Formal Specification with Alloy: Specification of Algorithms

Jan van Eijck

jve@cwi.nl

Master SE, 6 October 2010

Overview

- Alloy utilities
- Assignments and pre- and postconditions in Alloy
- Alloy for automated logical reasoning
- Alloy specifications of algorithms
- On your to do list:
  - Look through the example code in these slides,
  - make sure you understand what is happening.

Alloy utilities

- Operations on graphs: `util/graph.als`
- Operations on integers: `util/integer.als`
- Operations on naturals: `util/natural.als`
- Linear ordering: `util/ordering.als`

Assignments in Alloy

```
module myexamples/assignment
open util/ordering[State] as so
open util/integer as integer
sig State {
 // integer variable x
  x: Int,
 // integer variable y
  y: Int
}
pred transition[s,s': State] {
  so/lt[s,s'] and s'.x = add[s.x,s.y] and s'.y = s.y
}

run transition for 2 State, 5 int
```

Pre- and Postconditions in Alloy

Example:

$$\begin{array}{ccc} \{\, x == n^2 \,\} & x = x + 2 * n + 1 & \{\, x == (n{+}1)^2 \,\} \\ \{\, x == (n{+}1)^2 \,\} & n = n + 1 & \{\, x == n^2 \,\} \end{array}$$

Together:

$$\{\, x == n^2 \,\}\ x = x + 2 * n + 1; n = n + 1\ \{\, x == n^2 \,\}$$

State as an Ordered Domain

To represent this in Alloy, we have to represent state as an ordered domain, and view the variables x and n as functional relations on state: for every state, x and n yield integer values:

```
module myexamples/invar
open util/ordering[State] as so
open util/integer as integer

sig State {
 // integer variable x
  x: Int,
 // integer variable n
  n: Int
}
```

Incrementing and doubling:

```
fun inc [n : Int]: Int { add [n,Int[1]] }
fun double [n : Int]: Int { add [n,n] }
```

Initializing the first state by setting both x and n equal to 0:

```
pred init {
  let fs = so/first | { fs.x = Int[0] and fs.n = Int[0] }
}
```

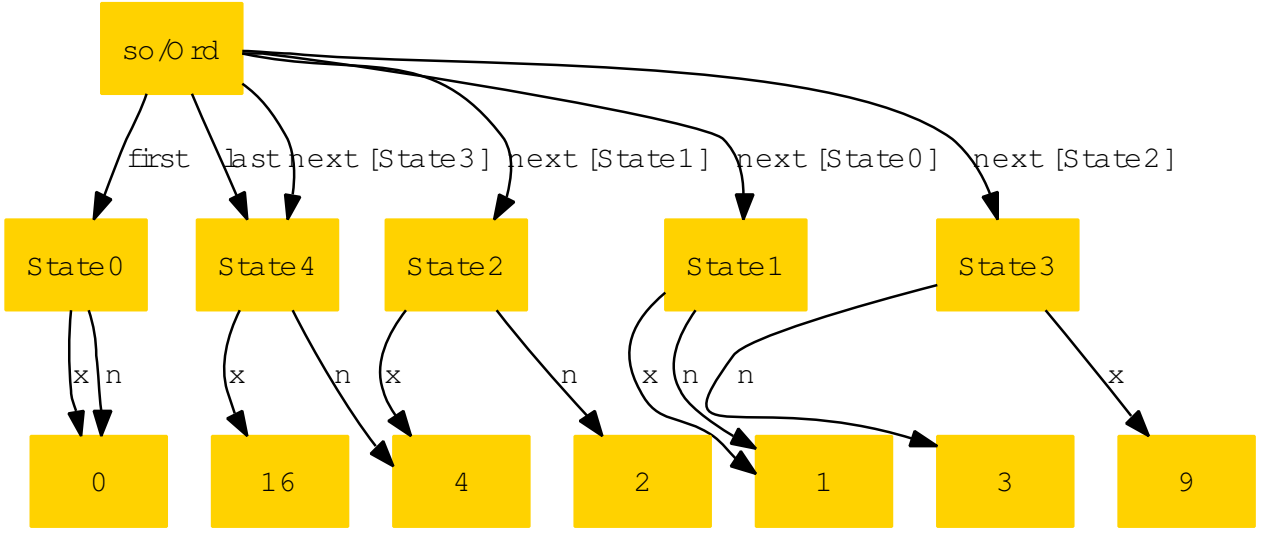
State change is the result of doing assignments to x and y.

