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

Desenvolvimento e Teste da Ferramenta HMR Sim:

Um Simulador de Ambientes Multi-Robôs

Giovanni Meneguette Guidini
Cristiane Naves Cardoso

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

Orientadora
Prof.a Dr.a Genaina Nunes Rodrigues

Brasília
2021

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

Desenvolvimento e Teste da Ferramenta HMR Sim:

Um Simulador de Ambientes Multi-Robôs

Giovanni Meneguette Guidini
Cristiane Naves Cardoso

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

Prof.a Dr.a Genaina Nunes Rodrigues (Orientadora)
CIC/UnB

Prof.a Dr.a Carla Cavalcante Koike
CIC/UnB

Prof.a Dr.a Carla Denise Castanho
CIC/UnB

Prof. Dr. Marcelo Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 26 de Maio de 2021

Dedicatória

Este trabalho é dedicado às nossas famílias, com as quais sempre temos a alegria de compartilhar as pequenas vitórias, e celebrar os bons momentos. Este trabalho é, na visão dos autores, uma pequena vitória.

Dedicamos esse trabalho também ao Laboratório de Engenharia de Software (LES) da Universidade de Brasília, na esperança que essa ferramenta possa auxiliá-los em suas pesquisas.

Agradecimentos

Agradecemos a nossa família que sempre nos apoiaram nos nossos estudos, se pudemos nos dedicar à este e tantos outros projetos foi apenas pelo seu apoio e cuidado. Especialmente aos nossos pais um grande agradecimento por tudo que nos transmitiram, muito obrigado. Por estarem sempre conosco ao longo da nossa jornada, muito obrigado. Como disse Isaac Newton, "se eu vi mais longe, foi por estar sobre ombros de gigantes", e vocês foram e sempre serão esses ombros que nos permitem ver mais longe.

Agradecemos também ao nosso mentor Gabriel Rodrigues que nos auxiliou com muita dedicação neste projeto de conclusão de curso e a nossa orientadora Prof.a Dr.a Genaína Rodrigues Nunes por ter aceitado e guiado esse desafio. Ao nosso colega Vítor Fernandes Dullens um agradecimento especial por ter participado do projeto no desenvolvimento do Seer.

Giovanni gostaria também de agradecer Mikael M. Mello por ter me contado sobre esse curso de computação com o qual eu me identifiquei muito, e sou muito grato por isso. Além do Mikael, muita gratidão também à Thiago Veras Machado, Vitor F. Dullens, Gabriel Bessa, André Cássio por compartilharem essa jornada da graduação comigo e deixarem ela muito melhor, mais animada e enriquecedora. Minha graduação termina aqui, mas a amizade de vocês espero levar para a vida toda.

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

Sistemas Multi-Robôs (SMR) têm sido usados há décadas em ambientes dinâmicos para cumprir diversas tarefas. Simuladores para esses tipos de sistemas são ferramentas populares para mitigar certos problemas relacionados ao desenvolvimento de SMRs, por exemplo diminuindo o custo e os recursos necessários para experimentos, permitir testes com hardware não disponível, etc. Apesar de populares entre desenvolvedores para testes, simuladores são menos utilizados para validação e verificação de SMRs devido a fatores como falta de documentação, necessidade de utilizar a interface gráfica acoplada com o simulador e de intervenção manual, dificuldade de criar cenários de testes, etc.

O Laboratório de Engenharia de Software (LES) na Universidade de Brasília (UnB) conduz pesquisas na área de SMRs. Algumas dificuldades relacionadas aos simuladores utilizados foram reportadas, principalmente com cenários com times maiores de robôs. Apesar de existirem diversos simuladores na literatura, cada qual com características próprias, em geral eles requerem recursos computacionais muito altos para simular um número maior de robôs ou possuem restrições no tipo de robôs que são capazes de simular.

Este projeto propõe uma nova ferramenta para simulação de SMRs, o simulador HMR Sim, com baixo nível de detalhamento físico, permitindo que times grandes de robôs sejam simulados. Atenção foi dada à facilidade de criação de simulações e reaproveitamento de entidades entre simulações. Além disso, levando em consideração os problemas relacionados à integração de simuladores no ciclo de verificação e validação de SMRs, o projeto também testou de forma automatizada o comportamento dos robôs, utilizando a metodologia Behavior Driven Development (BDD). Um *framework* para criação e validação de cenários de SMRs é proposto junto ao simulador.

Palavras-chave: Sistemas Multi-Robôs, simuladores, HMR Sim, validação e verificação, testes automatizados, Behavior Driven Development

Abstract

Multi-robots systems (MRS) have been used for decades in dynamic environments to handle various tasks. Simulators for those kinds of systems are popular tools to mitigate certain problems associated with MRS development, by reducing costs and resources needed for experiments, allowing unavailable hardware to be tested, etc. Albeit popular among developers for testing, simulators are less used for verification and validation of MRS because of factors such as lack of documentation, need to use the graphical interface with the simulator, need for manual intervention, difficulty creating test scenarios, etc.

The Software Engineering Laboratory (LES) at University of Brasília (UnB) conducts research in MRS field. Some difficulties have been reported related to the simulators used, specially with scenarios with large robot teams. Although there are many simulators in the literature, each with its own set of characteristics. In general they aare found to require high computational resources to simulate larger robot teams, or have restrictions on the kinds of robots they are able to simulate.

This project proposes a new tool for MRS simulation, the HMR Sim simulator, with low level of physical detailing, allowing for simulations of large teams of robots with reduced resources. Attention was given to the ease of creating simulations and reusing entities across simulations. Furthermore, taking into account the problems related to integrating simulators in the verification and validation cycle, the project also used automatic tests to validate robot behavior, using Behavior Driven Development (BDD). A framework is proposed for the creation and validation of MRS scenarios.

Keywords: Multi-Robot Systems, simulators, HMR Sim, validation and verification, automated tests, Behavior Driven Development

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Sistemas Multi-Robôs (SMR) são sistemas compostos por mais de um agente robótico. Por algumas décadas esses sistemas foram utilizados em diversos contextos para cumprir diferentes tarefas, especialmente em ambientes dinâmicos. SMRs atuam em um espaço ciber físico (i.e. parte do "mundo real"), logo seus agentes estão propensos a mudanças provenientes tanto de outros agentes do sistema como do ambiente em que estão inseridos [11]. Para aumentar a adaptabilidade do SMR, pode-se projetá-lo como um sistema auto-adaptativo, tornando-o capaz de responder a mudanças no ambiente, de maneira a continuar cumprindo objetivos e respeitando os limites impostos ao sistema [24].

