CONCURRENCY

©Magee/Kramer 1nd Edition

STATE MODELS & JAVA PROGRAMS

ANNY CAROLINE

PROF. ALEXANDRE SZTAJNBERG

Programa de Pós-Graduação em Engenharia Eletrônica (PEL)

Sumário

- conceitos de safety e liveness
- aplicação desses conceitos em modelos LTS
- exemplo de carros cruzando uma ponte de pista única
- leitores e escritores

Propriedades de segurança e vivacidade

- Propriedade
  - é um atributo que é verdadeiro para toda execução possível do programa

  - as propriedades interessantes para a programação concorrente são:
    - safety (segurança)
    - liveness (vivacidade)

Propriedades de segurança e vivacidade

- Segurança
  - nada de ruim acontece durante a execução
  - programa não deve atingir um “estado ruim”

- Algumas propriedades de segurança
  - o estado final do programa deve estar correto
  - exclusão mútua
  - ausência de deadlock
  - ausência de erros

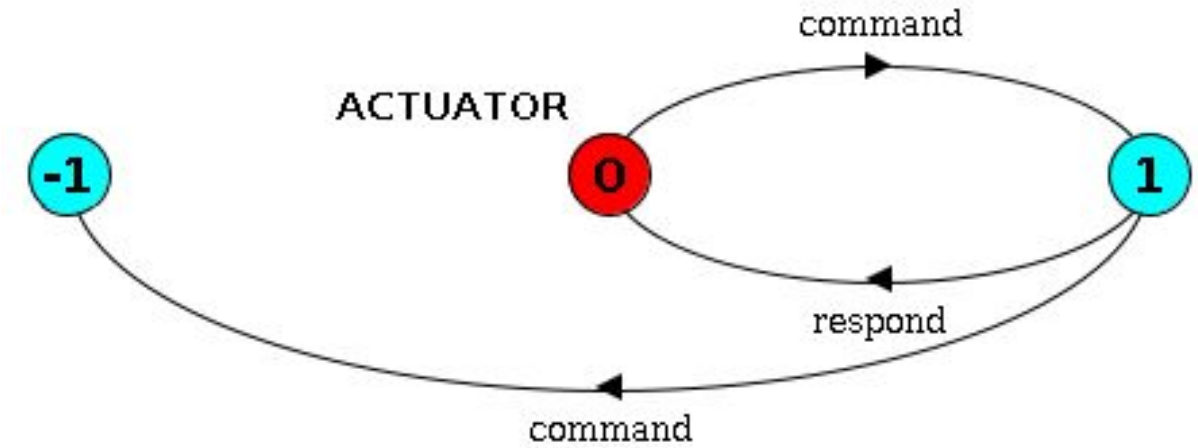
Propriedades de segurança e vivacidade

- Vivacidade
  - algo de bom acontece na execução
  - o programa eventualmente atinge um estado bom

- Algumas propriedades de vivacidade
  - o programa eventualmente termina
  - vivacidade associada ao acesso de recursos compartilhados
    - solicitações para acesso a recursos compartilhados são eventualmente atendidas?
  - são afetadas pelas políticas de escalonamento

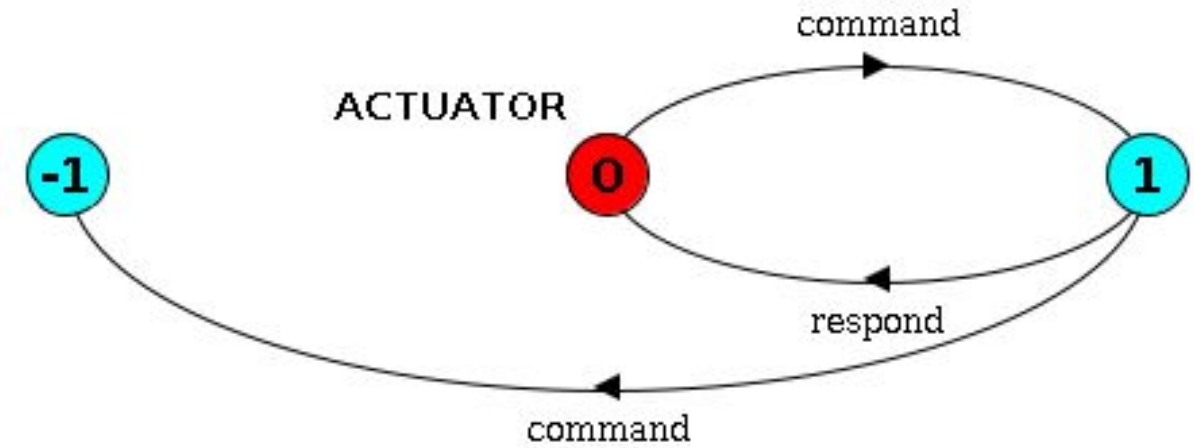
7.1 Segurança - Estados de erro

- deadlock
- estados de erro
  - um único identificador, -1



7.1 Segurança - Definindo estados de erro

- especificar situações consideradas como erro
- expressar as propriedades de segurança que devem ser preservadas

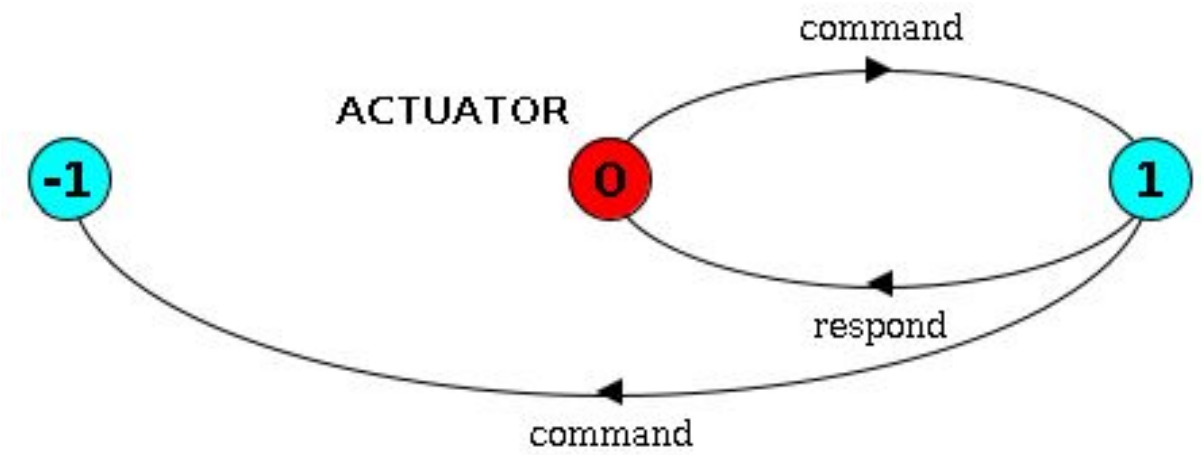


7.1 Segurança - Especificar os erros

- especificar situações consideradas como erro
  - estratégia já usada anteriormente (capítulos 4 e 5)
  - mais verbosa
  - tentar enumerar todas as possibilidades de erro não é uma boa estratégia

7.1 Segurança - Especificar os erros

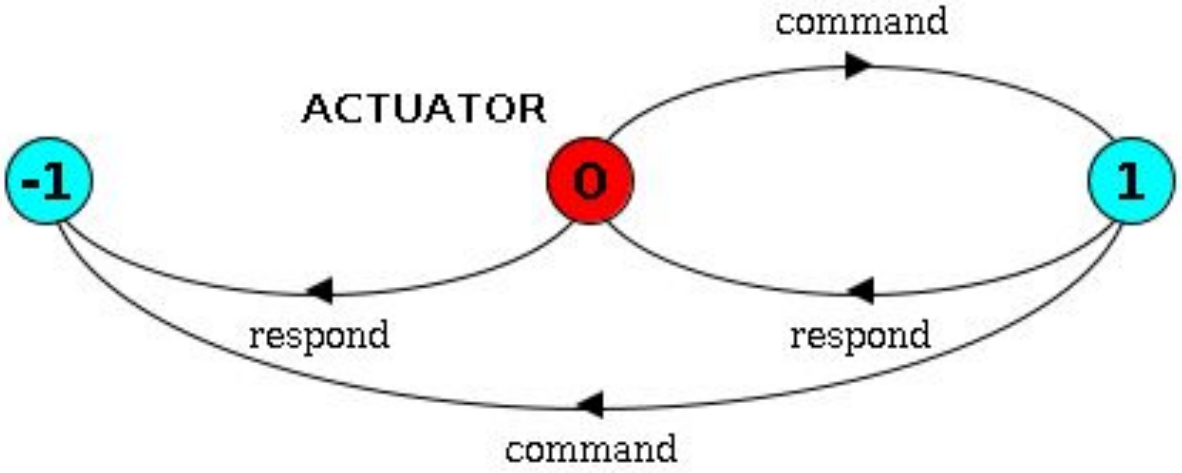
```
ACTUATOR = (command->ACTION),
ACTION = (respond->ACTUATOR | command->ERROR).
```



