

# **User's Guide for ContactCenters Simulation Library**

## **CTMC call center simulator API Specification**

Version: March 4, 2014

ERIC BUIST

This is the API specification for the CTMC-based simulator using the ContactCenters library. This API describes the classes of this simulator, as well as its extensions.

# Contents

<b>Package umontreal.iro.lecuyer.contactcenters.ctmc</b>	<b>2</b>
CallCenterCTMC	3
CallCenterCTMCWithQueues	19
TransitionType	20
TransitionListener	21
AgentGroupSelector	22
SimpleGroupSelector	23
ListGroupSelector	24
PriorityGroupSelector	25
SimpleQueueSelector	26
WaitingQueueSelector	27
ListQueueSelector	28
PriorityQueueSelectorQS	29
PriorityQueueSelectorWT	30
CallCenterCTMCQueues	31
CircularIntArray	33
CCEvent	35
CCEventFactory	36
FalseTransitionEvent	37
LookupEvent	38
EventWithSelection	39
EventWithTest	41
CallCenterCTMC11	43
CallCenterCTMC11WithQueues	44
CallCenterCTMCKI	45
CallCenterCTMCKIWithQueues	46
CTMCCreationException	47
QueueSizeThresh	48
StateThresh	50
InitStateThresh	53
ProbInAWT	54

ProbInAWTBinomial . . . . .	56
ProbInAWTGamma . . . . .	57
CallCenterCounters . . . . .	58
CallCenterStat . . . . .	61
RateChangeTransitions . . . . .	64
RateChangeInfo . . . . .	66
AbstractCallCenterCTMCSim . . . . .	67
BasicCallCenterCTMCSim . . . . .	68
IntMCallCenterCTMCSim . . . . .	69
CallCenterStatMP . . . . .	70
AbstractCallCenterCTMCSimMP . . . . .	71
BasicCallCenterCTMCSimMP . . . . .	72
<b>Package umontreal.iro.lecuyer.contactcenters.ctmc.splitmerge</b>	<b>73</b>
CallCenterCTMCSimSplit . . . . .	74

## Package `umontreal.iro.lecuyer.contactcenters.ctmc`

Provides call center simulators based on a quick uniformized continuous-time Markov chain model. Staffing and scheduling optimization in large multiskill call centers is time-consuming, mainly because it requires lengthy simulations to evaluate performance measures and their sensitivity. Simplified models that provide tractable formulas are unrealistic in general. This package implements an intermediate solution, based on an approximate continuous-time Markov chain model of the call center. This model is more accurate than the commonly used approximations, and yet can be simulated faster than a more realistic simulation (based on non-exponential distributions and additional details). To speed up the simulation, the Markov chain is uniformized and only its discrete-time version is simulated. Performance measures such as the fraction of calls of each type answered within a given waiting time limit can be recovered from this simulation.

The CTMC model supports multiple call types and agent groups, but it assumes that calls arrive following a Poisson process, and service and patience times are exponential. For more information about the model, see [1].

The main class for the simulator is `BasicCallCenterCTMCSimMP`. It can simulate a call center with multiple periods by using one CTMC for each period. The `BasicCallCenterCTMCSim` class, on the other hand, provides a simulator for a single period, using a single CTMC. Both simulators first generate the total number of transitions from the Poisson distribution, followed by the transitions. However, the `IntMCallCenterCTMCSim` class implements a single-period simulator generating a large number of transitions and integrating over the number of transitions using Poisson probabilities when estimating performance measures.

All these simulators use CTMC models represented by the interface `IntMCallCenterCTMCSim`. The package provides two different implementations for this interface: one adapted for a single call type and agent group using linear search, and another one for multiple call types and agent groups based on indexed search.

## CallCenterCTMC

Represents a continuous-time Markov chain (CTMC) modeling a call center with possibly multiple call types and agent groups. More specifically, calls are divided into  $K$  types, and are served by agents partitioned into  $I$  groups. Each agent group  $i = 0, \dots, I - 1$  has a skillset  $S_i \subseteq \{1, \dots, K\}$  giving the types of the calls the agents can serve. Callers arriving when no agent is available join a waiting queue, and can abandon if they do not receive service fast enough.

For each  $k = 0, \dots, K - 1$ , we suppose that calls of type  $k$  arrive following a Poisson process with rate  $\lambda_k$ . Service times for calls of type  $k$  served by agents in group  $i$  are i.i.d. exponential with rate  $\mu_{k,i}$ . A call of type  $k$  not served immediately can balk (abandon immediately) with probability  $\rho_k$ . The patience times of callers of type  $k$  joining the queue without balking are i.i.d. exponential with rate  $\nu_k$ . Moreover, for each  $i = 0, \dots, I - 1$ , agent group  $i$  contains  $N_i$  agents.

A router is used to assign agents to new calls, and queued calls to free agents using either static lists, or matrices of priorities. See `AgentGroupSelector`, and `WaitingQueueSelector` for more information. If no agent group contains free agents, the call joins a waiting queue specific to its type (and possibly balks). Abandoning calls are lost.

This interface specifies methods to obtain information about the modelled call center as well as methods to initialize the CTMC, and generate the next state randomly from the current state.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public interface CallCenterCTMC extends Cloneable
```

### Methods

```
public void initEmpty()
```

Initializes the system to an empty call center, and resets the counter giving the number of transitions done to 0.

```
public void init (CallCenterCTMC ctmc)
```

Initializes the state of this CTMC with the state of the other CTMC `ctmc`. The parameters of this CTMC, e.g., arrival rates, service rates, etc., are unchanged while the state is set to the state of the given CTMC. This method throws an `IllegalArgumentException` if the given CTMC is incompatible with this CTMC, e.g., the number of contact types or agent groups differ.

### Parameter

`ctmc` the CTMC to initialize the state from.

**Throws**

`IllegalArgumentException` if the given CTMC is not compatible with this CTMC.

```
public int getNumContactTypes()
```

Returns the number of contact types used in the modelled call center.

**Returns** the number of contact types.

```
public int getNumAgentGroups()
```

Returns the number of agent groups used in the modelled call center.

**Returns** the number of agent groups.

```
public int getNumContactsInQueue()
```

Returns the total number of contacts currently waiting in queue.

**Returns** the total number of queued contacts.

```
public int getNumContactsInService()
```

Returns the total number of contacts currently served by agents.

**Returns** the total number of contacts in service.

```
public int getNumContactsInQueue (int k)
```

Returns the number of contacts of type `k` currently waiting in queue.

**Parameter**

`k` the tested contact type.

**Returns** the number of contacts waiting in queue.

**Throws**

`IllegalArgumentException` if `k` is negative or greater than or equal to the result of `getNumContactTypes()`.

```
public int getNumContactsInServiceK (int k)
```

Returns the number of contacts of type `k` currently in service.

**Parameter**

`k` the tested contact type.

**Returns** the number of contacts in service.

**Throws**

`IllegalArgumentException` if `k` is negative or greater than or equal to the result of `getNumContactTypes()`.

```
public int getNumContactsInServiceI (int i)
```

Returns the number of contacts currently in service by agents in group `i`.

**Parameter**

*i* the tested agent group.

**Returns** the number of contacts in service.

**Throws**

`IllegalArgumentException` if *i* is negative or greater than or equal to the result of `getNumAgentGroups()`.

```
public int getNumContactsInService (int k, int i)
```

Returns the number of contacts of type *k* in service by agents in group *i*.

**Parameters**

*k* the tested contact type.

*i* the tested agent group.

**Returns** the number of contacts in service.

```
public int getQueueCapacity()
```

Returns the maximal queue capacity.

**Returns** the maximal queue capacity.

```
public void setQueueCapacity (int q)
```

Sets the capacity of the waiting queue to *q*.

**Parameter**

*q* the new queue capacity.

**Throws**

`IllegalArgumentException` if the new capacity is smaller than the current number of contacts in queue, returned by `getNumContactsInQueue()`, or larger than the maximum returned by `getMaxQueueCapacity()`.

```
public int getMaxQueueCapacity()
```

Returns the current bound on the queue capacity used to determine the transition rate of the CTMC.

**Returns** the bound on the queue capacity.

```
public void setMaxQueueCapacity (int q)
```

Sets the bound on the queue capacity to *q*. Note that this method can change the transition rate, which usually involves recreating search indexes. Calling this method too often can thus decrease performance; one should use `setQueueCapacity (int)` instead.

**Parameter**

*q* the new maximal queue capacity.

**Throws**

`IllegalArgumentException` if `q` is smaller than the queue capacity returned by `getQueueCapacity()`.

```
public int getNumStateThresh()
```

Returns the number of thresholds on the state space. When the queue size or the number of busy agents are small enough, the CTMC simulator can use a smaller transition rate, and generate a random number of successive false transitions before every transition. Since multiple transitions are generated using a single random number, this can save CPU time. This method returns the total number of transition rates the simulator can use depending on the queue size, and number of busy agents. Note that using too many vectors of thresholds can increase memory usage, because a separate search index is required for each vector of threshold.

**Returns** the number of state thresholds.

```
public int[] [] getStateThresholds()
```

Returns the  $R \times I + 1$  matrix of thresholds whose used for state space partitioning. Each row of the matrix corresponds to a vector of thresholds, column  $i = 0, \dots, I - 1$  corresponds to thresholds on the number of agents in group  $i$ , and the last column corresponds to the queue size.

**Returns** the matrix of thresholds.

