Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

HMR Sim: simulador de sistemas multi-robôs com alto nível de abstração

Giovanni Meneguette Guidini

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientadora
Prof.a Dr.a Genaína Rodrigues

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

HMR Sim: simulador de sistemas multi-robôs com alto nível de abstração

Giovanni Meneguette Guidini

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof.a Dr.a Genaína Rodrigues (Orientadora)
CIC/UnB

Prof. Dr. Donald Knuth
Stanford University

Dr. Leslie Lamport
Microsoft Research

Dr.a Gabriel Rodrigues
Coordenador do Bacharelado em Ciência da Computação

Brasília, 22 de maio de 2021

Dedicatória

Na *dedicatória* o autor presta homenagem a alguma pessoa (ou grupo de pessoas) que têm significado especial na vida pessoal ou profissional. Por exemplo (e citando o poeta): *Eu dedico essa música a primeira garota que tá sentada ali na fila. Brigado!*

Agradecimentos

Nos *agradecimentos*, o autor se dirige a pessoas ou instituições que contribuíram para elaboração do trabalho apresentado. Por exemplo: *Agradeço aos gigantes cujos ombros me permitiram enxergar mais longe. E a Google e Wikipédia.*

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

O *resumo* é um texto inaugural para quem quer conhecer o trabalho, deve conter uma breve descrição de todo o trabalho (apenas um parágrafo). Portanto, só deve ser escrito após o texto estar pronto. Não é uma coletânea de frases recortadas do trabalho, mas uma apresentação concisa dos pontos relevantes, de modo que o leitor tenha uma ideia completa do que lhe espera. Uma sugestão é que seja composto por quatro pontos: 1) o que está sendo proposto, 2) qual o mérito da proposta, 3) como a proposta foi avaliada/validada, 4) quais as possibilidades para trabalhos futuros. É seguido de (geralmente) três palavras-chave que devem indicar claramente a que se refere o seu trabalho. Por exemplo: *Este trabalho apresenta informações úteis a produção de trabalhos científicos para descrever e exemplificar como utilizar a classe LaTeX do Departamento de Ciência da Computação da Universidade de Brasília para gerar documentos. A classe* `UnB-CIC` *define um padrão de formato para textos do CIC, facilitando a geração de textos e permitindo que os autores foquem apenas no conteúdo. O formato foi aprovado pelos professores do Departamento e utilizado para gerar este documento. Melhorias futuras incluem manutenção contínua da classe e aprimoramento do texto explicativo.*

Palavras-chave: LaTeX, metodologia científica, trabalho de conclusão de curso

Abstract

O *abstract* é o resumo feito na língua Inglesa. Embora o conteúdo apresentado deva ser o mesmo, este texto não deve ser a tradução literal de cada palavra ou frase do resumo, muito menos feito em um tradutor automático. É uma língua diferente e o texto deveria ser escrito de acordo com suas nuances (aproveite para ler `http://dx.doi.org/10.6061%2Fclinics%2F2014(03)01`). Por exemplo: *This work presents useful information on how to create a scientific text to describe and provide examples of how to use the Computer Science Department's LaTeX class. The `UnB-CIC` class defines a standard format for texts, simplifying the process of generating CIC documents and enabling authors to focus only on content. The standard was approved by the Department's professors and used to create this document. Future work includes continued support for the class and improvements on the explanatory text.*

Keywords: LaTeX, scientific method, thesis

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Sistemas Multi-Robôs (SMR) são sistemas que consistem em mais de um agente robótico. Por algumas décadas esses sistemas foram utilizados em diversos contextos para cumprir diversas tarefas, especialmente em ambientes dinâmicos. SMRs atuam em um espaço ciber físico (i.e. parte do "mundo real"), logo seus agentes estão propensos à mudanças provenientes tanto de outros agentes do sistema como do ambiente em que estão inseridos [1]. Para aumentar a adaptabilidade do SMR, pode-se projetá-lo como um sistema auto-adaptativo, tornando-o capaz de responder à mudanças no ambiente, de maneira a continuar cumprindo objetivos e respeitando os limites impostos ao sistema [2].

Os agentes desses sistemas (i.e. robôs) existem no mundo físico e interagem com ele e entre si de maneiras mais complexas do que agentes de outros sistemas (e.g. computadores, bancos de dados, etc) [3]. Isso traz desafios para o desenvolvimento desse tipo de sistema, principalmente a preparação de experimentos com vários robôs [4]. Essa dificuldade pode ser superada com o uso de simuladores. Simuladores podem ser empregados tanto para testar a segurança, eficiência e robustez do sistema, quanto para prototipação de SMRs e robôs [4], [5]. Outras vantagens de simuladores incluem: (1) menor custo de tempo e recursos para preparação e execução do experimento; (2) ambientes simulados podem ser mais ricos, complexos e seguros que ambientes reais ou em laboratório; (3) é possível testar hardware que não está disponível [5], [6].

Diversos simuladores para SMRs existem na literatura, por exemplo Gazebo [7], Simbad [8], CoppeliaSim [9], MORSE [6] e Dragonfly [10], entre outros. Cada um desses simuladores foi criado com propostas diferentes, desde simulação precisa das partes que compõem um robô e sua interação com o ambiente (Gazebo, CoppeliaSim, Morse), até simulações de mais alto nível focando principalmente no comportamento dos robôs (Simbad, Dragonfly). Simulações multi-robô são suportadas por simuladores atuais, mas geralmente em menor número - devido ao alto uso de recursos computacionais necessários para simular cada robô (i. e. experimentos feitos com Gazebo mostraram que o simula-

dor tem dificuldades ao simular mais de 10 robôs [4]) - ou são muito específicos quando conseguem simular mais robôs (i.e. Dragonfly supostamente é capaz de simular até 400 entidades, mas está restrito à simulação de drones [10]).

O Laboratório de Engenharia de Software (LES) da Universidade de Brasília (UnB) conduz pesquisas na área de sistemas multi-agentes, incluindo sistemas multi-robôs. Entre os simuladores empregados nas pesquisas do LES, encontram-se Gazebo e MORSE, porém tem sido relatadas dificuldades com o uso desses simuladores em cenários com times maiores de robôs. Isso se dá pelo alto nível de detalhamento físico das simulações, que exige recursos computacionais consideráveis. Quando o objetivo da pesquisa é mais voltado para os algoritmos que coordenam os diferentes agentes do sistema, ese nível alto de detalhamento é desnecessário, mas aumenta consideravelmente o tempo de cada experimento.

Nesse cenário, a proposta deste projeto é fornecer uma ferramenta direcionada para simulação de sistemas multi-robôs auto-adaptativo com baixo nível de detalhamento físico. Esta ferramenta será usada na avaliação e comparar algoritmos de distribuição de tarefas entre agentes de um SMR. Também pode ser usado para prototipação das características de cada agente do SMR, bem como validação de requisitos do time de robôs de algum sistema.

O capítulo 2 comenta conceitos que foram relevantes na construção do simulador, e comenta um pouco mais sobre os outros simuladores que foram pesquisados na literatura. O capítulo 3 apresenta o simulador que foi criado nesse projeto, HMR Sim, sua arquitetura, decisões de projeto importantes e características principais, além de detalhar alguns resultados obtidos.

Capítulo 2

Referencial Teórico

O referencial teórico apresenta conceitos importantes para o projeto e trabalhos relacionados encontrados na literatura. A Seção 2.1 descreve um breve levantamento feito de alguns simuladores bem estabelecidos na literatura. Esses projetos foram usados de inspiração para a criação do HMR Sim. A Seção 2.2 apresenta a *design pattern* ECS (*Entity-Compoenent-System*), analisada por ser bastante utilizada na criação de jogos, pela complexidade dos jogos de videogame atuais e como estes são de certa forma simuladores. Esta arquitetura foi utilizada no projeto do HMR Sim pelas suas vantagens. Finalmente a Seção 2.3 comenta sobre duas técnicas de simulação encontradas nos simuladores que foram levantados e na literatura.

2.1 Simuladores na Literatura

O levantamento da literatura iniciou com um breve levantamento de simuladores já estabelecidos para sistemas robóticos. Foram selecionados os simuladores Gazebo [7], Simbad [8], CoppeliaSim [9], MORSE [6] e Dragonfly [10]. Uma descrição breve de como os robôs são definidos em cada simulador é incluída, junto com um link para a documentação oficial do projeto.