Trace to property violation in ACTUATOR:
command
command

7.1 Segurança - Especificar os erros

```
ACTUATOR = (command->ACTION | respond->ERROR),
ACTION = (respond->ACTUATOR | command->ERROR).
```



Trace to property violation in ACTUATOR:
respond

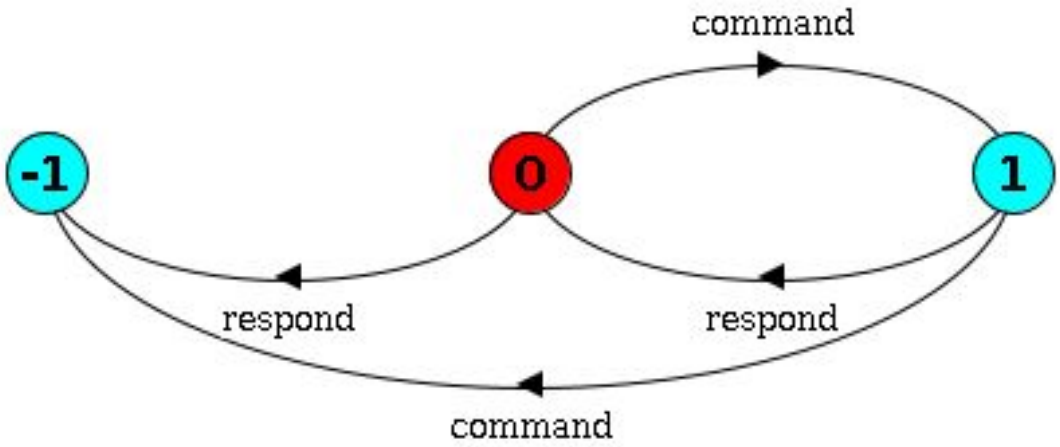
7.1.1 Segurança - Especificar as propriedades de segurança

- expressar as propriedades de segurança que devem ser preservadas
  - normalmente é melhor especificar diretamente as propriedades que devem ser preservadas ao invés de especificar o que não deve ocorrer

  - property process
  - sintaticamente, são processos prefixados com a palavra **property**
  - o compilador, então, gera automaticamente as transições para o estado de erro

7.1.1 Segurança - Especificar as propriedades de segurança

```
property ACTUATOR = (command->respond->ACTUATOR).
```



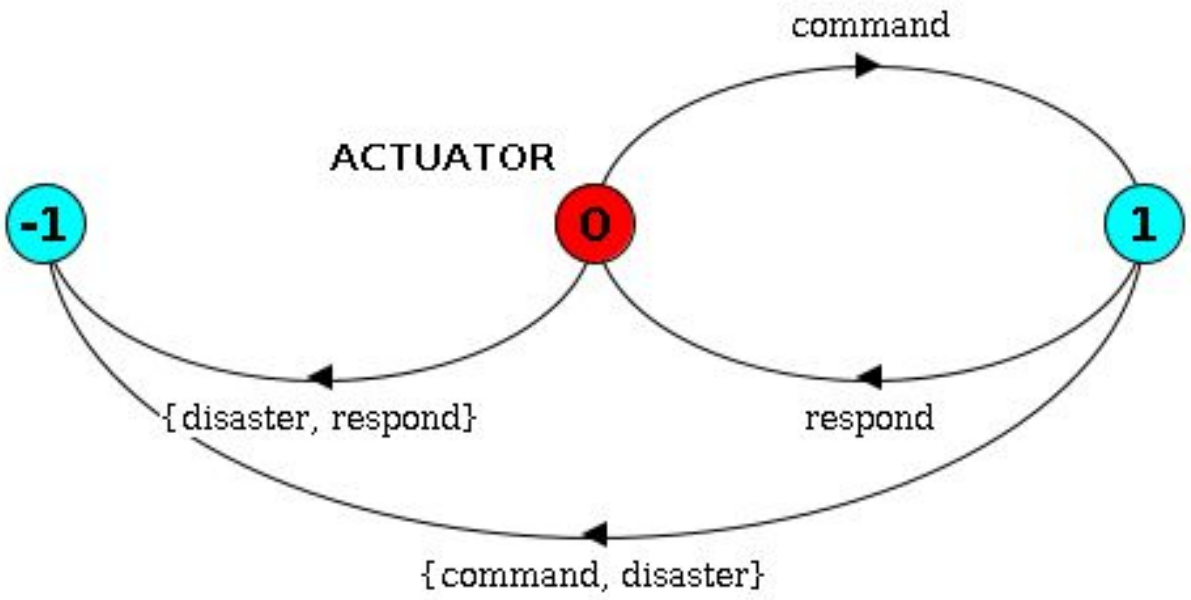
Trace to property violation in ACTUATOR:
respond

7.1.1 Segurança - Especificar as propriedades de segurança

- property process
  - todas as ações do alfabeto passam a ser elegíveis
  - todas as ações que não fazem parte da **property process** são transições para o estado de erro

7.1.1 Segurança - Especificar as propriedades de segurança

```
property ACTUATOR = (command->respond->ACTUATOR) + {disaster}.
```



Trace to property violation in ACTUATOR:
respond

7.1.2 Segurança - Exclusão mútua

Baseado no exemplo de semáforos do capítulo 5

```
LOOP = (mutex.down->enter->exit->mutex.up->LOOP).

||SEMADEMO = (p[1..3]:LOOP
              || {p[1..3]}::mutex:SEMAPHORE(1)).
```

- Para verificar se esse sistema realmente garante a exclusão mútua, podemos especificar uma propriedade de exclusão mútua e compor com o sistema da seguinte maneira

```
property MUTEX = (p[i:1..3].enter->p[i].exit->MUTEX).
||CHECK = (SEMADEMO || MUTEX).
```

7.1.2 Segurança - Exclusão mútua

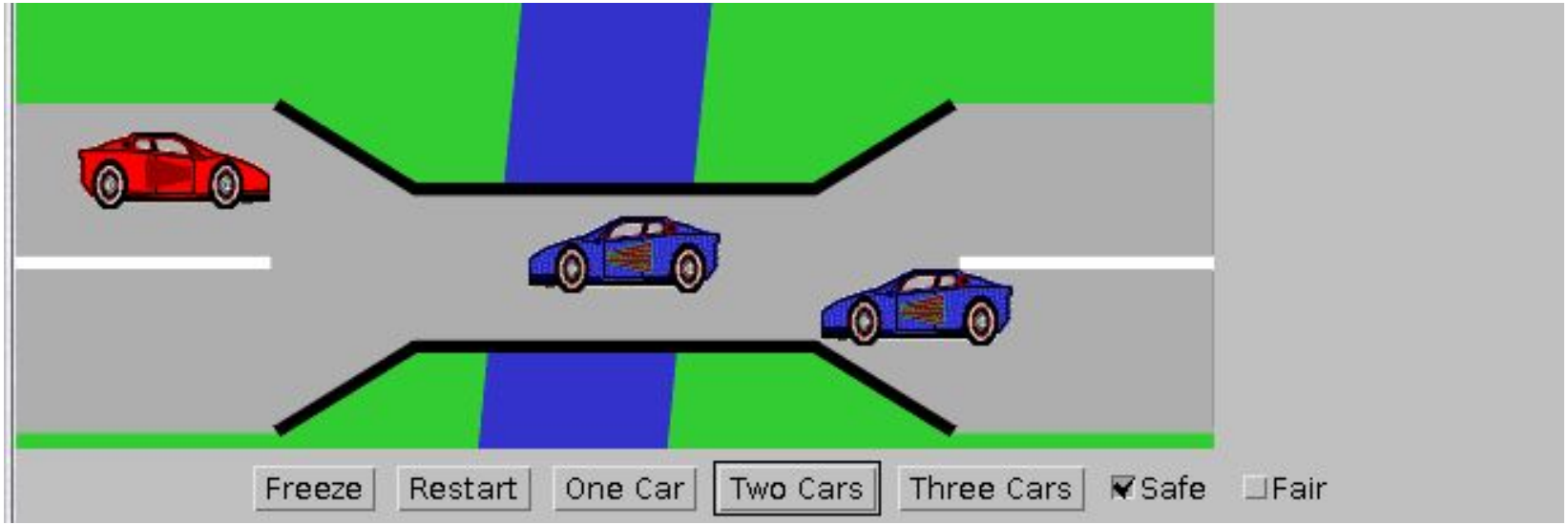
- A propriedade não é violada no sistema
- Mas, e se alterarmos o valor com que o semáforo é inicializado para 2 (i.e. `SEMAPHORE(2)`)?