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) [5]. Isso traz desafios para o desenvolvimento desse tipo de sistema, principalmente a preparação de experimentos com vários robôs [19]. 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 [19], [20]. Outras vantagens de simuladores incluem:

- Menor custo de tempo e recursos para preparação e execução do experimento;
- Ambientes simulados podem ser mais ricos, complexos e seguros que ambientes reais ou em laboratório [1];
- É possível testar hardware que não está disponível; [20], [6], [1].

De acordo com o *Survey* realizado [1], muitos desenvolvedores de sistemas robóticos confirmam que simuladores são ferramentas populares entre eles e um dos casos de uso mais comuns são os testes.

Diversos simuladores para SMRs existem na literatura, por exemplo Gazebo [14], Simbad [9], CoppeliaSim [21], MORSE [6], Dragonfly [16], entre outros. Cada um desses simuladores foi criado com propostas diferentes. Alguns simulam com alto grau de precisão as partes que compõem um robô e sua interação com o ambiente (e.g. Gazebo, CoppeliaSim, Morse), outros realizam simulações com grau de abstração mais alto, focando principalmente no comportamento dos robôs (e.g. 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 simulador tem dificuldades ao simular mais de 10 robôs [19]) - 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 [16]).

Ainda que simulações tenham um bom potencial, testes de campo são mais utilizados para validação e para verificação em sistemas robóticos do que os testes baseados em simulações. Alguns problemas dificultam o uso de simulações para fazer a validação e verificação (V&V), dentre eles [1]:

- **Documentação**: Falta de documentação ou documentação errada;
- **Reprodutibilidade**: É difícil repetir o resultado de uma simulação, pois o resultado pode ser não determinístico; e
- **Construção de cenários e ambientes**: Os desenvolvedores relataram ter dificuldade ao criar os cenários para os testes.

E também existem outros desafios que afetam a automação dos testes [1]:

- Não ser possível, ou ser muito custoso, testar a aplicação com a Interface Gráfica desabilitada;
- Não ser possível executar a simulação sem intervenção manual;
- Não ter uma terminação clara da simulação;
- Interfaces de programação instáveis.

Mesmo com os diversos desafios, Afzal et al. [1] concluem que os desenvolvedores estão usando simulações para testar os seus robôs e muitos querem incorporar a simulação na rotina de automação de testes, mas é também sugerido fazer algumas mudanças:

- Prover suporte para simulações sem a parte gráfica (GUI);
- Fornecer interfaces de programação estáveis e com bom design;

- Possibilitar reprodução dos resultados;
- Reduzir os custos de hardware (no caso de simulações com alto custo computacional).

O Laboratório de Engenharia de Software (LES) da Universidade de Brasília (UnB) conduz pesquisas na área de 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, esse nível alto de detalhamento é desnecessário, além de aumentar consideravelmente o tempo de cada experimento.

1.1 Objetivos

Dentro do cenário apresentado, o projeto tem dois focos:

1. Desenvolver uma ferramenta direcionada para simulação de sistemas multi-robôs auto-adaptativo com baixo nível de detalhamento físico.
2. Testar de forma automatizada comportamento de robôs em missões simuladas, levando em consideração as dificuldades citadas pelos desenvolvedores [1].

Dentro do primeiro foco, espera-se que a ferramenta possa ser usada na avaliação e comparação de algoritmos de distribuição de tarefas entre agentes de um SMR. Também pode ser usado para prototipação das características de cada agente do SMR e bem como validação de requisitos do time de robôs de algum sistema. Além disso o processo de criação de simulações deve ser o mais simples possível. Não é esperado que esta nova ferramenta substitua de nenhuma forma os outros simuladores apresentados, mas que seja usada em conjunto com estes. Ela foi desenvolvida para uso em uma etapa anterior no processo de desenvolvimento de um SMR, dentro da abordagem utilizada pelo LES, como detalhado no Capítulo 3.

Considerando o segundo foco, foi utilizado alguns aspectos da metodologia *Behavior Driven Development* (BDD) para ajudar no processo de verificação e validação dos comportamentos das missões. Para facilitar a construção dos testes e a automação deles, também foi desenvolvido um *framework* para ajudar na criação e na validação de cenários de missões robóticas desenvolvidas no HMR Sim. O *framework* proposto foi utilizado durante todo o processo de construção e validação dos testes e também poderá ser utilizado para criar novos cenários em trabalhos futuros, conforme a necessidade.

Objetivos Específicos

Como pontos de validação da ferramenta proposta foram estabelecidos os seguintes objetivos:

- Possibilidade de criar simulações de maneira rápida, preferencialmente aproveitando partes da definição entre diferentes testes;
- O simulador deve ser genérico e capaz de simular diferentes tipos de cenários com diferentes agentes heterogêneos; e
- O simulador deve ser capaz de simular cenários com pelo menos 50 robôs de maneira eficiente.

Para analisar a compatibilidade do BDD com verificações de missões em ambiente de robótica, foram estabelecidos objetivos específicos:

- Implementação da infraestrutura de simulação para o cenários (mapas, robôs, ações);
- Construir um *framework* para auxiliar nos testes das especificações das missões;
- Avaliar o desempenho do uso do BDD em ambiente de simulação de robótica.

1.2 Organização do Trabalho

Este texto está organizado da seguinte maneira: o Capítulo 2 apresenta o referencial teórico que fundamentou a realização do projeto e os simuladores levantados na literatura; o Capítulo 3 apresenta a ferramenta HMR Sim, detalhando seu principal caso de uso, funcionamento geral e detalhes de implementação; o Capítulo 4 aborda os testes realizados no simulador, bem como o *framework* proposto para facilitar e automatizar testes; finalmente o Capítulo 5 traz as conclusões do trabalho e trabalhos futuros.

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 discute sobre o uso de composição e a diferença entre herança e composição. A composição permite criar tipos complexos combinando componentes de vários tipos, sendo ela a base do padrão *Entity-Component-System*. A Seção 2.3 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. A Seção 2.4 comenta sobre duas técnicas de simulação encontradas nos simuladores que foram levantados na literatura. Por fim, a Seção 2.5 apresenta o BDD (*Behavior Driven Development*), que foi utilizado no processo de verificação e validação do HMR Sim. Uma das ideias do BDD é validar o comportamento de uma determinada funcionalidade, sendo que no contexto desse projeto, ele foi utilizado para criar diversos cenários de missões de simulação e para verificar o comportamento da execução destes cenários.