Gazebo. Robôs são definidos em modelos, que seguem uma estrutura de arquivos definida. O robô em si é descrito em arquivos .sdf. Cada modelo é definido através de uma série de <links> que definem as partes do modelo. Sensores ou outros componentes - outros modelos - podem ser ligados através de <joints>. Modelos podem ter plug-ins com funcionalidade extra. O projeto é open source e pode acessado no link `http://gazebosim.org`.

CoppeliaSim. Modelos são definidos como uma seleção de scene objects (e.g. joints, shapes, sensors, cameras, paths, etc). Existem muitas maneiras de controlar uma simulação, dentre elas destaca-se embedded scripts. Esse scripts podem ser definidos como parte

de um scene object e executam alguma funcionalidade relacionada à este objeto. CoppeliaSim (antigamente V-Rep) é uma solução comercial da empresa Coppelia Robotics, disponível no link `https://www.coppeliarobotics.com/features`.

MORSE. Robôs são plataformas que definem o formato e certas propriedades, como área de colisão massa, etc. É nessas plataformas que sensores e atuadores são montados. Estes são fornecidos pelo simulador MORSE para serem adicionados à robôs. Apenas sensores e atuadores interagem com o mundo real com alguma funcionalidade. Os sensores e atuadores são fornecidos em diversos níveis de realismo, permitindo maior ou menor grau de abstração na simulação. MORSE é uma solução open source, baseado no software de modelagem Blender. Infelizmente o projeto se encontra abandonado desde 2020, mas ainda está disponível no link `https://morse-simulator.github.io`.

Simbad. Robôs são definidos em classes Java que estendem a classe `Agent`. Sensores são adicionados como atributos da classe. Status e movimentação do robô é alcançado através de APIs próprias. Robôs implementam as funções `initBehavior()` e `performBehavior()`, que definem o que acontece com o robô ao ser criado e o comportamento dele em cada loop de simulação. Um projeto open source disponível no link `http://simbad.sourceforge.net`.

Dragonfly. Drones possuem uma classe de controle - `DroneKeyboardController` ou `DroneAutomaticController`, respectivamente para ser controlado pelo usuário ou automaticamente. E classes que definem seus comportamentos, através de modelos. Além disso outras configurações, como nível de bateria, consumo por bloco, alvo e os wrappers (para fornecer comportamento adaptativo) podem ser alterados pela interface gráfica, para cada drone na cena. Esse simulador está limitado à simuações de drones. Também open source, disponível em `https://github.com/DragonflyDrone/Dragonfly`.

Cada simulador possui características arquiteturais e objetivos próprios que foram analisados e comparados, fornecendo um arcabouço de técnicas que podem ser utilizadas (ver Tabela 2.1). Pontos relevantes que foram investigados sobre os simuladores incluem:

- **Nível de Abstração.** Pode ser *baixo* indicando grande detalhamento dos componentes que compõe o robô e suas características físicas; *médio* indicando necessidade de detalhamento dos movimentos individuais dos componentes que formam um robô; *alto* indicando abstração dos componentes do robô.
- **Número de robôs** que o simulador é capaz de simular num tempo razoável
- Se o simulador é **genérico** ou não, ou seja, se existe restrição no tipo de robôs que o simulador é capaz de simular.
- **Arquitetura** utilizada na representação do robô (i.e. um robô é uma classe que deve ser implementada, ou um arquivo XML, etc)

Simulador	Nível de Abstração	Nº de robôs	Genérico	Arquitetura	Tipo de Simulação
Gazebo	Baixo	<20	SIM	Declarativa	Passos/DES
CoppeliaSim	Baixo	<20	SIM	Declarativa	Passos/DES
Simbad	Médio	<10	SIM	OOP	Passos
MORSE	Médio/Alto	20 - 100	SIM	OOP/Declarativa	Passos
Dragonfly	Alto	400	NÃO	MVC/AOP	Passos

Tabela 2.1: Comparação resumida das ferramentas com suporte de simulação multi-robôs.

- **Tipo de simulação** indica qual a técnica de simulação usada, em passos ou de eventos discretos (ver Seção 2.3)

2.2 Entity-Component-System

Entity-Component-System (ECS) é um padrão de desenho (*design pattern*) de software amplamente utilizada em jogos, tipicamente em sistemas interativos em tempo-real (e.g. jogos do tipo MMO, *Massive Multiplayer Online*) [11]. Nesse padrão, objetos da simulação são transformados em *entidades*. Cada entidade nada mais é que uma coleção de *componentes*. Um componente, por sua vez, armazena dados, mas tipicamente não implementa nenhuma lógica.

A lógica da simulação está nos *sistemas*, que modificam os dados de componentes de acordo com seu objetivo. Cada sistema age de maneira independente de outros sistemas sobre um conjunto de componentes que lhe interessa, ou seja, se uma entidade possui esse conjunto de componentes, então ela será afetada pelo sistema durante a simulação. O estado da simulação é o conjunto de estados de todos os componentes de todas as entidades presentes na simualção. Ele é alterado apenas pelos sistemas, cada um alterando uma pequena parte desse estado global.

Essa organização permite grande modularização e separação de lógica entre as difrentes partes do sistema. Cada sistema (ou conjunto de sistemas) e seu conjunto de componentes associados pode ser adicionado ou remivdo do simulador conforme necessário. Por exemplo, um sistema comunicação entre diferentes robôs pode ser implementado como um componente que guarde uma fila de mensagens e pode ser adicionado à cada robô, associado à dois sistemas: um sistema que faça a entrega das mensagens de um robô para o outro, e outro sistema que processa as mensagens de cada robô. Note que se o processamento não for adequado à uma simulação, basta trocar aquele sistema por outro que seja adequado. Além disso, se alguma simulação não faz uso desse sistema

de mensagens, basta removê-lo do simulador completamente, deixando a simulação mais leve.

Uma outra vantagem de utilizar o padrão ECS é a flexibilidade de adicionar ou remover capacidades das entidades durante a execução da simulação. Como cada entidade é simplesmente uma coleção de componentes, é possível associar certas capacidades dos robôs (e.g. sensores, atuadores) à presença ou ausência de certos componentes naquela entidade. Por exemplo, dada a existência de um componente `camera` e um sistema associado que simule a captura de imagens, qualquer entidade que possue esse componente vai possuir a capacidade de coletar imagens via componente camera. Além disso, al simular falhas catastróficas em componentes, basta remover o componente da entidade sendo analisada.

Apesar dessas vantagens, como apontado por Wiebush [11], o uso de ECS pode trazer complicações de compatibilidade entre sistemas desenvolvidos de maneira indepentente, como uso de componentes incompatíveis, e dificuldade em conhecer qual sistema é responsável por determinada funcionalidade e como utilizá-la. detalhes de como esses problemas foram sentidos durante o desenvolvimento do projeto e medidas tomadas para mitigá-los são discutidas no capítulo 3.

Foi utilizada a biblioteca `esper` para suporte do padrão ECS. Esper é uma biblioteca de ECS leve com foco em performance, escrita na linguagem Python por Benjamin Moran [12]. Ela cria uma classe `World` que mantém uma lista de entidades e de todos os componentes para cada entidade. Um componente pode ser qualquer estrutura em Python, no caso do projeto foram usadas classes (i.e. `class`). É possível ainda adicionar sistemas à classe `World`, que são implementados como funções, convencionalmente chamadas `process`. Alguns dos sistemas do projeto são adicionados ao `World`.

2.3 Técnicas de Simulação

Uma técnica de simulação bem estabelecida é a de tempo discreto com intervalo de incremento fixo [13]. Nesse modelo, o estado de um sistema no tempo t_{i+1} é uma função do estado do sistema no tempo t_i. Cada variável que compõe o estado do sistema é uma função de variáveis e estados até o momento anterior. O incremento de tempo da simulação entre t_i e t_{i+1} é sempre o mesmo, e pré-definido.

