Markov Regenerative Process - steady-state analysis

MVT exam

Stefano Martina
stefano.martina@stud.unifi.it

Tommaso Papini
tommaso.papini1@stud.unifi.it

UNIVERSITÀ DEGLI STUDI FIRENZE

November 27, 2015

Markov Regenerative Processes (MRPs)

Definition

A Markov Regenerative Process (MRP) is a stochastic process that sooner or later, with probability one, will reach a regenerative state (will be regenerated).

Regenerative state

A state where the process loses its memory.

Markov Regenerative Processes (MRPs)

Definition

A Markov Regenerative Process (MRP) is a stochastic process that sooner or later, with probability one, will reach a regenerative state (will be regenerated).

Regenerative state

A state where the process loses its memory.

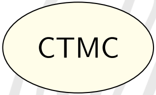
Markov Regenerative Processes (MRPs)

Definition

A Markov Regenerative Process (MRP) is a stochastic process that sooner or later, with probability one, will reach a regenerative state (will be regenerated).

Regenerative state

A state where the process loses its memory.



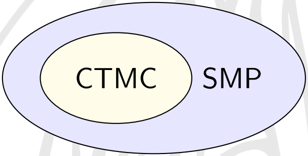
Markov Regenerative Processes (MRPs)

Definition

A Markov Regenerative Process (MRP) is a stochastic process that sooner or later, with probability one, will reach a regenerative state (will be regenerated).

Regenerative state

A state where the process loses its memory.



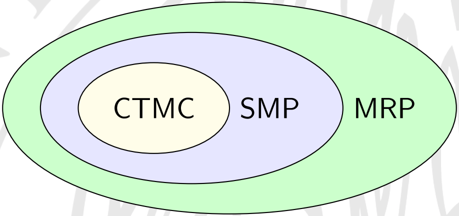
Markov Regenerative Processes (MRPs)

Definition

A Markov Regenerative Process (MRP) is a stochastic process that sooner or later, with probability one, will reach a regenerative state (will be regenerated).

Regenerative state

A state where the process loses its memory.



The steady-state problem

Transient probabilities

The probability distribution that the process will be in a certain state, after given t time.

Steady-state

For ergodic systems, it represents the probability distribution that the process will be in a certain state, as time goes to infinity.

- ✓ ORIS current state:
  - Transient analysis for Markov Regenerative Processes (MRPs)
  - Steady-state analysis for Continuous Time Markov Processes (CTMCs)
- ✓ Until now! ☺
- ✓ Warning: we assume that the MRP is ergodic!

The steady-state problem

Transient probabilities

The probability distribution that the process will be in a certain state, after given t time.

Steady-state

For ergodic systems, it represents the probability distribution that the process will be in a certain state, as time goes to infinity.

- ✓ ORIS current state:
  - Transient analysis for Markov Regenerative Processes (MRPs)
  - Steady-state analysis for Continuous Time Markov Processes (CTMCs)
- ✓ Until now! ☺
- ✓ Warning: we assume that the MRP is ergodic!

The steady-state problem

Transient probabilities

The probability distribution that the process will be in a certain state, after given t time.

Steady-state

For ergodic systems, it represents the probability distribution that the process will be in a certain state, as time goes to infinity.

- ✓ ORIS current state:
  - Transient analysis for Markov Regenerative Processes (MRPs)
  - Steady-state analysis for Continuous Time Markov Processes (CTMCs)
- ✓ Until now! ☺
- ✓ Warning: we assume that the MRP is ergodic!

The steady-state problem

Transient probabilities

The probability distribution that the process will be in a certain state, after given t time.

Steady-state

For ergodic systems, it represents the probability distribution that the process will be in a certain state, as time goes to infinity.

- ✓ ORIS current state:
  - Transient analysis for Markov Regenerative Processes (MRPs)
  - Steady-state analysis for Continuous Time Markov Processes (CTMCs)
- ✓ Until now! ☺
- ✓ Warning: we assume that the MRP is ergodic!

The steady-state problem

Transient probabilities

The probability distribution that the process will be in a certain state, after given t time.

Steady-state

For ergodic systems, it represents the probability distribution that the process will be in a certain state, as time goes to infinity.

- ✓ ORIS current state:
  - Transient analysis for Markov Regenerative Processes (MRPs)
  - Steady-state analysis for Continuous Time Markov Processes (CTMCs)
- ✓ Until now! $\ddot{\smile}$
- ✓ Warning: we assume that the MRP is ergodic!

The steady-state problem

Transient probabilities

The probability distribution that the process will be in a certain state, after given t time.

Steady-state

For ergodic systems, it represents the probability distribution that the process will be in a certain state, as time goes to infinity.

- ✓ ORIS current state:
  - Transient analysis for Markov Regenerative Processes (MRPs)
  - Steady-state analysis for Continuous Time Markov Processes (CTMCs)
- ✓ Until now! $\ddot\smile$
- ✓ Warning: we assume that the MRP is ergodic!

MRP steady-state analysis - The theory

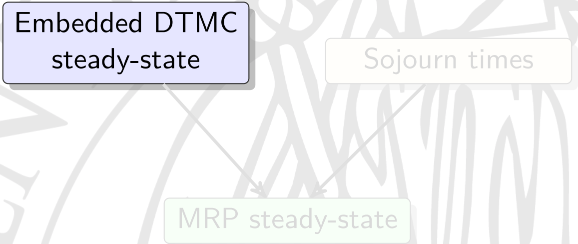
General idea:

1. Calculate the embedded DTMC steady-state on the regenerative states
2. Calculate the expected sojourn time in each marking, after reaching a regenerative state
3. Combine the two above in order to calculate the MRP steady-state

MRP steady-state analysis - The theory

General idea:

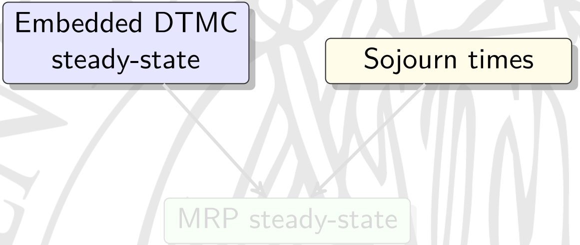
1. Calculate the embedded DTMC steady-state on the regenerative states
2. Calculate the expected sojourn time in each marking, after reaching a regenerative state
3. Combine the two above in order to calculate the MRP steady-state



MRP steady-state analysis - The theory

General idea:

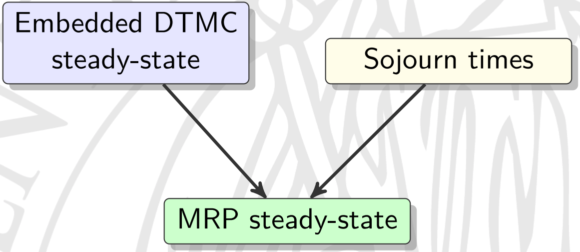
1. Calculate the embedded DTMC steady-state on the regenerative states
2. Calculate the expected sojourn time in each marking, after reaching a regenerative state
3. Combine the two above in order to calculate the MRP steady-state



MRP steady-state analysis - The theory

General idea:

1. Calculate the embedded DTMC steady-state on the regenerative states
2. Calculate the expected sojourn time in each marking, after reaching a regenerative state
3. Combine the two above in order to calculate the MRP steady-state



Main classes implemented

✓ `class EmbeddedDTMC`
  - written from scratch
  - calculate embedded DTMC steady-state

✓ `class RegenerativeSteadyStateAnalysis`
  - based on `class RegenerativeTransientAnalysis`
  - calculate MRP steady-state

Steady-state of the embedded DTMC on regenerative states

Steady-state in Discrete Time Markov Process (DTMC)

If the steady-state of a Discrete Time Markov Process (DTMC) exists and is unique (if is ergodic), then it's calculated by solving for v the linear system:

$$\begin{cases} v = vP \\ |v| = 1 \end{cases}$$

- ✓ We want to calculate the steady-state of the embedded DTMC of the MRP in the regenerative states
- ✓ But we don't have P! ☹

Steady-state of the embedded DTMC on regenerative states

Steady-state in Discrete Time Markov Process (DTMC)

If the steady-state of a Discrete Time Markov Process (DTMC) exists and is unique (if is ergodic), then it's calculated by solving for v the linear system:

$$\begin{cases} v = vP \\ |v| = 1 \end{cases}$$

- ✓ We want to calculate the steady-state of the embedded DTMC of the MRP in the regenerative states
- ✓ But we don't have P! ☹

Steady-state of the embedded DTMC on regenerative states

Steady-state in Discrete Time Markov Process (DTMC)

If the steady-state of a Discrete Time Markov Process (DTMC) exists and is unique (if is ergodic), then it's calculated by solving for v the linear system:

$$\begin{cases} v = vP \\ |v| = 1 \end{cases}$$

- ✓ We want to calculate the steady-state of the embedded DTMC of the MRP in the regenerative states
- ✓ But we don't have P! $\frown$

Reaching probability feature

- ✓ We add a new reaching probability feature to each state: `class ReachingProbabilityFeature`
- ✓ Inside `SteadyStateInitialStateBuilder`: set it to 1
- ✓ Inside `SteadyStatePostProcessor`: multiply the parent's reaching probability by the probability to chose a certain child

If we run a transient analysis on the Petri Net (PN) we get:

- ✓ `regenerationClasses`
- ✓ `Map<DeterministicEnablingState,Map<DeterministicEnablingState,Set<State>>>`
- ✓ sum reaching probability feature of each `State` to compute elements of P

Now we can solve the linear system:

$$\begin{cases} v = vP \\ |v| = 1 \end{cases} = \begin{cases} (P' - I)v' = 0 \\ \sum_i v_i = 1 \end{cases}$$

- ✓ `RealMatrix` & `RealVector`
- ✓ QR decomposition solver
  - `DecompositionSolver solver = new QRDecomposition(coefficients).getSolver();`
  - `RealVector steadyState = solver.solve(constants);`
- ✓ Convert `steadyState` into a `Map<DeterministicEnablingState,BigDecimal>`

Sojourn time a_{ij}

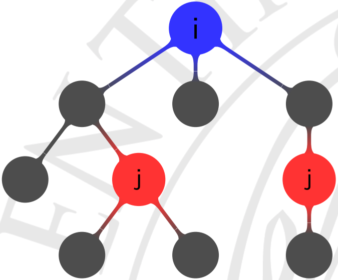
Definition

The sojourn time a_{ij} represents the average time spent in the j-th marking after the (last) i-th regeneration.

How to compute a_{ij}?

a_{ij} is:

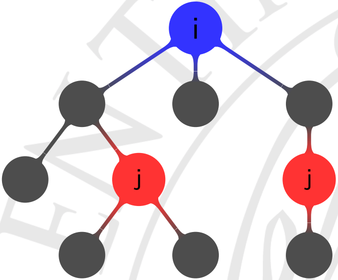
- ✓ sum of avg time spent in marking j occurrences
  - sum of avg times before each variable fires weighted by the probability of chosing that variable
    - ⋆ condition each variable to be the minimum (i.e. the one that fires)
    - ⋆ compute avg time before that variable fires (thanks Marco!)

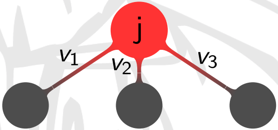


How to compute a_{ij}?

a_{ij} is:

- ✓ sum of avg time spent in marking j occurrences
  - sum of avg times before each variable fires weighted by the probability of chosing that variable
    - ⋆ condition each variable to be the minimum (i.e. the one that fires)
    - ⋆ compute avg time before that variable fires (thanks Marco!)

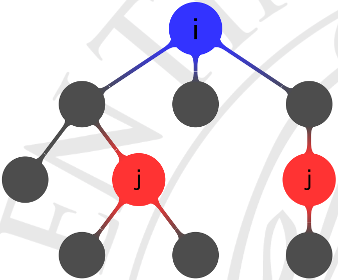


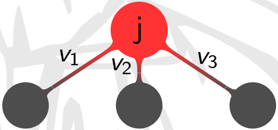


How to compute a_{ij}?

a_{ij} is:

- ✓ sum of avg time spent in marking j occurrences
  - sum of avg times before each variable fires weighted by the probability of chosing that variable
    - ⋆ condition each variable to be the minimum (i.e. the one that fires)
    - ⋆ compute avg time before that variable fires (thanks Marco!)

