Projeto e construção de uma ferramenta de simulação

Andrew M. Silva*, Estela M. Vilas Boas†

Universidade Federal da Fronteira Sul (UFFS) – 2021

Resumo

Através do estudo da teoria no componente curricular optativo de Modelagem e Simulação, este artigo relata o desenvolvimento do trabalho prático, o qual possui como objetivo demonstrar e trabalhar de forma prática a compreensão dos processos e métodos de simulação computacional. O projeto prático desenvolvido consiste na implementação de uma ferramenta de simulação baseada no conceito de 'centros de serviço', nas ferramentas utilizadas e nos conhecimentos adquiridos em sala de aula. O objetivo é que a ferramenta seja genérica o suficiente para executar diversos tipos de simulações diferentes. O desenvolvimento desta ferramenta foi de grande importância para aumentar a clareza quanto ao tema estudado.

Palavras-chaves: Simulação computacional, ferramenta de simulação, centros de serviço, relatórios de simulação.

Introdução

Simulação é utilizada em diferentes campos e faz contribuições significativas onde é aplicada (TAYLOR et al., 2013). Como todo campo de pesquisa, modelagem e simulação também possuem diversos desafios que até hoje são enfrentados pelos pesquisadores de forma a evoluir as técnicas já existentes e propor novas (TAYLOR et al., 2012).

*andrewsaxx@gmail.com

†estelavilasboas01@gmail.com

Modelagem é definida como uma representação matemática, física, lógica de um sistema ou processo. O grande desafio da modelagem é sua representação. Por sua vez, simulação é definida como uma maneira de implementar um modelo ao longo de determinado tempo, tendo como seu grande desafio a infraestrutura. (TAYLOR et al., 2013)

Existem as mais diferentes aplicações para simulação, todas com um grande potencial de uso e com combinações a serem realizadas e exploradas (STRASSBURGER, 2006). Sendo de grande ajuda para compreender padrões de um sistema e, para o caso de simulações de centros de serviço, pode contribuir para a delimitação das necessidades de um estabelecimento, visando melhorias como o fluxo de atendimento.

Portanto, é possível perceber o quão significativa é a existência e utilização de modelagem e simulação. Visto isso, este artigo possui como objetivo explicar brevemente o que é, como são e apresentar uma ferramenta de modelagem em simulação discreta .

1 Referencial teórico

A simulação computacional possui métodos desenvolvidos desde a década de 1950, sendo baseada em uma abordagem experimental que pode servir de apoio para tomadas de decisão (WOOLFSON; PERT, 1999). Ou seja, no conceito de centros de serviço, a ideia é simular mudanças, políticas e/ou regras antes de serem aplicadas no mundo real e analisar seus efeitos.

As mudanças e problemas de um sistema poderiam ser facilmente testados no mundo real de maneira controlada, porém essa abordagem se mostra problemática e custosa em diversos aspectos (PIDD, 1994). Uma opção seria o desenvolvimento de um modelo matemático do sistema para ser estudado. Todavia, devida a complexidade que um modelo matemático entregaria, a simulação do sistema em um modelo computacional é apresentada como a solução mais prática.

Um modelo de base computacional é o processo de simplificação e abstração (PIDD, 1994). Nele é possível isolar entidades e fatores cruciais para o funcionamento do sistema com base nos dados adquiridos. Entretanto, existem sistemas que não podem ser processados em uma simulação computacional. Para que um sistema possa ser adequado para uma simulação, deve-se conter as seguintes características:

- Dinâmico - deve ter uma variação entre os fatores que o compõem;
- Interativo - deve conter componentes que interagem entre si e que sua interação produza efeitos diferentes no sistema (PIDD, 1994);
- Complexo - deve conter muitos fatores dinâmicos e interativos, e que sua dinâmica individual seja passível de análise.

É possível classificar uma simulação computacional em duas classes distintas, sendo elas simulação contínua e discreta.