Se o tempo t_{calc} necessário para computar o estado t_{i+1} do sistema a partir do estado t_i é menor do que o tempo do incremento t_{incr}, então a simulação será computada mais rápido do que o tempo do relógio (e.g. o tempo real); da mesma forma, se $t_{calc} > t_{incr}$, então a simulação é computada mais devagar do que o tempo do relógio. Essas situações são conhecidas como simulação *offline* [13], porque não há sincronia entre o tempo da

simulação e o tempo do relógio. Essa é uma situação aceitável para este projeto, onde o objetivo é obter a simulação desejada no menor tempo possível.

Essa técnica de simulação é indicada para simular sistemas que mudam constantemente, como por exemplo a temperatura de um ambiente ao longo do tempo, ou um sinal recebido por um sensor que trabalha a uma frequência conhecida. No entanto o "relógio"da simulação é sincronizado, e todas as funções do estado são processadas a cada incremento de tempo, o que pode levar a cálculos desnecessários. Por exemplo, em uma simulação que involva uma função que altera temperatura de uma sala a cada 200ms, e um sensor que registra a temperatura da mesma sala com leituras a cada 100ms, a função que altera temperatura deve ser executada em todos os incrementos de tempo, que devem ser no máximo 100ms para suportar a leitura do sensor. Nesse cenário, metade das chamadas à função de alterar temperatura não afeta o estado do sistema, mas ainda tem que ser processadas.

Outra técnica de simulação é por eventos discretos (DES, *Discrete Event Simulation*) [14]. Nesse modelo uma fila de eventos é processado um por vez, e cada estado s_{i+1} é o resultado de processar o evento no topo da fila sobre o estado s_i. Um evento e possui um tempo t e uma função f que altera o estado, e potencialmente cria outros eventos, que serão adicionados à fila. A fila de eventos é uma fila de prioridades ordenada pelo tempo t de cada evento, sendo que o tempo de cada novo evento gerado pode ser igual ou maior que o tempo do evento que o gerou (nunca menor, porque não se pode alterar o passado da simulação). O tempo da simulação corresponde ao tempo t do evento atual sendo processado, e como os eventos são ordenados pelo tempo, ele só será incrementado quando todos os eventos naquele tempo foram processados.

Diferentemente da simulação com intervalo de incremento fixo, onde as mesmas funções são executadas em intervalos conhecidos de tempo, na simulação do tipo DES funções diferentes alteram o estado da simulação, e o tempo da simulação no estado $s_i + 1$ não depende apenas do estado s, mas também do evento sendo processado. Esse novo tempo pode não crescer de maneira uniforme ao longo da simulação. Essa técnica é adequada para simular sistemas que mudem de maneira infrequênte ao longo do tempo, por exemplo o inventário de um armazém [13], ou a operação de robôs de serviço dentro do armazém.

A simulação de incremento fixo de tempo pode ser implementada utilizando a técnica de eventos discretos, desde que os eventos sejam criados com tempos que possuam um intervalo constante. Uma outra característica interessante que pode ser alcançada com eventos discretos é separar a função em subsistemas que são executados de maneira independente e assíncrona. Retomando o exemplo da sala que muda de temperatura e possui o sensor, cada evento de leitura do sensor pode criar o próximo evento de leitura para o tempo $t + 100ms$; de forma similar cada evento de mudança de temperatura cria um novo

evento de mudança para o tempo $t + 200ms$. Dessa forma, evita-se o problema de funções de alteração do estado da simulação tendo que ser executadas antes da hora.

O simulador utiliza a técnica de simulação de eventos discretos, através da biblioteca `simpy` [15]. Esse framework de simulação DES é baseado em processos e faz todo o gerenciamento dos eventos e sua execução. A simulação acontece dentro de um ambiente, onde diversos processos interagem entre si e com o ambiente através de eventos. QUalquer função geradora em Python pode ser um process no simpy.

Esse framework também tem suporte para recursos (*Resources*), que são compartilhados entre os processos. Recursos podem simular desde recursos a serem disputados (i.e. uma impressora, uma estação de carga) até recursos que são armazenados em contâiners (i.e. 10L de aguá de um reservatório com capacidade para 10000L). Recursos podem ainda ser preemptivos, ou filtrados de algum contâiner. Essa última capacidade foi bastante utilizada para a comunicação entre sistemas do simulador, como será discutido no Capítulo 3.

Capítulo 3

O simulador HMR Sim

Conforme comentado no Capítulo 1, a proposta deste projeto é fornecer uma ferramenta direcionada para simulação de sistemas multi-robôs auto-adaptativo com baixo nível de detalhamento físico. Essa porposta foi realizada através do simulador HMR Sim. É um projeto *open source*, disponível em `https://github.com/lesunb/HMRSsim`.

A linguagem escolhida foi Python, uma linguagem de programação popular e versátil. Outros simuladores analisados têm suporte para essa linguagem, como MORSE e CoppeliaSim. Arquitetura do simulador é baseado na *design pattern* Entity-Component-System (ECS), e a técnica de simulação é de eventos discretos. Grande foco foi dado para modularização e reaproveitamento de sistemas na construção do simulador. Alguns dos principais sistemas (e.g. Navegação, Script) foram construídos para serem extensíveis. Além disso foi dado atenção à facilidade e rapidez de construir simulações. A arquitetura do simulador é detalhada na Seção 3.1. A criação e uso de sistemas na Seção 3.4. Os principais sistemas já construídos são apresentados na Seção 3.5, incluindo o sistema que permite visualização da simulação, Seer. Finalmente a Seção 3.6 verifica o desempenho do simulador numa simulação com grande número de robôs.

Uma simulação no HMR Sim tem dois estágios: (1) fase de carregamento, onde os componentes disponíveis são carregados e as entidades da simulação criadas e (2) fase de execução, onde os sistemas da simulação são inicializados e a simulação em si é executada. Componentes são definidos como classes em Python e são carregados automaticamete utilizando um sistema de nomenclatura apropriado. Sistemas são construídos como funções Python, e podem ser processos da biblioteca `simpy` ou funções aceitas como sistemas do `esper` (ver Seções 2.3 e 2.2 para detalhes sobre as bibliotecas). Os sistemas devem ser inicializados e adicionados ao simulador antes da fase de execução. A Seção 3.3 cobre a criação de componentes e como eles ficam disponíveis no simulador.

Uma simulação pode ser definida em um mapa (veja figura 3.1), programaticamente através de um objeto de configuração (dicionário Python), ou uma mistura das duas

opções. Mapas são arquivos XML construídos com a biblioteca `JGraph` (disponível em `https://github.com/jgraph`), com algumas restrições. Qualquer programa compatível com essa biblioteca pode ser utilizado, por exemplo `diagrams.net`, bastante popular. Também é possível criar entidades na simulação através de *EntityDefinition*.

As formas desenhadas no mapa da simulação podem ser anotadas para especializá-las (marcar a figura como um certo tipo de entidade), adicionar componentes, ou diferenciá-la de alguma forma. Na figura 3.2, por exemplo, as anotações marcam aquele objeto do mapa como um robo, que possui um componente `Claw` inicializado com os valores `[80, 1]`, e um componente `Script` inicializado com os valores da figura.