```
public void setStateThresholds (int[] [] thresholds)
```

Sets the matrix of thresholds for this CTMC to `thresholds`.

**Parameter**

`thresholds` the matrix of thresholds.

**Throws**

`NullPointerException` if `thresholds` is null.

`IllegalArgumentException` if `thresholds` has invalid dimensions.

```
public StateThresh getStateThresh()
```

Returns the thresholds on the state of the CTMC.

**Returns** the state thresholds.

```
public double getArrivalRate (int k)
```

Returns the arrival rate  $\lambda_k$  for contacts of type `k`.

**Parameter**

`k` the tested contact type.

**Returns** the arrival rate.



**Throws**

`IllegalArgumentException` if `k` is negative or greater than or equal to the result of `getNumContactTypes()`.

```
public void setArrivalRate (int k, double rate)
```

Sets the arrival rate for contacts of type `k` to `rate`.

**Parameters**

`k` the contact type.

`rate` the arrival rate.

**Throws**

`IllegalArgumentException` if `rate` is negative, or greater than the bound returned by `getMaxArrivalRate (int)`.

```
public void setArrivalRates (double[] rates)
```

Sets the arrival rate for each contact type `k` to `rates[k]`.

**Parameter**

`rates` the arrival rates.

**Throws**

`IllegalArgumentException` if `rates` does not have length  $K$ , contains at least one negative element, or if `rates[k]` is greater than the result of `getMaxArrivalRate (k)` for at least one `k`.

```
public double getArrivalRate()
```

Returns the total arrival rate  $\lambda = \sum_{k=0}^{K-1} \lambda_k$  for all contact types.

**Returns** the total arrival rate.

```
public double getMaxArrivalRate (int k)
```

Returns the maximal arrival rate  $\tilde{\lambda}_k$  for contacts of type `k`.

**Parameter**

`k` the tested contact type.

**Returns** the maximal arrival rate.

**Throws**

`IllegalArgumentException` if `k` is negative or greater than or equal to the result of `getNumContactTypes()`.

```
public void setMaxArrivalRate (int k, double rate)
```

Sets the maximal arrival rate for contacts of type `k` to `rate`. This can change the transition rate, and force the simulator to recompute some search indexes. Using this method often can therefore degrade performance, so it is recommended to call `setArrivalRate (int, double)` instead.

**Parameters**

**k** the contact type.

**rate** the arrival rate.

**Throws**

**IllegalArgumentException** if **rate** is smaller than the rate returned by **getArrivalRate** (**int**).

```
public void setMaxArrivalRates (double[] rates)
```

Sets the maximal arrival rate for each contact type **k** to **rates[k]**.

**Parameter**

**rates** the arrival rates.

**Throws**

**IllegalArgumentException** if **rates** does not have length  $K$ , or if **rates[k]** is smaller than the result of **getArrivalRate** (**k**) for at least one **k**.

```
public double getMaxArrivalRate()
```

Returns the total maximal arrival rate  $\tilde{\lambda} = \sum_{k=0}^{K-1} \tilde{\lambda}_k$  for all contact types.

**Returns** the total maximal arrival rate.

```
public double getProbBalking (int k)
```

Returns the probability of balking  $\rho_k$  for contacts of type **k**.

**Parameter**

**k** the tested contact type.

**Returns** the probability of balking.

**Throws**

**IllegalArgumentException** if **k** is negative or greater than or equal to the result of **getNumContactTypes** ().

```
public void setProbBalking (int k, double rhok)
```

Sets the balking probability to  $\rho_k$  for contacts of type **k** to **rhok**.

**Parameters**

**k** the affected contact type.

**rhok** the new value of  $\rho_k$ .

**Throws**

**IllegalArgumentException** if **rhok** is negative or greater than 1.

```
public double getPatienceRate (int k)
```

Returns the patience rate  $\nu_k$  for contacts of type **k**.

**Parameter**

**k** the tested contact type.

**Returns** the patience rate.

**Throws**

**IllegalArgumentException** if **k** is negative or greater than or equal to the result of `getNumContactTypes()`.

```
public void setPatienceRate (int k, double nuk)
```

Sets the patience rate  $\nu_k$  for contacts of type **k** to **nuk**.

**Parameters**

**k** the affected contact type.

**nuk** the new value of  $\nu_k$ .

**Throws**

**IllegalArgumentException** if **k** is out of range, **nuk** is negative, or **nuk** is greater than the maximum returned by `getMaxPatienceRate (int)`.

```
public double getMaxPatienceRate (int k)
```

Returns the maximal patience rate  $\tilde{\nu}_k$  for contacts of type **k**.

**Parameter**

**k** the tested contact type.

**Returns** the maximal patience rate.

**Throws**

**IllegalArgumentException** if **k** is negative or greater than or equal to the result of `getNumContactTypes()`.

```
public void setMaxPatienceRate (int k, double nuk)
```

Sets the maximal patience rate  $\tilde{\nu}_k$  for contacts of type **k** to **nuk**. This method can change the transition rate and recompute search indexes so using it repeatedly might degrade performance. It is recommended to use `setPatienceRate (int, double)` instead.

**Parameters**

**k** the affected contact type.

**nuk** the new patience rate.

**Throws**

**IllegalArgumentException** if **nuk** is smaller than the result of `getPatienceRate (int)`, or if **k** is out of range.

```
public double getServiceRate (int k, int i)
```

Returns the service rate  $\mu_{k,i}$  for contacts of type **k** served by agents in group **i**.

**Parameters**

`k` the tested contact type.

`i` the tested agent group.

**Returns** the service rate.

**Throws**

`IllegalArgumentException` if `k` or `i` are out of range.

```
public void setServiceRate (int k, int i, double muki)
```

Sets the service rate  $\mu_{k,i}$  to for contacts of type `k` served by agents in group `i` to `muki`.

**Parameters**

`k` the affected contact type.

`i` the affected agent group.

`muki` the new service rate.

**Throws**

`IllegalArgumentException` if `k` or `i` are out of range, if `muki` is negative, or if `muki` is greater than the maximum returned by `getMaxServiceRate (int, int)`.

```
public double getMaxServiceRate (int k, int i)
```

Returns the maximal service rate  $\tilde{\mu}_{k,i}$  for contacts of type `k` served by agents in group `i`.

**Parameters**

`k` the tested contact type.

`i` the tested agent group.

**Returns** the service rate.

**Throws**

`IllegalArgumentException` if `k` or `i` are out of range.

```
public void setMaxServiceRate (int k, int i, double muki)
```

Sets the maximal service rate for contacts of type `k` served by agents in group `i` to `muki`. This method can change the transition and recompute search indexes, so using it repeatedly might degrade performance. It is recommended to use `setServiceRate (int, int, double)` instead.

**Parameters**

`k` the affected contact type.

`i` the affected agent group.

`muki` the new maximal service rate.

**Throws**

`IllegalArgumentException` if `k` or `i` are out of range, or if `muki` is smaller than the service rate returned by `getServiceRate (int, int)`.

```
public double getJumpRate()
```

Returns the uniformized transition rate used by this CTMC.

**Returns** the uniformized transition rate.

```
public int getNumAgents()
```

Returns the total number of agents available for serving contacts.

**Returns** the total number of agents.

```
public int getMaxNumAgents()
```

Returns the maximal total number of agents that can be used for the CTMC.

**Returns** the maximal number of agents.

```
public int getNumAgents (int i)
```

Returns the number of agents in group `i`.

**Parameter**

`i` the tested agent group.

**Returns** the number of agents.

**Throws**

`IllegalArgumentException` if `i` is negative or greater than or equal to the result of `getNumAgentGroups()`.

```
public int[] getNumAgentsArray()
```

Returns an array of length `I` containing the number of agents in each agent group.

**Returns** the number of agents in each group.

```
public int getMaxNumAgents (int i)
```

Returns the maximal number of agents in group `i`.

**Parameter**

`i` the tested agent group.

**Returns** the maximal number of agents.

```
public int[] getMaxNumAgentsArray()
```

Returns an array of length `I` containing the maximal number of agents in each agent group.

**Returns** the maximal number of agents in each group.

```
public void setNumAgents (int i, int n)
```

Sets the number of agents in group  $i$  to  $n$ . This method might cause the transition rate to increase so it should never be called during the simulation of the CTMC.

**Parameters**

$i$  the tested agent group.

$n$  the new number of agents in the group.

**Throws**

`IllegalArgumentException` if  $i$  is negative or greater than or equal to the result of `getNumAgentGroups()`, or if  $n$  is negative.

```
public void setNumAgents (int[] numAgents)
```

For each agent group  $i = 0, \dots, I - 1$ , sets the number of agents in group  $i$  to `numAgents[i]`.

**Parameter**

`numAgents` the array containing the number of agents.

**Throws**

`IllegalArgumentException` if `numAgents` has a length different from  $I$ , or if it contains at least one negative value.

```
public void setMaxNumAgents (int i, int n)
```

Sets the maximal number of agents in group  $i$  to  $n$ . This method might cause the maximal transition rate returned by `getJumpRate()` to increase so it should never be called during the simulation of the CTMC.

**Parameters**

$i$  the tested agent group.

$n$  the new maximal number of agents in the group.

**Throws**

`IllegalArgumentException` if  $i$  is negative or greater than or equal to the result of `getNumAgentGroups()`, or if  $n$  is negative.