But Alloy is a declarative system, so instead of 'doing' the assigments, we describe the relation between the pre- and post-states.

```
pred extend [pre, post: State] {
  some X,N: Int | pre.x = X
                  and pre.n = N
                  and post.x = inc[add[X,double[N]]]
                  and post.n = inc[N]
}
```

Creating the state space:

```
fact createStates {
   init
   all s: State - so/last |
          let s' = so/next[s] | extend[s,s']
}

run {} for exactly 5 State, 6 int
```


so/Ord
first
last
next [State3]
next [State1]
next [State0]
next [State2]
State0
State4
State2
State1
State3
x
n
x
n
x
n
x
n
n
x
0
16
4
2
1
3
9

How to draw logical conclusions from a list of givens

Here is a story. Someone invites six people A, B, C, D, E, F to attend a conference. The email exchanges that follow yield the following information:

1. At least one of A, B will attend.
2. From the set $\{A, E, F\}$ exactly two will attend.
3. Either both B and C will attend or neither of them will.
4. One of A and D will attend, the other will not.
5. Same for C and D.
6. If D does not attend, then neither will E.

Use an Alloy specification to figure out who will attend the conference.

Solution

```
abstract sig Person {}
one sig A,B,C,D,E,F extends Person {}
sig Congress in Person {}

fact{
  some (A + B) & Congress
  #((A+E+F) & Congress) = 2
  B in Congress iff C in Congress
  A in Congress iff not D in Congress
  C in Congress iff not D in Congress
  not D in Congress => not E in Congress
}

run {}
```

Result

A (Congress)	B (Congress)	C (Congress)	D	E	F (Congress)

Spanning Tree of a Graph

Let a graph G be given. Assume G is symmetric and connected. A spanning tree of G is a tree structure with the same node set as G, and with every parent pointer along an edge of G.

Algorithm for finding a spanning tree:

- Start with an arbitrary root node, and put it in the set of tree nodes.
- As long as there are nodes not in the set of tree nodes, proceed as follows:
  - Pick a pair of graph nodes (x, y) such that (i) x is in the set of tree nodes, (ii) y is outside the set of tree nodes, and (iii) there is an edge from x to y in the graph.
  - put the parent link $y \mapsto x$ in the tree.

Specifying the Algorithm

To arrive at an Alloy specification, we need to represent the graph, and the sequence of execution states of the algorithm. We assume a linear ordering of execution states.

```
module myexamples/st

open util/graph[Node] as graph
open util/ordering[State] as so
```

We assume an arbitrary root. The graph should be undirected (symmetric) and connected. For convenience, we also assume that there are no self-loops:

```
sig Node { e: set Node }
one sig Root extends Node {}

fact niceGraph {
       noSelfLoops[e]
       undirected[e]         // edge relation is symmetric
       stronglyConnected[e] // no lose parts
}
```

Recall the definitions

```
// graph in undirected
pred undirected [r: node->node] {
  symmetric[r]
}

// graph has no self-loops
pred noSelfLoops[r: node->node] {
  irreflexive[r]
}
// graph is weakly connected
pred weaklyConnected[r: node->node] {
  all n1, n2: node | n1 in n2.*(r + ~r)
}
```

```
// graph is strongly connected
pred stronglyConnected[r: node->node] {
  all n1, n2: node | n1 in n2.*r
}
```

Question

Suppose we replace

`stronglyConnected[e]`

by

`weaklyConnected[e]`.

Would that make a difference? Why (not)?