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 [14], Simbad [9], CoppeliaSim [21], MORSE [6] e Dragonfly [16]. 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 *plugins* 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 a 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 a 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. 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 a simulações 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

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
MORSE	Baixo/Médio	<30	SIM	OOP/Declarativa	Passos
Simbad	Médio	<10	SIM	OOP	Passos
Dragonfly	Alto	400	NÃO	MVC/AOP	Passos

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

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). *Declarativa* se refere à utilização de arquivos que representam um robô ou suas partes (e.g. descrição em arquivo XML); *OOP* se refere à Programação Orientada a Objetos; *MVC* indica o padrão *Model View Controller*; *AOP* faz referência à Programação Orientada a Aspectos.
- **Tipo de simulação** indica qual a técnica de simulação usada, em passos ou de eventos discretos (ver Seção 2.4)

2.2 Composição ao invés de herança

Como pode ser observado na Tabela 2.1, os simuladores observados utilizam em sua maioria métodos declarativos para definir os robôs, ou então Programação Orientada a Objetos (OOP, do inglês *Object Oriented Programming*). OOP tradicionalmente utiliza *herança* entre classes. O simulador criado nesse projeto, detalhado no Capítulo 3, utiliza uma combinação de métodos declarativos e *composição*. Esta Seção discute brevemente a diferença entre herança e composição.

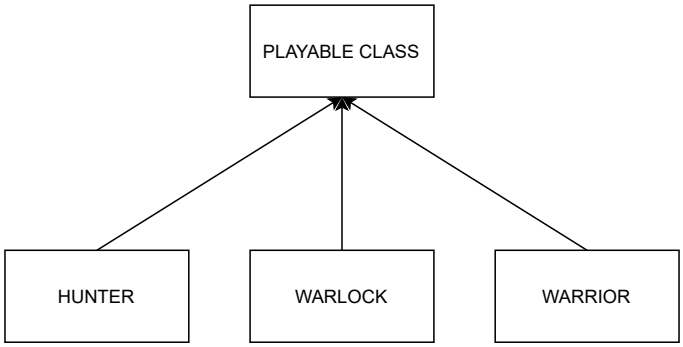


Figura 2.1: Herança Simples.

2.2.1 Herança

Herança permite o compartilhamento de dados e métodos entre classes de OOP, oferecendo uma solução para problemas de reuso e organização de código. Cada objeto no código é criado como uma instância de uma classe de forma que essa classe pode ou não herdar de outra classe. Quando uma classe filha/derivada herda de uma classe pai/-base ela adquire todos os comportamentos do pai. A herança vai formar uma estrutura hierárquica em formato de árvore [4], conforme mostra a Figura 2.1.

Uma hierarquia de classes utilizando Herança Simples é organizada como uma árvore, na qual cada nodo pode ter apenas um parente, ou seja, pode herdar de apenas uma classe. No caso, se uma entidade precisar herdar mais de um componente, este formato de árvore não é suficiente, sendo necessário o uso de Herança Múltipla. Neste tipo de herança, uma classe derivada pode estender de mais de uma classe pai.

Embora pareça uma solução fácil e simples no início, questões relacionadas ao uso de heranças múltiplas irão surgir. Um dessas questões é o problema do diamante, também conhecido como *Deadly Diamond*. Este é um problema de ambiguidade que ocorre quando duas ou mais classes, por exemplo, B e C herdam de uma classe A, e uma outra classe D herda de B e C, conforme mostra a Figura 2.2 adaptada da ilustração de Toni Härkönen [10]. Se existe um método em A no qual ambos B e C subscrevem e o método D não subscreve, então D vai herdar este método de B ou C?

De acordo com Toni Härkönen, este tipo de Herança pode rapidamente se tornar trabalhosa para manter e expandir, uma vez que questões relacionadas a qual cópia da classe base será usada precisam ser resolvidas [10]. Para resolver este problema, será necessário reorganizar a hierarquia de classes para se livrar do *Deadly Diamond*, provavelmente aumentando a repetição de código ou usando uma solução específica da linguagem de programação utilizada [10].

Outro problema no uso de Heranças é o acoplamento rígido da estrutura hierárquica de classes. Utilizando herança, um código com uma grande quantidade de entidades que

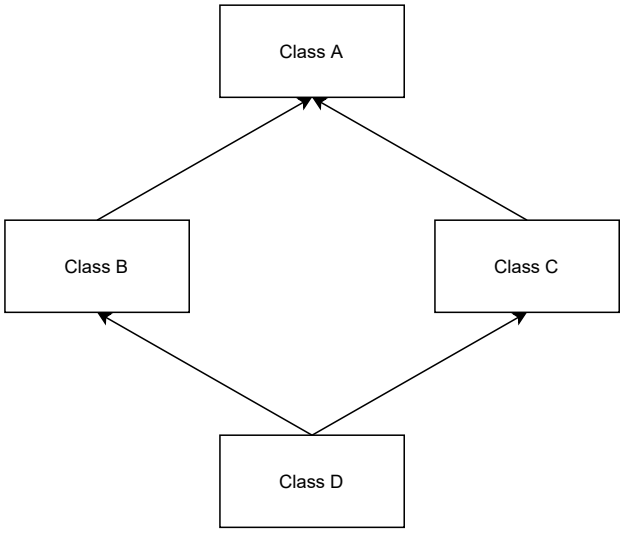


Figura 2.2: *Deadly Diamond Problem.*

herdam de outras entidades rapidamente vai possuir uma hierarquia de classes profunda. Alterar uma classe base, neste caso, pode causar efeitos colaterais indesejados em suas classes derivadas ou mesmo em todo o código [4].

O problema citado anteriormente também é conhecido como *Fragile base class*. Este é um problema de arquitetura da Programação Orientada a Objetos. Nele, as classes base são consideradas frágeis porque modificações aparentemente seguras na classe base, quando herdadas pelas classes derivadas podem causar um mal-funcionamento. Não dá para determinar se uma mudança é segura apenas examinando os métodos da classe base.