A modelagem em uma simulação contínua caracteriza-se por conter variáveis dependentes que mudam de forma contínua no tempo simulacional. Para definição dos comportamentos da simulação, aplica-se um conjunto de equações que estabelecem a dinâmica do modelo com as relações entre suas variáveis de estado.

Já na simulação discreta, um modelo computacional é composto pelo comportamento de objetos que representam os atores da simulação, chamados entidades (PIDD, 1994). O comportamento de cada entidade segue um padrão lógico determinada especificamente para a simulação do sistema em que está inserida, considerando o objeto que representa (pessoas, veículos, etc) e suas características.

Modela-se o comportamento de uma entidade a partir de uma série de eventos, onde um evento é estabelecido em um tempo simulacional que a entidade muda de estado. Os intervalos entre os eventos não são sempre previsíveis, dependendo das características da simulação (PIDD, 1994). É necessário que os intervalos sejam determinados estocasticamente através de uma amostragem.

Para a criação de um modelo de simulação discreta, deve-se estabelecer as mudanças nos estados, definir e descrever as atividades/eventos, e definir o processo seguido pelas entidades do sistema.

Para o desenvolvimento da ferramenta proposta na disciplina de Modelagem e Simulação, será utilizada a modelagem de simulação discreta. A partir da análise do comportamento das entidades em tempo simulacional, será possível definir a conclusão do estudo de caso da clínica médica hipotética e apresentar seus resultados.

2 Apresentação da ferramenta

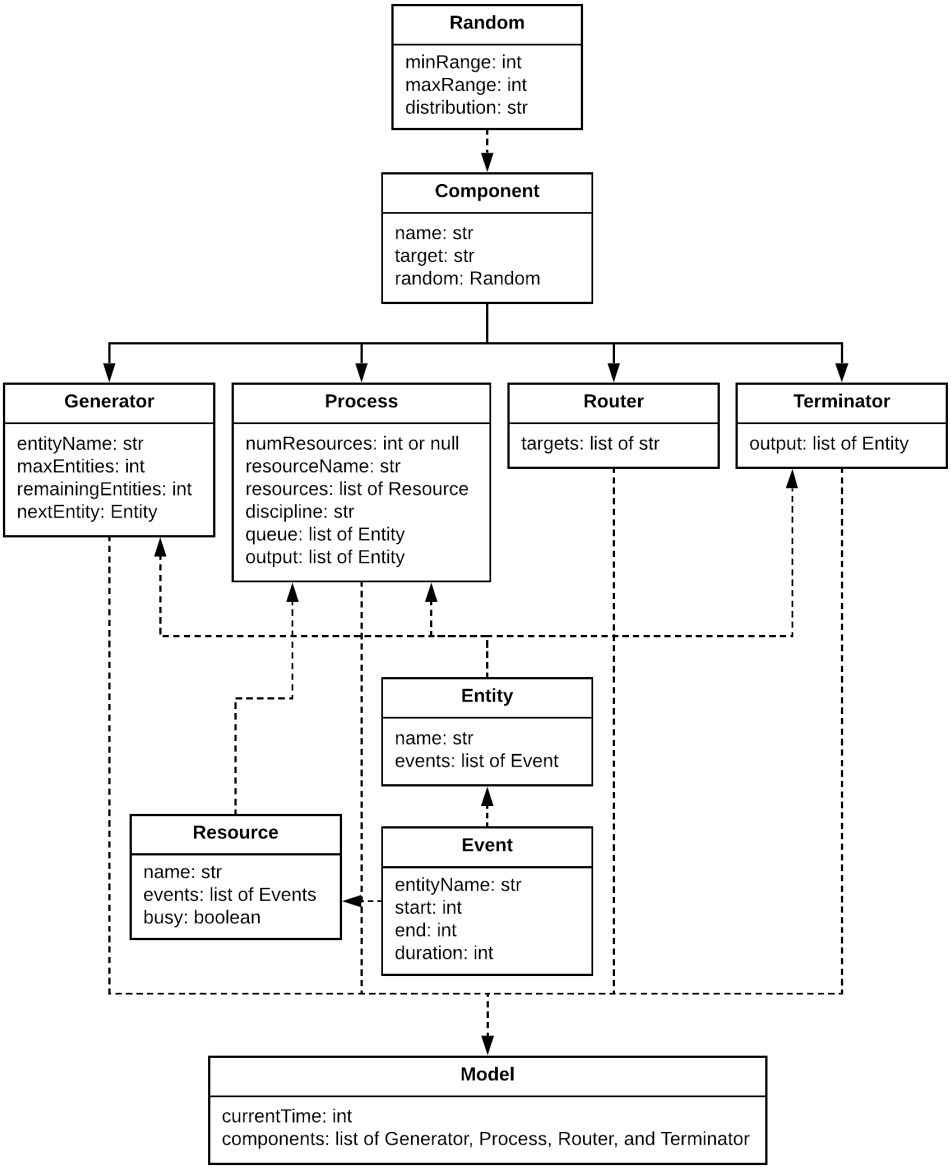
A ferramenta de simulação implementada é chamada de *SimTool*, o qual é baseada no conceito modelos simulacionais genéricos que utilizam geradores de entidades temporárias, roteadores e centros de serviços com filas de espera e múltiplos servidores. Seu objetivo é permitir ao usuário montar seu próprio modelo simulacional utilizando os componentes disponíveis para simular o comportamento de uma determinada situação de estudo.

