

**Desenvolvimento e Teste de uma *engine* de simulação de modelos discretos estocásticos**  
**(orientados a eventos e processos) v. 0.91 maio/21**

Este trabalho consiste na elaboração de uma *engine* (motor) que permita a definição e simulação de um modelo discreto estocástico, através de uma API (apresentada a seguir) bem como do teste desta *engine* por meio do seu emprego no desenvolvimento de um modelo, assim como sua simulação e validação.

**1. Definição de uma API de simulação** (essa é uma proposta mínima e flexível: pode ser modificada/adaptada)

a. Classes

i. **Entity**<sup>1</sup>

- Propriedades
  - name: *string*
  - id: *integer* ☞ atribuído pelo *Scheduler*
  - creationTime: *double* ☞ atribuído pelo *Scheduler*
  - priority: *integer* ☞ sem prioridade: -1 (0: + alta e 255: + baixa)
  - petriNet: *PetriNet*
- Métodos
  - Entity(name)
  - Entity(name, petriNet)
  - getId(): *integer*
  - getPriority(): *integer* e setPriority(priority)
  - getTimeSinceCreation(): *double*
  - getSets(): *EntitySet List* ☞ retorna lista de *EntitySets* nas quais a entidade está inserida
  - setPetriNet(*PetriNet*) e getPetriNet(): *PetriNet*

ii. **Process**

- Propriedades
  - name: *string*
  - processId: *integer* ☞ atribuído pelo *Scheduler*
  - duration: *double* ☞ é a duração (temporal) do processo; seu valor é calculado no início da execução da execução desta instância;
  - active: *boolean* ☞ processo pode ser (des)habilitado;
- Métodos
  - Process(name, duration)
  - getDuration(): *double* e setDuration(duration)
  - isActive(): *boolean*
  - activate(*boolean*)

iii. **Event**

- Propriedades
  - name: *string*
  - eventId: *integer* ☞ atribuído pelo *Scheduler*
- Métodos
  - Event(name)

iv. **EntitySet**

- Propriedades
  - name: *string*
  - id: *integer* ☞ atribuído pelo *Scheduler*
  - mode: FIFO/LIFO/Priority based/None ☞ no mode None, método remove() sorteia qual entidade será removida; neste mode, é + interessante empregar removeById(id)
  - size: *integer*

<sup>1</sup> As entidades criadas no modelo e instanciadas na simulação são derivadas (estendem) a classe *Entity* e acrescentam propriedades e métodos específicos da entidade; a mesma idéia se aplica as demais classes...

- `maxPossibleSize`: *integer* (zero for not size limited) ☞ tamanho máximo que o conjunto pode ter
- Métodos
  - `EntitySet(name, mode, maxPossibleSize)`
  - `getMode()`: *mode* e `setMode(mode)`
  - `insert(Entity)` ☞ similar a `enqueue` ou `push...`
  - `remove()`: *Entity* ☞ similar a `dequeue` ou `pop...`
  - `removeById(id)`: *Entity*
  - `isEmpty()`: *boolean* e `isFull()`: *boolean*
  - `findEntity(id)`: *Entity* ☞ retorna referência para uma *Entity*, se esta estiver presente nesta *EntitySet*

*coleta de estatísticas*

- `averageSize()`: *double* ☞ retorna quantidade média de entidades no conjunto
- `getSize()`: *integer* ☞ retorna quantidade de entidades presentes no conjunto no momento
- `getMaxPossibleSize()`: *integer* e `setMaxPossibleSize(size)`
- `averageTimeInSet()`: *double* ☞ retorna tempo médio que as entidades permaneceram neste conjunto
- `maxTimeInSet()`: *double* ☞ retorna tempo mais longo que uma entidade permaneceu neste conjunto
- `startLog(timeGap)` ☞ dispara a coleta (log) do tamanho do conjunto; esta coleta é realizada a cada `timeGap` unidades de tempo
- `stopLog()`
- `getLog()` ☞ retorna uma lista contendo o log deste *Resource* até o momento; cada elemento desta lista é um par <`tempoAbsoluto`, `tamanhoConjunto`>

**v. Resource**

- Propriedades
  - `name`: *string*
  - `id`: *integer* ☞ atribuído pelo *Scheduler*
  - `quantity`: *integer* ☞ quantidade de recursos disponíveis
- Métodos
  - `Resource(name, quantity)`
  - `allocate(quantity)`: *boolean* ☞ *true* se conseguiu alocar os recursos
  - `release(quantity)`