De acordo com Vittorio Romeo [22], considerando uma aplicação complexa ou um jogo que faz uso de uma arquitetura tradicionalmente orientada a objetos, as classes base são definidas como raízes de grandes hierarquias, das quais as entidades derivam. Com a adição de novos tipos de entidade, a complexidade do código aumenta, enquanto a capacidade de reutilização, flexibilidade e desempenho diminuem [22].

Uma solução possível, segundo Toni Härkönen [10], é utilizar componentes ao invés de utilizar herança para agregar funcionalidades a um objeto. Resolvendo o problema do diamante e o problema do acoplamento rígido entre as classes.

Para Vittorio Romeo, uma abordagem alternativa mais poderosa consiste em usar um Design Orientado a Dados ou Composição, onde o código é projetado em torno dos dados e seu fluxo, e as entidades são definidas como um agregado de componentes [22].

O uso de Composição surge como uma alternativa ao uso de Herança.

Figura 2.3: Objeto construído combinando vários componentes [8].

2.2.2 Composição

A Composição permite a criação de tipos complexos combinando componentes de vários tipos, em vez de herdar de uma classe base ou pai [4]. Segundo Brian [4], a composição contém instâncias de outras classes que implementam a funcionalidade desejada.

Fazendo uma analogia, é possível imaginar a Composição como um objeto montado de blocos de Lego. Várias peças de Lego (componentes) combinadas criam algo mais complexo, conforme mostra a Figura 2.3.

Este tipo de solução considera as mudanças futuras que podem ocorrer nos requisitos do software. Utilizando composição, não é necessário fazer uma reestruturação da árvore de classes, é possível simplesmente adicionar um novo componente à classe composta, sem modificar a superclasse para adaptar as mudanças [4].

A Figura 2.4 adaptada da ilustração de Toni Härkönen [10] mostra como ficaria essa solução que utiliza um padrão conhecido como *Object-Oriented Composition*. Neste padrão, as entidades são compostas de pequenos componentes, que acrescentam funcionalidades a entidade. Estes componentes podem ser reutilizados em várias outras entidades.

Um problema desta abordagem é que ela não dá suporte a separação dos dados e da lógica, ou seja, o componente possui os dados e a lógica na mesma estrutura. Uma outra abordagem proposta é utilizar um design que gerencia os dados de maneira mais eficaz, utilizando o padrão *Entity Component System*.

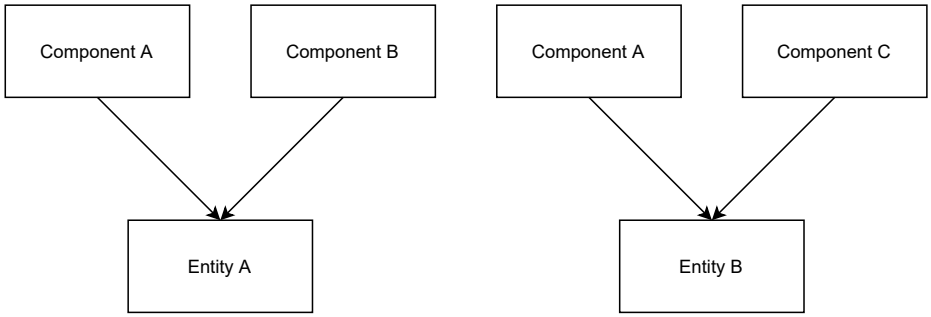


Figura 2.4: *Object Oriented Composition.*

2.3 *Entity-Component-System*

O desenvolvimento de aplicações de tempo real e de jogos pedem por um sistema de gerenciamento de entidades que seja flexível e eficiente. *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*) [25].

Nesse padrão, objetos da simulação são transformados em *entidades*. Uma entidade está relacionada a um conceito específico (e.g. uma parede, um robô, uma pessoa etc), com dados relacionados. Entidades podem ser criadas e destruídas ao longo da execução da simulação, e são usados apenas como identificadores. Cada entidade identifica uma coleção de *componentes*.

Um componente, por sua vez, armazena dados, mas tipicamente não implementa nenhuma lógica [10]. Componentes também podem ser adicionados ou removidos durante a execução da simulação, alterando o estado da entidade a qual pertencem. Componentes representam algum atributo, estado ou capacidade da entidade (e.g. Posição, Velocidade, Sensores, etc). Dessa forma, o padrão ECS consegue separar dados da lógica da aplicação.

A lógica da simulação está nos *sistemas*, que modificam os dados de componentes de acordo com seu objetivo [22]. Em outras palavras, em vez de focar diretamente em uma entidade, os sistemas estão interessados em grupos de componentes que possuem os mesmos aspectos do sistema e que serão processados para realizar uma determinada tarefa [10]. Cada sistema age de maneira independente de outros sistemas sobre um conjunto de componentes que lhe interessa. Ao mudar valores dos componentes, os sistemas efetivamente mudam o estado da simulação. O estado da simulação é o conjunto de estados de todos os componentes de todas as entidades presentes na simulação. Ele é alterado apenas pelos sistemas, cada um alterando uma pequena parte desse estado global.

A Figura 2.5, que é inspirada na ilustração de Cristiano Ferreira et al. [7], mostra o funcionamento de um sistema que usa o *ECS*. Neste exemplo existem duas entidades: Robot e RechargeStation. A entidade Robot possui um componente Battery, Collidable,

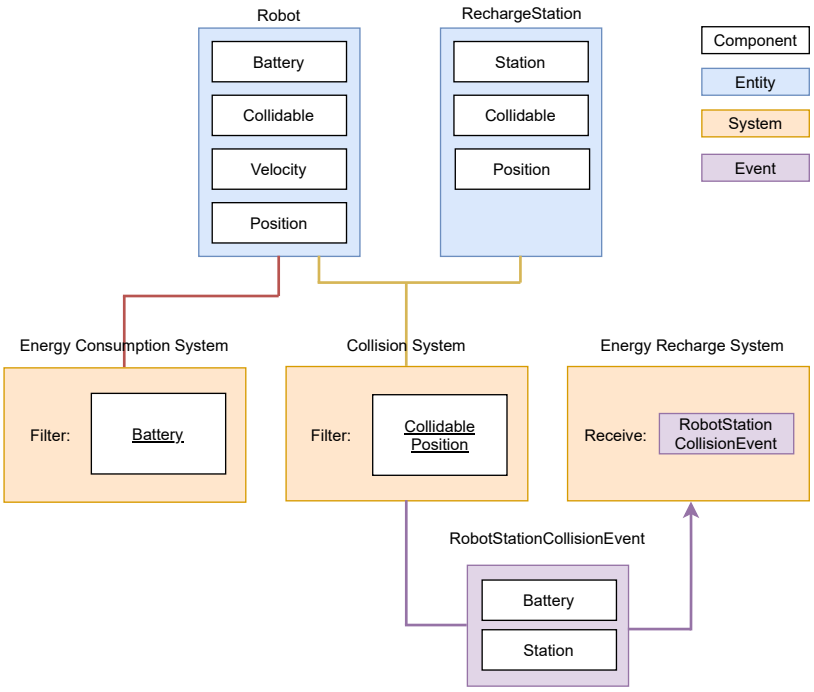


Figura 2.5: Exemplo de sistema construído no padrão ECS.

