```
/*
MIT License

Copyright (c) [2021] [Tamjid Hossain]

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
*/

open util/ordering[Node] as ring

one sig Base { size: Int } { size = 4 }

sig Node {}

sig Succs {list: seq Node}{ lastIdx[list] = 2 }

abstract sig Status {}
    one sig Active, Failed extends Status {}

abstract sig LiveStatus {}
  one sig Stabilizing, Rectifying extends LiveStatus{}

```

```
   lt[n1,n2] =>   ( lt[n1,nb] && lt[nb,n2] )
pred between [n1, nb, n2: Node] {
             else ( lt[n1,nb] || lt[nb,n2] )  }

conc state System {

    conc state [id : Node] NodeProc {

        env event Fail {}
        env event Join {}

        succ: one Succs
        prdc: one Node
        status: lone Status
        saved: lone Node
        bestSucc: lone Node
        liveStatus: lone LiveStatus


      state Live {

        trans NodeFailure {
          on Fail
          when {
              status = Live

              //Node cannot fail if it will leave a
              //member with no successors
              all otherNode: Node |
              (NodeProc[ids]/status = Active) &&
          not (otherNode = this) &&
              this in NodeProc[ids]/succ.list.elemes
               =>  some ids': Node |
              ((NodeProc[ids]/status = Active) - id) |
              ids' in (NodeProc[ids]/succ.list.elems )

            }
          }
          do status' = Failed
          goto Failed
        }

        default state Stabilizing{

          trans StabilizeFromSuccessor {
              when (no liveStatus)
              do{
                let succ1 = succ.list[0] | one p, q: Node
                | {

```

```
                //Successor is Live
                NodeProc[succ1]/statis = Active => (
                some u: Succs |
                ((u.list =
                insert [NodeProc[succ1]/succ.list, 0,
                succ1]
                and
                succ' = u
                and p in succ'.list[0])
                and
                //Check if the succ's pred is better
                (between [this, NodeProc[succ1]/prdc,
succ1])
                =>
                //Save it for next step
                (saved' = NodeProc[succ1]/prdc and
                //Update status
                liveStatus' = Stabilizing)
                else
                (NodeProc[p]/liveStatus' = Rectifying and
                NodeProc[p]/saved = id) ) )

                //Successor is dead
                else
                (( some u : Succs |
                //Remove it from succList
                u.list = add[rest[succ.list],
                ring/next[last[succ.list]]]
                and succ' = u
                //q is the new successor
                and q in succ'list[0]
                //Have new successor rectify
                and NodeProc[q]/liveStatus' = Rectifying
                and NodeProc[q]/saved' = this

                }
              }
            }


            trans StabilizeFromPredecessor {
              when (liveStatus = Stabilizing)
              //Make sure pred is still better succ
              and between[id, saved, succ.list[0])
              do{
                let newSucc = saved {
                one p: Node | p in succ.list[0] | (
                //Pred is alive
                NodeProc[newSucc]/status = Active
                =>
```

```
                  (some u: Succs |
                  u.list = insert[succ.list, 0, newSucc]
                  //Adopt its succ list
                  and succ' = u
                  and liveStatus' = no status
                  //Inform it to update pred
                  and NodeProc[newSucc]/liveStatus' =
      Rectifying
                  and NodeProc[newSucc]/saved' = id
                  )
                  //Pred is dead
                  else
                  (
                  succ' = succ
                  and liveStatus' = no status
                  //Tell succ to update pred
                  and NodeProc[p]/liveStatus' = Rectifying
                  and saved' = no saved
                  and NodeProc[p]/saved' = id
                  ))
                  }
                }
              }

            state Rectifying {
              when (liveStatus = Rectifying)
              do{
                saved' = no saved
                status = no status

                between[prdc, saved, this] =>
                prdc' = saved
              else
              prdc in members
              =>  prdc' = prdc
                else
                prdc' = saved
              }
            }
          }
        }

        state Failed {
          trans NodeJoin {
            on Join
            when status = Failed
            do{
              status = Active
              some otherNode: Node |
          not (otherNode = this) &&
              NodeProc[otherNode]/stauts = Active &&
              between[otherNode, this,
              NodeProc[otherNode]/succ.list[0]] &&
              succ' = NodeProc[otherNode]/succ &&
              prdc' = otherNode

            }
            goto Live
          }
        }
    }

  }




  /********** PROPERTIES **********/


  //Every member process has a live succ
  pred oneLiveSucessor {
    all id: Node | (NodeProcess[id]/status = Active)
      => (NodeProc[NodeProc[id]/bestSucc]/status = Active)
    gte [#principals, Base.size]
  }

  //Every member NodeProcess has an ordered list
  //of successors
  pred OrdederedSuccessorList {
    all id: Node | NodeProc[id]/status = Active | (

    let curr = NodeProc[id] | (

    (all disj j, k: curr/succ.list.inds |
      lt [j, k] =>
      between [id, curr/succ.list[j], curr/succ.list[k]]

      all disj j, k, l: curr/succ.list.inds |
      lt [j, k] && lt [k, l]
      => between[curr/succ.list[j], curr/succ.list[k],
      curr/succ.list[l]]) )
      )
  }

  assert InvariantImpliesOrderedList {
    ag(oneLiveSucessor => OrdederedSuccessorList)
  }

  pred NoDuplicates [s: NetState] {
      all m: Node | (NodeProcess[m]/status = Active) |
      let curr = NodeProc[id] | {
      no j: curr/succ.list.inds | m = curr/succ.list[j]
      no disj j, k: curr/succ.list.inds |
        curr/succ.list[j] =  curr/succ.list[k]}
  }

  assert InvariantImpliesNoDuplicatet {
    ag(oneLiveSucessor => NoDuplicates)
  }
```