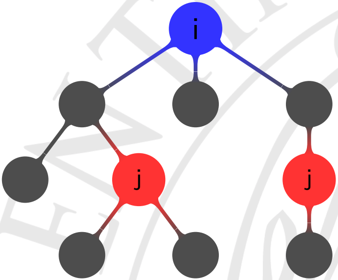


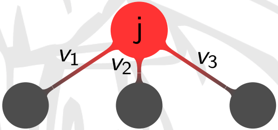


How to compute a_{ij}?

a_{ij} is:

- ✓ sum of avg time spent in marking j occurrences
  - sum of avg times before each variable fires weighted by the probability of chosing that variable
    - ⋆ condition each variable to be the minimum (i.e. the one that fires)
    - ⋆ compute avg time before that variable fires (thanks Marco!)





When to compute a_{ij}?

During the transient analysis!

- ✓ transient analysis generates succession trees for each regenerative state
  - regenerative state as root
  - following regenerative states as leaves
  - reachable markings as inner nodes
- ✓ during the tree generation compute and accumulate a_{ij} for each marking occurrence found

When to compute a_{ij}?

During the transient analysis!

✓ transient analysis generates succession trees for each regenerative state
  - regenerative state as root
  - following regenerative states as leaves
  - reachable markings as inner nodes

✓ during the tree generation compute and accumulate a_{ij} for each marking occurrence found

When to compute a_{ij}?

During the transient analysis!

- ✓ transient analysis generates succession trees for each regenerative state
  - regenerative state as root
  - following regenerative states as leaves
  - reachable markings as inner nodes
- ✓ during the tree generation compute and accumulate a_{ij} for each marking occurrence found

When to compute a_{ij}?

During the transient analysis!

- ✓ transient analysis generates succession trees for each regenerative state
  - regenerative state as root
  - following regenerative states as leaves
  - reachable markings as inner nodes
- ✓ during the tree generation compute and accumulate a_{ij} for each marking occurrence found

Markov Regenerative Process (MRP) steady-state

Let's combine the embedded DTMC steady-state and the sojourn times!

$$\pi_j = \frac{\sum_i v_i a_{ij}}{K}$$

- ✓ We multiply the sojourn time in the marking j after the regeneration i by the probability of reaching the i-th regeneration
- ✓ We do this for each regeneration that leads to the marking j before another regeneration
- ✓ K is a normalization factor calculated as the sum of π_j

Markov Regenerative Process (MRP) steady-state

Let's combine the embedded DTMC steady-state and the sojourn times!

$$\pi_j = \frac{\sum_i v_i a_{ij}}{K}$$

- ✓ We multiply the sojourn time in the marking j after the regeneration i by the probability of reaching the i-th regeneration
- ✓ We do this for each regeneration that leads to the marking j before another regeneration
- ✓ K is a normalization factor calculated as the sum of π_j

Markov Regenerative Process (MRP) steady-state

Let's combine the embedded DTMC steady-state and the sojourn times!

$$\pi_j = \frac{\sum_i v_i a_{ij}}{K}$$

- ✓ We multiply the sojourn time in the marking j after the regeneration i by the probability of reaching the i-th regeneration
- ✓ We do this for each regeneration that leads to the marking j before another regeneration
- ✓ K is a normalization factor calculated as the sum of π_j

Markov Regenerative Process (MRP) steady-state

Let's combine the embedded DTMC steady-state and the sojourn times!

$$\pi_j = \frac{\sum_i v_i a_{ij}}{K}$$

- ✓ We multiply the sojourn time in the marking j after the regeneration i by the probability of reaching the i-th regeneration
- ✓ We do this for each regeneration that leads to the marking j before another regeneration
- ✓ K is a normalization factor calculated as the sum of π_j

Markov Regenerative Process (MRP) steady-state

Let's combine the embedded DTMC steady-state and the sojourn times!

$$\pi_j = \frac{\sum_i v_i a_{ij}}{K}$$

- ✓ We multiply the sojourn time in the marking j after the regeneration i by the probability of reaching the i-th regeneration
- ✓ We do this for each regeneration that leads to the marking j before another regeneration
- ✓ K is a normalization factor calculated as the sum of π_j

Test

Unit test

✓ Class `SteadyStateTest` with JUnit tests

✓ Three different models:

- TestCaseSMP
- TestCase2ParallelTasks
- TestCaseRejuvenation

✓ For each test:

1. launch MRP steady state analysis
2. check if the result is comparable to the expected value (with a tolerance)

Test

Unit test

- ✓ Class `SteadyStateTest` with JUnit tests
- ✓ Three different models:
  - TestCaseSMP
  - TestCase2ParallelTasks
  - TestCaseRejuvenation
- ✓ For each test:
  1. launch MRP steady state analysis
  2. check if the result is comparable to the expected value (with a tolerance)

Test

Unit test

- ✓ Class `SteadyStateTest` with JUnit tests
- ✓ Three different models:
  - TestCaseSMP
  - TestCase2ParallelTasks
  - TestCaseRejuvenation
- ✓ For each test:
  1. launch MRP steady state analysis
  2. check if the result is comparable to the expected value (with a tolerance)

Test

Unit test

- ✓ Class `SteadyStateTest` with JUnit tests
- ✓ Three different models:
  - TestCaseSMP
  - TestCase2ParallelTasks
  - TestCaseRejuvenation
- ✓ For each test:
  1. launch MRP steady state analysis
  2. check if the result is comparable to the expected value (with a tolerance)

Test

Unit test

- ✓ Class `SteadyStateTest` with JUnit tests
- ✓ Three different models:
  - TestCaseSMP
  - TestCase2ParallelTasks
  - TestCaseRejuvenation
- ✓ For each test:
  1. launch MRP steady state analysis
  2. check if the result is comparable to the expected value (with a tolerance)

Test

Unit test

- ✓ Class `SteadyStateTest` with JUnit tests
- ✓ Three different models:
  - TestCaseSMP
  - TestCase2ParallelTasks
  - TestCaseRejuvenation
- ✓ For each test:
  1. launch MRP steady state analysis
  2. check if the result is comparable to the expected value (with a tolerance)

Test

Unit test

- ✓ Class `SteadyStateTest` with JUnit tests
- ✓ Three different models:
  - TestCaseSMP
  - TestCase2ParallelTasks
  - TestCaseRejuvenation
- ✓ For each test:
  1. launch MRP steady state analysis
  2. check if the result is comparable to the expected value (with a tolerance)