```
public void setMaxNumAgents (int[] numAgents)
```

For each agent group  $i = 0, \dots, I - 1$ , sets the maximal number of agents in group  $i$  to `numAgents[i]`.

**Parameter**

`numAgents` the array containing the number of agents.

**Throws**

`IllegalArgumentException` if `numAgents` has a length different from  $I$ , or if it contains at least one negative value.

```
public double[][] getRanksTG()
```

Returns the type-to-group matrix of ranks associating a priority to each contact type and agent group when selecting an agent group for a new arrival.

**Returns** the matrix of ranks being used.

```
public double[][] getRanksGT()
```

Returns the group-to-type matrix of ranks associating a priority to each agent group and contact type when selecting a waiting queue for a free agent.

**Returns** the matrix of ranks being used.

```
public TransitionType getLastTransitionType()
```

Returns the type of the last transition, or `null` if no transition occurred since the last call to `initEmpty()`.

**Returns** the type of the last transition.

```
public int getLastSelectedContactType()
```

Returns the last contact type selected by the `nextState (double)` method.

**Returns** the last selected contact type.

```
public int getLastSelectedContact()
```

Returns the index of the last selected contact having abandoned. This returns the position of the contact having abandoned within a queue containing contacts of type  $k$  only. The returned value is thus in  $0, \dots, Q_k - 1$  where  $k$  is the result of `getLastSelectedContactType()`, and  $Q_k$  is the number of contacts of type  $k$  in queue.

**Returns** the last selected contact.

```
public int getLastSelectedAgentGroup()
```

Returns the last agent group selected by the `nextState (double)` method.

**Returns** the last selected agent group.

```
public int getLastSelectedQueuedContactType()
```

Returns the type of the last contact removed from a waiting queue for service by the `nextState (double)` method.

**Returns** the last selected queued contact type.

```
public TransitionType nextState (double u)
```

Generates the next state of the CTMC randomly from the current state, using the given uniform `u`, and changes the current state to this new state. The method then returns the type of transition being generated. Depending on the transition type, additional information about the selected contact type or agent group can be obtained using `getLastSelectedContactType()`, `getLastSelectedQueuedContactType()`, or `getLastSelectedAgentGroup()`.

**Parameter**

`u` the uniform used to generate the new state.

**Returns** the type of the generated transition.

```
public TransitionType nextStateInt (int v)
```

Similar to `nextState (double)`, except that the given random variate `v` is uniformly distributed over  $[0, 2^{31} - 1]$ .

**Parameter**

`v` the uniform random integer.

**Returns** the type of the generated transition.

```
public TransitionType getNextTransition (double u)
```

Returns the type of the next transition generated using the random number `u`. This method is similar to `nextState (double)`, except that it does not alter the state of the CTMC.

**Parameter**

`u` the random number for the state generation.

**Returns** the type of the next transition.

```
public TransitionType getNextTransitionInt (int u)
```

Similar to `getNextTransition (double)`, using a random integer rather than a uniform number.

**Parameter**

`u` the random number used for generating the transition.

**Returns** the type of the next transition.

```
public void generateArrivalServed (int k, int i, int np, int nf)
```

Generates the arrival of a contact of type `k` served by an agent in group `i`. If all agents are busy in group `i`, this method throws an `IllegalStateException`.



**Parameters**

- k** the contact type.
- i** the agent group.
- np** the number of false transitions preceding the main transition.
- nf** the number of false transitions following the main transition.

**Throws**

- IllegalStateException** if all agents in group **i** are busy.

```
public void generateArrivalQueued (int k, int np, int nf)
```

Generates the arrival of a contact of type **k**, and adds the new contact to the waiting queue. This method throws an **IllegalStateException** if the queue is full before the arrival.

**Parameters**

- k** the type of the new contact.
- np** the number of false transitions preceding the main transition.
- nf** the number of false transitions following the main transition.

**Throws**

- IllegalStateException** if the queue capacity is exceeded.

```
public void generateArrival (int k, int np, int nf)
```

Generates the arrival of a contact of type **k** being blocked or balking.

**Parameters**

- k** the type of the arrival.
- np** the number of false transitions preceding the main transition.
- nf** the number of false transitions following the main transition.

```
public void generateEndService (int k, int i, int np, int nf)
```

Generates the end of the service for a contact of type **k** served by an agent in group **i**. If no contact of type **k** are in service by agents in group **i**, this method throws an **IllegalStateException**.

**Parameters**

- k** the type of the contact.
- i** the group of the agent.
- np** the number of false transitions preceding the main transition.
- nf** the number of false transitions following the main transition.

**Throws**

`IllegalStateException` if no contact of type `k` is in service by agents in group `i`.

```
public void generateEndService (int k, int i, int kp, int np, int nf)
```

Generates the end of the service for a contact of type `k` served by an agent in group `i`, and assigns the `kposth` queued contact of type `kp` to the free agent. If no contact of type `k` are in service by agents in group `i`, this method throws an `IllegalStateException`.

**Parameters**

`k` the type of the contact ending service.

`i` the group of the agent ending service.

`kp` the type of the dequeued contact.

`np` the number of false transitions preceding the main transition.

`nf` the number of false transitions following the main transition.

```
public void generateAbandonment (int k, int kpos, int np, int nf)
```

Generates the abandonment of the `kposth` contact of type `k`. This method throws an `IllegalStateException` if `kpos` is negative or greater than or equal to the number of queued contacts of type `k`.

**Parameters**

`k` the contact type.

`kpos` the position of the contact in queue.

`np` the number of false transitions preceding the main transition.

`nf` the number of false transitions following the main transition.

```
public boolean selectContact (int i)
```

Selects a new queued contact for a free agent in group `i`, and returns a boolean indicator determining if a contact could be selected. After this method returns `true`, the method `getLastSelectedAgentGroup()` returns the value of `i` while `getLastSelectedQueuedContactType()` returns the type of the contact assigned to the free agent. This method can be used, e.g., when agents are added in some groups during a simulation.

**Parameter**

`i` the agent group index.

**Returns** determines whether a contact is removed from a queue.

```
public void generateFalseTransition (int np, int nf)
```

Generates a false transition. This method only updates the transition counter of the CTMC.

## Parameters

**np** the number of false transitions preceding the main transition.

**nf** the number of false transitions following the main transition.

**public int getNumPrecedingFalseTransitions()**

Returns the number of additional false transitions generated by the last call to **nextState (double)** before the main transition. Some implementation of this interface might generate several false transitions using a single random number. In this case, the call to **nextState (double)** will return a transition type while this method should be used to obtain the number of additional false transitions that were generated before the main transition.

**Returns** the number of additional false transitions generated by the last call to **nextState (double)** before the main transition.

**public int getNumFollowingFalseTransitions()**

Similar to **getNumPrecedingFalseTransitions()**, but for the number of false transitions generated after the main transition.

**Returns** the number of additional false transitions generated by the last call to **nextState (double)** after the main transition.

**public int getNumTransitionsDone()**

Returns the number of generated transitions. This corresponds to the number of times the **nextState (double)** method was called since the last call to **initEmpty()**.

**Returns** the number of transitions done.

**public CallCenterCTMC clone()**

Returns an independent copy of this call center CTMC. In particular, calling **nextState (double)** on the returned CTMC should not affect the state of any other CTMC.

**Returns** the clone of the chain.

**public int hashCodeState()**

Computes and returns a hash code using the current state of the CTMC. The returned hash code should be  $\sum_{i=1}^I K_i \sum_{k=1}^K S_{k,i} + K_{I+1} \sum_{k=1}^K Q_k + K_{I+2}n$  where  $K_i = \prod_{j=1}^{i-1} \tilde{N}_j$  for  $i = 1, \dots, I + 1$ , and  $K_{I+1} = K_{I+1}H$ .

**Returns** the computed hash code.

**public boolean equalsState (Object o)**

Determines if the state of this CTMC is the same as the state of the CTMC **o**. If **o** does not correspond to an instance of **CallCenterCTMC**, this method should return **false**.

## Parameter

**o** the object to test.

**Returns** the result of the equality test.

```
public int getTargetNumTransitions()
```

Returns the current target number of transitions. Sometimes, `nextState (double)` or `nextStateInt (int)` generate several false transitions with a single random number. When the number of transitions done is near the target number of transitions, the final number of transitions may then exceed the target.

When a target number of transition is specified, no more state change occur after the target is reach. Moreover, when the return value of `getNumTransitionsDone()` is greater than or equal to the value returned by this method, `nextState (double)`, and `nextStateInt (int)` throw an `IllegalStateException`.

The default target number of transitions is `Integer.MAX_VALUE`.

**Returns** the current target number of transitions.

```
public void setTargetNumTransitions (int tntr)
```

Sets the target number of transitions to `tntr`.

#### Parameter

`tntr` the new target number of transitions.

#### Throws

`IllegalArgumentException` if `tntr` is negative.

## CallCenterCTMCWithQueues

Extends the `CallCenterCTMC` interface for keeping track of the transition number for any queued contact. This additional bookkeeping allows one to obtain the waiting time of the last contact having entered service or abandoned using the `getLastWaitingTime (int)` method, or the longest waiting time in queue using the `getLongestWaitingTime (int)` method.

Note that waiting times returned by these two methods are expressed in numbers of transitions. Because the simulated Markov chain is uniformized, the expected waiting time can be retrieved by dividing this integer by `CallCenterCTMC.getJumpRate()`.

One can use `CallCenterCTMCQueues` to implement the two methods specified by this interface.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public interface CallCenterCTMCWithQueues extends CallCenterCTMC
```

### Methods

```
public int getLastWaitingTime (int k)
```

?returns waiting time of the last contact of type `k` having entered service, or abandoned. If no such contact exists, this method returns 0.