Trace to property violation in MUTEX:

p.1.mutex.down
p.1.enter
p.2.mutex.down
p.2.enter

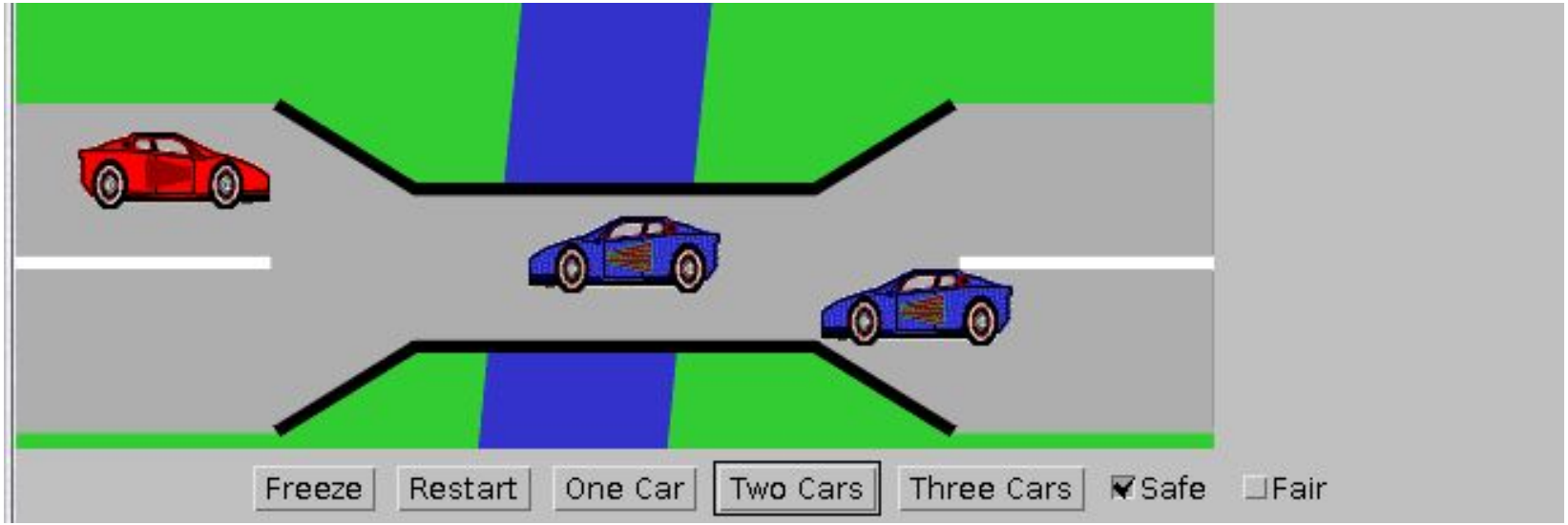
7.2 Segurança - Single-lane bridge model

- Uma ponte sobre um rio é grande o suficiente para permitir uma única faixa de tráfego.



7.2 Segurança - Single-lane bridge model

- O modelo também deve garantir que carros que se movem na mesma direção não ultrapassem uns aos outros



7.2 Segurança - Single-lane bridge model

```
const N = 3 // number of each type of car
range T = 0..N // type of car count
range ID= 1..N // car identities
```

- a essência do problema é o acesso a ponte, então os únicos eventos de interesse para o carro são o de entrar na ponte e sair da ponte

```
CAR = (enter->exit->CAR).
```

7.2 Segurança - Single-lane bridge model

- para modelar o fato que os carros não podem ultrapassar outros, foram definidos os seguintes processos. Eles limitam a ordem das ações de entrada e saída.

```
NOPASS1    = C[1],
C[i:ID]    = ([i].enter
NOPASS2    = C[1],
C[i:ID]    = ([i].exit

||CONVOY = ([ID]:CAR || NOPASS1 || NOPASS2).

||CARS = (red:CONVOY || blue:CONVOY).
```

O processo CONVOY modela um conjunto de carros viajando na mesma direção que entram na ponte um depois do outro e deixam a ponte um após o outro.

7.2 Segurança - Single-lane bridge model

- o modelo da ponte armazena a quantidade de carros vermelhos e azuis

```
BRIDGE = BRIDGE[0][0],

BRIDGE[nr:T][nb:T] =
    (when (nb==0)
          red[ID].enter  -> BRIDGE[nr+1][nb]
    |red[ID].exit    -> BRIDGE[nr-1][nb]
    |when (nr==0)
          blue[ID].enter -> BRIDGE[nr][nb+1]
    |blue[ID].exit   -> BRIDGE[nr][nb-1]
).
```

7.2 Segurança - Single-lane bridge model

- falta definir a **propriedade de segurança** (a ser composta com o sistema) para verificarmos se não irá acontecer nenhuma colisão
- o índice é usado para contar os carros vermelhos (ou azuis) atualmente na ponte

```
property ONEWAY = (red[ID].enter  -> RED[1]
         |blue[ID].enter -> BLUE[1]
         ),

RED[i:ID]  = (red[ID].enter -> RED[i+1]
           |when(i==1) red[ID].exit  -> ONEWAY
           |when(i>1)  red[ID].exit  -> RED[i-1]
           ),
```

7.2 Segurança - Single-lane bridge model

```
BLUE[i:ID] = (blue[ID].enter -> BLUE[i+1]
             |when(i==1)blue[ID].exit  -> ONEWAY
             |when( i>1)blue[ID].exit  -> BLUE[i-1]
             ).
```

7.2 Segurança - Single-lane bridge model

```
||SingleLaneBridge = (CARS || BRIDGE || ONEWAY ).
```

No deadlocks/errors

```
||SingleLaneBridge = (CARS || ONEWAY ).
```

Trace to property violation in ONEWAY:

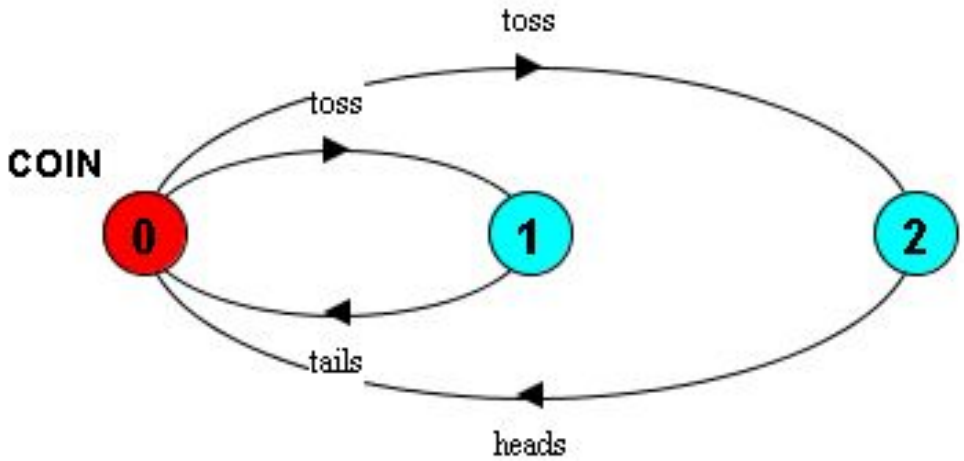
red.1.enter
blue.1.enter

7.3.1 Vivacidade - Propriedade de progresso

- Uma **progress property** define uma ação específica a ser executada eventualmente a partir de qualquer estado do sistema
- **Progress** é o oposto de **Starvation**