*coleta de estatísticas*

- `allocationRate()`: *double* ☞ percentual do tempo (em relação ao tempo total simulado) em que estes recursos foram alocados<sup>2</sup>
- `averageAllocation()`: *double* ☞ quantidade média destes recursos que foram alocados (em relação ao tempo total simulado)<sup>3</sup>

**vi. Scheduler**

- Propriedades
  - `time`: *double*
- Métodos
  - `getTime()`: *double* ☞ retorna o tempo atual do modelo

*disparo de eventos e processos*

- `scheduleNow(Event)`
- `scheduleIn(Event, timeToEvent)`
- `scheduleAt(Event, absoluteTime)`
- `startProcessNow(processId)`
- `startProcessIn(processId, timeToStart)`
- `startProcessAt(processId, absoluteTime)`
- `waitFor(time)` ☞ se a abordagem para especificação da passagem de tempo nos processos for explícita

<sup>2</sup> Ex.: em 10 seg, durante 2 seg houveram recursos alocados (essa taxa seria então de  $2/10 = 0.2$  recursos/seg.)

<sup>3</sup> Ex.: ao longo de 10 seg., houveram 5 unidades deste recurso alocado (esta quant. média ficaria  $5/10 = 0.5$  recursos)

*controlando tempo de execução*

- `simulate` ☞ executa até esgotar o modelo, isto é, até a engine não ter mais nada para processar (FEL vazia, i.e., lista de eventos futuros vazia)
- `simulateOneStep` ☞ executa somente uma primitiva da API e interrompe execução; por ex.: dispara um evento e para; insere numa fila e para, etc.
- `simulateBy(duration)`
- `simulateUntil(absoluteTime)`

*criação, destruição e acesso para componentes*

- `createEntity(Entity)` ☞ instancia nova *Entity* e `destroyEntity(id)`
- `getEntity(id): Entity` ☞ retorna referência para instância de *Entity*
- `createResource(name, quantity): id`
- `getResource(id): Resource` ☞ retorna referência para instância de *Resource*
- `createProcess(name, duration): processId`
- `getProcess(processId): Process` ☞ retorna referência para instância de *Process*
- `createEvent(name): eventId`
- `getEvent(eventId): Event` ☞ retorna referência para instância de *Event*
- `createEntitySet(name, mode, maxPossibleSize): id`
- `getEntitySet(id): EntitySet` ☞ retorna referência para instância de *EntitySet*

*random variates*

- `uniform(minValue, maxValue): double`
- `exponential(meanValue): double`
- `normal(meanValue, stdDeviationValue): double`

*coleta de estatísticas*

- `getEntityTotalQuantity(): integer` ☞ retorna quantidade de entidades criadas até o momento
- `getEntityTotalQuantity(name): integer` ☞ retorna quantidade de entidades criadas cujo nome corresponde ao parâmetro, até o momento
- `averageTimeInModel(): double` ☞ retorna o tempo médio que as entidades permanecem no modelo, desde sua criação até sua destruição
- `maxEntitiesPresent(): integer` ☞ retorna o número máximo de entidades presentes no modelo até o momento

- b. A construção de um modelo pode ser feita:
  - i. de maneira interativa, através de
  - ii. através da leitura de um arquivo onde está a definição do modelo
  - iii. por meio de chamadas dos métodos da API em uma classe de teste
- c. A execução do modelo deverá suportar dois modos de execução:
  - i. Passo a passo
  - ii. Execução por uma fatia de tempo

