Stable Path Problem Specification in Alloy

Sanjai Narain
narain@research.telcordia.com

The network state is a set of node-rib pairs. Each pair defines the node's current route (rib) to a fixed sink node. Define a state transition relation trans modeling the route selection and dissemination activity at each node. Then, ask Alloy to check if there is a sequence of states s_0, …, s_k such that trans[s_0, s_1] && .. && trans[s_k, s_0]. Any solution returned by Alloy represents a bad gadget. The gadget consists of:

- A topology specifying the endpoints of each edge
- The preferred and backup paths for each node. These are the permitted paths
- An oscillating sequence of states

Note that Alloy not only generates the oscillating sequence but also the topology and path preferences. One only supplies as inputs the definition of trans and *sizes* of sets of nodes, edges and node-rib pairs. Empty paths are not modeled as yet.

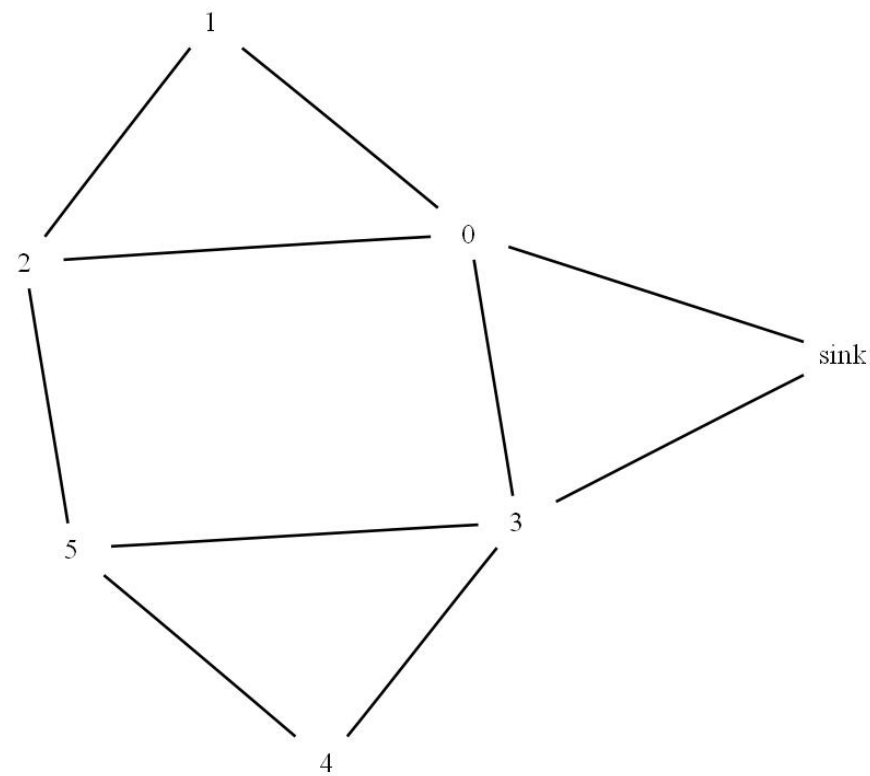
Given states s and t, trans[s, t] provided:
  s and t are different, and
  For every non-sink node:
    At least one permitted path for the node can be constructed from s, and
    If node's preferred path can be constructed, then it is node's rib in t, and
    If node's preferred path cannot be constructed, then node's backup path is the node's rib in t

A node's preferred path can be constructed from a state, provided:
  A path can be constructed for that node from the state, and
  This path is the preferred path

A path can be constructed for a node from a state, provided:
  The node is directly connected to sink, or
  The node is directly connected to another node and that node has a rib in the state

Similarly, for a node's backup path being constructed from a state.

Alloy generated the following bad gadget consisting of 7 nodes, 10 edges and an oscillating sequence of four states, in 50 seconds:



Network Topology

Permitted Paths and Preferences					
0S	**120S**	**210S**	**3S**	**43S**	**530S**
03S	**10S**	**20S**	**30S**	**453S**	**53S**