Test

Unit test

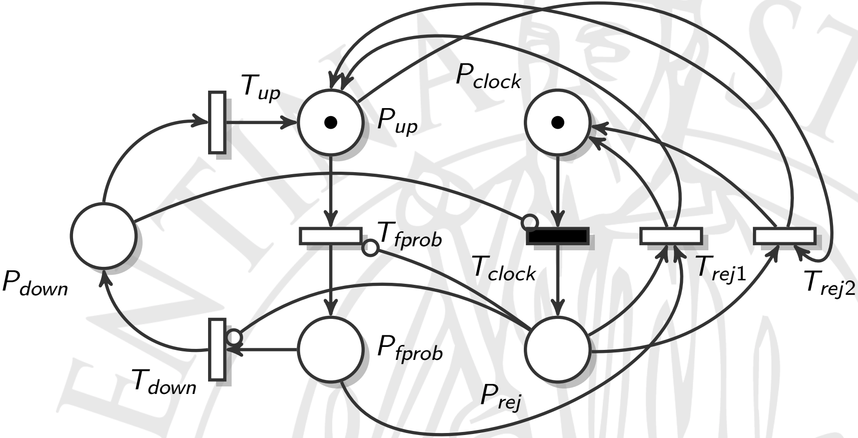
- ✓ Class `SteadyStateTest` with JUnit tests
- ✓ Three different models:
  - TestCaseSMP
  - TestCase2ParallelTasks
  - TestCaseRejuvenation
- ✓ For each test:
  1. launch MRP steady state analysis
  2. check if the result is comparable to the expected value (with a tolerance)

Test

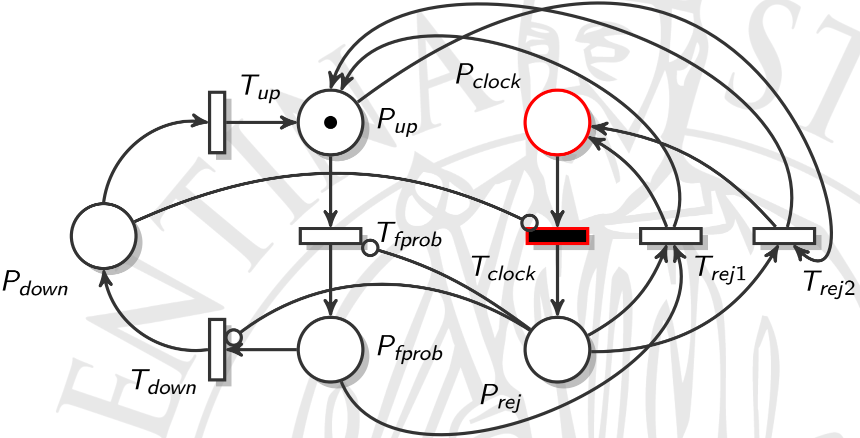
Unit test

- ✓ Class `SteadyStateTest` with JUnit tests
- ✓ Three different models:
  - TestCaseSMP
  - TestCase2ParallelTasks
  - TestCaseRejuvenation
- ✓ For each test:
  1. launch MRP steady state analysis
  2. check if the result is comparable to the expected value (with a tolerance)