Representing State

In each state, there is a current set of tree nodes, and a current set of parent pointer links:

```
sig State {
 // set of tree nodes in current state
  tree: set Node,
 // parent pointers in current state
  parent: Node -> lone Node
}
```

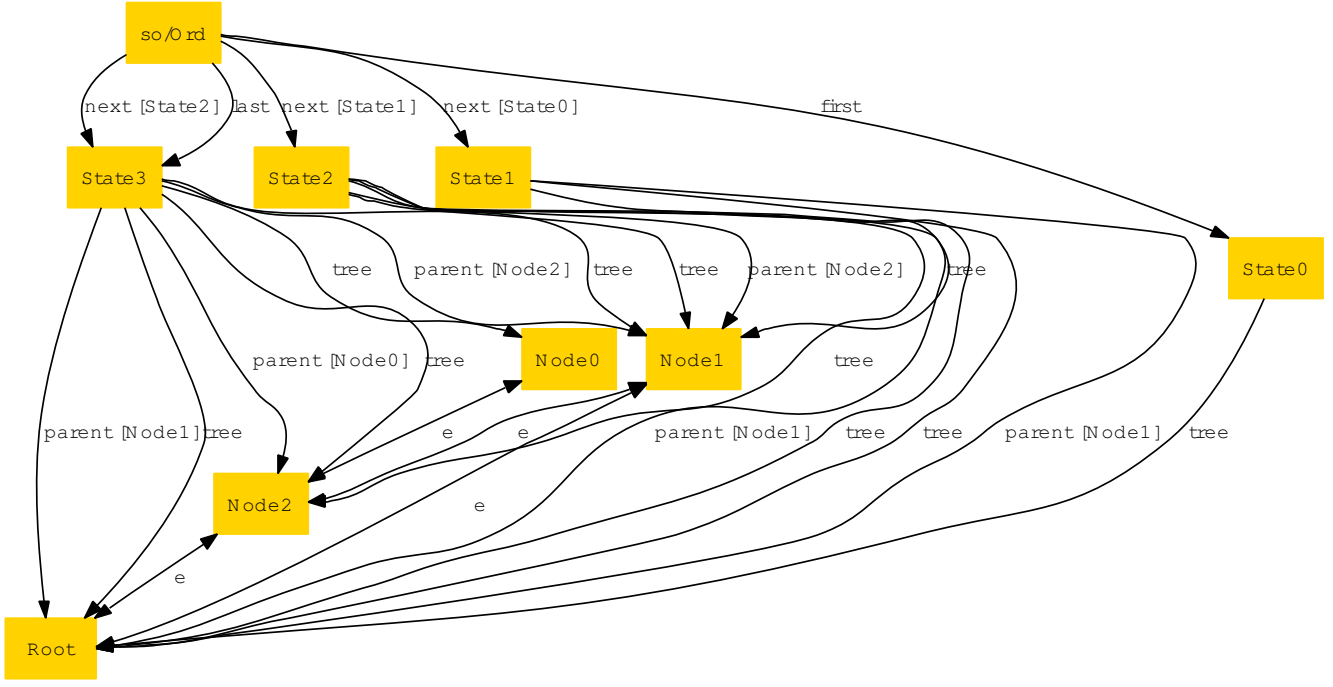
Initialisation, Tree Extension

```
pred init {
  let fs = so/first | { fs.tree = Root and no fs.parent }
}

pred extend [pre, post: State] {
  some x,y: Node | x in pre.tree
                   and y !in pre.tree
                   and y in x.e
                   and post.tree = pre.tree + y
                   and post.parent = pre.parent + y->x
}
```

Tree Creation

```
fact createTree {
     init
     all s: State - so/last |
        let s' = so/next[s] | extend[s,s']
}

run {} for exactly 4 Node, 4 State
```

Example Run


so/Ord
next [State2]
last
next [State1]
next [State0]
first
State3
State2
State1
State0
tree
parent [Node2]
tree
tree
parent [Node2]
tree
parent [Node0]
tree
Node0
Node1
tree
parent [Node1]
tree
e
e
parent [Node1]
tree
tree
parent [Node1]
tree
Node2
e
e
Root

Checking the Algorithm

Check that the structure found at the last state is indeed a tree over all nodes of the graph.