#### Parameter

`k` the queried type of contact.

**Returns** the last waiting time.

```
public int getLongestWaitingTime (int k)
```

Returns the longest waiting time among all contacts of type `k`. This returns 0 if no contact of type `k` is waiting in queue.

#### Parameter

`k` the queried contact type.

**Returns** the longest waiting time.

## TransitionType

Represents the type of a transition performed by the `CallCenterCTMC nextState` (double) method.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public enum TransitionType
```

### Constants

#### ARRIVALSERVED

Arrival with immediate service.

#### ARRIVALBALKED

Arrival with balking (immediate abandonment).

#### ARRIVALQUEUED

Arrival and waiting in queue.

#### ARRIVALBLOCKED

Arrival and blocking due to exceeded queue capacity.

#### ENDSERVICEANDDEQUEUE

An agent terminates a service, and receives a new queued contact to serve.

#### ENDSERVICENODEQUEUE

An agent terminates a service, and remains free because of no available queued contacts.

#### ABANDONMENT

A queued contacts abandons, i.e., leaves the queue without receiving service.

#### FALSETRANSITION

Fictitious transition not affecting the state of the CTMC.

# TransitionListener

Represents a listener that can be notified when a transition occurs during a DTMC simulation of a call center. After an object of a class implementing this interface is constructed, it can be registered with a DTMC call center simulator.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public interface TransitionListener
```

## Methods

```
public void init (CallCenterCTMC ctmc, int r, int mp)
```

This method is called when the registered call center CTMC is initialized, during replication `r`, at the beginning of main period `mp`.

### Parameters

`ctmc` the initialized CTMC.

`r` the replication number.

`mp` the main period index.

```
public void newTransition (CallCenterCTMC ctmc, int r, int mp,  
                           TransitionType type)
```

This method is called when a new transition occurs in the CTMC `ctmc`, during replication `r` of the simulation of main period `mp`. The type of the simulated transition is given by `type`.

### Parameters

`ctmc` the continuous-time Markov chain.

`r` the replication index.

`mp` the main period index.

`type` the transition type.

## AgentGroupSelector

Represents a policy selecting an agent group for an incoming contact, in a CTMC call center model. A CTMC model supporting multiple call types can have a different agent group selector for each contact type. The available implementations are `ListGroupSelector`, and `PriorityGroupSelector`.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public interface AgentGroupSelector
```

### Methods

```
public int selectAgentGroup (CallCenterCTMC ctmc, int tr)
```

Selects an agent group for the newly arrived contact, and returns the index of the selected agent group. If no agent group can be selected, this method returns a negative value.

#### Parameters

`ctmc` the call center CTMC model.

`tr` the current transition number.

**Returns** the selected agent group.

```
public double[] getRanks()
```

Returns an array giving the rank associated with each agent group by this agent group selector.

**Returns** the array of ranks.



# SimpleGroupSelector

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class SimpleGroupSelector implements AgentGroupSelector
```

## ListGroupSelector

Represents an agent group selector using static lists for agent selection. When a call enters the center, the group of the serving agent is given by the first group, among the user-specified groups  $i_0, i_1, \dots$ , containing at least one free agent.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class ListGroupSelector implements AgentGroupSelector
```

### Constructor

```
public ListGroupSelector (int numGroups, int[] groupList)  
    Constructs a new list-based agent group selector using the given static list groupList.
```

### Parameter

`groupList` the list of agent groups being queried by `selectAgentGroup (CallCenter-CTMC, int)`.

# PriorityGroupSelector

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class PriorityGroupSelector implements AgentGroupSelector
```

## SimpleQueueSelector

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class SimpleQueueSelector implements WaitingQueueSelector
```

# WaitingQueueSelector

Represents a policy selecting a queued contact for an agent becoming free, in a CTMC call center model. A CTMC model supporting multiple agent groups can have a different waiting queue selector for each agent group. The available implementations are `ListQueueSelector`, `PriorityQueueSelectorQS`, and `PriorityQueueSelectorWT`.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public interface WaitingQueueSelector
```

## Methods

```
public int selectWaitingQueue (CallCenterCTMC ctmc, int k, int tr)
```

Selects a waiting queue for the free agent, and returns the index of the selected queue. If no waiting queue can be selected, this method returns a negative value.

### Parameters

`ctmc` the call center CTMC model.

`tr` the current transition number.

**Returns** the selected waiting queue.

```
public double[] getRanks()
```

Returns an array giving the rank associated with each waiting queue by this waiting queue selector.

**Returns** the array of ranks.

## ListQueueSelector

Represents a waiting queue selector using static lists. When an agent becomes free, the router selects the first waiting queue, among the user-specified queues  $k_0, k_1, \dots$ , containing at least one call. If such a queue exists, the first queued call is removed, and assigned to the free agent. Otherwise, the agent stays free until a new call arrives.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public class ListQueueSelector implements WaitingQueueSelector
```

### Constructor

```
public ListQueueSelector (int numQueues, int[] queueList)
```

Constructs a new list-based waiting queue selector using the given list `queueList`.

### Parameter

`queueList` the list of waiting queried by `selectWaitingQueue (CallCenterCTMC, int, int)`.

## PriorityQueueSelectorQS

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class PriorityQueueSelectorQS implements WaitingQueueSelector
```

## PriorityQueueSelectorWT

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public class PriorityQueueSelectorWT implements WaitingQueueSelector
```



## CallCenterCTMCQueues

Provides helper method used to maintain information on queued calls, for a CTMC model of a call center. This class encapsulates an array of circular arrays of integers representing the waiting queues, and provides the `init()` that should be called after `CallCenterCTMC.initEmpty()` or `CallCenterCTMC.init (CallCenterCTMC)`. It also implements the `update (CallCenterCTMC, TransitionType)` method which should be called after each transition to update the waiting queues. The methods `getLastWaitingTime (int)`, and `getQueue (int)` can then be used by CTMC models to implement the interface `CallCenterCTMCWithQueues`.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class CallCenterCTMCQueues implements Cloneable
```

### Constructor

```
public CallCenterCTMCQueues (CallCenterCTMC ctmc)
```

Constructs a new object holding queueing information from the call center CTMC model `ctmc`.

#### Parameter

`ctmc` the call center CTMC model.

### Methods

```
public CircularIntArray getQueue (int k)
```

Returns the circular array of integers containing the transition number at which each call of type `k` entered the queue, provided that the `update (CallCenterCTMC, TransitionType)` method has been called at each transition.

#### Parameter

`k` the tested call type.

**Returns** the circular array representing the queue.

```
public int getLastWaitingTime (int k)
```

Returns the number of transitions spent by the last call of type `k` having left the queue, provided that the `update (CallCenterCTMC, TransitionType)` method has been called after each transition.

#### Parameter

`k` the tested call type.

**Returns** the number of transitions spent in queue.

`public void init()`

Empties all the circular arrays representing waiting queues, and resets the last waiting times to 0.

`public void init (CallCenterCTMCQueues q)`

Initializes this object with the contents of the other object `q`.

**Parameter**

`q` another object holding queue information.

`public void update (CallCenterCTMC ctmc, TransitionType type)`

Updates the status of the waiting queues after a transition of type `type` of the CTMC model `ctmc`.

**Parameters**

`ctmc` the CTMC model in which the transition occurred.

`type` the type of transition.

`public CallCenterCTMCQueues clone()`

Constructs and returns a deep copy of this object, including copies of the waiting queues.

# CircularIntArray

Represents a resizable circular array of integers. Any object of this class encapsulates an ordinary array of integers as well as an index of the starting position. Elements can be added to the array which grows as necessary. The first and last element of the array can be removed in constant time while removing any other element requires a linear time shift of the other elements in the array.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class CircularIntArray implements Cloneable
```

## Constructors

```
public CircularIntArray()
```

Constructs a new circular array of integers with a default initial capacity of 5 elements.

```
public CircularIntArray (int initialCapacity)
```

Constructs a new array of integers with the given initial capacity.

### Parameter

`initialCapacity` the initial capacity of the array.

## Methods

```
public void clear()
```

Clears this circular array. After a call to this method, the size of the array returned by `size()` is 0.

```
public void init (CircularIntArray a)
```

Initializes this circular array of integers with the contents of another circular array `a`. This method replaces the internal array of this object with a clone of the internal array of `a`, and also copies the size and starting position of the given array.

### Parameter

`a` the other circular array to copy.

```
public void add (int x)
```

Adds the element `x` at the end of this circular array. If adding the element would exceed the capacity of the internal array, the internal array grows.

**Parameter**

**x** the new element to add.

```
public int removeFirst()
```

Removes and returns the first element of this circular array. If the size of the array returned by the `size()` method is 0, this method throws a `NoSuchElementException`.

**Returns** the value of the (removed) first element.

```
public int removeLast()
```

Removes and returns the last element of this circular array. If the size of the array returned by the `size()` method is 0, this method throws a `NoSuchElementException`.

**Returns** the value of the (removed) last element.

```
public int size()
```

Returns the size of this circular array.

**Returns** the size of the circular array.

```
public int get (int i)
```

Returns the value of element `i` in this circular array.

**Parameter**

**i** the index of the queried element.

**Returns** the value of the element.

```
public void set (int i, int e)
```

Sets the value of element `i` to `e`.

**Parameters**

**i** the index of the element to modify.

**e** the new value of the element.