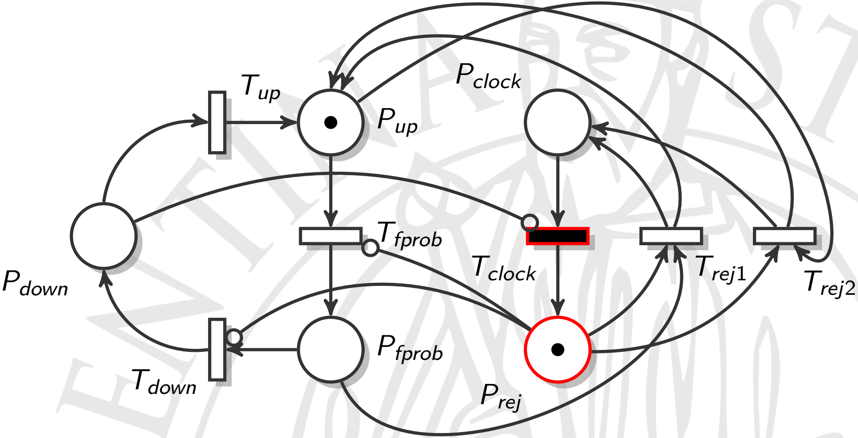
Rejuvenation



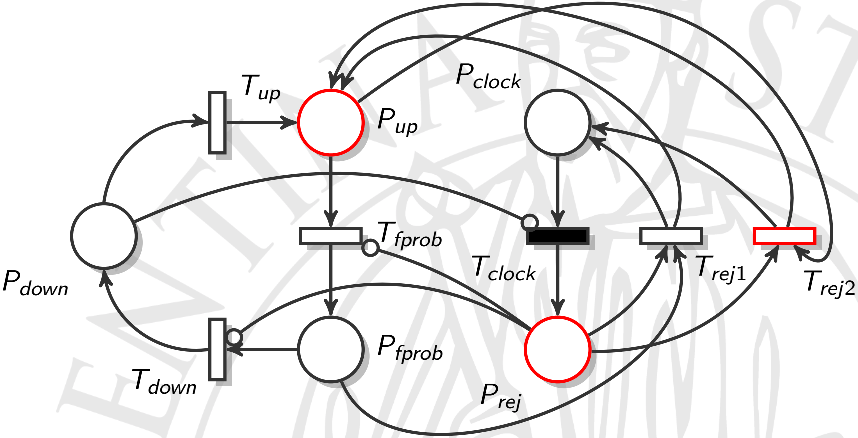
Rejuvenation



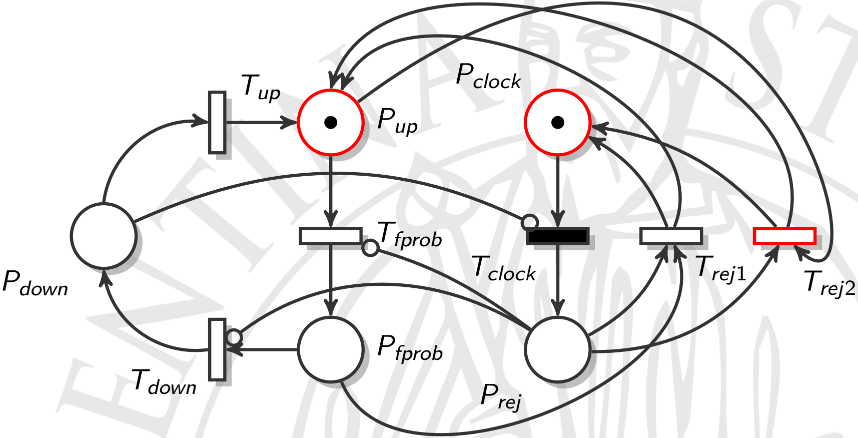
Rejuvenation



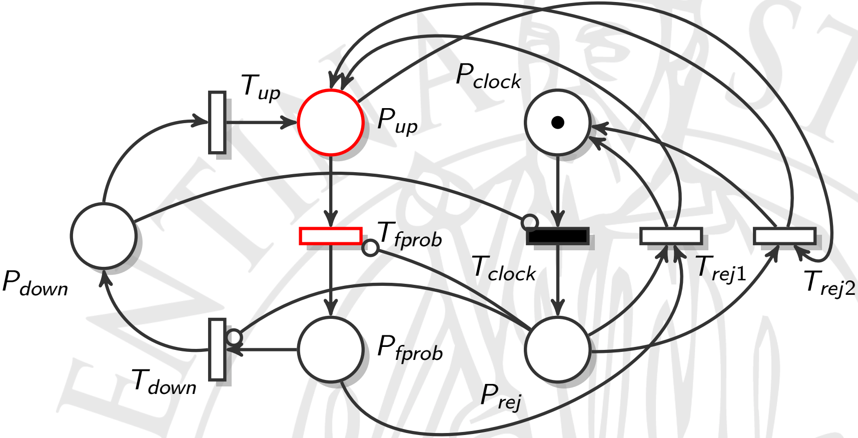
Rejuvenation



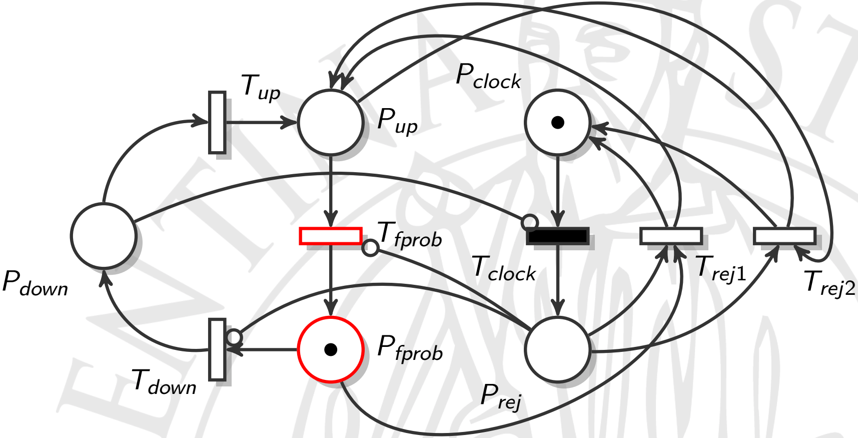
Rejuvenation



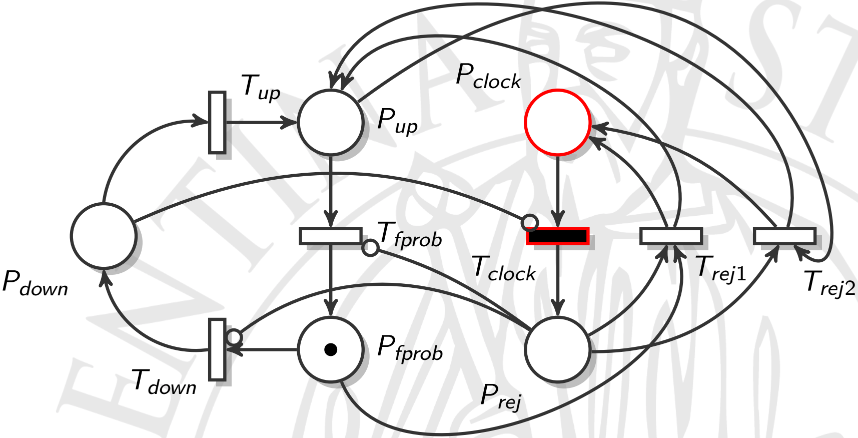
Rejuvenation



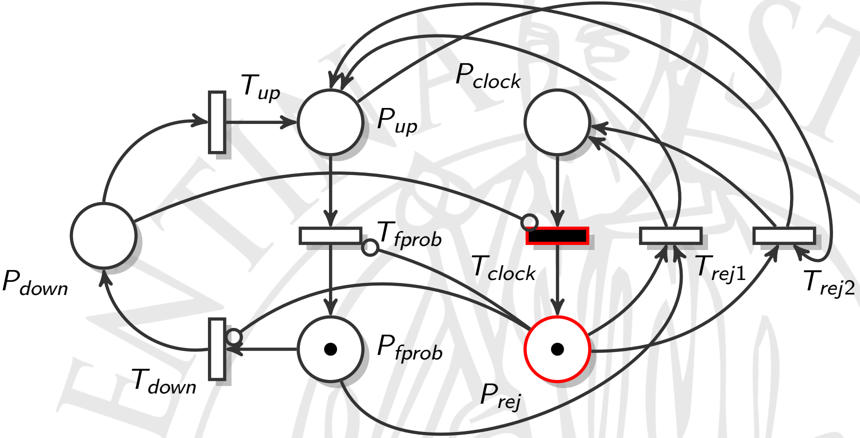
Rejuvenation



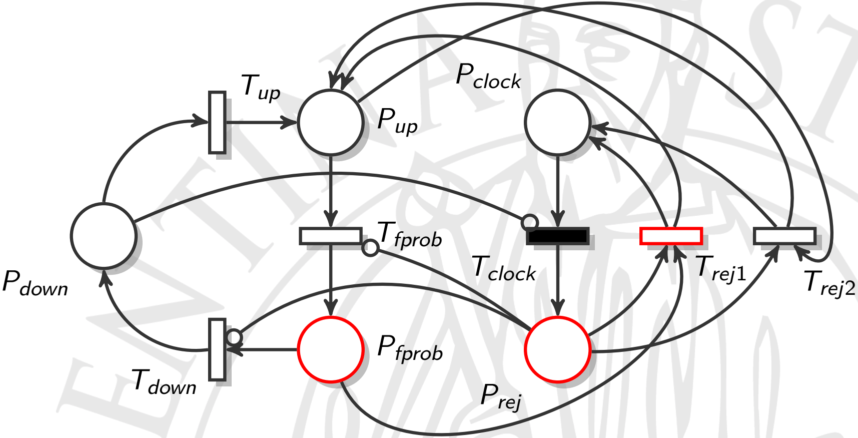
Rejuvenation



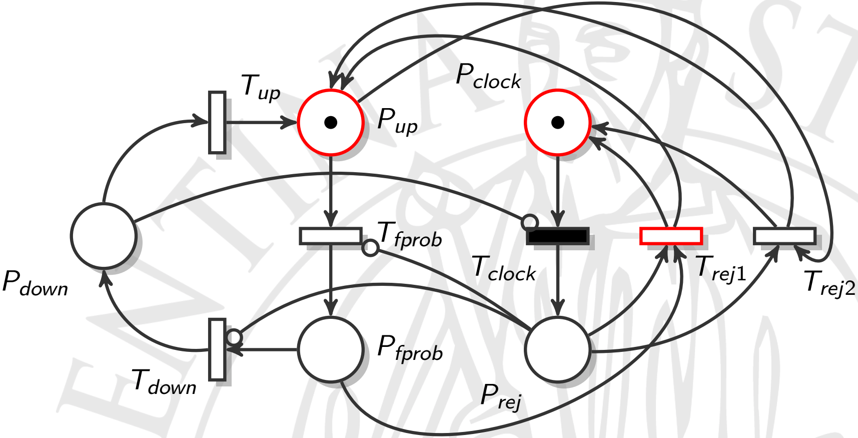
Rejuvenation



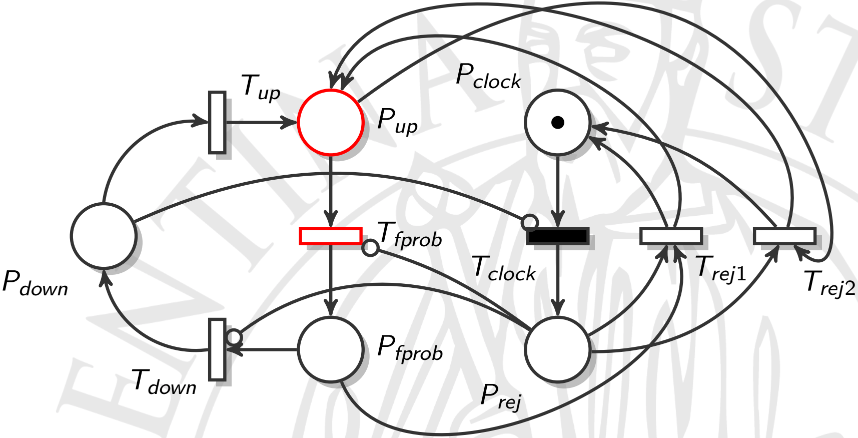
Rejuvenation



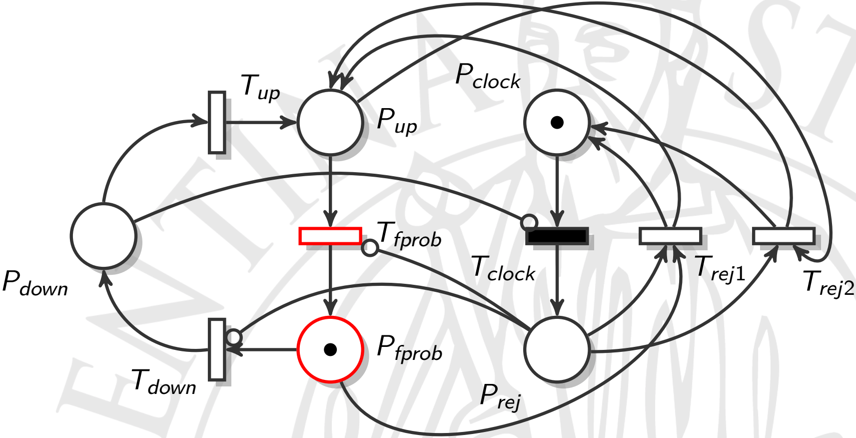
Rejuvenation