A implementação foi realizada utilizando orientação a objetos para que cada objeto da simulação fosse auto-contido e funcionasse por si mesmo, sem necessariamente depender de outras áreas do algoritmo. A Figura 1 apresenta o diagrama de classes empregado para a implementação, o qual mostrou-se complexo. Portanto, as seções seguintes irão dividir este diagrama e explicar cada classe separadamente para que tudo seja devidamente compreendido.

2.1 *Events*

A classe *Event* possui como objetivo representar um evento ocorrido em um determinado momento da simulação. Conforme mostrado na Figura 2, esta classe possui quatro atributos: *entityName*, *start*, *end* e *duration*. O atributo *entityName* armazena o nome da *Entity* no qual aquele evento está relacionado, enquanto os demais atributos lidam com o tempo simulacional em que o evento foi disparado.

Figura 1 – Diagrama de classes da ferramenta implementada

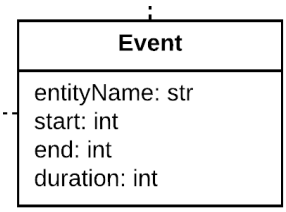


O atributo *start* armazena o momento em que o evento foi iniciado, enquanto o atributo *end* armazena o momento em que o evento foi ou será concluído. Por fim, o atributo *duration* armazena a duração do evento, o qual pode ser facilmente calculado como sendo a subtração entre *end* e *start*.

2.2 *Entities*

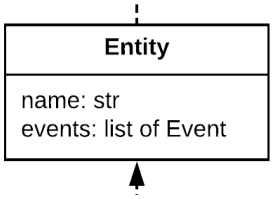
As entidades temporárias que percorrem o modelo e disparam eventos ao serem atendidas pelos centros de serviços são representadas pela classe *Entity*. Conforme mostrado na Figura 3, esta classe possui apenas dois atributos: *name* e *events*. O atributo *name* armazena o nome de uma *Entity*, o qual deve ser um identificador

Figura 2 – Definição da classe *Event*



único.

Figura 3 – Definição da classe *Entity*

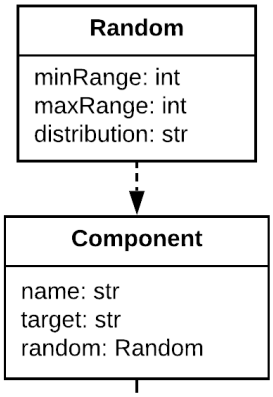


O atributo *events* armazena uma lista contendo um histórico de todos os eventos ocorrido com a *Entity*, sendo estes eventos intâncias da classe *Event*. Quando uma *Entity* é criada, seu primeiro evento é automaticamente criado. Este evento inicial é seu evento de "nascimento", que possui *duration* igual a zero e *start* e *end* iguais ao momento em que a *Entity* foi gerada ou criada.