7.3.1 Vivacidade - Propriedade de progresso - COIN

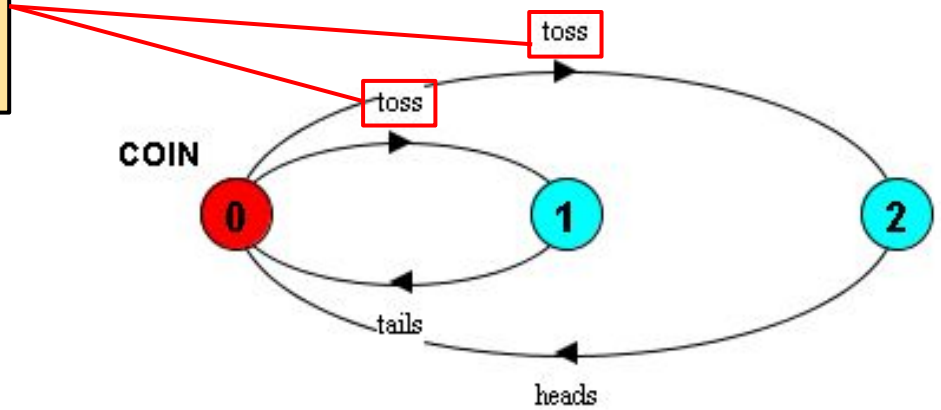
```
COIN = (toss->heads->COIN|toss->tail->COIN).
```



7.3.1 Vivacidade - Propriedade de progresso - Escolha justa

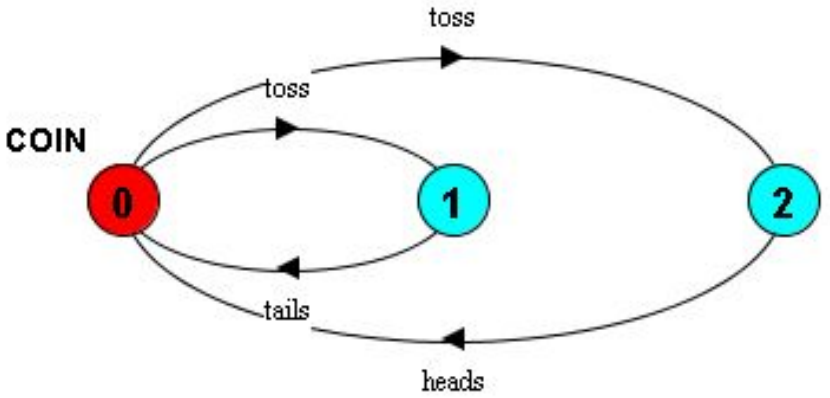
- se a moeda fosse lançada infinitas vezes
  - depende da política de escalonamento
  - se não for justa, poderíamos sempre escolher a transição **toss** que leva à **cara**

Escolha justa: se uma escolha sobre um conjunto de transições for executada infinitas vezes, todas as transições no conjunto serão executadas infinitas vezes.



7.3.1 Vivacidade - Propriedade de progresso - COIN

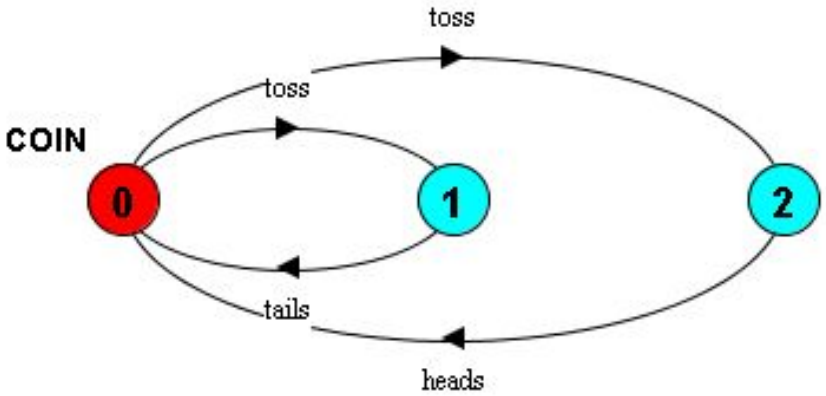
- o sistema **COIN** eventualmente escolherá **heads** e eventualmente escolherá **tails**
  - assumindo uma escolha justa, é claro
- podemos conferir essa propriedade definindo uma **progress property**



7.3.1 Vivacidade - Propriedade de progresso - COIN

```
progress HEADS = {heads}  progress HEADSorTAILS = {heads,tails}
progress TAILS = {tails}
```

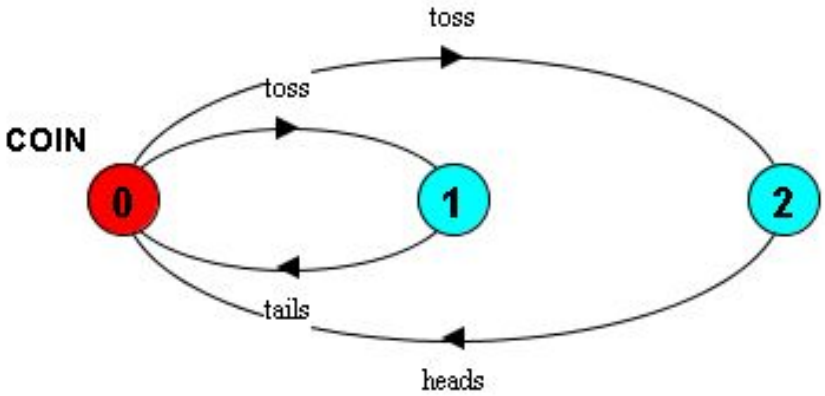
`progress P = {a1,a2..an}` define uma progress property P que afirma que em um execução infinita de um sistema S, pelo menos uma das ações `a1,a2..an` será executada infinitas vezes.



7.3.1 Vivacidade- Propriedade de progresso - COIN

```
progress HEADS = {heads}
progress TAILS = {tails}
```

`progress P = {a1,a2..an}` define uma progress property P que afirma que em um execução infinita de um sistema S, pelo menos uma das ações `a1,a2..an` será executada infinitas vezes.



LTSA check progress: No progress violations detected.

7.3.1 Vivacidade - Propriedade de progresso - TWOCOINS

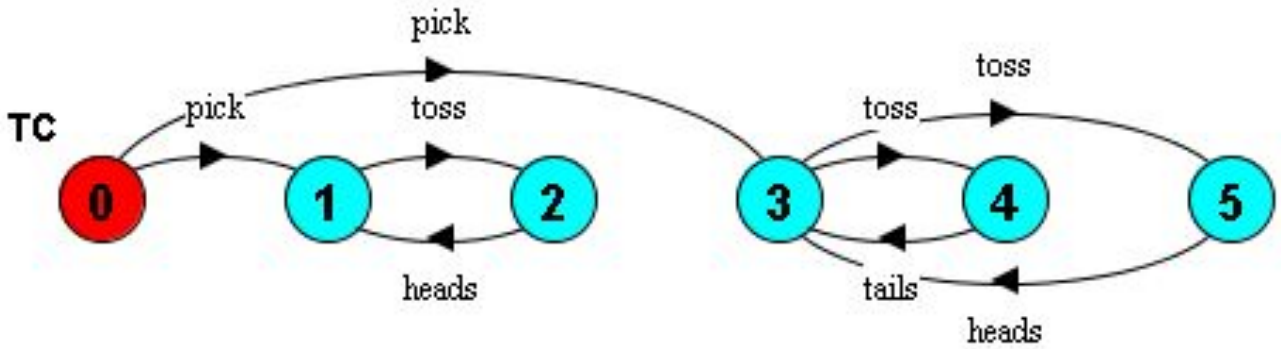
- antes de lançar as moedas, deve-se escolher entre duas moedas:
  - um moeda normal, com um lado cara e outro coroa
  - uma moeda modificada, com cara nos dois lados

7.3.1 Vivacidade - Propriedade de progresso - TWOCOINS

```
TWOCOIN = (pick->COIN|pick->TRICK),
TRICK   = (toss->heads->TRICK),
COIN    = (toss->heads->COIN|toss->tails->COIN).

progress HEADS = {heads}
progress TAILS = {tails}
```

Progress violation: **TAILS**
Trace to terminal set of states:
pick
Actions in **terminal set**:
{heads, toss}

