Static Program Checking - Modeling dynamic systems in Alloy
Jun.-Prof. Mana Taghdiri

Computing a spanning tree

The goal of this exercise is to model a step-by-step algorithm in Alloy. We will use a simple spanning tree computation to learn about the different ways of expressing dynamic behaviors in Alloy. Consider an undirected graph with no self loops, represented by an adjacency relation. Suppose that the graph contains a distinct node, called *Root*, such that every node is reachable from Root. To simplify your model, you can use the Alloy graph library as follows:

```
open util/graph[Node] as Graph
sig Node {
   adj: set Node
}
one sig Root extends Node {}
fact configuration {
  Graph/noSelfLoops[adj]     // for simplicity
  Graph/undirected[adj]      // adjacency is symmetric
  Node in Root.*adj          // all nodes are reachable from root
}
```

The spanning tree algorithm is as follows: At the beginning, only the root belongs to the tree. At each subsequent step, any node that has a neighbor that is already in the tree will be included in the tree, and its parent will be set to that neighbor node. Once a node is part of the tree, it does not change its parent any more. The algorithm terminates when no more changes are possible. At this point, all nodes belong to the tree, and the parent relation gives the exact edges of the tree.

Model this algorithm in Alloy step by step. Write an empty *show* predicate and run it to see some instances of the algorithm execution.

Hint: to distinguish between different steps of the algorithm, you can define a *Time* signature, and use the ordering library to define a total order over Time. All dynamic attributes of your model can now be defined as relations with a Time column.

```
open util/ordering[Time] as T
sig Time {}
T/first, T/last, T/next[t], T/prev[t]  //For accessing the time order
```

Optional. Once you are sure that you have modeled the algorithm correctly, try to think of at least one other way to model the same algorithm in Alloy.

Solution 1 – Local State (Time)

```
open util/ordering[Time] as T
open util/graph[Node] as Graph

sig Time {}

abstract sig Status {}
one sig inTree extends Status {}
one sig Waiting extends Status {}

sig Node {
  adj: set Node,
  parent: Time -> lone Node,
  status: Time -> one Status
}
one sig Root extends Node {}

fact configuration {
  Graph/noSelfLoops[adj]     // for simplicity
  Graph/undirected[adj]      // adjacency is symmetric
  Node in Root.*adj          // everything reachable from root
}

fact initial {
  all n: Node-Root | n.status[T/first] = Waiting
  Root.status[T/first] = inTree
  all n: Node | no n.parent[T/first]
}

fact algorithm {
  all n: Node, t: Time - T/last | step[n, t]
}

pred step(n: Node, t: Time) {
  readyToAct[n, t] => act[n, t]
  else noChange[n, t]
}
```

```
pred readyToAct(n: Node, t: Time) {
  (n.status[t] = Waiting) &&
  (some x: n.adj | x.status[t] = inTree)
}

pred act(n: Node, t: Time) {
  let t' = T/next[t] | {
    n.status[t'] = inTree
    (some n.parent[t']) && (n.parent[t'] in potentialParents[n, t])
  }
}

pred noChange(n: Node, t: Time) {
  let t' = T/next[t] | {
    n.status[t'] = n.status[t]
    n.parent[t'] = n.parent[t]
  }
}

fun potentialParents(n: Node, t: Time): set Node {
  {x: n.adj | x.status[t] = inTree}
}

pred show() {}
run show for 5
```

Solution 2 – Global State

```
open util/ordering[State] as S
open util/graph[Node] as Graph

sig Node {
  adj: set Node,
}
one sig Root extends Node {}

sig State {
  parent: Node -> lone Node,
  inTree: set Node
}

fact configuration {
  Graph/noSelfLoops[adj]     // for simplicity
  Graph/undirected[adj]      // adjacency is symmetric
  Node in Root.*adj          // everything reachable from root
}

fact initial {
  let s0 = S/first | {
    s0.inTree = Root
    no s0.parent
  }
}

fact algorithm {
  all s: State - S/last | step[s]
}

pred step(s: State) {
  let s' = S/next[s] | {
    s'.inTree = s.inTree + s.inTree.adj
    all n: Node |
      readyToAct[n, s] => some s'.parent[n] &&
                          s'.parent[n] in {x: n.adj | x in s.inTree}
      else s'.parent[n] = s.parent[n]
  }
}
```

```
pred readyToAct(n: Node, s: State) {
  n !in s.inTree &&
  some x: n.adj | x in s.inTree
}

pred show() {}
run show for 5
```