Velocity e Position e a entidade RechargeStation possui os componentes Station, Collidable e Position.

O Exemplo 2.5 apresenta dois sistemas. `Energy Consumption System` é responsável por consumir (decrementar) o total de energia de todas as entidades que possuem o componente `Battery`, sendo que nesse caso apenas a entidade Robot possui este componente. `Collision System` é responsável por verificar e notificar colisões entre entidades da simulação. Se ocorrer uma colisão entre uma entidade `Robot` e uma `RechargeStation`, significa que esse `Robot` quer recarregar a sua bateria. Sistemas são uma parte importante do simulador e são discutidos em maior detalhe na Seção 3.2.4.

Essa organização permite grande modularização e separação de lógica entre as diferentes partes do sistema. Cada sistema (ou conjunto de sistemas) e seu conjunto de componentes associados pode ser adicionado ou removido do simulador conforme necessário, de maneira independente. Por exemplo, um sistema de comunicação entre diferentes robôs pode ser implementado como um componente que guarde uma fila de mensagens e pode ser adicionado a cada robô, associado a 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 a 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 possui esse componente vai possuir a capacidade de coletar imagens via componente câmera. Remover um componente da entidade remove aquele atributo ou capacidade dela.

Apesar dessas vantagens, como apontado por Wiebush [25], o uso de ECS pode trazer complicações de compatibilidade entre sistemas desenvolvidos de maneira independente, como uso de componentes incompatíveis, e dificuldade em conhecer qual sistema é responsável por determinada funcionalidade e como utilizá-lo. 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 [18]. 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`. `esper` também fornece funções que facilitam a obtenção de componentes ou conjuntos de componentes das entidades salvas, criação e remoção de entidades e componentes, e gerencia a execução dos sistemas.

2.4 Técnicas de Simulação

Uma técnica de simulação bem estabelecida é a de tempo discreto com intervalo de incremento fixo [3]. 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* [3], 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 envolva 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*) [17]. Um evento e possui um tempo t e uma função f que altera o estado da simulação, e potencialmente cria outros eventos, que serão adicionados a uma fila de eventos. Cada estado s_{i+1} é o resultado de processar o evento no começo da fila sobre o estado s_i. 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. 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 infrequente ao longo do tempo, por exemplo o inventário de um armazém [3], 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 contêineres (i.e. 10L de aguá de um reservatório com capacidade para 10000L). Recursos podem ainda ser preemptivos, ou filtrados de algum contêiner. Essa última capacidade foi bastante utilizada para a comunicação entre sistemas do simulador, como será discutido no Capítulo 3.

2.5 *Behavior Driven Development*

O *Behavior-Driven Development*, ou desenvolvimento orientado a comportamento, é uma técnica de desenvolvimento ágil que foi originada do TDD (*Test-Driven Development*) e do DDD (*Domain-Driven Design*). De acordo com John Ferguson Smart [23], o BDD é um conjunto de práticas da Engenharia de Software que tem como propósito melhorar a qualidade do código e a entrega de um software correto.

O BDD foca no comportamento do produto e incentiva a colaboração entre equipes. Lembrando que o BDD é uma técnica de desenvolvimento de produto e os princípios dele são aplicáveis em todos os níveis do desenvolvimento do software [23], incluindo no nível de testes.

Portanto, o BDD envolve mais do que apenas a etapa de testes. O processo se inicia na etapa de levantamento de requisitos. Nessa etapa, os requisitos do software serão discutidos entre pessoas interessadas de diferentes equipes e uma pessoa responsável, geralmente da área de testes, vai descrever estes requisitos em arquivos que podem ser entendidos por todos os envolvidos no processo. Estes arquivos guiarão o desenvolvimento do produto e servirão como uma forma de documentação para que não se perca informações ao longo do processo. No final, será validado se todos os comportamentos descritos nos cenários de cada arquivo estão funcionando de forma correta. Essa validação pode ser feita de forma automatizada, através da automação de testes. Dessa forma, cada cenário de uma funcionalidade é um caso de teste.

Os casos de teste, compostos por entradas, conjunto de passos e saídas, são descritos utilizando uma formatação sugerida pelo BDD, baseada em histórias de usuário. Esses

casos de teste vão descrever, em diversos cenários, o comportamento das regras de negócio de cada funcionalidade.

O processo de desenvolvimento dos testes é similar ao do TDD. Primeiro são feitos testes que vão falhar, depois é escrito o código até que esses testes passem e por fim, o código é refatorado. E esse ciclo se repete. A diferença principal é que os testes no BDD são testes de comportamento.

Um problema que o BDD tenta resolver é a comunicação entre os times e demais envolvidos e a perda de informações dos requisitos das funcionalidades ao longo dos diversos processos. Por exemplo, utilizando o BDD, quando uma pessoa da área de negócios pede por uma nova funcionalidade, ela vai se reunir com o time de desenvolvedores e de testadores e vai explicar os requisitos dessa nova funcionalidade. Nesta reunião, todos os envolvidos vão traduzir essas especificações, utilizando a linguagem `Gherkin`, em cenários e regras que podem ser entendidos por todos os envolvidos. Os diversos cenários e regras descritos vão guiar o desenvolvimento e vão servir como casos de teste [23].

De acordo com [23], o BDD vai ajudar o time a focar na identificação, no entendimento e na implementação das funcionalidades que realmente importam para os clientes.