2.3 *Components*

Todos os componentes que podem ser criados em uma simulação recebem como herança a classe *Component*. Nela estão os atributos e métodos que todos os componentes possuem, como nome, alvo, e gerador de números aleatórios. A Figura 4 mostra mais nitidamente a definição básica das classes *Component* e *Random*.

Figura 4 – Definição das classes *Random* e *Component*



O atributo *name* armazena o nome do componente, que deve ser um identificador único. O atributo *target* armazena o nome de um componente V ligado a um componente C, criando uma relação processual entre esses dois componentes.

Por exemplo, o componente “Recepção” pode estar conectado com o componente “Médicos”, podendo enviar pacientes para consulta após terem sido atendidos.

Além disso, o atributo *random* armazena uma instância de um objeto Random, que possui como função gerar números aleatórios com base num mínimo, num máximo e numa distribuição de aleatoriedade, isto é, *minRange*, *maxRange* e *distribution*, respectivamente. O mínimo e o máximo devem ser valores inteiros, enquanto a distribuição deve ser definida como “uniform” ou “normal”, detonando uma distribuição uniforme ou normal de probabilidades, respectivamente.

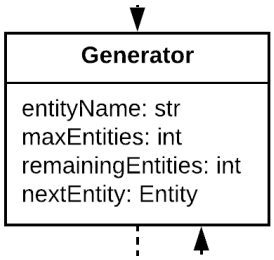
Componentes que não possuem *target* e/ou não precisam gerar número aleatórios podem deixar esses atributos com valores nulos. Existem quatro tipos de componentes: *Generators*, *Processes*, *Routers* e *Terminators*. Cada componente possui suas próprias características, mas todos herdam o que é definido na classe *Component*. As sub-seções a seguir explicam o funcionamento de cada um dos tipos de componentes.

2.3.1 *Generators*

Um componente *Generator* possui como objetivo gerar um determinado número de *Entities* e enviá-las para a simulação. Cada *Entity* gerada possui um intervalo de tempo que é definido aleatoriamente utilizando o objeto *Random* herdado da classe *Component*.

Além dos outros atributos herdados da classe *Component*, um *Generator* possui seu próprio conjunto de atributos, conforme apresentado na Figura 5. O atributo *entityName* armazena o nome que o *Generator* dará às *Entities* geradas, o quais serão nomes únicos, pois serão acompanhados de um número de identificação. Por exemplo, se um *Generator* gerar *Entities* com o nome “Pessoa”, as gerações terão o nome de “Pessoa 0”, “Pessoa 1”, “Pessoa 2” e assim por diante.

Figura 5 – Definição da classe *Generator*

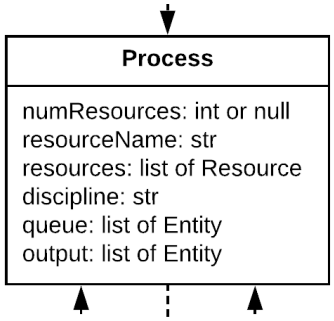


O atributo *maxEntities* define o número *Entities* que o *Generator* vai gerar. Enquanto houverem *remainingEntities*, isto é, *Entities* a serem geradas, a simulação não termina, pois espera que todas as *Entities* sejam devidamente geradas e processadas. Por fim, o atributo *nextEntity* armazena a próxima instância da classe *Entity* que será enviada para a simulação quando o tempo simulacional apropriado for alcançado.

2.3.2 *Processes*

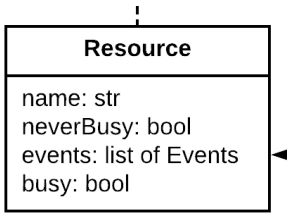
Todo centro de serviço é representado pela classe *Process*, que está sendo definida conforme apresentado na Figura 6. O objetivo da classe *Process* é receber *Entities*, processá-las, disparar *Events* e realizar o controle da fila e dos recursos (servidores) disponíveis.