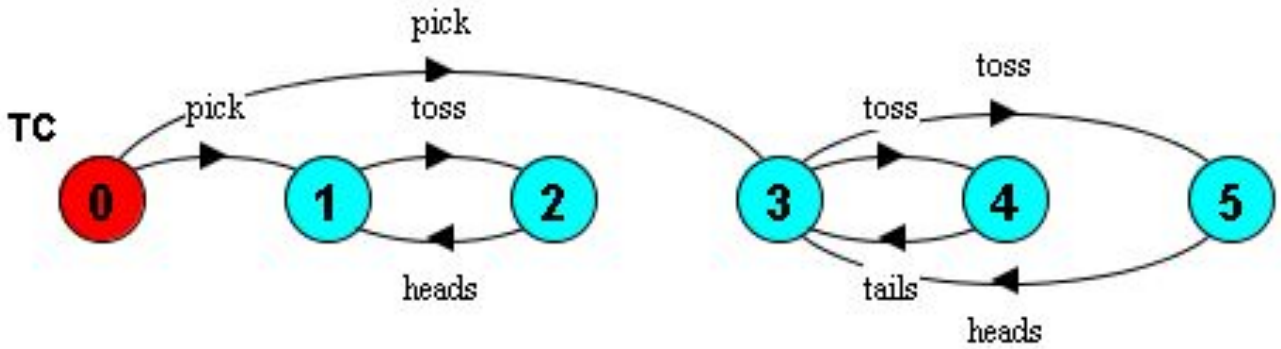


7.3.1 Vivacidade - Propriedade de progresso - TWOCOINS

```
TWOCOIN = (pick->COIN|pick->TRICK),
TRICK   = (toss->heads->TRICK),
COIN    = (toss->heads->COIN|toss->tails->COIN).

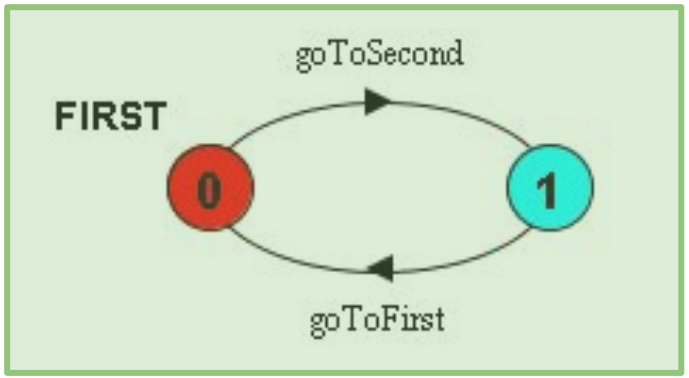
progress HEADSorTAILS = {heads,tails}
```

No progress violations detected.



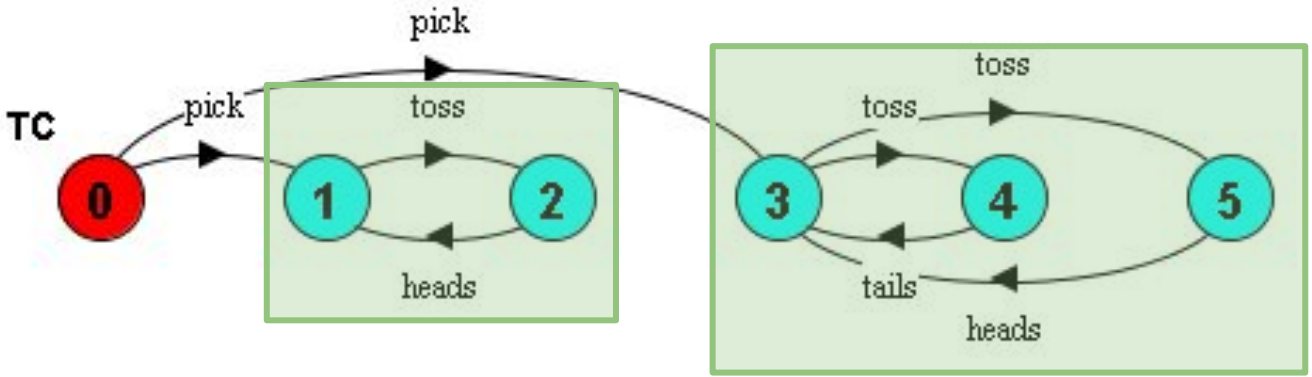
7.3.2 - Vivacidade - Análise de progresso do LTSA

- conjunto terminal
  - cada estado é alcançável de todos os outros estado no conjunto através de uma ou mais transições
  - não há transição de dentro do conjunto para qualquer estado fora do conjunto



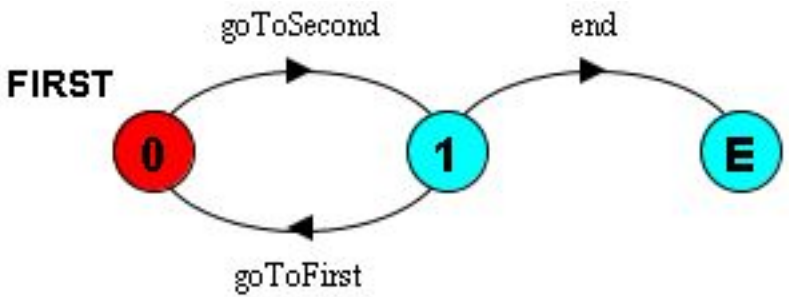
7.3.2 - Vivacidade - Análise de progresso do LTSA

- conjunto terminal
  - cada estado é alcançável de todos os outros estado no conjunto através de uma ou mais transições
  - não há transição de dentro do conjunto para qualquer estado fora do conjunto



7.3.2 - Vivacidade - Análise de progresso do LTSA

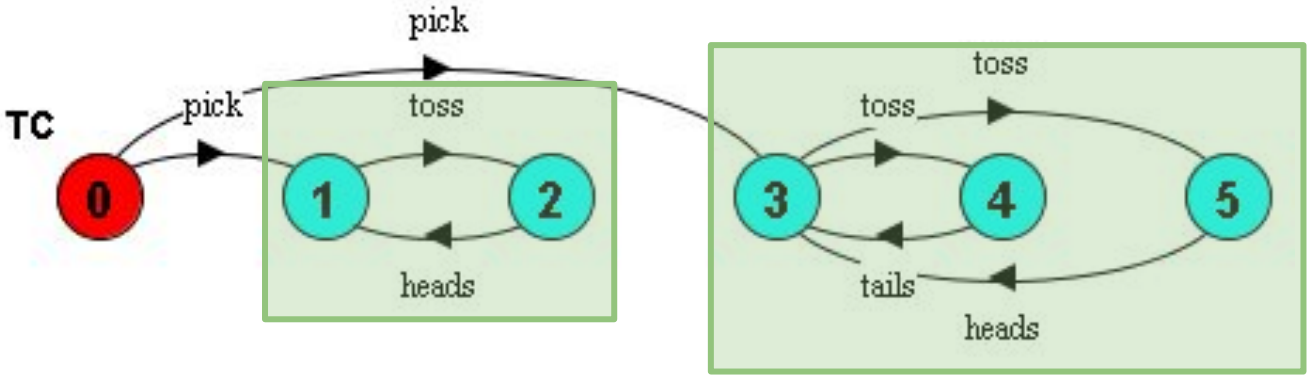
- conjunto terminal
  - cada estado é alcançável de todos os outros estado no conjunto através de uma ou mais transições
  - não há transição de dentro do conjunto para qualquer estado fora do conjunto



Conjunto terminal vazio

7.3.2 - Vivacidade - Análise de progresso do LTSA

- mas por que conjunto terminal é importante?
  - os estados visitados infinitas vezes formam um conjunto terminal
  - considerando uma **escolha justa**, todos os estados de um conjunto terminal são executados infinitas vezes
  - como não existem transições “para fora” de um conjunto terminal, todas as transições que não estiverem em TODOS os conjuntos terminais do sistema não executam infinitas vezes (para todas as execuções do sistema)