2.5.1 Linguagem de especificação de cenários: Gherkin

O BDD propõe utilizar uma linguagem que é entendível por todos os *stakeholders* para descrever os requisitos do software, conforme descrito acima. Uma dessas linguagens é o `Gherkin`.

O comportamento das funcionalidades do software serão descritos em vários arquivos que terminam com o sufixo `.feature`. Estes arquivos serão utilizados para guiar os desenvolvedores como parte da documentação e também serão utilizados para testar os vários comportamentos do produto. Cada arquivo `.feature` vai descrever uma funcionalidade, sendo que essa funcionalidade pode ser dividida em mais de um arquivo. Estes arquivos serão compostos por um ou mais cenários.

Os cenários são compostos de várias regras e cada regra pode ser mapeada a um método que vai verificar se ela é executada corretamente.

Uma *feature* pode ser especificada da seguinte forma [2]:

Feature: breve descrição da funcionalidade

Background: agrupa regras que são comuns em todos os cenários.

Scenario: descrição de um cenário específico da funcionalidade.

Given uma pre-condição

And outra pre-condição

When ocorre uma ação

And outra ação ocorre

Then é obtido um resultado testável

Cada keyword do Gherkin[1] descrita acima é responsável por:

- **Feature**: O propósito da *keyword* `Feature` é dar uma descrição de alto nível de uma funcionalidade do sistema e agrupar os cenários relacionados a esta *feature*.
- **Background**: Quando uma ou mais regras `Given` estão sendo repetidas em todos os cenários, elas podem ser agrupadas na seção do `Background`. As regras agrupadas nesta seção serão executadas antes dos cenários.
- **Scenario**: A *keyword* `Scenario` descreve uma regra de negócio da funcionalidade e é composto por várias regras `Given`/`When`/`Then`.
- **Scenario Outline**: Permite executar um cenário várias vezes e a cada vez utiliza exemplos de valores diferentes. Os valores serão definidos com a *keyword* `Examples`.
- **Examples**: Permite criar uma tabela de exemplos. Cada coluna da tabela representa uma variável e cada linha vai conter um valor diferente para a variável. Estes exemplos podem ser usados quando um cenário é executado várias vezes, dessa forma, a cada vez que um cenário é executado é lida uma linha da tabela de exemplos.
- **Steps** Descreve os passos necessários para completar um cenário. Estes passos podem ser dos seguintes tipos:
  - **Given**: Define um estado inicial antes de começar a testar. Nesta etapa podem ser feitas instâncias de objetos, configurações de banco de dados, busca por uma página da web, entre outras configurações necessárias para deixar o sistema em um estado inicial.
  - **When**: Define um evento ou uma ação que será realizada no sistema que está sendo testado. Esta ação pode ser feita por uma pessoa ou por um outro sistema.
  - **Then**: Descreve o resultado esperado da execução das etapas anteriores. Nesta etapa será realizado a assertiva para verificar se a saída recebida é igual a saída que era esperada.
  - **And/But**: Serve como um sinônimo para os passos anteriores a ele e é utilizado para evitar reescritas.

```
Feature: Robot Displacement
```

[1] https://cucumber.io/docs/gherkin/reference/

```
Background:
        Given a simulation

Scenario: The robot stands in the same place
        Given a robot in position 0.0, 0.0
        When the robot does not move
        Then a robot is in 0.0, 0.0

Scenario: The robot moves to a position
        Given a robot in <initial_position>
        When the robot moves to <move_to_position>
        Then a robot is in <final_position>

Examples:
        | initial_position | move_to_position | final_position |
        | 0.0, 0.0         | 1.5, 1.5         | 1.5, 1.5       |
        | 5.0, 5.0         | 12.5, 10.0       | 12.5, 10.0     |
```

Listagem 2.1: Exemplo de um arquivo .feature.

A Listagem 2.1 mostra um exemplo de um arquivo `.feature`.

Capítulo 3

O simulador HMR Sim

Conforme comentado no Capítulo 1, um dos focos deste projeto é fornecer uma ferramenta direcionada para simulação de sistemas multi-robôs auto-adaptativo com baixo nível de detalhamento físico. Essa proposta foi realizada através do simulador HMR Sim[1] - *Heterogeneous Multi-Robot Simulator*, um projeto *open source*, cujo ícone é mostrado na Figura 3.1.

Figura 3.1: Ícone do projeto HMR Sim.

A linguagem escolhida para a construção do simulador 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, nesse sentido 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.

[1] https://github.com/lesunb/HMRSsim

Esta ferramente foi desenvolvida primariamente para ser usada pelo Laboratório de Engenharia de Software (LES) da UnB. Como comentado no Capítulo 1, o LES realiza pesquisas sobre o desenvolvimento de sistemas ciber-físicos (e.g. SMRs), do ponto de vista de engenharia de software. É empregada uma abordagem *top-down*, que parte dos objetivos gerais de um sistema e seus requisitos e utiliza modelos para definir uma série de tarefas que devem ser realizadas pelos agentes do sistema. O controlador do sistema é responsável por calcular o plano da missão, distribuir as tarefas e manter a comunicação entre todos os membros do time, alterando o plano conforme necessário. Cada agente é responsável por realizar a sua tarefa e atualizar o controlador sempre que necessário.

Dentro desse processo, a ferramente HMR Sim pode auxiliar principalmente no desenvolvimento dos algoritmos de *controle* de SMRs e prototipação dos times empregados em missões, agilizando a criação de cenários de validação e oferecendo dados sobre potenciais configurações tanto do time quanto do controlador. Nesse momento do desenvolvimento os detalhes de implementação dos agentes do sistema não é tão relevante, sendo mais importante as suas *habilidades* e *capacidades*. Por exemplo, assumindo que um robô tem a capacidade de pegar objetos de até xKg que estejam a uma distancia de até y metros dele, a maneira como essa operação é realizada e a configuração real do mecanismo que a permite são totalmente abstraídas e somente os efeitos dessa capacidade são simulados (e.g. o robô pega um objeto próximo após algum tempo de processamento que acarreta um gasto de bateria, por exemplo). Por conta disso, o compromisso com a realidade neste simulador, ainda que fundamental, é muito menos restritivo do que em simuladores como Gazebo e MORSE.

É claro que este é apenas o foco principal do simulador, e outros casos de uso podem surgir futuramente conforme o projeto evoluir.