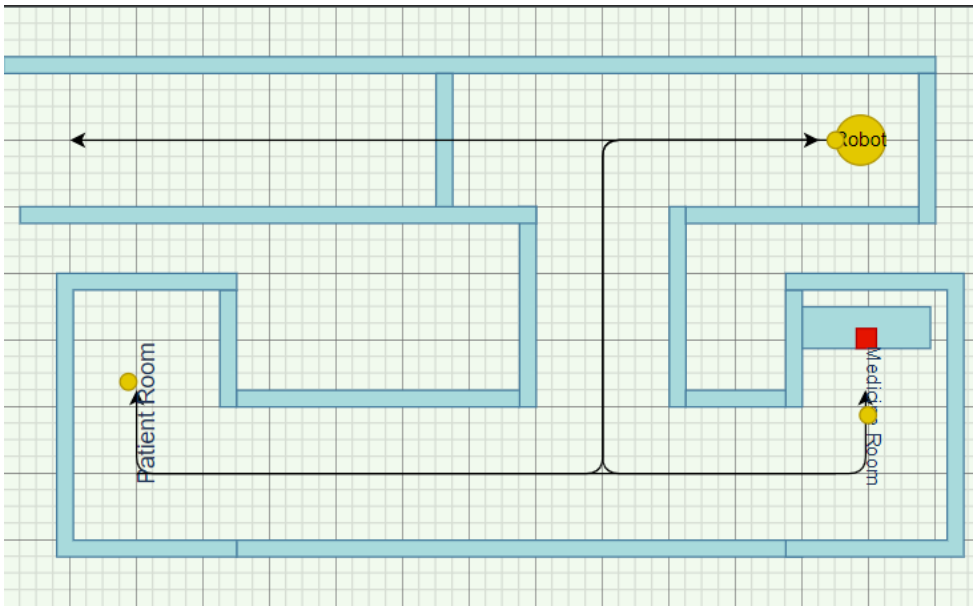

Figura 3.1: Exemplo de mapa de uma simulação

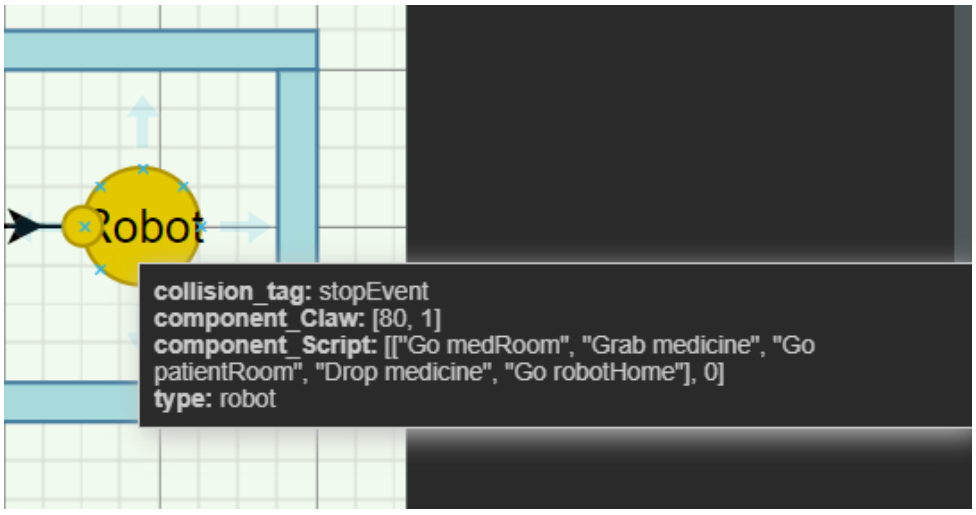

Figura 3.2: Exemplo de anotações em uma entidade

3.1 Arquitetura

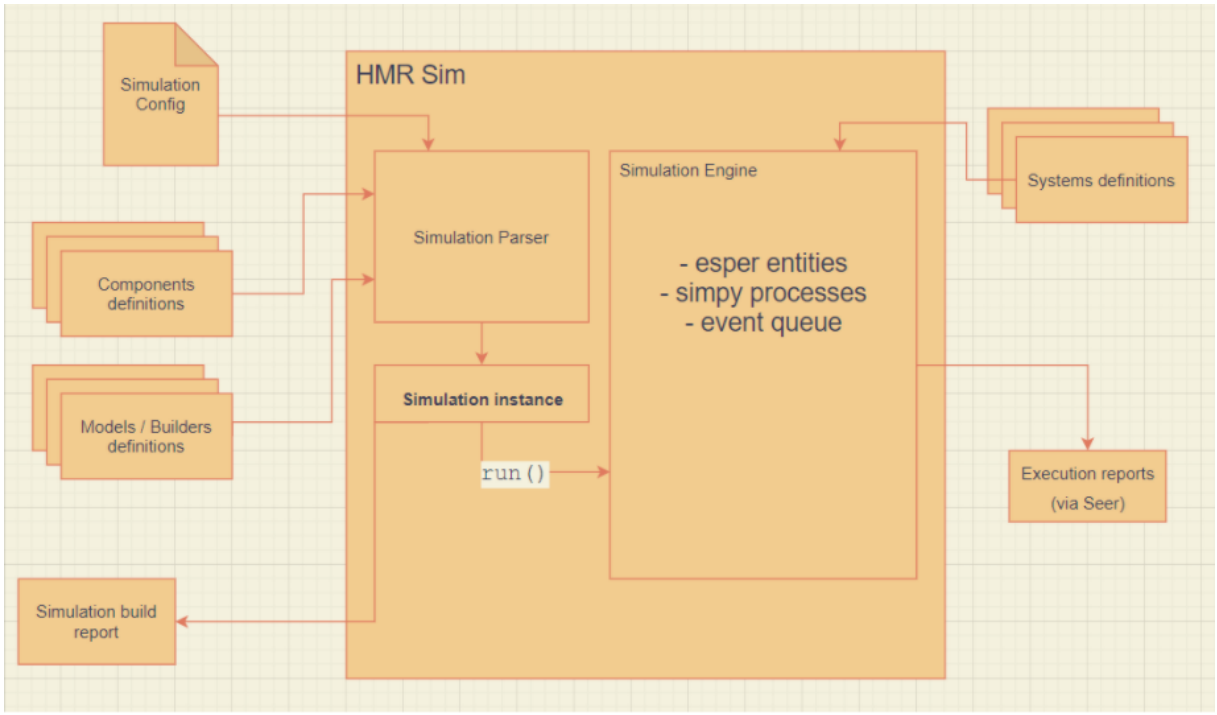

Figura 3.3: Diagrama representando um resumo da arquitetura do HMR Sim

A figura 3.3 mostra um resumo da arquitetura do simulador HMR Sim. A construção da simulação acontece separada da simulação em si. Para construir uma simulação, um objeto de configuração tem que ser passado para o simulador, bem como as definições dos componentes disponíveis e os `models` e `builders` disponíveis. A classe `Simulator` cria a simulação, e depois a executa. A instancia da simulação mostrada na figura 3.3 (*Simulation instance*) é exatamente a instância da classe `Simulator` criada.

O objeto de configuração é um dicionário Python, que pode ser salvo como um arquivo `json`. Algumas das opções mais importantes estão listadas abaixo. Para ver todas as opções verifique a documentação do projeto, no repositório.

- **context** (string) - A raíz do projeto, de onde componentes, sistemas e builders extras serão incluídos;
- **map** (string) - O arquivo XML do mapa
- **FPS** (int) - Frequência com que serão executados os sistemas do `esper`. Esses sistemas não geram eventos, são indicados para representar sistemas que acontecem de forma frequênte e previsível. Caso nenhum sistema `esper` seja utilizado não é necessário informar esse valor.

- **duration** (int) - Tempo limite da simulação (no relógio da simulação). Por exemplo, um valor 6 vai fazer o simulador encerrar a simulação após 6s serem simulados.
- **simulationComponents** (dict) - Componentes "globais". Eles são adicionados à entidade 1, reservada à simulação. Podem ser acessados por todos os robôs (e.g. um mapa compartilhado de rotas).
- **extraEntities** (EntityDefinition[]) - Lista de `EntityDefinition` (ver documentação), que definiem entidades que não estão presentes no mapa mas devem ser incluídas na simulação. É possível declarar todas as entidades com essa opção e usar o mapa apenas para coisas estáticas, reaproveitando-o para vários cenários.

O simulador importa automaticamente `components`, `models` e `builders` do sistema de arquivos. Por conta disso os projetos que usem o HMR Sim precisam de uma organização expecífica, como mostrado abaixo. Dentro de cada pasta os arquivos deve estar no formato apropriado.

```
project_root
├── models
├── components
└── builders
```

A simulação construída se torna uma instância da classe `Simulator` cujo atributo `World` está preenchido com as entidades que foram criadas, e as opções de configuração salvas. Essa instância é um objeto Python, podendo ser salvo através da biblioteca `pickle`, por exemplo, e distribuído. Antes de poder ser executada, no entanto, é necessário inicializar e adicionar os sistemas à simulação. Para adicionar sistemas os métodos `Simulator.add_system` e `Simulator.add_des_system` podem ser usados. Qual dos dois usar depende do sistema a ser adicionado, detalhes na Seção 3.4.