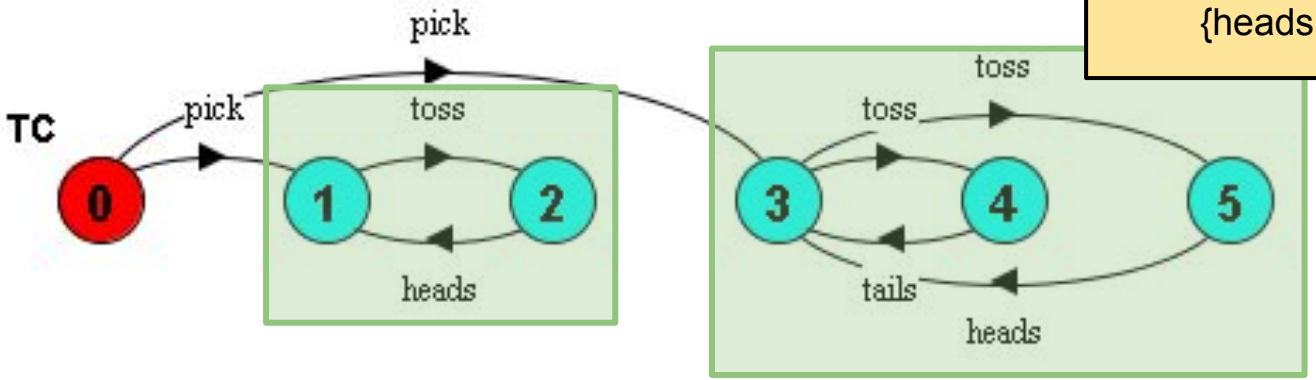


7.3.2 - Vivacidade - Análise de progresso do LTSA

- mas por que conjunto terminal é importante? (cont.)
  - logo, verificar uma **progress property** é simplesmente verificar se em cada conjunto terminal existe pelo menos uma ação de cada **progress property**

```
progress HEADS = {heads}
progress TAILS = {tails}
```

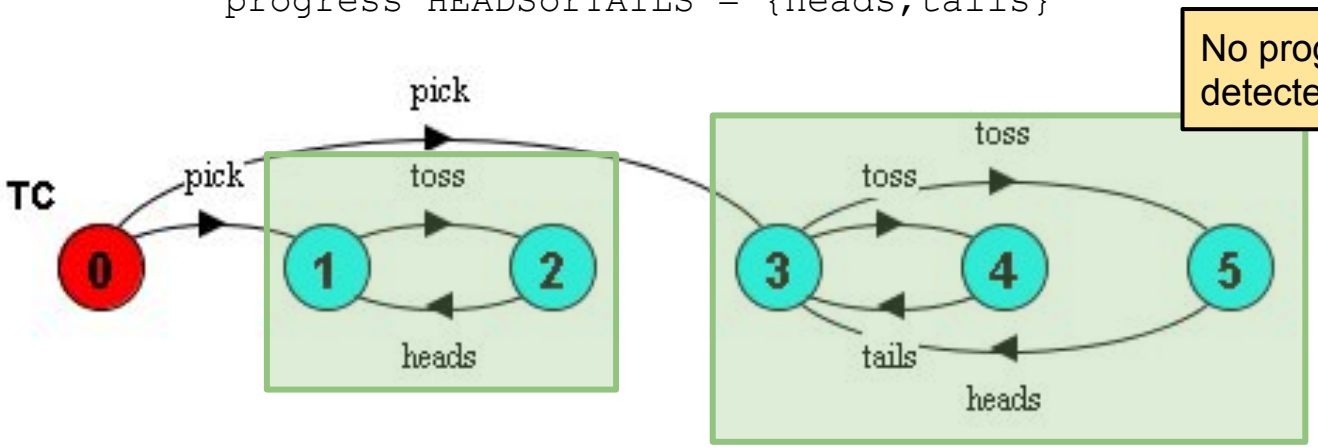
Progress violation: **TAILS**
Trace to terminal set of states:
pick
Actions in **terminal set**:
{heads, toss}



7.3.2 - Vivacidade - Análise de progresso do LTSA

- mas por que conjunto terminal é importante? (cont.)
  - logo, verificar uma **progress property** é simplesmente verificar se em cada conjunto terminal existe pelo menos uma ação de cada **progress property**

```
progress HEADSorTAILS = {heads,tails}
```



7.3.2 - Vivacidade - Análise de progresso do LTSA

- e se nenhuma propriedade de progresso for especificada?
  - o LTSA executará a análise de progresso usando uma propriedade padrão.
  - essa propriedade afirma que para cada ação no alfabeto do sistema de destino, dada a escolha justa, essa ação será executada infinitas vezes

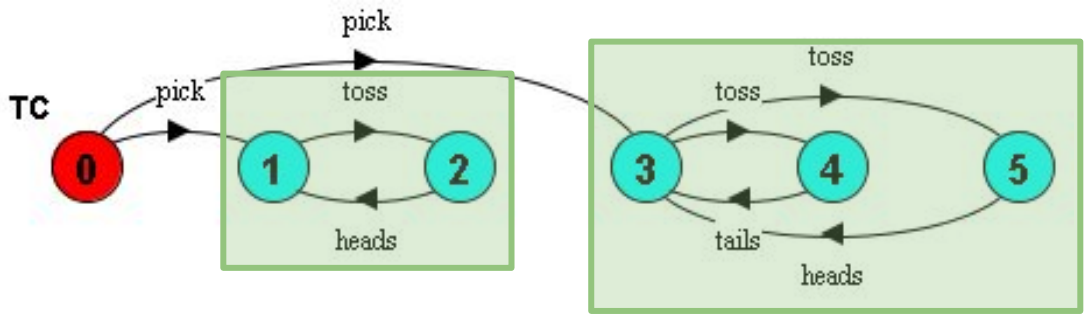
7.3.2 - Vivacidade - Análise de progresso do LTSA

- e se nenhuma propriedade de progresso for especificada? (cont.)
  - em outras palavras, é como se definíssemos uma **progress property** para todas as ações do alfabeto

```
progress HEADS = {heads}
progress TAILS = {tails}
progress PICKS = {pick}
```

7.3.2 - Vivacidade - Análise de progresso do LTSA

- e se nenhuma propriedade de progresso for especificada? (cont.)



Progress violation for actions:
{pick, tails}
Trace to terminal set of states:
pick
Cycle in terminal set:
toss
heads
Actions in terminal set:
{heads, toss}

7.3.3 - Vivacidade - Prioridade de ação

- pode-se definir prioridades de ações
  - **high** priority (<<)

    `||C = P << {a1,...,an}` especifica uma composição em que as ações `a1,...,an` possuem uma prioridade maior que qualquer outra ação no alfabeto de P

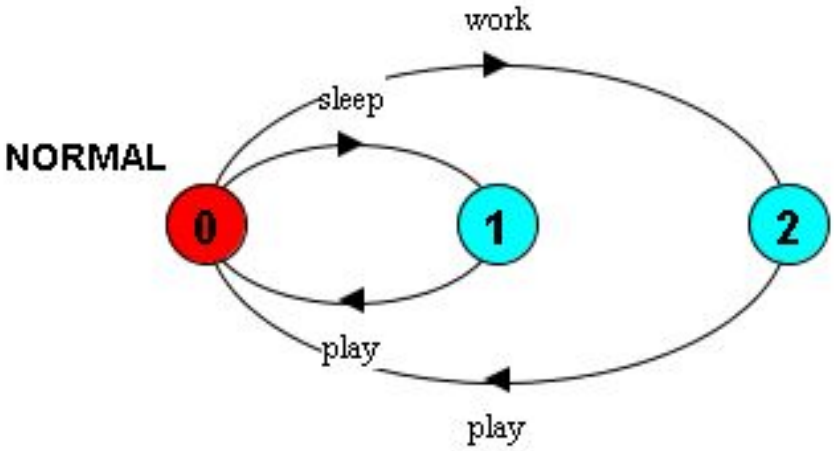
    em qualquer escolha que possua ações `a1,...,an`, todas as outras são descartadas

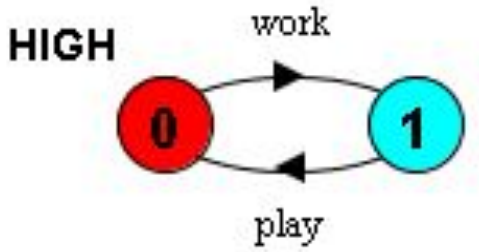
  - **low** priority (>>)

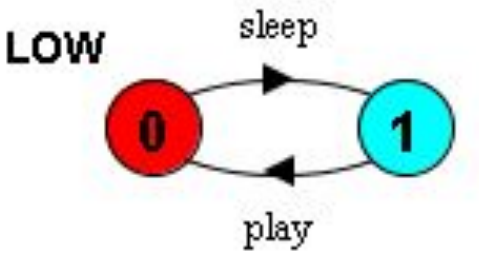
    em qualquer escolha que possua ações `a1,...,an`, as ações `a1,...,an` são descartadas