```
public int remove (int i)
```

Removes and returns the element with index `i` in the array. If the given index is 0, this calls `removeFirst()`. If the given index is `size()` minus 1, this returns the result of `removeLast()`. Otherwise, elements of the array are shifted appropriately to remove the element.

**Parameter**

**i** the index of the element to remove.

**Returns** the removed element.

```
public CircularIntArray clone()
```

Returns a copy of this circular array of integers. The returned copy is completely independent from this instance since the internal array is also cloned.

```
public String toString()
```

Constructs and returns a string representation of the form `[e1, e2, ...]` of this array.

## CCEvent

Represents an event occurring during a transition of a CTMC representing a contact center with multiple contact types and agent groups.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public interface CCEvent
```

### Method

```
public TransitionType actions (CallCenterCTMCKI ctmc, int tr, int rv, int  
                               usedBits, boolean changeState)
```

Performs the necessary actions for the transition, and returns the appropriate transition type. This method is called by `CallCenterCTMCKI.nextStateInt (int)` in order to generate a transition. Random bits can be obtained as needed by using the given integer `rv`, but the `numUsedBits` least significant bits of `rv` are already used before the method is called, i.e., to select the index of an event in a lookup table.

### Parameters

`ctmc` the CTMC representing the call center.

`tr` the number of transitions already done.

`rv` the random integer used to simulate the transition.

`usedBits` the number of bits already used in `rv`.

`changeState` determines if the event can change the state of the CTMC.

**Returns** the type of the simulated transition.

## CCEventFactory

Represents a factory for creating new call center events.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public interface CCEventFactory
```

### Method

```
public CCEvent newInstance (double u1, double u2, int maxExtraBits)
```

Creates a new call center event resulting from a uniform generated in interval  $[u_1, u_2]$ , and using a maximum of `maxExtraBits` additional random bits to take decisions.

#### Parameters

`u1` the uniform  $u_1$ .

`u2` the uniform  $u_2$ .

`maxExtraBits` the maximal number of additional bits that can be used by the created events to take decisions.

**Returns** the constructed event.

## FalseTransitionEvent

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class FalseTransitionEvent implements CCEvent
```

# LookupEvent

Represents a call center event using random bits to select the index of a subinterval corresponding to an event. The executed selected event might perform some action or a new indexed search. A lookup event can be constructed from any array of call center events. Alternatively, a method `createIndex (double[], CCEventFactory[], int, int)` is provided to construct a search index.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public class LookupEvent implements CCEvent
```

## Constructor

```
public LookupEvent (CCEvent[] events)
```

Creates a new lookup event selecting events from the given array `events`.

### Parameter

`events` the array of events.

## Method

```
public static LookupEvent createIndex (double[] prob, CCEventFactory[]  
                                     factories, int numIntervals, int  
                                     maxBits)
```

Creates a search index by partitioning the  $[0, 1]$  in `numIntervals` subintervals, and using a maximum of `maxBits` random bits for the indexed search. A subinterval is assigned an event `l` created with the factory `factories[l]` with probability `prob[l]`.

### Parameters

`prob` the probabilities of occurrence of the events.

`factories` the event factories.

`numIntervals` the number of subintervals.

`maxBits` the maximal number of bits.

**Returns** the event performing the indexed search.



## EventWithSelection

Represents an event that can select the integers  $k = 0, \dots, K - 1$  and  $p = 0, \dots, n_k - 1$  such that  $0 \leq u - \sum_{j=0}^{k-1} W_j n_j - p W_k < w_k$ . Here  $0 \leq u_1 < u_2 \leq WN$  where  $w_k \leq W_k \leq W$  for  $k = 0, \dots, K - 1$ , and  $\sum_{k=0}^{K-1} n_k \leq N$ . This can be interpreted as selecting an event  $p = 0, \dots, n_k - 1$  of type  $k$ , with each event of type  $k$  having weight  $w_k$ .

This class can be used, e.g., to determine if a uniform  $u$  generates a false transition, or an abandonment of type  $k$ . In this case,  $n_k$  gives the number of queued contacts of type  $k$ ,  $w_k = \nu_k$  is the abandonment rate for contacts of type  $k$ ,  $W_k = \tilde{\nu}_k$  is the maximal possible abandonment rate for contacts of type  $k$ , and  $N$  is the total queue capacity. The integer  $p$  then corresponds to the position of the contact in queue having abandoned.

To use this class, one must extend it to provide implementations for `getNumValues (CallCenterCTMCKI)`, and `getNumValues (CallCenterCTMCKI, int)`. In the `CCEvent.actions (CallCenterCTMCKI, int, int, int, boolean)` method, one then calls `select-Type (CallCenterCTMCKI, int, int, int)` to select the event type.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;

public abstract class EventWithSelection extends EventWithTest
```

### Constructor

```
public EventWithSelection (double minU, double maxU, int numBits, int
                           numTypes, double maxWeight)
```

Constructs a new event with selection.

#### Parameters

`minU` the value of  $u_1$ .

`maxU` the value of  $u_2$ .

`numBits` the number  $b$  of bits used to generate  $u$  when the test cannot be completed only with  $u_1$ , and  $u_2$ .

`numTypes` the number  $K$  of event types.

`maxWeight` the maximal weight  $W$ .

### Methods

```
public int getNumTypes()
```

Returns the number  $K$  of event types.

**Returns** the value of  $K$ .

`public int getLastSelectedEvent()`

Returns the index  $p$  of the last selected event among events of type  $k$ .

**Returns** the last selected position.

`public abstract double getWeight (CallCenterCTMCKI ctmc, int k)`

Returns the weight  $w_k$  corresponding to events of type  $k$ .

**Parameters**

`ctmc` the tested CTMC.

`k` the tested type.

**Returns** the weight  $w_k$ .

`public abstract double getMaxWeight (CallCenterCTMCKI ctmc, int k)`

Returns the weight  $W_k$  corresponding to events of type  $k$ .

**Parameters**

`ctmc` the tested CTMC.

`k` the tested type.

**Returns** the weight  $W_k$ .

`public abstract int getNumValues (CallCenterCTMCKI ctmc, int k)`

Returns the current value of  $n_k$ .

**Parameters**

`ctmc` the tested CTMC.

`k` the tested type.

**Returns** the value of  $n_k$ .

`public abstract int getNumValues (CallCenterCTMCKI ctmc)`

Returns the sum  $\sum_{k=0}^{K-1} n_k$ .

**Parameter**

`ctmc` the tested CTMC.

**Returns** the sum of  $n_k$ 's.

`public int selectType (CallCenterCTMCKI ctmc, int tr, int rv, int usedBits)`

Selects and returns an event type  $k$ . If the event corresponds to a false transition, this returns  $K$ . Otherwise, the method `getLastSelectedEvent()` can be used to obtain  $p$ .

**Parameters**

`ctmc` the CTMC representing the call center.

`tr` the number of transitions already done.

`rv` the random integer used to simulate the transition.

`usedBits` the number of bits already used in `rv`.

**Returns** the selected event type.

# EventWithTest

Abstract event type with some helper methods to generate  $u_1 \leq u < u_2$  randomly and uniformly, and to test that  $u < t$  for any value of  $t$ .

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;

public abstract class EventWithTest implements CCEvent
```

## Constructor

```
public EventWithTest (double minU, double maxU, int numBits)
```

Constructs a new event with the associated interval  $[u_1, u_2)$ , and using `numBits` additional random bits to generate  $u$  when needed. The value of  $u_1$  and  $u_2$  are given using the fields `minU` and `maxU`.

### Parameters

`minU` the value of  $u_1$ .

`maxU` the value of  $u_2$ .

`numBits` the number of bits used to generate  $u$ .

## Methods

```
public double getMinU()
```

Returns the value of  $u_1$  associated with this event.

**Returns** the associated value of  $u_1$ .

```
public double getMaxU()
```

Returns the value of  $u_2$  associated with this event.

**Returns** the associated value of  $u_2$ .

```
public double getU (int rv, int usedBits)
```

Generates the value of  $u$  using bits in `rv` but ignoring the first least significant `usedBits` bits.

### Parameters

`rv` the random bits to generate  $u$  from.

`usedBits` the number of used random bits.

**Returns** the value of  $u$ .

```
public boolean isUSmallerThan (int rv, int usedBits, double t)
```

Returns **true** if and only if  $u < t$ , using `getU (rv, usedBits)` to generate  $u$  randomly. This method returns **true** without generating  $u$  if  $u_2 < t$  since in that case,  $u < u_2 < t$ . Similarly, it returns **false** without generating  $u$  if  $u_1 \geq t$  since in this case,  $u \geq u_1 \geq t$ . Otherwise, `getU (int, int)` is called to get the value of  $u$ .

#### Parameters

`rv` the random bits to generate  $u$  from.

`usedBits` the number of used random bits.

`t` the tested threshold.

**Returns** the success indicator of the test.