A Seção 3.1 fornece uma visão em alto nível do funcionamento do simulador. A Seção 3.2 traz detalhes de implementação, e como utilizar o simulador, sendo complementada pela documentação do projeto.

3.1 Visão Geral

Sendo a arquitetura do simulador baseada em ECS os conceitos de *entidade*, *componente* e *sistema* são fundamentais dentro do HMR Sim. Tanto os objetos dentro de uma simulação como a simulação em si são construídos por composição: os objetos compostos por diferentes componentes e a simulação através dos sistemas e configuração.

Entidades na simulação estão geralmente relacionadas a objetos físicos. Uma entidade pode representar, por exemplo, um robô, uma parede, um sensor independente, etc. Dentro da simulação, as entidades são gerenciadas pela biblioteca `esper`, e cada uma

possui um identificador próprio. Esse identificador associa essa entidade a um conjunto de componentes que pertencem a essa entidade. No início da simulação, tipicamente uma entidade terá um determinado conjunto de componentes que pode ser alterado ao longo da simulação. Entidades também podem ser criadas ou removidas ao longo da simulação.

Componentes armazenam dados sobre as entidades as quais eles pertencem. Por exemplo, a presença de um componente `Velocity` indica a capacidade de movimentação, e armazena a velocidade atual da entidade; a presença de um componente `Camera` indica que a entidade possui um sensor de câmera, e pode armazenar a última imagem registrada, etc. Componentes podem ser definidos pelo usuário como classes em Python e são carregados automaticamente utilizando um sistema de nomenclatura apropriado. A Seção 3.2.3 cobre a criação de componentes e como eles ficam disponíveis no simulador.

Sistemas definem os comportamentos na simulação. Cada sistema tipicamente acessa um conjunto específico de componentes e utiliza os dados desses componentes para executar alguma ação, atualizando dados conforme necessário. Sistemas podem também gerar eventos. Por exemplo, um sistema de movimentação pode acessar os componentes de posição e velocidade de uma entidade, e usar essas informações para atualizar a posição da entidade, movendo-a pela simulação. Geralmente um sistema, ou conjunto de sistemas, fica responsável por implementar algum comportamento específico na simulação. Por exemplo, um sistema de movimentação é responsável por movimentar entidades na simulação; um sistema de controle pode ser responsável por planejar uma missão e distribuir tarefas entre agentes do sistema, etc.

Um ponto importante é que alguma habilidade associada a um componente qualquer é definida pelo sistema que age sobre esse componente, de forma que com um mesmo componente é possível atribuir diferentes comportamentos à uma entidade. Por exemplo, se um robô possui um componente `Camera`, dependendo do sistema usado essa câmera pode representar uma câmera normal, uma câmera infra-vermelho, uma câmera com visão noturna, etc. Uma consequência desse fato é que o compromisso com a realidade (dentro do que é aceitável na simulação) é em grande parte responsabilidade dos sistemas utilizados na simulação. O usuário deve empregar na simulação sistemas que garantam o grau de realismo esperado por ele nos comportamentos que implementam.

Sistemas são construídos como funções Python, e podem ser processos da biblioteca `simpy` ou funções aceitas como sistemas do `esper` (ver Seções 2.4 e 2.3 para detalhes sobre as bibliotecas). Os sistemas devem ser inicializados e adicionados ao simulador antes da fase de execução.

Uma simulação pode ser definida em um mapa, programaticamente através de um objeto de configuração (dicionário Python), ou uma mistura das duas opções. Mapas são arquivos XML construídos com a biblioteca `JGraph` [12], com algumas restrições.

Qualquer programa compatível com essa biblioteca pode ser utilizado, como por exemplo o editor online e *open source* `diagrams.net` [13]. Também é possível criar entidades na simulação através de *EntityDefinition* (ver documentação do projeto).