Após adicionar os sistemas, a simulação pode ser executada utilizando o método `Simulator.run`, da instância do simulador gerada. A simulação é executada por `duration` segundos de simulação, se essa opção foi passada na configuração, ou até que o evento `KILL_SWITCH` seja processado. É possível manter ver logs da simulação durante a execução. A visualização gráfica da simulação é implementada através do sistema Seer, discutido na seção 3.5.5.

Alguns sistemas com funcionalidades consideradas essenciais na maior parte das simulações já foram implementados, como parte da validação do simulador. Eles serão detalhados na Seção 3.5, e incluem sistema de navegação, movimentação, colisão, controle de robôs, sensores, e visualização.

3.2 `builders` e `models`

Os arquivos XML que podem ser usados de mapas representam os objetos desenhados dentro de tags `<mxCell>`, com alguma geometria. Diferentes formas possuem diferentes conteúdos na tag `<mxCell>`. `models` são funções que traduzem o XML em uma `<mxCell>` para uma lista de componentes da simulação. Cada `model` deve ser armazenado em um arquivo próprio que exporte: (1) uma função `from_mxCell`, que recebe uma `<mxCell>` e retorna uma lista de componentes; (2) uma constante `MODEL` com o nome da forma que esse model traduz.

`builders` são semelhantes aos `models`, mas fazem a tradução do XML contido em `<object>`. Qualquer forma (`<mxCell>`) que tenha anotações (como as mostradas na figura 3.2) é envolvida em uma tag `<object>` que guarda as anotações. `builders` traduzem tanto as anotações no objeto quanto o XML da `<mxCell>` contido nele. Eles podem ou não fazer uso de `models` para isso. UM `builder` deve estar em um arquivo próprio que exporte: (1) uma função `build_object`, que transforma o XML do objeto e (2) uma constante `TYPE`, indicando que esse `builder` deve ser aplicado em objetos que tenham a anotação `type` com esse valor.

Os `models` e `builders` são opcionais. Caso todas as entidades sejam criadas através da opção `extraEntities` da configuração eles serão desnecessários. Atualmente cerca de 10 formas podem ser usadas, e cerca de 7 `builders` foram criados.

3.3 Entidades e Componentes

HMR Sim utiliza a biblioteca `esper` [12] (ver detalhes na Seção 2.2) para gerenciar as entidades e seus componentes na simulação. Cada entidade do mapa ou das entidades passadas pela configuração é representada por um inteiro, armazenado dentro da classe `World` do `esper`, e pode ser acessado através da instância da simulação no atributo `world`, e também pelos sistemas. O identificador de cada objeto do mapa e qual entidade corresponde a ele fica armazenado no atributo `draw2ent` do simulador. Entidades também podem ser adicionadas diretamente à simulação através do método `Simulator.add_entity`.

Entidades possuem um conjunto de componentes. componentes podem ser adicionados ou removidos de entidades usando métodos da biblioteca `esper`. Existem também métodos para verificar a existência de um componente em uma entidade, buscar componentes específicos de entidades, buscar todos os componentes de um certo tipo no `World`, etc. Componentes são simplesmente classes Python, que devem ter o mesmo nome que o arquivo que as declara (uma por arquivo). É convencionado no padrão ECS que os componentes não tenham lógica implementada, sugere-se que apenas os métodos `__init__` e

`__str__` sejam implementados em um componente. `@dataclass` podem ser utilizadas.

O identificador de um componente é o nome da sua classe. Dessa forma, se o componente `simulator.componentes.MyComponent` for declarado, para utilizá-lo em algum sistema pode-se importá-lo normalmente (e.g. `from simulator.components.MyComponent import MyComponent`) e usar essa importação ao se referir ao componente (buscando-o no `World`, por exemplo).

3.4 Sistemas

Sistemas são uma parte essencial do simulador, pois são os responsáveis por fazerem a simulação acontecer. Sistemas são também muito versáteis, podendo ser usados não só para representar uma parte da simulação, mas também para extrair informações dela. Existem 2 tipos diferentes de sistemas que podem ser usados: (1) um sistema compatível com sistemas da biblioteca `esper`, referidos como "sistemas normais"e sistemas compatíveis com a biblioteca `simpy`, referidos como "sistemas DES". Ambos são opcionais e o uso de um ou outro depende do que será simulado pelo sistema. As seções 3.4.1 e 3.4.2 explicam a diferença entre eles e as particularidades de cada um.

O conceito central do HMR Sim é que sistemas são construídos para utilizar um conjunto de componentes e representar uma parte da simulação. Cada conjunto de sistemas relacionados (e seus componentes) confere novas capacidades ao simulador, e portanto à simulação sendo feita, por exemplo sistemas podem representar sensores, movimentação, comunicação entre robôs, etc. Grande parte da flexibilidade que o simulador proporciona está na relativa facilidade de criar e usar sistemas. Se algum sistema não se adequa às suas necessidades, basta substituí-lo por outro, ou modificá-lo, sem ter que alterar outras partes da simulação. Por exemplo, se o objetivo de uma simulação é comparar dois algoritmos de gerenciamento de robôs, 2 sistemas diferentes (que usem os mesmos componentes) serão criados, um para cada algoritmo. Testar o algoritmo A ou B se torna uma simples questão de adicionar o sistema A ou B na simulação.

3.4.1 Sistemas compatíveis com `esper`

Os "sistemas normais"são aqueles que ficam armazenados dentro do `World` do `esper`. Eles são definidor como classes que extendem a classe `esper.Processor` e devem implementar 2 funções: `__init__` e `process`. A função `process` recebe dois argumentos, `self`, a instância do `World` e `kwargs`, os parâmetros passados pelo `Simulator` aos sistemas (ver documentação).

Esses sistemas (e apenas eles) são executados pelo `esper`. Para utilizzá-los é necessário passar a opção `FPS` na configuração da simulação. Esses sistemas serão então executados

uma vez a cada $\frac{1}{FPS}$ segundos de simulação. Note que o uso desses sitemas força a simulação a correr em passos (i.e. o relógio da simulação vai andar em intervalos de, no máximo, $\frac{1}{FPS}$).

Esse tipo de sistema é indicado para simular comportamentos constantes e repetitivos, com frequência definida. Por exemplo movimentação, verificação de colisão, aumento de temperatura em uma sala, gasto de bateria, etc. Eles geralmente seguem o formato mostrado no código 3.1

```
# ...
from simulator.components.MyComponent import MyComponent
from simulator.components.MyOtherComponent import MyOtherComponent
# ...

class MySystem(esper.Processor):
    def __init__(self, args):
        super().__init__()
        # initialize system

    def process(self, kwargs: SystemArgs) -> None:
        # Iterate over components the system is interested in
        for ent, (my_comp, my_other_comp) in \
            self.world.get_components(MyComponent, MyOtherComponent):
            # Does stuff
            # ...
```

Listing 3.1: Formato básico de um sistema normal

3.4.2 Sistemas compatíveis com `simpy`

Os "sistemas DES"são gerenciados pelo `simpy`. Eles são definidos como processos do `simpy`, e podem ser qualquer processo aceito pelo `simpy`, ou seja, qualquer função geradora. Sugere-se, como convenção, usar uma função chamada `process`. Opcionalmente podem também definir uma função de limpeza, que será executada ao final da simulação, e serve para fechar arquivos que foram abertos, por exemplo. Esses sistemas "vivem"todos dentro do mesmo ambiente do `simpy`, que é armazenado como um dos atributos da simulação.

A comunicação entre esse tipo de sistema aproveita os eventos suportados pelo `simpy`, utilizando o recurso `EVENT_STORE`, outro atributo do simulador. Convenciona-se que apenas eventos (`EVENT` ou `ERROR`, ver `typehints` na documentação do simulador) sejam utilizados dentro da `EVENT_STORE`. Cada sistem pode exportar seu próprio payload e tag para

eventos. Assim, qualquer outro sistema que queira enviar uma mensagem para o sistema em questão só precisa criar um novo evento com o payload e tag apropriado e adicioná-lo a `EVENT_STORE`. Esse sistema de comunicação permite a criação de sistemas reativos, que apenas processam eventos esperados por eles, e no resto do tempo ficam desativados.