State Transitions						
s0	**53S**	**43S**	**3S**	**0S**	**210S**	**120S**
s1	**53S**	**43S**	**3S**	**0S**	**20S**	**10S**
s2	**53S**	**43S**	**3S**	**0S**	**210S**	**120S**
s3	**53S**	**43S**	**3S**	**0S**	**20S**	**10S**
s0	**53S**	**43S**	**3S**	**0S**	**210S**	**120S**

```
-- The Alloy model is given below

-- A rib is a node's static route to a fixed destination sink
sig node_rib_pair {device: node, rib: seq node}

-- A state is the set of rib of all nodes
sig state {node_rib_pairs: set node_rib_pair}

-- A node has two permitted paths to sink, path_1 and path_2
sig node {path_1: seq node,path_2: seq node} -- path_1 and path_2 are
the node's permitted paths

-- A sink is the fixed destination
sig sink extends node {}

-- An edge has two end points.
sig edge {end_point_1: node, end_point_2: node}

-- All edges are distinct
fact {all disj e1, e2:edge | not identical_edge[e1, e2]}

-- Edge endpoints are distinct
fact {all e:edge | e.end_point_1 != e.end_point_2}

-- Each state has a single rib for each node except for the sink node
fact {all s:state | all x:node | x!=sink => (one y:s.node_rib_pairs |
y.device=x)}

-- No state has a rib for the sink node
fact {all s:state | not (some x:s.node_rib_pairs | x.device=sink)}

-- All ribs are permitted
fact {all x:node_rib_pair | permitted[x.device, x.rib]}

-- All permitted paths lead to sink and are distinct
fact {all x:node | not(x=sink) => (is_path[x, sink, x.path_1] &&
is_path[x, sink, x.path_2] && x.path_1 != x.path_2)}

----------------------------------------------------------------------
-
pred permitted[x:node, y:seq node] {x.path_1=y || x.path_2=y}

----------------------------------------------------------------------
-
pred directly_connected[n1:node, n2:node]
{
   some e:edge |
        (e.end_point_1 = n1 && e.end_point_2 = n2) ||
        (e.end_point_1 = n2 && e.end_point_2 = n1)
}
----------------------------------------------------------------------
-
pred is_path [S:node, D: node, P:seq node]
{
   not(hasDups[P]) && -- Removing this creates a solution
   first[P]=S &&
   last[P]=D &&
```

```
   all i: P.inds -P.lastIdx | directly_connected[P[i], P[i+1]]
}

-----------------------------------------------------------------------
-
-- Two edges are identical if their end points match in any order. Not
used in the model
pred identical_edge[e1, e2:edge] {
    (e1.end_point_1=e2.end_point_1 &&
     e1.end_point_2=e2.end_point_2) ||
    (e1.end_point_1=e2.end_point_2 &&
     e1.end_point_2=e2.end_point_1)
}

-----------------------------------------------------------------------
-
-- Node n's path_1 is available in the current state s
pred path_1_available[s:state, n:node]
{       (n.path_1[0]=n && n.path_1[1]=sink) || -- n is directly
connected to sink
   (some x:s.node_rib_pairs |
       directly_connected[n, x.device] &&
       n.path_1=(x.rib).insert[0, n])
}

-----------------------------------------------------------------------
-
-- Node n's path_2 is available in the current state s
pred path_2_available[s:state, n:node]
{       (n.path_2[0]=n && n.path_2[1]=sink) || -- n is directly
connected to sink
   some x:s.node_rib_pairs |
       directly_connected[n, x.device] &&
       n.path_2 = (x.rib).insert[0, n]
}

-----------------------------------------------------------------------
-
-- Legal transition from state s to state t
-- s and t are different
-- For every non-sink node n:
--    n's path_1 or path_2 are available in s
--    If path_1 is available, then it is n's rib in t
--    If path_1 is unavailable, then path_2 is n's rib in t
pred trans[s:state, t:state]
{     different_states[s, t] &&
  all n:node |
   (n != sink) =>
   ((path_1_available[s, n] || path_2_available[s, n]) && -- one path
is available in s
    (path_1_available[s, n] => has_path_1[n, t]) &&
    ((not path_1_available[s, n]) => has_path_2[n, t]))
}
-----------------------------------------------------------------------
-
pred has_path_1[n:node, s: state]
{
```

```
   some y:s.node_rib_pairs | y.device=n && y.rib=n.path_1
}
pred has_path_2[n:node, s: state]
{
   some y:s.node_rib_pairs | y.device=n && y.rib=n.path_2
}
pred different_states[s, t:state]
{
   some x:s.node_rib_pairs, y:t.node_rib_pairs | x.device=y.device &&
x.rib!=y.rib
}
----------------------------------------------------------------------
-
-- Find a three state bad gadget
pred test_3_state_3_node_3_edge []
{
    some disj s0, s1, s2:state |
          trans[s0, s1] && trans[s1, s2] && trans[s2, s0]
}

----------------------------------------------------------------------
-
-- Find a 6 state bad gadget with 7 nodes
pred test_4_state_7_node_10_edge []
{
    some s0, s1, s2, s3:state |
            trans[s0, s1] &&
            trans[s1, s2] &&
            trans[s2, s3] &&
            trans[s3, s0]
}

----------------------------------------------------------------------
-
-- Explore all gadgets up to a certain size
pred test_all []
{
   some s:seq state | #s>3 &&  all i:s.inds -s.lastIdx | trans[s[i],
s[i+1]] && trans[s[s.lastIdx], s[0]]
}

----------------------------------------------------------------------
-
run test_3_state_3_node_3_edge for exactly 3 state, exactly 3 node,
exactly 3 edge, 6 node_rib_pair
run test_4_state_7_node_10_edge for exactly 4 state, exactly 7 node, 10
edge, 10 node_rib_pair
```