## 2. O teste da engine através da validação do modelo de teste (modelo Restaurante)

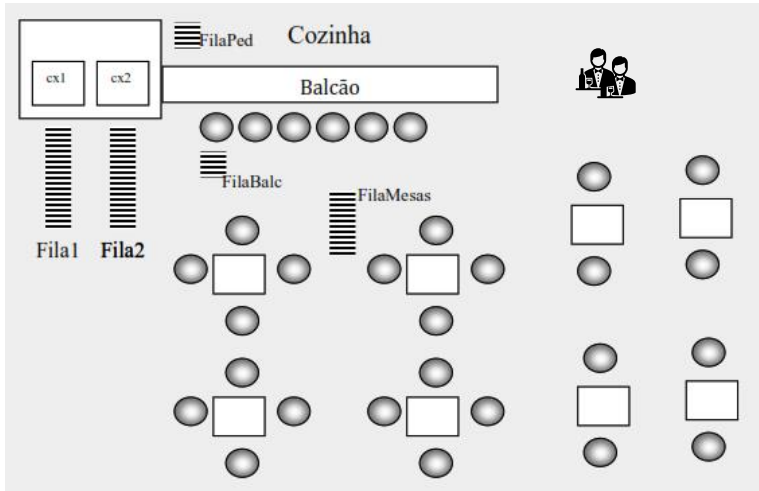
O processo de teste da *engine* será realizado através do emprego deste motor para a construção e simulação do modelo do restaurante (<https://www.moodle.unisinos.br/mod/resource/view.php?id=686290>).

A versão Anylogic deste modelo desempenhará o papel de sistema, sendo seu desempenho (resultados gerados pela sua simulação) comparados com o desempenho do modelo que emprega a *engine*.

Esta comparação (isto é, a **validação** do modelo, e consequentemente da *engine*) será feita através do processo de **análise estatística** (<https://www.moodle.unisinos.br/mod/resource/view.php?id=686434>). Para a realização desta análise pode-se comparar (entre o modelo Anylogic, que faz o papel de sistema, e o modelo que emprega o motor) o **tamanho das filas** (nos caixas, dos pedidos na cozinha, clientes nas filas das mesas e balcão) e o **tempo de espera** nas mesmas ao longo da simulação (ou de uma janela de tempo).

A abordagem orientada a Eventos e a Processos é baseada no documento visto em aula: <https://www.moodle.unisinos.br/mod/resource/view.php?id=769450>.

Em relação ao modelo original do Restaurante, será acrescentada a esta versão uma entidade adicional que é o **garçom**, cujo comportamento será modelado através de **Rede de Petri**.



Em relação a nova entidade garçom:

- É responsável pelo transporte da refeição da cozinha para a mesa ou balcão;
- Antes do grupo de clientes sentar a mesa ou no balcão, o garçom realiza a higienização da mesa ou balcão;
- De tempos em tempos um dos caixas vai ao banheiro, e neste momento é substituído por um dos garçons.

Este conjunto de comportamentos é implementado nesta entidade através de uma Rede de Petri, empregando-se a engine desenvolvida no trabalho do GA.

3. O algoritmo de **scheduling de eventos e processos** pode ser baseado no artigo *An Introduction to Discrete-Event Modeling and Simulation* disponível em <https://www.moodle.unisinos.br/mod/resource/view.php?id=686247>

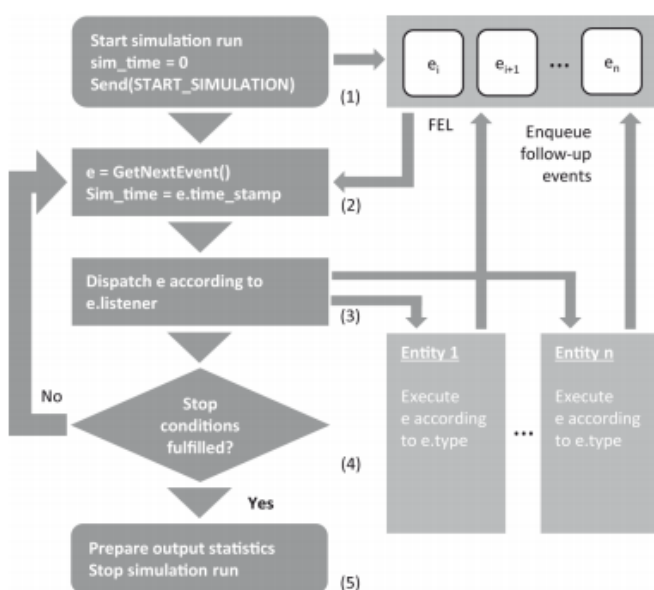


Diagrama do algoritmo de **scheduling**, baseado na consulta e consumo da FEL (Future Events List)

4. **O processo de validação**, isto é, da análise estatística pode ser realizado através do uso das planilhas vistas em sala de aula. Estas podem ser postadas juntamente com o software desenvolvido, ou todo material pode ser disponibilizado para acesso online.