```
public int getPosition (int rv, int usedBits, double weight, double
                        maxWeight, int maxS)
```

Returns the value  $s$  for which  $0 \leq u - sW < w$ , where  $s = 0, \dots, S - 1$ , or  $S$  if such  $s$  does not exist. Here,  $w \leq W$ . For example, if  $0 \leq u < \tilde{q}$ ,  $w = \nu$ , and  $W = \tilde{\nu}$ ,  $s$  can be interpreted as the position in queue of a contact having abandoned.

#### Parameters

`rv` the random bits to generate  $u$  from.

`usedBits` the number of used random bits.

`weight` the value of  $w$ .

`maxWeight` the value of  $W$ .

`maxS` the maximal value  $S$  of  $s$ .

**Returns** the value of  $s$ .

## CallCenterCTMC11

CTMC model for a call center with a single call type and a single agent group.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class CallCenterCTMC11 implements CallCenterCTMC
```

## CallCenterCTMC11WithQueues

Extension of the CTMC model for a single contact type and agent group, with information on contacts waiting in queue.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class CallCenterCTMC11WithQueues extends CallCenterCTMC11  
    implements CallCenterCTMCWithQueues
```

## CallCenterCTMCKI

CTMC model of a call center with multiple call types and agent groups.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class CallCenterCTMCKI implements CallCenterCTMC
```

## CallCenterCTMCKIWithQueues

Extends the CTMC model for multiple call types and agent groups with information on queued calls.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class CallCenterCTMCKIWithQueues extends CallCenterCTMCKI  
    implements CallCenterCTMCWithQueues
```



## CTMCCreationException

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class CTMCCreationException extends Exception
```

## QueueSizeThresh

Encapsulates thresholds on the queue size with the corresponding transition rates and geometric distributions for the number of successive self jumps preceding any generated transition.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class QueueSizeThresh
```

### Constructor

```
public QueueSizeThresh (double jumpRate, double nu, int queueCapacity, int  
                        numThresh)
```

Constructs a new manager for queue size thresholds, for a CTMC with the given jump rate `jumpRate`, representing a call center with maximal abandonment rate `nu`, and maximal queue capacity `queueCapacity`. The computed thresholds is an increasing sequence of `numThresh` integers distributed evenly on the interval  $[0, H]$ , where  $H$  is the queue capacity.

### Parameters

`jumpRate` the jump rate for the CTMC.  
`nu` the maximal abandonment rate.  
`queueCapacity` the queue capacity.  
`numThresh` the number of thresholds to create.

### Methods

```
public int getNumThresholds()
```

Returns the number of thresholds managed by this object.

**Returns** the number of managed thresholds.

```
public double getJumpRate (int r)
```

Returns the maximal transition rate if the queue size is smaller than or equal to the threshold with index `r`.

### Parameter

`r` the index of the tested threshold.

**Returns** the transition rate.

```
public int getQueueSizeThresh (int r)
```

Returns the threshold on the queue size with index `r`.

**Parameter**

`r` the index of the threshold.

**Returns** the threshold corresponding to the index.

```
public GeometricDist getNumFalseTrDist (int r)
```

Returns the geometric distribution for the successive number of self jumps before any transition, while the queue size is smaller than or equal to threshold with index `r`.

**Parameter**

`r` the index of the queue size threshold.

**Returns** the distribution of the successive number of self jumps before any transition.

```
public int updateQIdx (int qidx, int queueSize)
```

Returns the smallest index for which the queue size is smaller than or equal to the corresponding threshold, given that the current index is `qidx`. The given index is used as a starting point for searching the correct index.

**Parameters**

`qidx` the current threshold index.

`queueSize` the current queue size.

**Returns** the appropriate queue size threshold index.

## StateThresh

Represents thresholds on the queue size, and the number of agents in each group. The transition rate, and the distribution for the number of successive self jumps preceding any transition are also computed and stored.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public class StateThresh implements Cloneable
```

### Constructor

```
public StateThresh (CallCenterCTMC ctmc, int[] [] thresholds)
```

Constructs a new state thresholds object using the given thresholds on the queue size and agent groups. This constructor accepts a CTMC `ctmc`, a matrix of thresholds `thresholds` on the number of agents, and the queue size. Elements `thresholds[r][i]`, for  $i = 0, \dots, I - 1$ , give the  $r$ th threshold on the number of agents in group  $i$ , while element `thresholds[r][I]`, the threshold on queue size. If `thresholds` is null, a single vector of thresholds  $(\tilde{N}_0, \dots, \tilde{N}_{I-1}, H)$  is used. This vector of thresholds is also added if `thresholds` is not null, and the vector is not present in the 2D array `thresholds`.

### Parameters

`ctmc` the call center CTMC.

`thresholds` the thresholds on the number of agents, and queue size.

### Throws

`IllegalArgumentException` if `thresholds` has invalid dimensions, or contains a negative value.

### Methods

```
public int getNumVectorsOfThresholds()
```

Returns the number of vectors of thresholds stored by this object.

**Returns** the number of vectors of thresholds.

```
public double getJumpRate (int r)
```

Returns the transition rate corresponding to vector of thresholds with index  $r$ .

### Parameter

`r` the index of the vector of thresholds.

**Returns** the corresponding transition rate.

```
public GeometricDist getNumFalseTrDist (int r)
```

Returns the geometric distribution for the successive number of self jumps before any transition, while the queue size and number of agents are smaller than or equal to thresholds with index  $r$ .

**Parameter**

$r$  the index of the vector of thresholds.

**Returns** the distribution of the successive number of self jumps before any transition.

```
public int getThreshNumAgents (int r, int i)
```

Returns the threshold on the number of agents in group  $i$  corresponding to vector with index  $r$ .

**Parameters**

$r$  the index of the vector of thresholds.

$i$  the index of the agent group.

**Returns** the value of the threshold.

```
public int[] [] getThreshNumAgents()
```

Returns a 2D array containing the thresholds on the number of agents. Element  $[r][i]$  of the array gives the threshold with index  $r$  for agent group  $i$ .

**Returns** the array of thresholds.

```
public int getThreshQueueSize (int r)
```

Returns the threshold on the queue size corresponding to vector with index  $r$ .

**Parameter**

$r$  the index of the vector of thresholds.

**Returns** the threshold on the queue size.

```
public int[] getThreshQueueSize()
```

Returns the array of thresholds on the queue size. Element  $r$  of the returned array gives the threshold for index  $r$ .

**Returns** the array of thresholds on the queue size.

```
public void initOperatingMode (CallCenterCTMC ctmc)
```

Determines the current operating mode  $r$  depending on the state of the given CTMC  $ctmc$ . This method is called after the CTMC is initialized. The more efficient `updateOperatingMode (CallCenterCTMC, TransitionType)` can be used to update  $r$  at each transition. The value of  $r$  can be obtained using `getOperatingMode()`.

**Parameter**

`ctmc` the call center CTMC.

```
public int getOperatingMode()
```

Returns the current operating mode. The operating mode corresponds to the vector of thresholds  $r$  offering the smallest jump rate, and thresholds greater than or equal to the current number of agents and queue size.

**Returns** the current operating mode.

```
public boolean updateOperatingMode (CallCenterCTMC ctmc, TransitionType  
                                   type)
```

Updates the current vector of thresholds after a transition of type **type** of the CTMC model `ctmc`. This returns **true** if the operating mode changed, and needs to be queried using `getOperatingMode()`.

**Parameters**

`ctmc` the call center CTMC.

`type` the transition type.

**Returns** **true** if the operating mode changed.

## InitStateThresh

Used to initialize vectors of thresholds automatically. This program provides a method `getThresholds` (`CallCenterCTMC`, `int`, `int`, `boolean`) returning a matrix of thresholds. The program can also be called from the command-line to perform the initialization.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public class InitStateThresh
```

### Methods

```
public static int[][] getThresholds (CallCenterCTMC ctmc, int
                                   numStateThresh, int numGroupSlices,
                                   boolean threshOnQueueSize)
```

Returns a matrix with `numStateThresh*numGroupSlices` rows representing vectors of thresholds. Columns  $i = 0, \dots, I - 1$  of the returned matrix give thresholds for the number of agents while column  $I$  gives thresholds on the waiting queue. The vectors of thresholds are constructed based on the CTMC model `ctmc`. The constructed matrix has  $M = \text{numGroupSlices}$  sets of `numStateThresh` vectors of thresholds. Set  $m$ , for  $m = 0, \dots, M - 1$ , has thresholds with number of agents in  $(\lfloor m\tilde{N}_i/M \rfloor, \lfloor (m+1)\tilde{N}_i/M \rfloor]$ .

### Parameters

`ctmc` the call center CTMC model.

`numStateThresh` the number of vectors of thresholds on the state.

`numGroupSlices` the number of slices for the number of agents in groups.

**Returns** the 2D array of thresholds.

### Throws

`NullPointerException` if `ctmc` is null.

`IllegalArgumentException` if `numStateThresh` or `numGroupSlices` are smaller than 1.

```
public static void main (String[] args)
```

Main method of the class, to be called from the command-line. This method accepts the name of the parameter file for the call center, the name of the parameter file for the experiments, the number of vectors of thresholds to create, and an output parameter file. The program computes the vectors of thresholds for each main period in the model, and outputs a modified version of the given experiment parameter file, with the vectors of thresholds.

### Parameter

`args` the command-line arguments of the program.

# ProbInAWT

Represents an object that can compute information on the waiting time distribution conditional on the total number of transitions during a time horizon, the transition rate, and the number of transitions spent in queue by a particular contact.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public interface ProbInAWT
```

## Methods

```
public double getProbInAWT (int delta)
```

Returns the probability that the waiting time of a contact having spent **delta** transitions in the queue is smaller than the current acceptable waiting time.

### Parameter

**delta** the number of transitions spent in queue by the contact.

**Returns** the probability of the waiting time being smaller than the threshold.

```
public double getExpectedWaitingTime (int delta)
```

Returns the expected waiting time for a contact having spent **delta** transitions in queue.