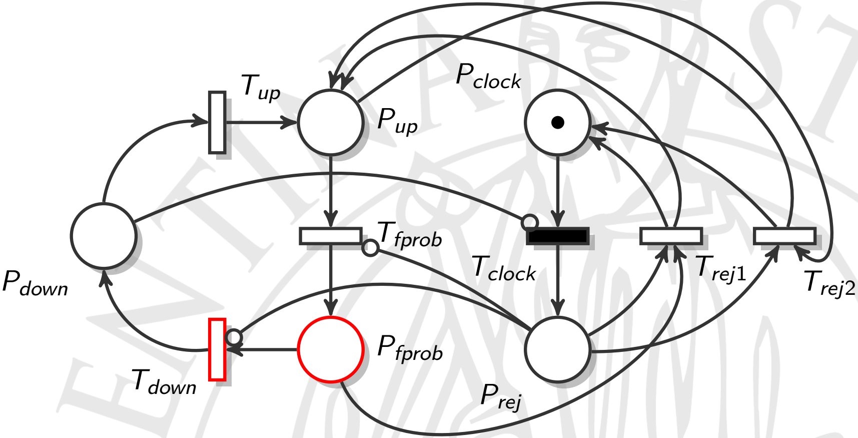
Rejuvenation



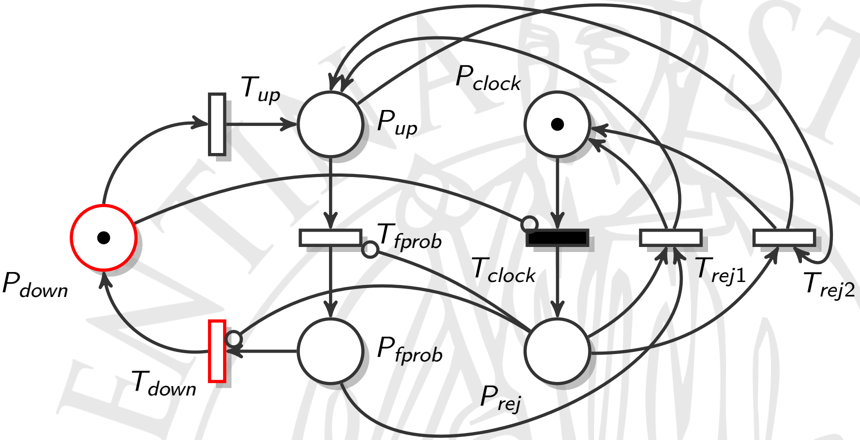
Rejuvenation



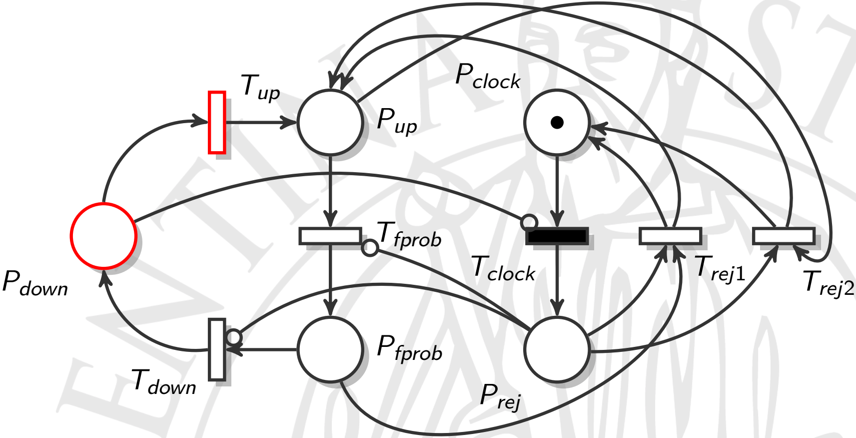
Rejuvenation



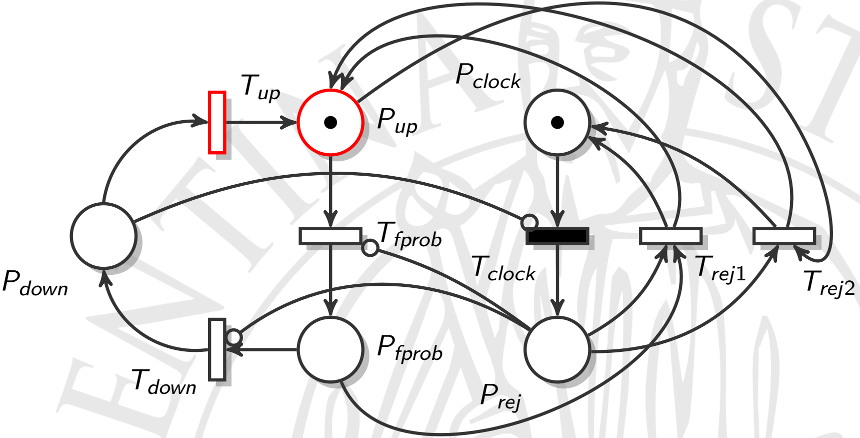
Rejuvenation



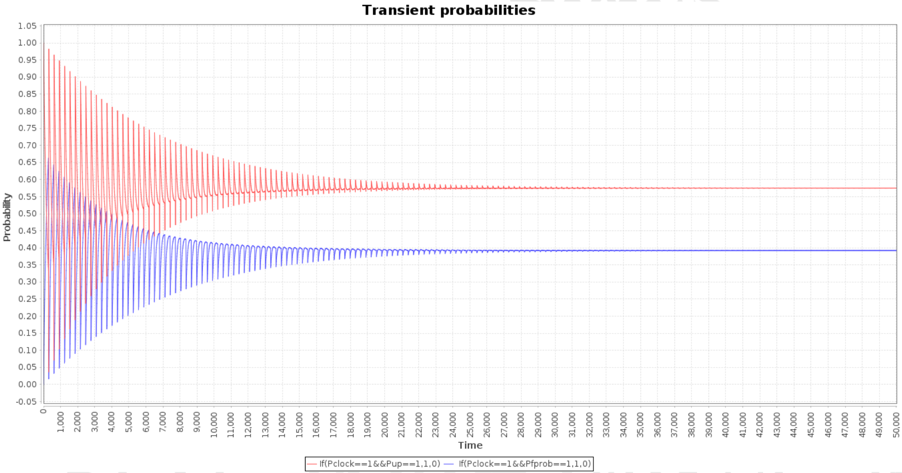
Rejuvenation



Rejuvenation



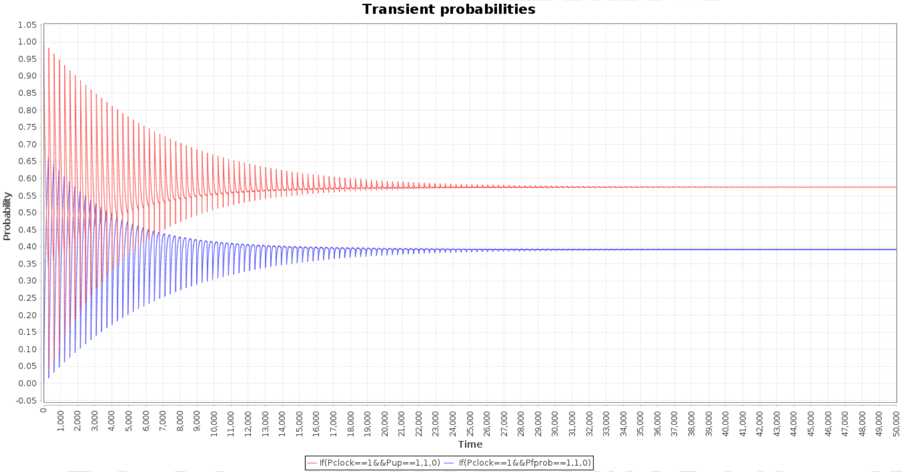
Transient analysis



✓ Steady-state estimation using transient analysis:

- $\text{Prob}(P_{clock} = 1 \wedge P_{up} = 1) \approx 0.58$
- $\text{Prob}(P_{clock} = 1 \wedge P_{fprob} = 1) \approx 0.40$

Transient analysis



- ✓ Steady-state estimation using transient analysis:
  - Prob($P_{clock} = 1 \wedge P_{up} = 1$) ≈ 0.58
  - Prob($P_{clock} = 1 \wedge P_{fprob} = 1$) ≈ 0.40

Steady state analysis

```java
Map<String, Integer> tmpPlacesMarking = new HashMap<String, Integer>();
tmpPlacesMarking.put("Pup", Integer.parseInt("1"));
tmpPlacesMarking.put("Pclock", Integer.parseInt("1"));
getTestPlacesMarkings().put(tmpPlacesMarking, new BigDecimal("0.58"));

tmpPlacesMarking = new HashMap<String, Integer>();
tmpPlacesMarking.put("Pfprob", Integer.parseInt("1"));
tmpPlacesMarking.put("Pclock", Integer.parseInt("1"));
getTestPlacesMarkings().put(tmpPlacesMarking, new BigDecimal("0.40"));
```