Figura 6 – Definição da classe *Process*



Para que um *Process* funcione, ele depende da classe *Resource*, que possui o papel de representar os recursos que o *Process* tem à disposição. Conforme apresentado pela Figura 7, um *Resource* possui quatro atributos. O atributo *name* armazena o nome do recurso, que não precisa ser necessariamente único.

Figura 7 – Definição da classe *Resource*



O atributo *busy* define se o *Resource* em questão está ocupado com uma *Entity* ou se está disponível para realizar outros atendimentos. No entanto, o atributo *neverBusy* pode ser utilizado para definir o *Resource* nunca fica ocupado e, portanto, sempre está disponível para realizar atendimentos. Isso é útil para simular o comportamento de um centro de serviço com infinitos servidores, pois, ao invés de criar infinitos servidores de acordo com a demanda, basta manter apenas um servidor que nunca fica ocupado e pode atender a todas as solicitações imediatamente.

Além disso, o atributo *events* armazena um histórico de todos os eventos que ocorreram naquele *Resource*. Isso possibilita calcular algumas métricas e a verificar se ele está ocupado. Por exemplo, se o último *Event* disparado pelo *Resource* tiver seu tempo de término igual a 15 e o tempo simulado atual for 14, quer dizer que o atendimento relacionado a este *Event* ainda não foi concluído, mas será no tempo simulado seguinte.

Um *Process* depende muito de seus *Resources* para funcionar corretamente. Portanto, o atributo *numResources* é essencial para definir quantos *Resources* um *Process* possui. Este atributo deve ser um valor inteiro maior que zero para ter um

ou mais *Resources* ou pode ser um valor nulo para que o *Process* se comporte com infinitos *Resources*.

Os *Resources* de um *Process* são nomeados com base no atributo *resourceName*. Por exemplo, um *Process* com *resourceName* igual a "Atendente" nomeará seus *Resources* como "Atendente 0", "Atendente 1" e assim por diante. Isso é útil para diferenciar as ações dos *Resources* durante a simulação.

O atributo *resources* armazena uma lista contendo todos os *Resources* que o *Process* possui. No caso de um *Process* com infinitos *Resources*, essa lista possuirá apenas um *Resource* com o atributo *neverBusy* ativado. Por consequência, este único *Resource* atenderá todas as *Entities* que surgirem e nunca ficará ocupada.

Se o *Process* possuir um número finito de *Resources*, eles ficarão ocupado quando estiverem atendendo *Entities*. Por consequência, pode haver espera na fila quando todos os *Resources* disponíveis estão ocupados. Por conta disso, o atributo *queue* armazena uma lista de todos as *Entities* que estão esperando para serem atendidas, gerando algumas estatísticas sobre essa espera.

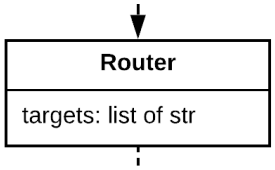
Para que uma *Entity* saia da fila para ser atendida, é necessário que haja ao menos um *Resource* disponível. A disciplina da fila é definida pelo atributo *discipline*, que pode ser "FCFS" ou "LCFS", que significam First Come First Served e Last Come First Served, respectivamente. Na disciplina de fila FCFS, a próxima *Entity* a ser atendida sempre a que tiver chegado na fila primeiro, enquanto a disciplina LSFC atende sempre as *Entities* que chegaram na fila por último.

A duração dos atendimento de cada *Entity* é definido com base num sorteio realizado pelo objeto *Random* herdado da classe *Component*. Finalmente, o atributo *output* armazena uma lista das *Entities* que já foram atendidas. Quando o tempo simulado apropriado é alcançado, essas *Entities* são repassadas para os componentes seguintes do modelo.

2.3.3 *Routers*

A classe *Router* possui como objetivo representar componentes roteadores, isto é, que definem caminhos entre múltiplos componentes do modelo simulacional. O único atributo que um *Router* possui de diferente é o atributo *targets*, que armazena uma lista com os nomes dos componentes que a saída do *Router* está conectada.

