```
1  open util/ordering[Node] as ring
2
3  one sig Base { size: Int } { size = 4 }
4
5  sig Node {}
6
7  sig Succs {list: seq Node}{ lastIdx[list] = 2 }
8
9  abstract sig Status {}
10     one sig Active, Failed extends Status {}
11
12 abstract sig LiveStatus {}
13   one sig Stabilizing, Rectifying extends LiveStatus{}
14
15    lt[n1,n2] =>   ( lt[n1,nb] && lt[nb,n2] )
16 pred between [n1, nb, n2: Node] {
17              else ( lt[n1,nb] || lt[nb,n2] )  }
18
19 conc state System {
20
21     conc state [id : Node] NodeProc {
22
23         env event Fail {}
24         env event Join {}
25
26         succ: one Succs
27         prdc: one Node
28         status: lone Status
29         saved: lone Node
30         bestSucc: lone Node
31         liveStatus: lone LiveStatus
32
33
34       state Live {
35
36         trans NodeFailure {
37           on Fail
38           when {
39               status = Live
40
41               //Node cannot fail if it will leave a
42               //member with no successors
43               all otherNode: Node |
44               (NodeProc[ids]/status = Active) &&
45           not (otherNode = this) &&
46               this in NodeProc[ids]/succ.list.elemes
47                =>  some ids': Node |
48               ((NodeProc[ids]/status = Active) - id) |
49               ids' in (NodeProc[ids]/succ.list.elems )
```

```
50              }
51            }
52            do status' = Failed
53            goto Failed
54          }
55
56          default state Stabilizing{
57
58            trans StabilizeFromSuccessor {
59                when (no liveStatus)
60                do{
61                  let succ1 = succ.list[0] | one p, q: Node
62                  | {
63
64                  //Successor is Live
65                  NodeProc[succ1]/statis = Active => (
66                  some u: Succs |
67                  ((u.list =
68                  insert [NodeProc[succ1]/succ.list, 0,
69                  succ1]
70                  and
71                  succ' = u
72                  and p in succ'.list[0])
73                  and
74                  //Check if the succ's pred is better
75                  (between [this, NodeProc[succ1]/prdc,
    succ1])
76                  =>
77                  //Save it for next step
78                  (saved' = NodeProc[succ1]/prdc and
79                  //Update status
80                  liveStatus' = Stabilizing)
81                  else
82                  (NodeProc[p]/liveStatus' = Rectifying and
83                  NodeProc[p]/saved = id) ) )
84
85                  //Successor is dead
86                  else
87                  (( some u : Succs |
88                  //Remove it from succList
89                  u.list = add[rest[succ.list],
90                  ring/next[last[succ.list]]]
91                  and succ' = u
92                  //q is the new successor
93                  and q in succ'list[0]
94                  //Have new successor rectify
95                  and NodeProc[q]/liveStatus' = Rectifying
96                  and NodeProc[q]/saved' = this
97                  }
```

```
98                }
99              }
100
101
102             trans StabilizeFromPredecessor {
103               when (liveStatus = Stabilizing)
104               //Make sure pred is still better succ
105               and between[id, saved, succ.list[0])
106               do{
107                 let newSucc = saved {
108                 one p: Node | p in succ.list[0] | (
109                 //Pred is alive
110                 NodeProc[newSucc]/status = Active
111                 =>
112                 (some u: Succs |
113                 u.list = insert[succ.list, 0, newSucc]
114                 //Adopt its succ list
115                 and succ' = u
116                 and liveStatus' = no status
117                 //Inform it to update pred
118                 and NodeProc[newSucc]/liveStatus' =
        Rectifying
119                 and NodeProc[newSucc]/saved' = id
120                 )
121                 //Pred is dead
122                 else
123                 (
124                 succ' = succ
125                 and liveStatus' = no status
126                 //Tell succ to update pred
127                 and NodeProc[p]/liveStatus' = Rectifying
128                 and saved' = no saved
129                 and NodeProc[p]/saved' = id
130                 ))
131                 }
132               }
133             }
134
135           state Rectifying {
136             when (liveStatus = Rectifying)
137             do{
138               saved' = no saved
139               status = no status
140
141               between[prdc, saved, this] =>
142               prdc' = saved
143             else
144             prdc in members
145             =>  prdc' = prdc
146               else
```

```
147                 prdc' = saved
148               }
149             }
150           }
151         }
152
153         state Failed {
154           trans NodeJoin {
155             on Join
156             when status = Failed
157             do{
158               status = Active
159               some otherNode: Node |
160           not (otherNode = this) &&
161               NodeProc[otherNode]/stauts = Active &&
162               between[otherNode, this,
163               NodeProc[otherNode]/succ.list[0]] &&
164               succ' = NodeProc[otherNode]/succ &&
165               prdc' = otherNode
166
167             }
168             goto Live
169           }
170         }
171     }
172
173 }
174
175
176
177
178 /********** PROPERTIES **********/
179
180
181 //Every member process has a live succ
182 pred oneLiveSucessor {
183   all id: Node | (NodeProcess[id]/status = Active)
184     => (NodeProc[NodeProc[id]/bestSucc]/status = Active)
185   gte [#principals, Base.size]
186 }
187
188 //Every member NodeProcess has an ordered list
189 //of successors
190 pred OrdederedSuccessorList {
191   all id: Node | NodeProc[id]/status = Active | (
192
193   let curr = NodeProc[id] | (
194
195   (all disj j, k: curr/succ.list.inds |
196     lt [j, k] =>
```

```
197       between [id, curr/succ.list[j], curr/succ.list[k]]
198
199       all disj j, k, l: curr/succ.list.inds |
200       lt [j, k] && lt [k, l]
201       => between[curr/succ.list[j], curr/succ.list[k],
202       curr/succ.list[l]]) )
203       )
204  }
205
206  assert InvariantImpliesOrderedList {
207    ag(oneLiveSucessor => OrdederedSuccessorList)
208  }
209
210  pred NoDuplicates [s: NetState] {
211     all m: Node | (NodeProcess[m]/status = Active) |
212     let curr = NodeProc[id] | {
213     no j: curr/succ.list.inds | m = curr/succ.list[j]
214     no disj j, k: curr/succ.list.inds |
215        curr/succ.list[j] =  curr/succ.list[k]}
216  }
217
218  assert InvariantImpliesNoDuplicatet {
219    ag(oneLiveSucessor => NoDuplicates)
220  }
```