7.3.3 - Vivacidade - Prioridade de ação

```
NORMAL = (work->play->NORMAL | sleep->play->NORMAL).
||HIGH = (NORMAL)<<{work}.
||LOW  = (NORMAL)>>{work}.
```

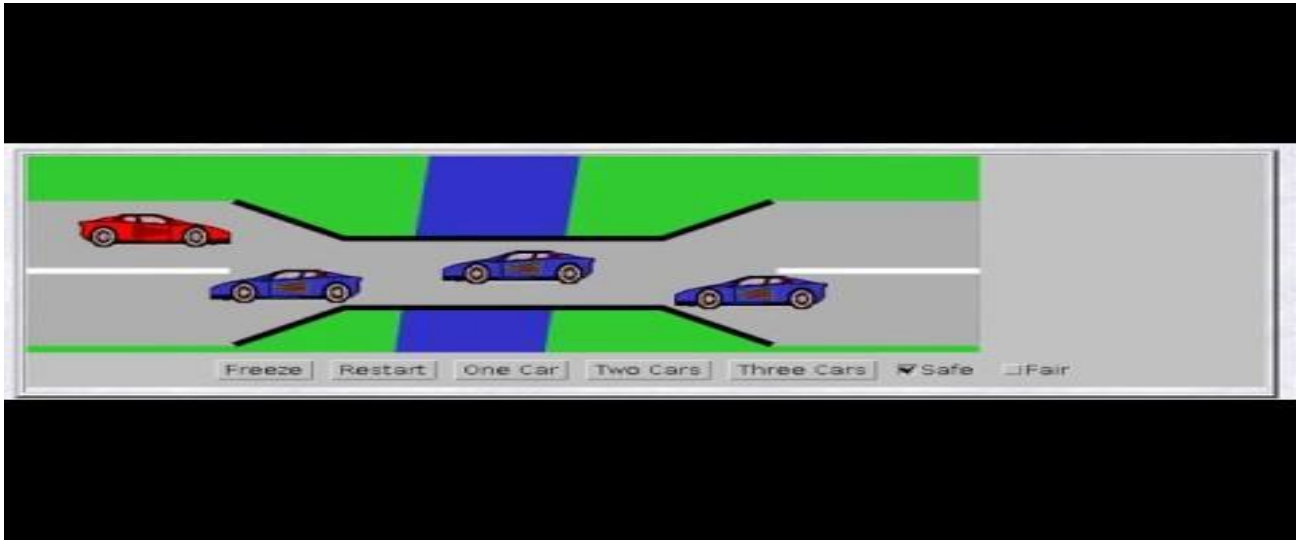






7.4 Vivacidade do modelo Single-Lane Bridge

- mas todo carro acaba tendo a oportunidade de cruzar a ponte?
- como pode ser visto no applet, não



7.4 Vivacidade do modelo Single-Lane Bridge

- no entanto, se a análise de progresso padrão for aplicada ao modelo, nenhuma violação será detectada. **Por que?**

Porque assumimos um escalonamento justo.

- solução em dividia em duas etapas:
  - aproximar o modelo da implementação, fazendo-o, assim, apresentar as violações de progresso
  - solucionar o problema de starvation de forma a poder usar a solução na implementação

7.4 Vivacidade do modelo Single-Lane Bridge - Fazer o modelo falhar

- aproximar o modelo da implementação
- podemos usar **propriedades de progresso** e **prioridades de ações**
  - propriedades de progresso
    - ```
      progress BLUECROSS = {blue[ID].enter}
      progress REDCROSS = {red[ID].enter}
      ```
    - essas prioridades verificam se os carros azuis (e vermelhos) eventualmente entrarão na ponte
    - mas ainda não temos nenhuma violação de prioridade, pois os carros sempre conseguem entrar
      ```

7.4 Vivacidade do modelo Single-Lane Bridge - Fazer o modelo falhar

- aproximar o modelo da implementação (cont.)
  - prioridades de ações
    - entrada
      - se dermos prioridade para os carros azuis entrarem na ponte, os carros vermelhos nunca entrarão (violação da `REDCROSS)` e vice-versa
    - saída
      - diminuir a prioridade da saída de carros
      - toda vez que ocorrer uma escolha entre uma ação de entrada na ponte e de saída, sempre será escolhida a de entrada
      - o objetivo é modelar uma ponte congestionada

```
||CongestedBridge = (SingleLaneBridge)
                     >>{red[ID].exit, blue[ID].exit}.
```

7.4 Vivacidade do modelo Single-Lane Bridge - Fazer o modelo falhar

Progress violation: BLUECROSS

Path to terminal set of states:
red.1.enter
red.2.enter

Actions in terminal set:
{red.1.enter, red.1.exit, red.2.enter, red.2.exit, red.3.enter, red.3.exit}

Progress violation: REDCROSS

Path to terminal set of states:
blue.1.enter
blue.2.enter

Actions in terminal set:
{blue.1.enter, blue.1.exit, blue.2.enter, blue.2.exit, blue.3.enter, blue.3.exit}

7.4 Vivacidade do modelo Single-Lane Bridge - Solução

- a ponte passa a decidir dinamicamente, e a qualquer momento, quando admitir carros azuis e quando admitir carros vermelhos
- ela precisa saber quantos carros estão esperando para entrar

- modificações no modelo
  - do carro

```
CAR = (request->enter->exit->CAR)
```

7.4 Vivacidade do modelo Single-Lane Bridge - Solução

- modificações no modelo (cont.)
  - da ponte

```
BRIDGE = BRIDGE[0][0][0][0],
BRIDGE[nr:T][nb:T][wr:T][wb:T] =
      (red[ID].request -> BRIDGE[nr][nb][wr+1][
      |when (nb==0 && wb==0)
            red[ID].enter   -> BRIDGE[nr+1][nb]
        |red[ID].exit       -> BRIDGE[nr-1][nb]
        |blue[ID].request -> BRIDGE[nr][nb][w
        |when (nr==0 && wr==0)
           blue[ID].enter -> BRIDGE[nr][nb+1]
        |blue[ID].exit     -> BRIDGE[nr][nb-1]
      ).
```

Trace to DEADLOCK:
red.1.request
red.2.request
red.3.request
blue.1.request
blue.2.request
blue.3.request

7.4 Vivacidade do modelo Single-Lane Bridge - Solução

- utilizar uma variável **turn** qual cor de carro deve entrar
- variável booleana, **bt** (blue turn)
  - inicia como **true**
  - quando um carro azul sai da ponte, ela recebe **false**
  - quando um carro vermelho sai, ela recebe **true** novamente

7.4 Vivacidade do modelo Single-Lane Bridge - Solução

```
BRIDGE = BRIDGE[0][0][0][0][True],

BRIDGE[nr:T][nb:T][wr:T][wb:T][bt:B] =
    (red[ID].request  -> BRIDGE[nr][nb][wr+1][wb][bt]
    |when (nb==0 && (wb==0 || !bt))
       red[ID].enter  -> BRIDGE[nr+1][nb][wr-1][wb][bt]
    |red[ID].exit     -> BRIDGE[nr-1][nb][wr][wb][True]
    |blue[ID].request->BRIDGE[nr][nb][wr][wb+1][bt]
    |when (nr==0 && (wr==0 || bt))
       blue[ID].enter -> BRIDGE[nr][nb+1][wr][wb-1][bt]
    |blue[ID].exit    -> BRIDGE[nr][nb-1][wr][wb][False]
).
```

7.5 Leitores e escritores

- acesso a uma região compartilhada (bd, por exemplo) por dois tipos de processos:
  - leitores
  - escritores

7.5.1 Leitores e escritores - Modelo

```
set Actions = {acquireRead,releaseRead,acquireWrite,releaseWrite}

READER = (acquireRead->examine->releaseRead->READER) + Actions \ {examine}.

WRITER = (acquireWrite->modify->releaseWrite->WRITER) + Actions \ {modify}.
```

- a extensão de alfabeto é usada para garantir que as outras ações de acesso não ocorram livremente para qualquer instância prefixada do processo.
- as ações **examine** e **modify** foram ocultas pois não são relevantes para a sincronização de acesso

7.5.1 Leitores e escritores - Modelo

```
const False = 0 const True = 1
range Bool = False..True
const Nread = 2  //máximo de leitores
Nwrite= 2        //máximo de escritores

RW_LOCK = RW[0][False],
RW[readers:0..Nread][writing:Bool] =
     (when (!writing)
          acquireRead -> RW[readers+1][writing]
     |releaseRead -> RW[readers-1][writing]
     |when (readers==0 && !writing)
          acquireWrite -> RW[readers][True]
     |releaseWrite -> RW[readers][False]
     ).
```

O RW_LOCK conta a quantidade de leitores “ativos”.