É possível também um sistema em determinado momento criar um canal temporário para aguardar uma resposta de alguma operação asíncrona. É possível também que um sistema execute em mais de uma thread, mas a thread principal deve executar na mesma que o resto do simulador, e deve ser adicionada ao ambiente `simpy`. A comunicação entre as threads so sistema é de responsabilidade dele.

Durante a execução argumento `kwargs` é passado aos sistemas, dando acesso ao `World`, ao ambiente `simpy` da simulação, à `EVENT_STORE`, o evento `KILL_SWITCH`, etc. Essas informações permitem ao sistema grande poder sobre a simulação. O evento `KILL_SWITCH` é um evento especial que termina a simulação se for disparado.

Esses tipo de sistema é indicado para simular comportamentos inconstantes ou imprevisíveis, sistemas reativos ou ainda como plugins, exetendendo o simulador. O código 3.2 representa o formato típico de um sistema DES.

```
MyEventPayload = NamedTuple('myPayload', [('ent', int), ('info', str)])
MyEventTag = 'MyEvent'

def process(kwargs: SystemArgs):
    event_store: FilterStore = kwargs.get('EVENT_STORE', None)
    # Initialize other variables
    while True:
        # Activates process when event arrives
        ev = yield event_store.get(lambda e: e.type == MyEventTag)
        # Do stuff
        # ...
```

Listing 3.2: Formato básico de um sistema DES

3.5 Sistemas Disponíveis

Alguns sistemas considerados críticos foram implementados no desenvolvimento desse projeto. Eles serviram parcialmente como validação das decisões tomadas ao longo do projeto. Ao longo do seu desenvolvimento foi dado foto na modularização e extensibilidade, como evidenciado principalmente nos sistemas de controle de entidades e navegação. Todos os sistemas que serão citados nas próximas seções (Seções 3.5.1 até

3.5.5) foram testados com simulações que estão disponíveis nos exemplos do projeto (`https://github.com/lesunb/HMRSsim`).

3.5.1 Sistema de Movimentação e Colisão

O sistema de movimentação é intimamente relacionado ao sistema de colisão. Ambos foram modelados como sistemas compatíveis à biblioteca `esper`. Os componentes que habilitam esses sistemas são posição, velocidade e uma caixa de colisão (`Collidable`). Esses sistemas dão suporte para simulações 2D apenas, para uma simulação 3D outros sistemas precisam ser implementados.

Convenciona-se que apenas o sistema de movimentação modifica o componente posição das entidades. Outros sistemas que queiram movimentar uma entidade devem adicionar velocidade à ela, através do componente de velocidade. Isso permite que o sistema de movimentação seja leve, levando em média 0.003s para movimentar 200 entidades na simulação do enxame de drones (detalhes na Seção 3.6). O sistema de movimentação também é responsáver por dividir o espaço da simulação em setores, para aumentar a eficiência do sistema de colisão. O tamanho do setor pode ser alterado na inicialização do sistema de movimentação.

O sistema de colisão verifica colisão entre entidades que se mexem (i.e. possuem os componentes `Position`, `Collidable` e `Velocity`) e outras entidades da simulação que tenham uma posição e o componente `Collidable`. Por questões de desempenho apenas formas côncavas são utiliadas, porém é possível definir múltiplas formas para a mesma entidade, aproximando assim um formato convexo. Como mencionado anteriormente, é feita uma separação da simulação em setores, assim o sistema de colisão só verifica colisão de uma entidade com as entidades no mesmo setor e em setores adjacentes na simulação, aumentando a eficiência do sistema em simulações com muitas entidades. Na simulação do enxame de drones, esse sistema apresentou uma média de 0.097s à 0.085s para verificar a colisão das 200 entidades. A diferença ocorre porque em diferentes momentos da simulação as entidades estão mais próximas ou mais separadas, evidenciado que a separação da simulação em setores afeta diretamente o desempenho desse sistema.

Quando uma colisão é detectada, um evento de colisão é adicionado à fila de eventos (i.e. `EVENT_STORE`), com os IDs das entidades que colidiram. Um outro sistema deve ser implementado para tomar a resposta apropriada em relação a esse evento, preferencialmente um sistema DES reativo. `simulator/systems/StopCollisionDESProcessor` é um exemplo de sistema que pode ser utilizado.

3.5.2 Sistema de Navegação

O sistema de navegação foi construído para permitir que robôs encontrem caminhos até pontos do mapa por conta própria. Esse sistema funciona em conjunto com o sistema de caminhos, que é baseado em caminhos que o robõ deve seguir (componente `Path`). Existem duas maneiras de comandar um robô até um ponto específico do mapa: passando uma coordenada do mapa, ou definindo um ponto de interesse (POI) no mapa (ver figura 3.4, os pontos vermelhos), e passando a tag identificadora desse POI para o robô. Esse comando é feito através de eventos na fila de eventos.

É esperado ainda que a entidade do ambiente (entidade 1) tenha um componente `Map`, o mapa de caminhos que será compartilhado por todos os robôs, e que caminhos tenham sido definidos no mapa, utilizando entidades do tipo `map-path` (ver figura 3.4, as setas). Esses caminhos são considerados percursos seguros que os robôs podem seguir para se movimentar ao longo do mapa. Durante a criação da simulação, os `map-paths` são transformados em um grafo armazenado no componente `Map` do ambiente. Pontos próximo são combinados em "super-pontos"para diminuir a quantidade de nós no grafo, aumentando a eficiência das buscas. O tamanho de um super-ponto pode ser definido na inicialização do componente `Map`.

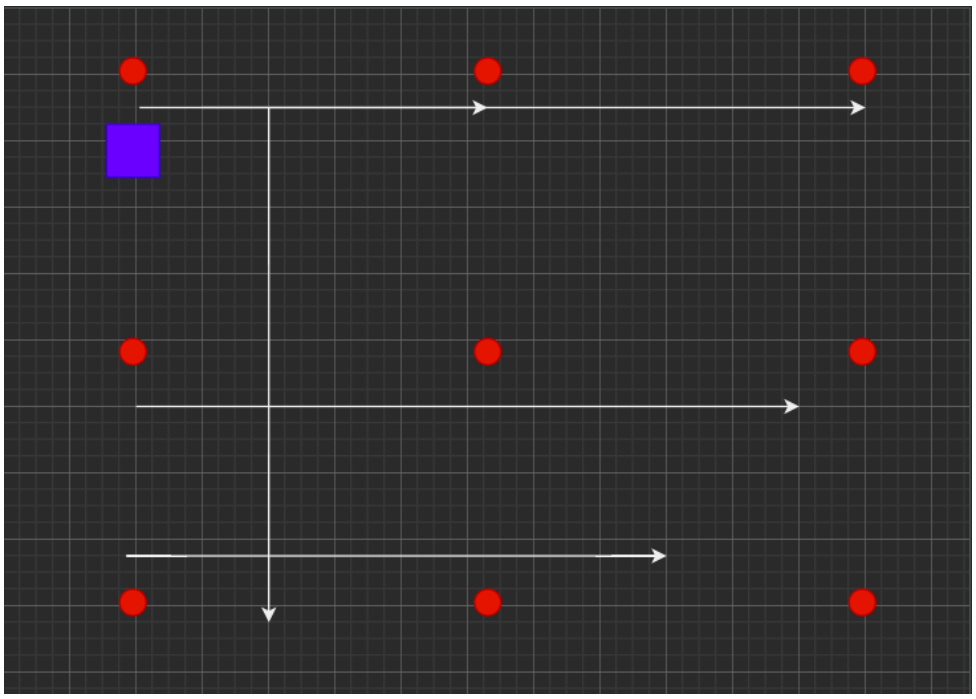

Figura 3.4: Mapa representando POIs (pontos vermelhos) e `map-paths` (setas) para o sistema de navegação

Ao inicializar o sistema de navegação, uma função de navegação deve ser informada. Essa função recebe como parâmetros o mapa do ambiente, a posição atual do robô e sua

posição de destino, e deve retornar um caminho (`Path`) da origem até o destino através dos caminhos seguros do mapa. É permitido que os robôs saiam dos caminhos seguros dentro de um limite que pode ser configurado no componente `Map`. A função `find_route` (`simulator/systemas/NavigationSystem`) pode ser usada como função que encontra caminhos.