Figura 8 – Definição da classe *Router*



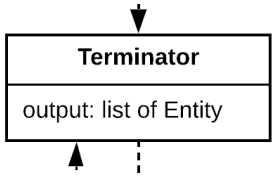
Ao obter receber uma *Entity*, o *Router* utilizará o objeto *Random* para sortear para qual dos possíveis destinos a *Entity* será roteada. Para realizar o sorteio, não é necessário definir o valor mínimo e máximo a ser considerado no sorteio, pois, como ele é feito com base na lista de possíveis destinos, o mínimo é a posição zero da lista

e o máximo é a última posição da lista. As distribuições de aleatoriedade continuam funcionando normalmente para *Routers*.

2.3.4 *Terminators*

A classe *Terminator* representa o componente de saída da simulação e possui apenas um atributo, conforme apresentado pela Figura 9. O atributo *output* armazena uma lista de todas as *Entities* que concluíram a simulação em um determinado *Terminator*. Isso pode ser útil para calcular algumas métricas relacionadas às *Entities* e seus *Terminators*.

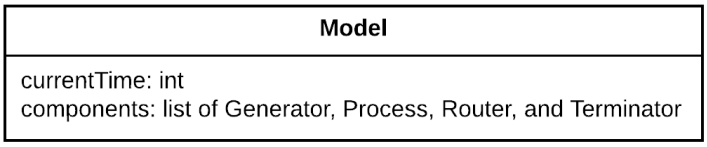
Figura 9 – Definição da classe *Terminator*



2.4 *Model*

A classe principal do *SimTool* é a classe *Model*, a qual é responsável por definir o modelo simulacional, conter todos os componentes e realizar a simulação. Conforme apresentado pela Figura 10, esta classe não possui muitos atributos. O atributo *currentTime* armazena o tempo simulado de uma simulação em andamento, enquanto o atributo *components* armazena uma lista contendo todos os componentes que o modelo possui, respeitando restrições de nomes únicos.

Figura 10 – Definição da classe *Model*



2.4.1 Criando e carregando um modelo

A parte mais importante da classe *Model* são seus métodos, que controlam diferentes aspectos do modelo. Existem duas formas de se criar um modelo: criação via instanciação e chamada de métodos ou a criação por carregamento de arquivo de configuração.

Para criar um modelo via instanciação e chamada de métodos, basta instanciar o modelo e utilizar os métodos *createGenerator()*, *createProcess()*, *createRouter()* e *createTerminator()* para criar os componentes que compõem o modelo. Cada um desses métodos recebem os parâmetros de cada componente e os criam para que possam ser utilizados para rodar a simulação.

É possível salvar o arquivo de configuração de um modelo já instanciado em um `.csv` gerado automaticamente ao executar o método *save()*. Para carregar um

modelo a partir de um arquivo de configuração, basta chamar o método *load()*, que terá o papel de carregar o arquivo `.csv`, criar os componentes utilizando os métodos mencionados anteriormente e retornar um *Model* já instanciado e pronto para uso.

Cada linha do arquivo `.csv` corresponde a um componente do modelo e cada coluna representa um atributo deste componente. No total, há onze colunas, cada uma com sua própria importância. Cada coluna está definia a seguir:

1. *type*: Define o tipo do componente pela sua letra inicial (G, P, R ou T).
2. *name*: Define o nome do componente, que deve ser único.
3. *target*: Define o nome do componente conectado a ele, podendo ser um valor nulo ou, no caso de *Routers*, múltiplos nomes separados por "$$".
4. *min_range*: Define o valor mínimo a ser gerado pelo objeto *Random*. Componentes como *Routers* e *Terminators* devem deixar esta coluna com um valor nulo.
5. *max_range*: Define o valor mínimo a ser gerado pelo objeto *Random*. Componentes *Routers* e *Terminators* devem deixar esta coluna com um valor nulo.
6. *distribution*: Define a distribuição de probabilidade a ser utilizada pelo *Random*. Componentes *Terminator* devem deixar esta coluna com um valor nulo.
7. *max_entities*: Define o número de *Entities* que um *Generator* vai gerar. Demais componentes devem deixar esta coluna com um valor nulo.
8. *entity_name*: Define o nome das *Entities* geradas por um *Generator*. Demais componentes devem deixar esta coluna com um valor nulo.
9. *num_resources*: Define o número de *Resources* um *Process* possui, podendo ser um número ou um valor nulo (para infinitos *Resources*). Demais componentes devem deixar esta coluna com um valor nulo.
10. *resource_name*: Define o nome que os *Resources* de um *Process* possuem. Demais componentes devem deixar esta coluna com um valor nulo.
11. *discipline*: Define a disciplinca que a fila de um *Process* utilizará. Demais componentes devem deixar esta coluna com um valor nulo.