Usa, também, um booleano para indicar se há um escritor na região crítica.

7.5.1 Leitores e escritores - Segurança

- Para verificar se o bloqueio se comporta como desejado, definimos uma propriedade de segurança, RW_SAFE

```
property SAFE_RW
     = (acquireRead -> READING[1]
        |acquireWrite -> WRITING
        ),

READING[i:1..Nread]
     = (acquireRead -> READING[i+1]
       |when(i>1) releaseRead -> READING[i-1]
       |when(i==1) releaseRead -> SAFE_RW
       ),

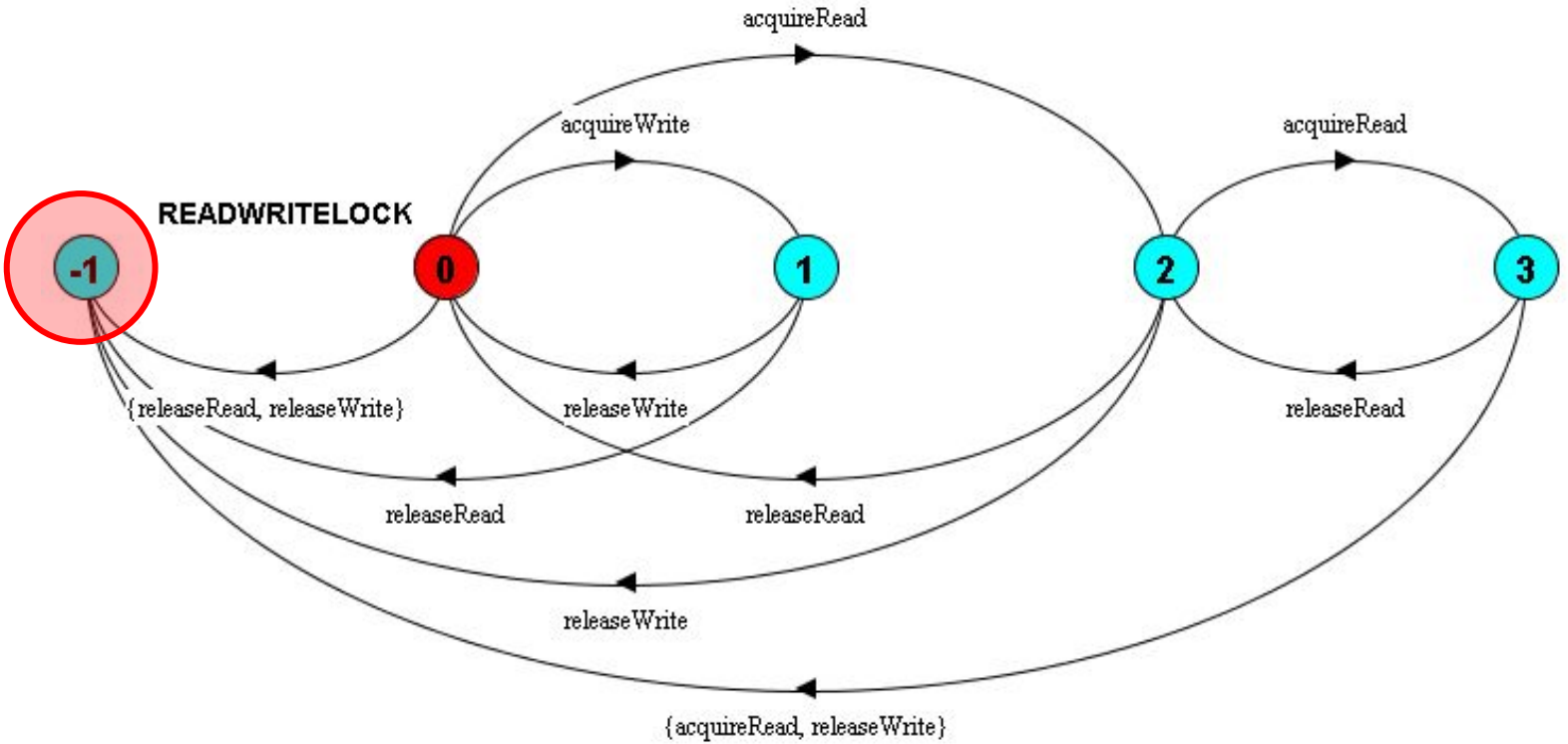
WRITING = (releaseWrite -> SAFE_RW).
```

7.5.1 Leitores e escritores - Segurança

- Para verificar se a implementação de bloqueio RW_LOCK satisfaz a propriedade, o bloqueio é composto com a propriedade da seguinte forma:

```
||READWRITELOCK = (RW_LOCK || SAFE_RW).
```

7.5.1 Leitores e escritores - Segurança



7.5.2 Leitores e escritores - Progresso

```
progress WRITE = {writer[1..Nwrite].acquireWrite}
progress READ  = {reader[1..Nread].acquireRead}
```

- WRITE
  - eventualmente, um escritor irá executar **acquireWrite**
- READ
  - eventualmente, algum leitor irá executar **acquireRead**

7.5.2 Leitores e escritores - Progresso

- Nenhuma violação de progresso detectada
  - por conta da escolha justa
- Passo a passo
  - fazer o modelo falhar
  - solucionar

7.5.2 Leitores e escritores - Fazer o modelo falhar

- Condições adversas usando prioridade de ação
  - diminuir a prioridade das ações de **release** para os leitores e escritores

```
||RW_PROGRESS = READERS_WRITERS
                        >>{reader[1..Nread].releaseRead,
                            writer[1..Nwrite].releaseWrite}.
```

7.5.2 Leitores e escritores - Fazer o modelo falhar

Progress violation: WRITE

Path to terminal set of states:
reader.1.acquireRead

Actions in terminal set:
{reader.1.acquireRead, reader.1.releaseRead,
reader.2.acquireRead, reader.2.releaseRead}

O número de leitores nunca diminui para 0

7.5.2 Leitores e escritores - Solucionar

- **leitores** são proibidos de acessar de acessar a região crítica se algum **escritor** está esperando para acessar
- para detectar se algum **escritor** está esperando

```
set Actions = {acquireRead,releaseRead,acquireWrite,
releaseWrite,requestWrite}

WRITER =(requestWrite->acquireWrite->modify
                     ->releaseWrite->WRITER
        )+Actions\{modify}.
```

7.5.2 Leitores e escritores - Solucionar

- o processo do escritor (READER) não foi modificado
- o RW_LOCK foi modificado para contar a quantidade de **escritores** esperando

```
RW_LOCK = RW[0][False][0],
RW[readers:0..Nread][writing:Bool][waitingW:0..Nwrite] =
     (when (!writing && waitingW==0)
          acquireRead -> RW[readers+1][writing][waitingW]
     |releaseRead -> RW[readers-1][writing][waitingW]
     |when (readers==0 && !writing)
          acquireWrite-> RW[readers][True][waitingW-1]
     |releaseWrite-> RW[readers][False][waitingW]
     |requestWrite-> RW[readers][writing][waitingW+1]
     ).
```

7.5.2 Leitores e escritores - Solucionar

- Sem deadlocks
- Sem erros
- Starvation dos leitores
  - se sempre existir um escritor aguardando

Progress violation: READ

Path to terminal set of states:
writer.1.requestWrite
writer.2.requestWrite

Actions in terminal set:
{writer.1.requestWrite, writer.1.acquireWrite, writer.1.releaseWrite,
writer.2.requestWrite, writer.2.acquireWrite, writer.2.releaseWrite}

Para satisfazer as propriedades de READ e WRITE, poderia ser utilizada uma variável **turn** (da mesma forma utilizada no problema das pontes).

Referências e links

- Ambiente de execução dos applets
  - firefox 40.0
    - http://ftp.mozilla.org/pub/firefox/releases/40.0/linux-x86_64/pt-BR/
  - plugin icedtea-plugin
  - How to install the Java plugin for Firefox?
    - https://askubuntu.com/questions/354361/how-to-install-the-java-plugin-for-firefox/354406
  - Install an older version of Firefox
    - https://support.mozilla.org/en-US/kb/install-older-version-of-firefox
- Imagem dos trens
  - https://pixabay.com/pt/trem-ferrovi%C3%A1ria-s-bahn-transportes-797072/