Encontrado um caminho do ponto de origem ao ponto de destino, o componente `Path` que representa esse caminho a ser seguido é adicionado à entidade. A partir desse ponto o sistema de caminhos (`PathProcessor`) se encarrega de adicionar velocidade ao robô para que passe por todos os pontos do caminho. Ao chegar no destino, o componente é removido. Caso um caminho não seja encontrado, um evento de erro é adicionado à fila de eventos, para que o controlador do robô (que passou o comando de movimentação) possa tomar a decisão apropriada.

A simulação `poi_test` (ver figura 3.4) foi feita para testar o sistema de navegação. Nessa simulação um robô (o quadrado) recebe instruções para ir a alguns POIs (os círculos) em sequência, utilizando o sistema de navegação. Ele segue os caminhos definidos no mapa (as setas).

Toda vez que uma entidade encontra um caminho até um objetivo, esse caminho encontrado (que pode passar por pontos ainda não mapeados) é incluído ao mapa compartilhado, na tentativa de expandi-lo. Assim, quanto mais as entidades navegam pelo mapa, mais pontos conhecem e mais pontos podem acessar.

3.5.3 Sistema de Controle

O sistema de controle (ou sistema de script) foi criado para facilitar o controle e passagem de instruções às entidades da simulação. Ele é especialmente indicado para criar o equivalente a NPCs (*non playable characters*) numa simulação, ou seja, entidades que precisam se movimentar de alguma forma, mas não são o foco da simulação. Esse sistema foi feito para ser extensível por outros sistemas, e funciona baseado em instruções. Na figura 3.2, um dos componentes do robô é `component_Script`, que é o componente utilizado pelo sistema de controle. O vetor de strings passados para esse componente é a sequência de instruções que ele vai realizar na simulação, gerenciadas pelo sistema de controle. Esse sistema é reativo e reage à eventos anotados como funções ou como eventos de interesse.

Uma instrução pode ser definida por qualquer sistema, e incluída no sistema de script durante sua inicialização. Elas são implementadas através de funções que aceitem argumentos específicos (detalhes na documentação do projeto), e cujo retorno é o estado do script daquela entidade ao executar a instrução. Por exemplo, na figura 3.2 a função `Go` foi definida pelo sistema de navegação, e as funções `Grab` e `Drop` pelo sistema da garra (um atuador que será discutido na Seção 3.5.4).

Existem 3 estados que um script pode estar ao longo da simulação: `READY`, `BLOCKED` ou `DONE`. `DONE` significa que todas as instruções do script foram executadas. `READY` indica que a entidade está pronta para executar a próxima instrução. `BLOCKED` sinaliza que a entidade está executando alguma ação do script, ou esperando que algo aconteça para poder executar a próxima instrução. Para dar suporte às ações asíncronas, existe uma lista de eventos que o script espera. Toda vez que uma instrução asíncrona é executada, espera-se que o retorno seja o estado `BLOCKED`, e que tenham sido adicionadas ao componente de script as tags dos eventos que marcam o final da ação sendo executada. O sistema de controle é ativado ao receber um evento com alguma dessas tags, e vai removendo elas da lista de espera do script da entidade apropriada. Quando essa lista fica vazia a entidade é desbloquada (i.e. passa do estado `BLOCKED` para o estado `READY`).

Por exemplo, se a entidade 2 vai executar a instrução `Go poi`. Essa instrução pode emitir um evento para o sistema de navegação movimentar a entidade 2 até a coordenada do ponto de interesse "poi". Quando o robô chegar até o ponto de destino, nesse sistema, é emitido um evento `EndOfPath`. Nesse caso, a função que implementa a instrução `Go` deve adicionar o evento `EndOfPath` na lista de espera do script da entidade 2 e retornar o estado `BLOCKED`. Passado algum tempo, a entidade 2 vai chegar até o seu destino, o evento `EndOfPath` será emitido e capturado pelo sistema de controle, esse evento será removido da lista de espera do script, deixando-a vazia e indicando que a entidade 2 pode executar a próxima instrução.

Esse sistema de controle foi testado nas simulações `poi_test` e `hospital_scenario`, com instruções implementadas por diferentes sistemas. O sistema de controle em si só necessita do componente `Script` e dos eventos para funcionar. Ele desconhece a implementação das instruções passadas em sua inicialização, o que permite grande flexibilidade ao sistema.

O sistema de controle também é capaz de tratar erros. Isso é feito passando um dicionário `error_handlers` ao componente `Script` de uma entidade. Esse dicionário possui tags de erros como chaves e uma ação a ser realizada como valor. Ao receber um erro, o sistema de controle verifica se existe uma ação para aquele erro e a executa caso exista. Se não existir ele tenta realizar uma ação genérica (e.g. panic!) que pode ser configurada. Esse comportamento de tratamento de erros foi testado na simulação `poi_test`, onde uma das instruções não retorna um caminho completo até o destino, apenas um caminho parcial que é seguido. É esperado que um sistema que implemente instruções implemente também as ações a serem realizadas em caso de erro da sua instrução e as adicione ao dicionário de erros.

3.5.4 Sensores e Atuadores

3.5.5 Seer

Até o momento não foi mencionado como extrair informações da simulação, e particularmente como visualizar a simulação. Visto que a construção de uma simulação utilizando um diagrama é predominantemente visual, é esperado que a visualização da mesma seja: (1) semelhante ao mapa construído e (2) tão intruitiva quanto construir a simulação. Isso é suportado através do sistema Seer, implementado como um sistema DES. Essa decisão foi tomada para permitir que o simulador seja executado sem qualquer visualização gráfica, por ser uma operação custosa durante a execução da simulação.

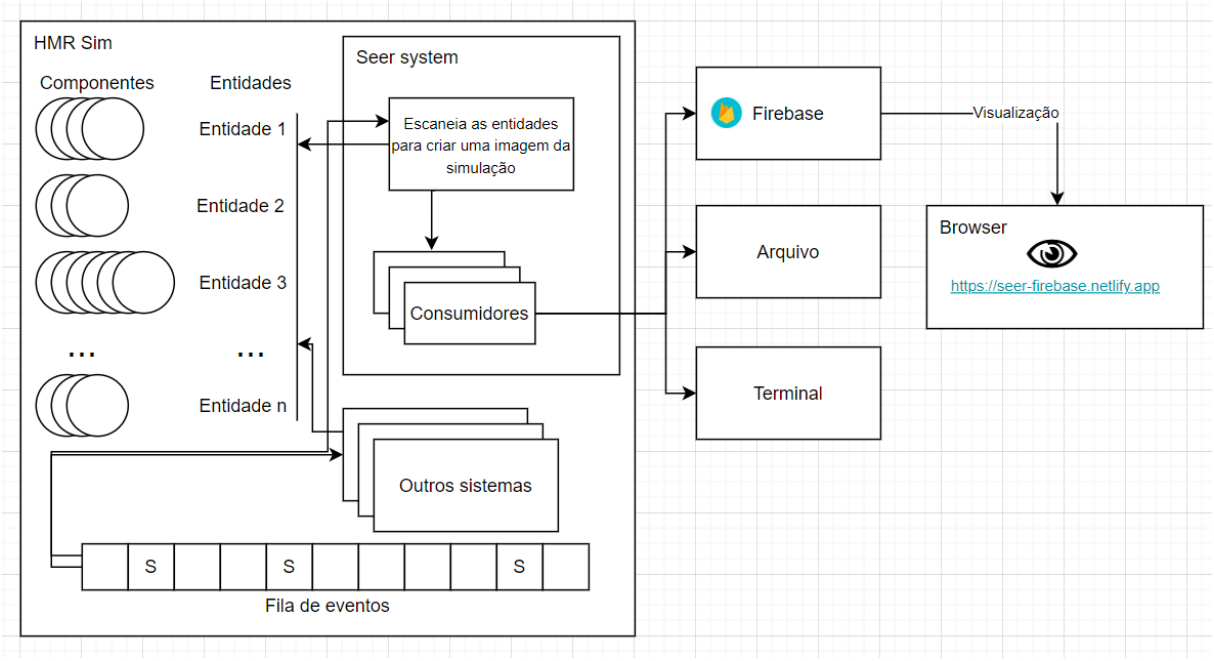

Figura 3.5: Esquema da arquitetura do sistema Seer

Seer é um sistema opcional como qualquer outro. A figura 3.5 mostra um esquema da arquitetura desse sistema, feito para ser extensível. Ele utiliza os componentes `Position` e `Skeleton`, esse segundo sendo utilizado para guardar o estilo das entidades no XML do mapa. A cada evento do Seer (ilustrados na figura 3.5 pelos espaços marcados com 'S' na fila de eventos) o sistema escaneias as entidades e cria uma mensagem que representa uma "fotografia"(e.g. *snapshot*) da simulação naquele ponto. A frequência com que o Seer tira essas fotografias é especificado na inicialização do sistema.