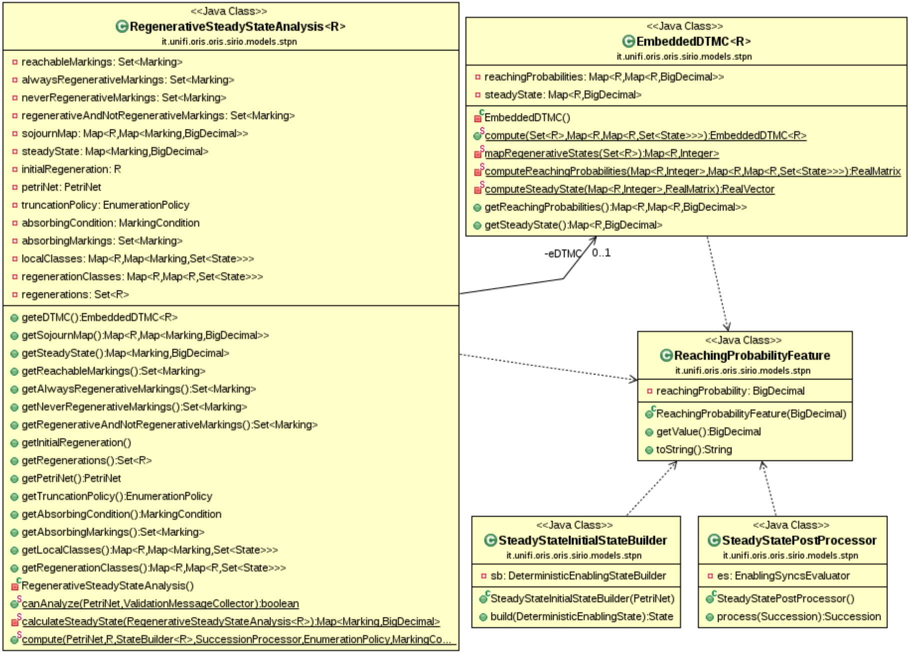
Test Passed

Steady state analysis

```java
Map<String, Integer> tmpPlacesMarking = new HashMap<String, Integer>();
tmpPlacesMarking.put("Pup", Integer.parseInt("1"));
tmpPlacesMarking.put("Pclock", Integer.parseInt("1"));
getTestPlacesMarkings().put(tmpPlacesMarking, new BigDecimal("0.58"));

tmpPlacesMarking = new HashMap<String, Integer>();
tmpPlacesMarking.put("Pfprob", Integer.parseInt("1"));
tmpPlacesMarking.put("Pclock", Integer.parseInt("1"));
getTestPlacesMarkings().put(tmpPlacesMarking, new BigDecimal("0.40"));
```

Test Passed


<<Java Class>>
RegenerativeSteadyStateAnalysis<R>
it.unifi.oris.oris.sirio.models.stpn
reachableMarkings: Set<Marking>
alwaysRegenerativeMarkings: Set<Marking>
neverRegenerativeMarkings: Set<Marking>
regenerativeAndNotRegenerativeMarkings: Set<Marking>
sojournMap: Map<R,Map<Marking,BigDecimal>>
steadyState: Map<Marking,BigDecimal>
initialRegeneration: R
petriNet: PetriNet
truncationPolicy: EnumerationPolicy
absorbingCondition: MarkingCondition
absorbingMarkings: Set<Marking>
localClasses: Map<R,Map<Marking,Set<State>>>
regenerationClasses: Map<R,Map<R,Set<State>>>
regenerations: Set<R>
geteDTMC():EmbeddedDTMC<R>
getSojournMap():Map<R,Map<Marking,BigDecimal>>
getSteadyState():Map<Marking,BigDecimal>
getReachableMarkings():Set<Marking>
getAlwaysRegenerativeMarkings():Set<Marking>
getNeverRegenerativeMarkings():Set<Marking>
getRegenerativeAndNotRegenerativeMarkings():Set<Marking>
getInitialRegeneration()
getRegenerations():Set<R>
getPetriNet():PetriNet
getTruncationPolicy():EnumerationPolicy
getAbsorbingCondition():MarkingCondition
getAbsorbingMarkings():Set<Marking>
getLocalClasses():Map<R,Map<Marking,Set<State>>>
getRegenerationClasses():Map<R,Map<R,Set<State>>>
C RegenerativeSteadyStateAnalysis()
S canAnalyze(PetriNet,ValidationMessageCollector):boolean
S calculateSteadyState(RegenerativeSteadyStateAnalysis<R>):Map<Marking,BigDecimal>
S compute(PetriNet,R,StateBuilder<R>,SuccessionProcessor,EnumerationPolicy,MarkingCo...
-eDTMC
0..1
<<Java Class>>
EmbeddedDTMC<R>
it.unifi.oris.oris.sirio.models.stpn
reachingProbabilities: Map<R,Map<R,BigDecimal>>
steadyState: Map<R,BigDecimal>
C EmbeddedDTMC()
S compute(Set<R>,Map<R,Map<R,Set<State>>>):EmbeddedDTMC<R>
S mapRegenerativeStates(Set<R>):Map<R,Integer>
S computeReachingProbabilities(Map<R,Integer>,Map<R,Map<R,Set<State>>>):RealMatrix
S computeSteadyState(Map<R,Integer>,RealMatrix):RealVector
getReachingProbabilities():Map<R,Map<R,BigDecimal>>
getSteadyState():Map<R,BigDecimal>
<<Java Class>>
ReachingProbabilityFeature
it.unifi.oris.oris.sirio.models.stpn
reachingProbability: BigDecimal
C ReachingProbabilityFeature(BigDecimal)
getValue():BigDecimal
toString():String
<<Java Class>>
SteadyStateInitialStateBuilder
it.unifi.oris.oris.sirio.models.stpn
sb: DeterministicEnablingStateBuilder
C SteadyStateInitialStateBuilder(PetriNet)
build(DeterministicEnablingState):State
<<Java Class>>
SteadyStatePostProcessor
it.unifi.oris.oris.sirio.models.stpn
es: EnablingSyncsEvaluator
C SteadyStatePostProcessor()
process(Succession):Succession

The End.

Questions? Thank you!

The End.

Questions? Thank you!

The End.

Questions? Thank you!