left and right fields of a node that is not in the
  // tree point to the node itself



}
```

(b) Isomorphism

With the fact `DisconnectedNodesHaveSelfLoops` included in your model, if you execute the command "`run Acyclic`" and enumerate the instances, do any two of these instances represent *isomorphic* binary trees[1] as solutions to the constraint `Acyclic`? If such instances appear as solutions, write two distinct instances using Alloy Analyzer's textual format (i.e., `Txt` in the GUI) as comments in your model:

```
/*
Isomorphic instances for Question 2 (b):

Instance #1:
...

Instance #2:
...

*/
```

(c) Linear order

Add the following declaration to your model to introduce four nodes, namely `N0`, `N1`, `N2`, and `N3`, in the model:

```
one sig N0, N1, N2, N3 extends Node {}
```

Implement the following fact `LinearOrder` to define a linear ordering on the 4 nodes as described in the comments:

```
one sig Ordering { // model a linear order on nodes
  first: Node, // the first node in the linear order
  order: Node -> Node // for each node n, n.(Ordering.order) represents the
                      //    node (if any) immediately after n in order
}

fact LinearOrder {
  // the first node in the linear order is N0; and
  // the four nodes are ordered as [N0, N1, N2, N3]





}
```

(d) Non-isomorphic enumeration

Use the linear order defined by the signature `Ordering` and the fact `LinearOrder` for the four nodes to implement the following predicate `NonIsomorphicTrees` as described in the comments:

```
pred SymmetryBreaking(t: BinaryTree) {
  // if t has a root node, it is the first node according to the linear order; and
  // a "pre-order" traversal of the nodes in t visits them according to the linear order
```

[1] Consider only the part of the instance reachable from the binary tree atom.

```
}

pred NonIsomorphicTrees(t: BinaryTree) {
  Acyclic[t]
  SymmetryBreaking[t]
}

run NonIsomorphicTrees // enumerates non-isomorphic binary trees with up to 4 nodes
```

Verify that your implementation is correct by executing the command `run NonIsomorphicTrees` and checking that it enumerates exactly 23 binary trees – which are all the (non-isomorphic) binary trees with up to 4 nodes.