Essa estrutura empregada no arquivo de configuração é suficiente para configurar um modelo simulacional completamente, embora não seja tão fácil e agradável de ser construída manualmente. Portanto, recomenda-se utilizar os métodos *save()* e *load()* para gerar e carregar o modelo facilmente.

2.4.2 Realizando a simulação

Ao ter um *Model* instanciado e com componentes já criados ou carregados, é possível realizar a simulação. Isso é feito através do método *run()*, utiliza uma estratégia de tempo simulado incremental. Em outras palavras, a simulação se inicia no tempo simulado zero e aumenta em um para cada iteração realizada. A execução de cada iteração possui duas etapas: a execução dos *Generators* e a execução dos *Processes*.

Na execução dos *Generators* é verificado se alguma *Entity* foi gerada e, se for o caso, ela é roteada para o *target* do *Generator* em questão. Se nenhuma *Entity* for gerada, esta etapa é ignorada. O método que realiza o roteamento das *Entities* para seus respectivos destinos é recursivo, pois considera a possibilidade de existência de um ou mais *Routers* no meio do caminho. Isso garante que as *Entities* alcancem seus destinos independente de quantos componentes precisem percorrer para tal.

Na execução dos *Processes* é verificado se algum deles possui *Entities* em sua *output*, sendo necessário roteá-las aos seus respectivos *targets*. Feito isso, cada *Process* é executado, realizando todas as suas funções internas, como escalonamento de fila, controle de *Resources*, etc. Após isso, o tempo simulado é incrementado e uma nova iteração se inicia.

Para que a simulação se encerre, foi definido um critério de parada. Segundo o critério, a simulação se encerrará apenas se os *Generators* gerarem todos as *Entities* que deveriam gerar e se os *Processes* estiverem em posse de mais nenhuma *Entity*, seja na *queue* ou na *output*. Por um lado, isso garante que a simulação não se encerre prematuramente, antes que tudo seja concluído. No entanto, isso pode gerar uma execução infinita se os as ligações entre os componentes do modelo gerarem *loops*.

Além disso, para tornar possível a reprodutibilidade de diferentes experimentos, o método *run()* possui um parâmetro opcional chamado *random_state* que pode ser utilizado para definir a semente de aleatoriedade do algoritmo. Isso significa que, sempre que uma simulação for executada com uma semente definida, dos resultados sempre serão os mesmos, permitindo diversas análises comparativas. Sem a semente, execuções de um mesmo modelo sempre posuirão resultados diferentes e aleatórios.

2.4.3 Resultados de uma simulação

Após a realização de uma simulação, seus resultados são armazenados em dois arquivos distintos. O arquivo `simulation.txt` armazena o registro de todos os eventos que aconteceram durante a simulação, em ordem de ocorrência e com informações sobre os atores envolvidos em cada evento. É possível utilizar este registro para analisar os eventos ocorridos e obter algumas informações interessantes.