### Parameter

**delta** the number of transitions spent in queue.

**Returns** the expected waiting time.

```
public double getExpectedWaitingTimeGTAWT (int delta)
```

Returns the expected waiting time conditional that the waiting time is greater than the acceptable waiting time, for a contact having spent **delta** transitions into the queue.

### Parameter

**delta** the number of transitions spent into the queue.

**Returns** the expected waiting time.

```
public double getJumpRate()
```

Returns the currently used transition rate.

**Returns** the transition rate.

```
public double getAWT()
```

Returns the currently used acceptable waiting time.



**Returns** the acceptable waiting time.

```
public double getTimeHorizon()
```

Returns the currently used time horizon.

**Returns** the time horizon.

```
public int getNumTransitions()
```

Returns the currently used number of transitions.

**Returns** the current number of transitions.

```
public void init (double awt, double jumpRate, double timeHorizon, int  
                  numTransitions)
```

Initializes this object with a new acceptable waiting time, transition rate, time horizon, and number of transitions.

#### Parameters

**awt** the new acceptable waiting time.

**jumpRate** the new transition rate.

**timeHorizon** the new time horizon.

**numTransitions** the new number of transitions.

## ProbInAWTBinomial

Computes information on the conditional distribution of the waiting time, for a deterministic horizon.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class ProbInAWTBinomial implements ProbInAWT
```

## ProbInAWTGamma

Computes information on the conditional distribution of the waiting time in the case of a random horizon.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class ProbInAWTGamma implements ProbInAWT
```

## CallCenterCounters

Represents statistical counters computing sums for individual replications of a simulation of a call center using a discrete-time Markov chain. After a simulator constructs an instance of this class, it calls `init (CallCenterCTMC, double, int)` at the beginning of each replication, and then uses `collectStat (CallCenterCTMC, TransitionType)` for each simulated transition. At the end of the simulation, the method `updateStatOnTime (CallCenterCTMC)` should also be called.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public class CallCenterCounters implements Cloneable
```

### Constructors

```
public CallCenterCounters (CallCenterCTMC ctmc, double[] awt, boolean
                           randomHorizon)
```

Constructs a new set of call center counters using the given CTMC to obtain the number of call types, agent groups, etc. The argument `awt` contains a vector giving the acceptable waiting times for estimating the expected number of calls waiting less than a given time limit. Element  $sK' + k$  of this array gives the  $s$ th AWT for calls of type  $k$  if  $k = 0, \dots, K - 1$ , or for calls of all types if  $k = K$ . Here,  $K' = K + 1$  if  $K > 1$ , or  $K$  otherwise. This can be `null` if performance measures based on acceptable waiting times are not estimated.

The boolean `randomHorizon` is set to `true` if we are simulating on a random horizon, or `false` for deterministic horizon. The horizon type has an impact on how some performance measures are estimated.

### Parameters

`ctmc` the call center model.

`awt` the vector of acceptable waiting times.

`randomHorizon` `true` for a random time horizon, `false` for a deterministic horizon.

```
public CallCenterCounters (CallCenterCTMC ctmc, double[] awt, double[]
                           gawt, boolean randomHorizon, double[] jumpRate,
                           int[] startingTransition, int counterPeriod)
```

Similar to constructor `CallCenterCounters (CallCenterCTMC, double[], boolean)`, for a case with multiple periods. The additional argument `gawt` plays a role similar to `awt`, but contains thresholds used for all periods; this is used to estimate performance measures based of acceptable waiting times over the entire horizon while `awt` is used for the estimates over a single period.

The last three arguments are used to estimate performance measures for calls arriving into and leaving the system during different periods. The argument `jumpRate` contains a  $(P + 1)$ -dimensional vector of transition rates for each of the  $P + 1$  period-specific CTMCs; there

is  $P$  CTMCs for the main periods, plus one CTMC for the wrap-up period. The argument `startingTransition`, on the other hand, is a  $(P + 1)$ -dimensional vector giving the starting transition for each period; this is updated as the simulation is made. The argument `counterPeriod` gives the index of the period for which this set of counters computes sums.

### Parameters

`ctmc` the call center model.

`awt` the vector of period-specific acceptable waiting times.

`gawt` the vector of global acceptable waiting times.

`randomHorizon` `true` for a random time horizon, `false` for a deterministic horizon.

`jumpRate` the per-period transition rates.

`startingTransition` the starting transitions for each period.

`counterPeriod` the period concerned by this counter.

### Methods

```
public void init (CallCenterCTMC ctmc, double timeHorizon, int ntr)
```

Initializes this set of counters using the given call center CTMC `ctmc`, for a simulation over a time horizon `timeHorizon` with `ntr` transitions. For random horizon, the number of transitions is ignored.

### Parameters

`ctmc` the call center CTMC.

`timeHorizon` the time horizon.

`ntr` the number of transitions to simulate.

```
public void collectSum (boolean lastTimeAvg, boolean statAWTG,
                       CallCenterCounters... counters)
```

For each counter of this set, replaces the current value with the sum of the values of all corresponding counters in the sets given by the array `counters`. This can be used to aggregate statistics from successive individual periods into a single counter.

The `lastTimeAvg` boolean determines if the last counter in the given array, which usually corresponds to counters concerning the wrap-up period, is taken into account when summing the time-average queue size and number of agents. This has no impact on other statistics.

If `statAWTG` is set to `true`, the number of calls waiting less than the acceptable waiting time, stored in fields `numServedBeforeAWT` and `numAbandonedBeforeAWT`, are determined by summing the numbers in fields `numServedBeforeAWTG` and `numAbandonedBeforeAWTG` in the counters of `counters`. If `statAWTG` is `false`, the fields `numServedBeforeAWT` and `numAbandonedBeforeAWT` are used instead.

**Parameters**

`lastTimeAvg` determines whether the last element in array `counters` is taken into account for average queue size and number of agents.

`statAWTG` determines how the number of calls waiting less than the acceptable waiting time is summed up.

`counters` the array of counters.

```
public void collectStat (CallCenterCTMC ctmc, TransitionType type)
```

Collects statistics concerning the last transition of the CTMC `ctmc` with type `type` by updating the appropriate counters.

**Parameters**

`ctmc` the call center CTMC.

`type` the transition type.

```
public void updateStatOnTime (CallCenterCTMC ctmc)
```

Updates the arrays `queueSize`, `busyAgents`, and `totalAgents` for this set of counters from the corresponding accumulates computing time-averages for the queue size, number of busy agents, and total number of agents, respectively. Note that the third quantity changes with time only in a multi-period setup.

**Parameter**

`ctmc` the call center CTMC.

```
public CallCenterCounters clone()
```

Returns a copy of this set of counters. This method creates a clone of each internal array in the set of counters.

```
public String toString()
```

Returns a string giving the number of counted arrivals, abandoned calls, and calls waiting less than the acceptable waiting time.

# CallCenterStat

Regroups tallies collecting observations obtained from independent replications of a simulation using a CTMC in the case of an individual period. After an object of this class is constructed, it can be initialized using `init` (`CallCenterCTMC`). While transitions are simulated, a set of counters represented by an instance of `CallCenterCounters` is updated. At the end of the replication, the set of counters is given to the `addObs` (`CallCenterCounters`, `double`) method of this class to collect the observations. Statistical collectors are regrouped into matrices concerning types of performance measures. The method `getMatrixOfStatProbes` (`PerformanceMeasureType`) can be used to obtain the matrix of statistical probes for a given type of performance measure.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public class CallCenterStat
```

## Constructor

```
public CallCenterStat (CallCenterCTMC ctmc, int ns, boolean keepObs)
```

Constructs a new set of statistical probes based on the CTMC model `ctmc`, with `ns` matrices of acceptable waiting times. The boolean `keepObs` indicates if observations are kept while collecting statistics.

### Parameters

`ctmc` the call center CTMC.

`ns` the number of matrices of acceptable waiting times.

`keepObs` determines if collected observations are kept.

## Methods

```
public Tally getStatNumFalseTransitions()
```

Returns the tally for statistics on the number of false transitions, also called self jumps.

**Returns** the tally for collecting the number of false transitions.

```
public Tally getStatNumTransitions()
```

Returns the tally for statistics on the number of simulated transitions.

**Returns** the tally for collecting the number of simulated transitions.

```
public PerformanceMeasureType[] getPerformanceMeasures()
```

Returns an array of types of performance measures for which statistics are collected by this object.

**Returns** the array of performance measures.

```
public Map<PerformanceMeasureType, MatrixOfStatProbes<?>>
getMatricesOfStatProbes()
```

Returns a map associating each supported type of performance measure with a matrix of statistical probes.

**Returns** the map associating the performance measures with matrices of statistical probes.

```
public MatrixOfStatProbes<?> getMatrixOfStatProbes (PerformanceMeasureType
                                                    m)
```

Returns the matrix of statistical probes corresponding to the performance measure type `m`.

**Parameter**

`m` the type of the performance measure.

**Returns** the associated matrix of statistical probes.

```
public void init (CallCenterCTMC ctmc)
```

Initializes the statistical probes in this object. The given CTMC model is used for initializing the arrival rates which are used for some statistics.

```
public void initLambda (CallCenterCTMC[] ctmcS)
```

Initializes the arrival rates used by this statistical collector by summing the arrival rates for all the CTMCs in the given array. The arrival rates are used by `addObs (CallCenterCounters, double)` to compute the expected number of arrivals during the considered period. By default, the arrival rates are initialized from the parameters of the CTMC given to the `init (CallCenterCTMC)` method. With this method, the arrival rates can be replaced with sums over several periods, to get the total arrival rate over all periods of an horizon.