Criada a mensagem representando o estado da simulação em um ponto ela é colocada numa fila de mensagens. O gerenciador dos consumidores, que executa numa thread separada do simulador, remove as mensagens dessa fila e as passa para uma lista de consumidores. Esses consumidores são funções que vão processar as mensagens, e possivelmente

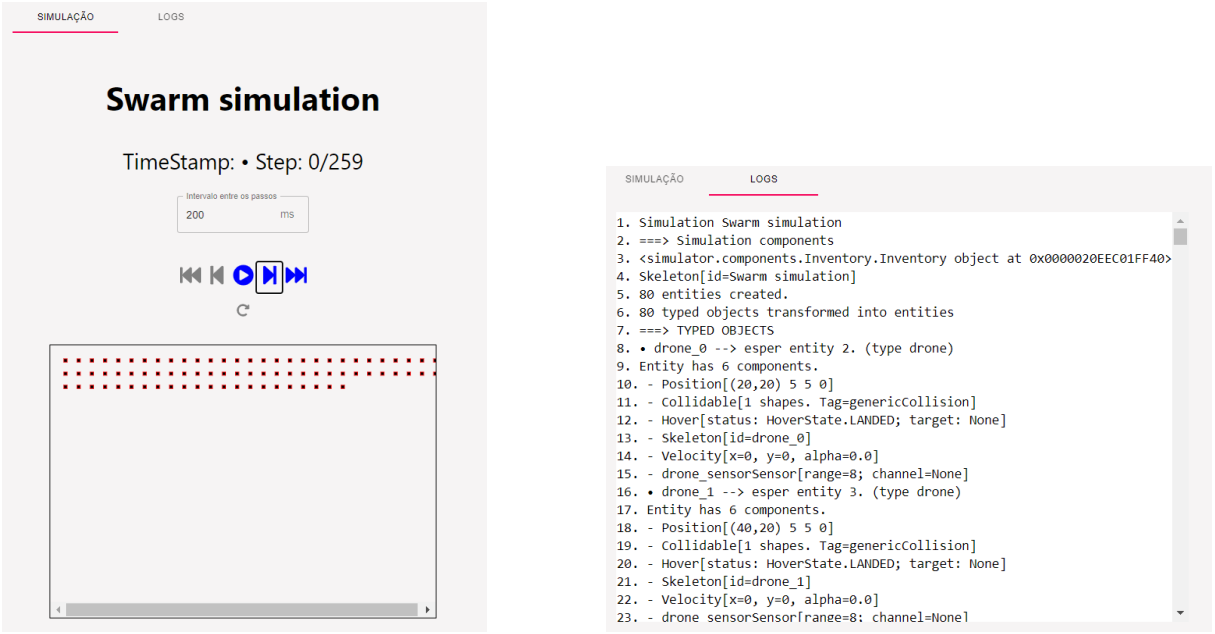

Figura 3.6: Visualizador do Seer disponível em `https://seer-firebase.netlify.app`

enviá-las para outros locais de armazenamento, como mostrado na figura. Os consumidores são passados para o Seer no momento de sua inicialização, e podem ser criados para atender às necessidades particulares do usuário.

Um consumidor de destaque, representado na figura pelo caminho que vai até o Firebase e está conectado ao browser é o consumidor do firebase. Esse consumidor utiliza o banco de dados em tempo real do firebase para enviar as mensagens ao firebase. Um programa auxiliar que faz a tradução das mensagens no firebase e reconstroi a simulação é disponibilizado no endereço `https://seer-firebase.netlify.app`. Ele tem suporte a múltiplas simulações, e também permite visualizar os logs da simulação (ver figura 3.6). A visualização da simulação em si utiliza a mesma biblioteca `jGraph` que a construção do mapa, para manter o máximo de compatibilidade, e pode ser executada passo a passo ou em sequência como um vídeo. As mensagens ficam salvas no banco de dados do Firebase e podem ser vistas depois, permitindo analises posteriores à simulação sem a necessidade de re-executar a simulação toda, um processo que pode ser custoso e demorado.

Outro sistema que exemplifica a possibilidade de extrair informações da simulação é o sistema `ClockSystem`, que registra o tempo gasto para simular 1 segundo da simulação e guarda essa informação em um arquivo externo.

3.6 Desempenho

Referências

[1] Iocchi, Luca, Daniele Nardi e Massimiliano Salerno: *Reactivity and deliberation: a survey on multi-robot systems.* Em *Workshop on Balancing Reactivity and Social Deliberation in Multi-Agent Systems*, páginas 9–32. Springer, 2000. 1

[2] Sykes, Daniel: *Autonomous architectural assembly and adaptation.* Tese de Doutoramento, Citeseer, 2010. 1

[3] Cao, Y Uny, Andrew B Kahng e Alex S Fukunaga: *Cooperative mobile robotics: Antecedents and directions.* Em *Robot colonies*, páginas 7–27. Springer, 1997. 1

[4] Noori, Farzan M, David Portugal, Rui P Rocha e Micael S Couceiro: *On 3d simulators for multi-robot systems in ros: Morse or gazebo?* Em *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, páginas 19–24. IEEE, 2017. 1, 2

[5] Pinciroli, Carlo, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle *et al.*: *Argos: a modular, parallel, multi-engine simulator for multi-robot systems.* Swarm intelligence, 6(4):271–295, 2012. 1

[6] Echeverria, Gilberto, Nicolas Lassabe, Arnaud Degroote e Séverin Lemaignan: *Modular open robots simulation engine: Morse.* Em *2011 IEEE International Conference on Robotics and Automation*, páginas 46–51. IEEE, 2011. 1, 3

[7] Koenig, Nathan e Andrew Howard: *Design and use paradigms for gazebo, an open-source multi-robot simulator.* Em *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, páginas 2149–2154. IEEE, 2004. 1, 3

[8] Hugues, Louis e Nicolas Bredeche: *Simbad: an autonomous robot simulation package for education and research.* Em *International Conference on Simulation of Adaptive Behavior*, páginas 831–842. Springer, 2006. 1, 3

[9] Rohmer, Eric, Surya PN Singh e Marc Freese: *V-rep: A versatile and scalable robot simulation framework.* Em *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, páginas 1321–1326. IEEE, 2013. 1, 3

[10] Maia, Paulo Henrique, Lucas Vieira, Matheus Chagas, Yijun Yu, Andrea Zisman e Bashar Nuseibeh: *Dragonfly: a tool for simulating self-adaptive drone behaviours.* Em *2019 IEEE/ACM 14th International Symposium on Software Engineering for*

Adaptive and Self-Managing Systems (SEAMS), páginas 107–113. IEEE, 2019. 1, 2, 3

[11] Wiebusch, Dennis e Marc Erich Latoschik: *Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems*. Em *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, páginas 25–32. IEEE, 2015. 5, 6

[12] Moran, Benjamin: *Esper*. `https://github.com/benmoran56/esper/commit/850ee6365dcdc6c36c7b02053bef121a98042849`, maio 2020. 6, 13

[13] Bélanger, Jean, P Venne e Jean Nicolas Paquin: *The what, where and why of real-time simulation*. Planet Rt, 1(1):25–29, 2010. 6, 7

[14] Matloff, Norm: *Introduction to discrete-event simulation and the simpy language*. Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August, 2(2009):1–33, 2008. 7

[15] Lünsdorf, Ontje e Stefan Scherfke: *Simpy*. `https://gitlab.com/team-simpy/simpy/-/commit/b29e72af9975aae6cc0f2ffeaff0c5c876c9e644`, abril 2020. 8