Note that in the graph utility, trees are defined with the arrows pointing towards the leaves. So we have to check for the converse of the `parent` relation.

```
assert STfound {
     so/last.tree = Node
     treeRootedAt[~(so/last.parent),Root]
}

check STfound for exactly 4 Node, 4 State
```

Result

```
Executing "Check STfound for exactly 4 Node, 4 State"
   Solver=minisat Bitwidth=4 Symmetry=ON
   797 vars. 96 primary vars. 2583 clauses. 75ms.
   No counterexample found. Assertion may be valid. 34ms.
```

Problem of Finding Minimum Spanning Tree

- A weighted graph is a graph with weights assigned to the edges. Think of the weight as an indication of distance.
- Let G be a weighted, symmetric and connected graph. Assume there are no self-loops. (Or, if there are self-loops, make sure their weight is set to 0.)
- A minimum spanning tree for weighted graph G is a spanning tree of G whose edges sum to minimum weight.
- Caution: minimum spanning trees are not unique.
- Applications: finding the least amount of wire necessary to connect a group of workstations (or homes, or cities, or . . .).

Minimum Spanning Tree: Prim's Algorithm

Finds a minimum spanning tree for an arbitrary weighted symmetric and connected graph. See Prim [1957], [Skiena, 1998, 4.7].

- Select an arbitrary graph node r to start the tree from.
- While there are still nodes not in the tree
  - Select an edge of minimum weight between a tree and non-tree node.
  - Add the selected edge and vertex to the tree.

It is not at first sight obvious that Prim's algoritm always results in a minimum spanning tree, but this fact can be checked by means of an Alloy specification.

Alloy Specification of Prim's Algorithm

Start as before, only now the graphs have weighted edges:

```
module myexamples/mst

open util/graph[Node] as graph
open util/ordering[State] as so
open util/ordering[W] as wo

// graphs with weighted edges
sig Node { w: Node -> lone W }
one sig Root extends Node {}

sig W {} // weights
```

```
fact wSymm { all x,y: Node | w[x,y] = w[y,x] }

fact wDiv { some x,y: Node | w[x,y]= wo/last }
```

Alloy Specification of Prim's Algorithm (ctd)

```
//define edge in terms of weighted edges
fun edge: Node -> set Node {
   { x,y: Node | some w[x,y] }
}

fact niceGraph {
     noSelfLoops[edge]
     undirected[edge]         // edge relation is symmetric
     stronglyConnected[edge] // no lose parts
}
```

Alloy Specification of Prim's Algorithm (ctd)

States and initialisation as before:

```
sig State {
 // set of tree nodes in current state
  tree: set Node,
 // parent pointers in current state
  parent: Node -> lone Node
}

pred init {
  let fs = so/first | { fs.tree = Root and no fs.parent }
}
```

Alloy Specification of Prim's Algorithm (ctd)

Extension now looks for the edge with minimum weight:

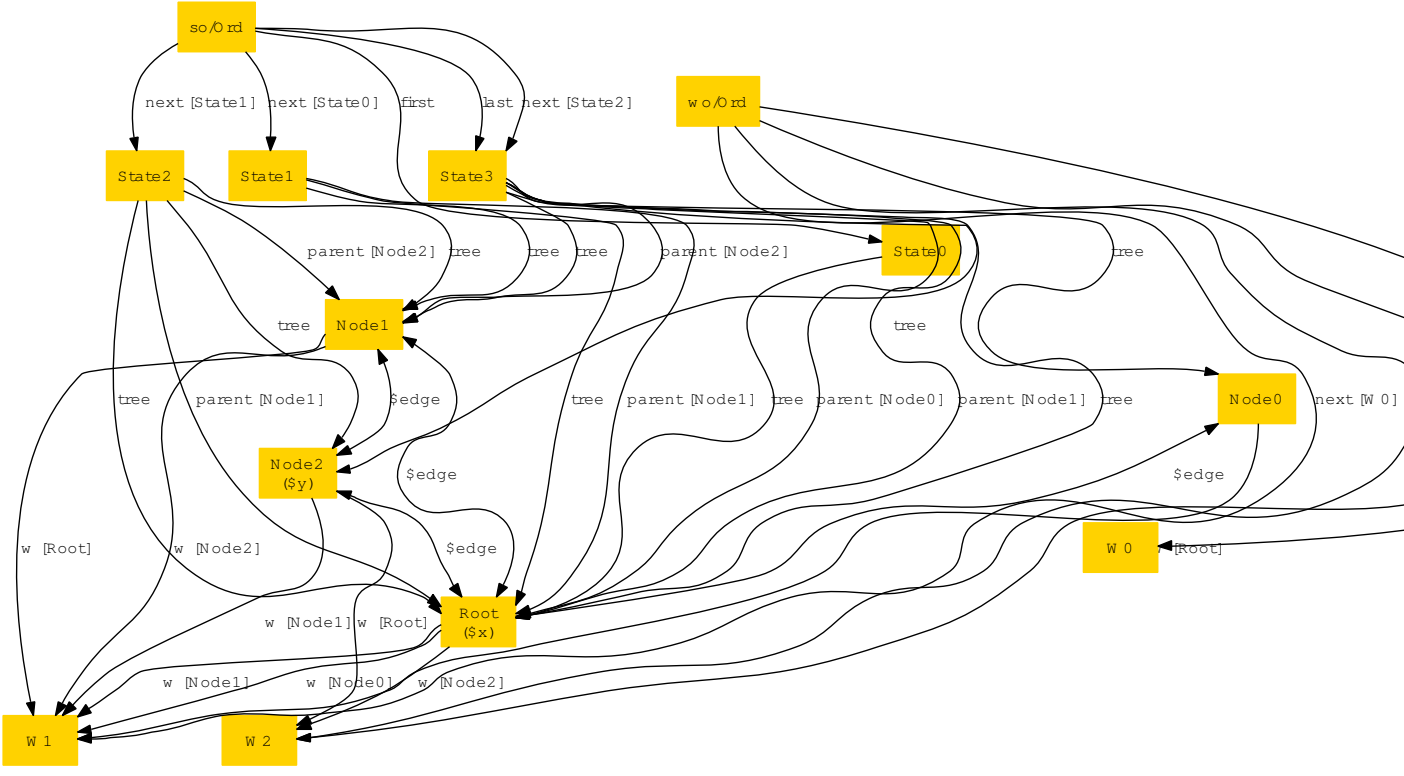
```
pred extend [pre, post: State] {
  some x,y: Node | x in pre.tree
                   and y !in pre.tree
                   and y in x.edge
                   and all y':Node | {
                       y' in x.edge => lte[w[x,y],w[x,y']]
                       }
                   and post.tree = pre.tree + y
                   and post.parent = pre.parent + y->x
}
```

Alloy Specification of Prim's Algorithm (ctd)

Rest as before:

```
fact createTree {
     init
     all s: State - so/last |
        let s' = so/next[s] | extend[s,s']
}

run {} for exactly 4 Node, 4 State, 3 W
```

Example Run


so/Ord
next [State1]
next [State0]
first
last
next [State2]
w o/Ord
State2
State1
State3
State0
parent [Node2]
tree
tree
tree
parent [Node2]
tree
tree
Node1
tree
tree
tree
parent [Node1]
$edge
tree
parent [Node1]
tree
parent [Node0]
parent [Node1]
tree
Node0
next [W 0]
Node2
($y)
$edge
$edge
w [Root]
w [Node2]
$edge
W 0
[Root]
Root
($x)
w [Node1]
w [Root]
w [Node1]
w [Node0]
w [Node2]
W 1
W 2

A Check

```
assert STfound {
     so/last.tree = Node
     treeRootedAt[~(so/last.parent),Root]
}

check STfound for exactly 4 Node, 4 State, 3 W expect 0
```

This check succeeds. The checks establishes that the result of the algorithm is a spanning tree (within the specified Alloy scope).

Still to be checked: ...

- We have not checked yet that the resulting tree is a minimum spanning tree.
- Checking this is part of your homework for this week.
- Of course, the check might reveal that there is still something wrong with the specification!

References

R.C. Prim. Shortest connection networks and some generalizations. Bell System Technical Journal, 36:1389–1401, 1957.

Steven S. Skiena. The Algorithm Design Manual. Springer Verlag, New York, 1998. Second Printing.