O arquivo `reports.json` armazena relatório sobre a simulação realizada. Este relatório consiste em um conjunto de estatísticas calculadas com base nos eventos ocorridos, calculando diversas métricas interessantes. Ao todo, são calculadas quatorze métricas para cada *Process*:

1. *resourceIdleTime*: Tempo de ociosidade de cada *Resource*;

2. *minIdleTime*: Menor tempo de ociosidade dos *Resources*.
3. *meanIdleTime*: Tempo médio de ociosidade dos *Resources*.
4. *minDurationTime*: Menor tempo de duração dos atendimentos.
5. *meanDurationTime*: Tempo médio de duração dos atendimentos.
6. *maxDurationTime*: Menor tempo de duração dos atendimentos.
7. *immediateProcessing*: Número de *Entities* que foram atendidas imediatamente (não esperaram na fila).
8. *minWaitingTime*: Menor tempo de espera na fila.
9. *meanWaitingTime*: Tempo médio de espera na fila.
10. *maxWaitingTime*: Maior tempo de espera na fila.
11. *minWaitingCount*: Menor número de *Entities* esperando na fila ao mesmo tempo.
12. *meanWaitingCount*: Número médio de *Entities* esperando na fila ao mesmo tempo.
13. *maxWaitingCount*: Maior número de *Entities* esperando na fila ao mesmo tempo.

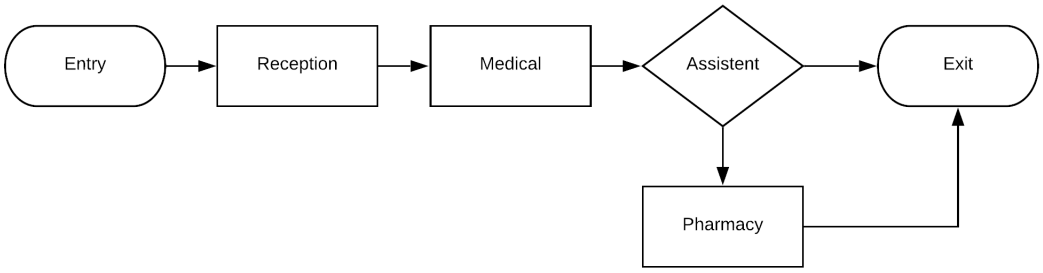
Processes com um número infinito de *Resources* possuem valores nulos em muitas dessas métricas por não possuírem dados para serem analisados, visto que não existe espera na fila e tempo de ociosidade. No entanto, *Processes* com um número finito de *Resources* podem apresentar estatísticas muito interessantes para uma possível análise.

3 Considerações finais

A implementação do *SimTool* foi realizada gradualmente e foi muito útil para entendimento dos processos que envolvem uma simulação. Foi uma tarefa desafiadora, mas os resultados atingidos foram muito positivos. Foram feitos diversos testes utilizando diferentes modelos e o mais utilizando foi o de uma clínica médica hipotética, conforme descrito na Figura 11.

Foi possível perceber diversas estatísticas interessantes deste exemplo, pois foram utilizados todo os tipod e componentes possíveis de serem criados, desde *Processes* com finitos e infinitos *Resources* a *Routers* com múltiplos *targets*. Por fim, o *SimTool* atingiu seus objetivos e expectativas quanto ao seu funcionamento e resultado.

Figura 11 – Definição da clínica médica hipotética



Referências

PIDD, M. An introduction to computer simulation. In: LAROQUE J. HIMMELSPACH, R. P. O. R. C.; UHRMACHER, A. (Ed.). *Proceedings of the 1994 Winter Simulation Conference*. [S.l.: s.n.], 1994. p. 7–14. 2, 3

STRASSBURGER, S. Overview about the high level architecture for modelling and simulation and recent developments. *Simulation News Europe*, v. 16, n. 2, p. 5–14, 2006. 2

TAYLOR, S. J. et al. Panel on grand challenges for modeling and simulation. In: IEEE. *Proceedings of the 2012 Winter Simulation Conference (WSC)*. [S.l.], 2012. p. 1–15. 1

TAYLOR, S. J. et al. Grand challenges on the theory of modeling and simulation. In: *SpringSim (TMS-DEVS)*. [S.l.: s.n.], 2013. p. 34. 1, 2

WOOLFSON, M. M.; PERT, G. J. *An introduction to computer simulation*. [S.l.]: Oxford University Press on Demand, 1999. 2