**Parameter**

`ctmcS` the array of CTMCs.

```
public void addObs (CallCenterCounters counters, double periodDuration)
```

Adds new observations obtained from `counters` to the statistical probes managed by this object. The period duration `periodDuration` is used to multiply the arrival rates in order to get the expected number of arrivals over the considered period.

**Parameters**

`counters` the counters to get statistics from.

`periodDuration` the period duration.

```
public void formatReport (Map<String, Object> evalInfo, double
                          numExpectedTransitions)
```

Adds statistical information about the number of transitions to a map of evaluation information. Usually, the map is obtained using `ContactCenterEval.getEvalInfo()`, and generated information is displayed in reports produced by a simulator.



## Parameters

`evalInfo` the evaluation information.

`numExpectedTransitions` the expected number of transitions.

## RateChangeTransitions

Provides methods to determine transitions at which arrival rates change in order to have piecewise-constant arrival rates in the CTMC simulator. By default, a simulator based on an implementation of `CallCenterCTMC` uses a fixed arrival rate for each call type. Each replication,  $N(T)$  transitions are simulated. This class provides methods to determine how many transitions to simulate with each arrival rate, and generate a sequence of  $(t, k, \lambda_k)$  tuples. Each tuple indicates that the arrival rate for call type  $k$  changes to  $\lambda_k$  at transition number  $t$ .

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public class RateChangeTransitions
```

### Constructor

```
public RateChangeTransitions (CallCenter cc)
```

Constructs a new object for managing changes of arrival rates for the call center `cc`. This constructor collects the arrival rates and time of changes for any call type with an arrival process having piecewise-constant arrival rate. Any call type with an arrival process other than `PoissonArrivalProcessWithTimeIntervals` is ignored; the associated arrival rate will remain constant with time.

### Parameter

`cc` the call center from which to collect the information.

### Methods

```
public boolean hasChanges()
```

Returns `true` if and only if the arrival rate for at least one call type of the associated model changes with time.

```
public double[][] getTimeDist (double startingTime, double endingTime)
```

Returns a 2D array with one row per call type. Row  $k$  of the returned array is created by calling `getTimeDist (double[], double, double)` with the time of changes for call type  $k$ , and the given values of `startingTime` and `endingTime`.

### Parameters

`startingTime` the starting time  $a$ .

`endingTime` the ending time  $b$ .

**Returns** the 2D array of time distributions.

```
public static double[] getTimeDist (double[] times, double startingTime,
                                   double endingTime)
```

Constructs an array of length  $L + 1$  giving the proportion of interval  $[a, b)$  taken by each interval  $[t_{j-1}, t_j)$ . Let  $0 \leq t_0 < \dots < t_{L-1} < \infty$  be an increasing sequence of times. For each  $j = 0, \dots, L$ , the method sets the element  $j$  of the returned array to

$$\max(\min(t_j, b) - \max(t_{j-1}, a), 0) / (b - a).$$

Here,  $t_{-1} = 0$  and  $t_L = \infty$ . Each element of the returned array is in  $[0, 1]$ , and the sum of the values is 1. Element  $j$  of the returned array gives the proportion of transitions, in a uniformized CTMC, that needs to be simulated with arrival rate  $\lambda_j$  corresponding to time interval  $[t_j, t_{j+1})$ . The arrival rate is always 0 with time  $t < t_0$  and  $t \geq t_{L-1}$ .

### Parameters

**times** the sequence of times  $t_0, \dots, t_{L-1}$ .

**startingTime** the starting time  $a$ .

**endingTime** the ending time  $b$ .

**Returns** an array containing the proportion of total time for each interval.

```
public RateChangeInfo[] generateRateChanges (RandomStream stream, double[] []
                                             timeDist, int ntr)
```

Generates and returns a sequence of changes of arrival rates for simulating a uniformized CTMC with time-varying arrival rates for one or more call types. For each call type  $k$  for which **timeDist**[**k**] is non-null, this method generates a vector from the multinomial distribution giving the number of transitions spent with each arrival rate. The vectors are generated using random stream **stream**, and parameter  $n$  of the multinomials is **ntr**, and the vector of probabilities for call type **k** is given by **timeDist**[**k**]. The method then creates a sequence of objects representing change of arrival rates, and sorts these objects in increasing number of transition.

### Parameters

**stream** the random stream used to generate random vectors.

**timeDist** the time distribution for each call type.

**ntr** the total number of transitions.

**Returns** the sequence of changes of arrival rates.

# RateChangeInfo

Represents information about a change in the arrival rate.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class RateChangeInfo implements Comparable<RateChangeInfo>
```

## Constructor

```
public RateChangeInfo (int tr, int k, double rate)
```

Constructs a new object representing a change to arrival rate **rate** of call type **k** at transition number **tr**.

### Parameters

**tr** the transition number at which the rate is supposed to change.

**k** the affected call type.

**rate** the new arrival rate.

## Methods

```
public int getTransition()
```

Returns the transition number.

```
public int getK()
```

Returns the index of the call type.

```
public double getRate()
```

Returns the new arrival rate.

```
public int compareTo (RateChangeInfo o)
```

Compares this object with another object **o**. This comparison method orders objects using the transition number, and the call type for objects with the same transition number.

## AbstractCallCenterCTMCSim

Base class for simulators of call centers using a continuous-time Markov chain. Any instance of this class encapsulates a CTMC, statistical counters concerning replications, and statistical probes for collecting observations for the replications. The simulator is constructed from an instance of `CallCenterParams` which is usually obtained from a XML parameter file.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;

public abstract class AbstractCallCenterCTMCSim extends
    AbstractContactCenterSim
    implements ContactCenterSimWithObservations
```

### Constructor

```
public AbstractCallCenterCTMCSim (CallCenterParams ccParams,
    CTMCRepSimParams simParams, int mp)
    throws CallCenterCreationException,
    CTMCCreationException
```

Constructs a new simulator using call center parameters `ccParams`, experiment parameters `simParams`, and concentrating on main period `mp` of the model.

### Parameters

`ccParams` the parameters of the call center.

`simParams` the parameters of the experiment.

`mp` the index of the simulated main period.

### Throws

`CallCenterCreationException` if an error occurs during the creation of the call center.

`CTMCCreationException` if an exception occurs during the creation of the CTMC.

### Methods

```
public abstract void simulate (RandomStream stream1, double timeHorizon,
    int n)

public abstract double getNumExpectedTransitions()
```

## BasicCallCenterCTMCSim

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class BasicCallCenterCTMCSim extends AbstractCallCenterCTMCSim
```

## IntMCallCenterCTMCSim

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class IntMCallCenterCTMCSim extends AbstractCallCenterCTMCSim
```

## CallCenterStatMP

Provides a merged view of several single-period `CallCenterStat` instances. More specifically, for each supported type of performance measure, a matrix of tallies resulting from the concatenation of all single-column matrices concerning the same type of performance measure can be obtained through this object. An object of this class is constructed using an array of `CallCenterStat` instances. Any update to these sets of statistical probes are reflected on the merged view. Matrices of statistical probes regrouping statistics for each separate period, for a given type of performance measure, can be obtained by using the method `getMatrixOfStatProbes (PerformanceMeasureType)`.

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;
```

```
public class CallCenterStatMP
```

### Constructor

```
public CallCenterStatMP (CallCenterCTMC[] ctmc, CallCenterStat[] ccStat)
```

Constructs a new set of statistical probes from the given  $(P + 1)$ -dimensional arrays of CTMCs and period-specific statistical counters. The last element of `ctmc` corresponds to the CTMC for the wrap-up period while the last element of `ccStat` contains the collectors concerning the complete horizon. All other elements concern a specific main period.

#### Parameters

`ctmc` the array of CTMCs.

`ccStat` the array of statistical probes.

### Method

```
public void formatReport (Map<String, Object> evalInfo, double[]
                          numExpectedTransitions)
```

Adds statistical information about the number of transitions during each period to a map of evaluation information. Usually, the map is obtained using `ContactCenterEval.getEvalInfo()`, and generated information is displayed in reports produced by a simulator.

#### Parameters

`evalInfo` the evaluation information.

`numExpectedTransitions` the expected number of transitions, for each period.



# AbstractCallCenterCTMCSimMP

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public abstract class AbstractCallCenterCTMCSimMP extends  
    AbstractContactCenterSim  
    implements ContactCenterSimWithObservations
```

## Methods

```
public abstract void simulate (RandomStream stream1, int n)  
  
public abstract double[] getNumExpectedTransitions()
```

## BasicCallCenterCTMCSimMP

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc;  
  
public class BasicCallCenterCTMCSimMP extends AbstractCallCenterCTMCSimMP
```

## **Package** `umontreal.iro.lecuyer.contactcenters.ctmc.splitmerge`

Provides an implementation of a split and merge scheme for simulating a call center with different staffing vectors simultaneously.

## CallCenterCTMCSimSplit

---

```
package umontreal.iro.lecuyer.contactcenters.ctmc.splitmerge;  
  
public class CallCenterCTMCSimSplit extends AbstractCallCenterCTMCSim
```

## References

- [1] E. Buist, W. Chan, and P. L'Ecuyer. Speeding up call center simulation and optimization by Markov chain uniformization. In *Proceedings of the 2008 Winter Simulation Conference*, pages 1652–1660, Piscataway, New-Jersey, 2008. IEEE Press.