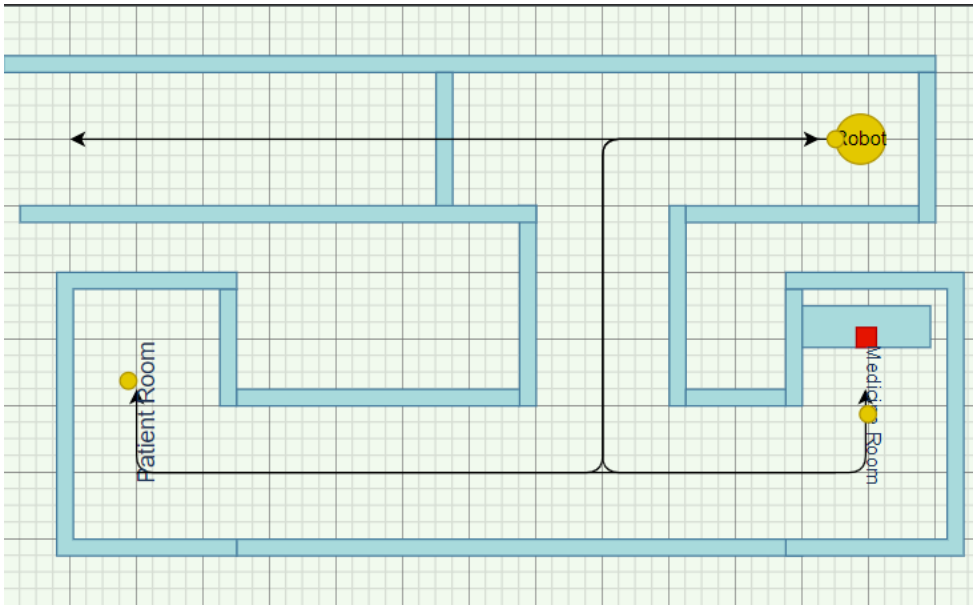


Figura 3.2: Exemplo de mapa de uma simulação

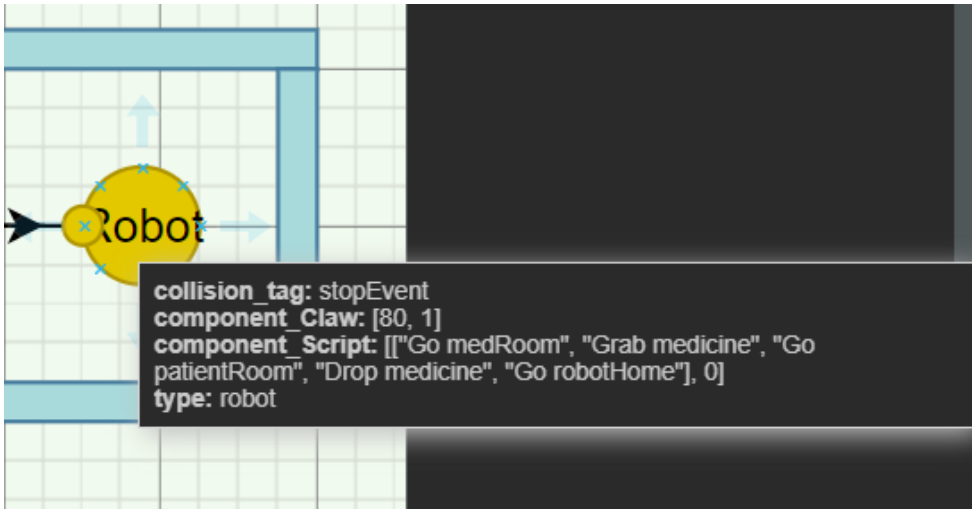


Figura 3.3: Exemplo de anotações em uma entidade

A Figura 3.2 mostra um exemplo de mapa que representa uma simulação. Os retângulos azul-claro são paredes, entidades estáticas na simulação. O círculo amarelo com "Robot" escrito dentro é o robô da simulação. Os outros círculos amarelos denotam pontos de interesse (ou POIs) desse mapa e as setas são rotas que o robô conhece. Essas entidades são usadas pelo sistema de navegação detalhado na Seção 3.2.5. O quadrado vermelho

representa um objeto que pode ser pego pelo robô, caso ele tenha essa habilidade. Esse processo é detalhado na Seção 3.2.5.

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 3.3, por exemplo, as anotações marcam aquele objeto do mapa como um robô, que possui um componente `Claw` inicializado com os valores `[80, 1]`, e um componente `Script` inicializado com os valores da Figura 3.3.

A simulação da Figura 3.2 está disponível nos exemplos de simulação, no repositório do projeto. É a simulação *hospitalScenario*. O robô desta simulação, detalhado na Figura 3.3, possui a habilidade de andar em qualquer direção (definida pelo `builder` utilizado) e habilidade de usar uma garra para pegar certos objetos, habilitada pela presença do componente `Claw`. Além disso, ele possui uma missão, codificada através dos comandos no componente `Script`, que será executada automaticamente.

Uma simulação no HMR Sim tem dois estágios:

1. *fase de carregamento*, onde os componentes disponíveis são carregados e as entidades da simulação criadas;
2. *fase de execução*, onde os sistemas da simulação são inicializados e a simulação em si é executada.

Essa organização permite que as partes estáticas de uma simulação (construídas na fase de carregamento) sejam carregadas um única vez e reaproveitadas em múltiplas execuções. Além disso, os sistemas da simulação são inicializados e adicionados entre as fases de carregamento e execução, como será detalhado mais a frente. Como os sistemas que são responsáveis por definir comportamentos na simulação, é possível que uma parte estática que tenha sido carregada pode ser executada com diferentes comportamentos, trocando os sistemas utilizados.

3.2 Implementação

Esta Seção apresenta detalhes sobre a implementação do simulador HMR Sim, sendo estruturada da seguinte forma: a arquitetura do simulador é detalhada na Seção 3.2.1; a criação e uso de sistemas na Seção 3.2.4; os principais sistemas já construídos são apresentados na Seção 3.2.5. A documentação do projeto está disponível em `https://github.com/lesunb/HMRSsim/wiki`.

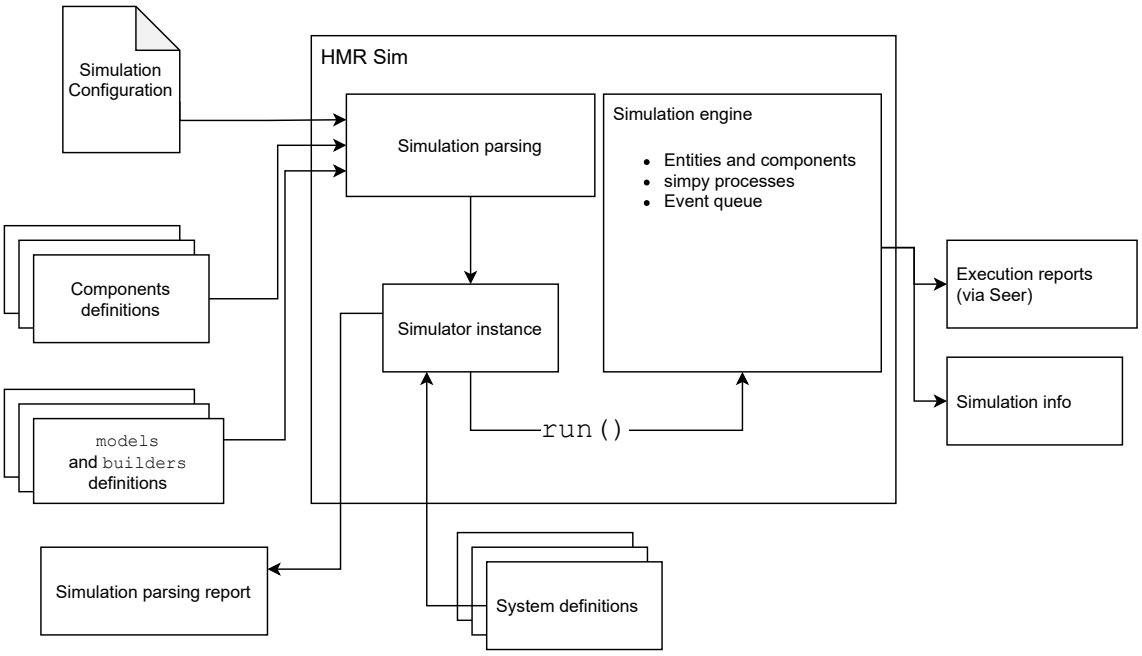


Figura 3.4: Diagrama representando a arquitetura do HMR Sim

3.2.1 Arquitetura

A Figura 3.4 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, `models` e `builders` disponíveis. A classe `Simulator` cria a simulação e depois a executa. A instancia da simulação mostrada na Figura 3.4 (*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 raiz 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 frequente 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 a entidade com identificador '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 definem 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 usam o HMR Sim precisam de uma organização específica, como mostrado na Figura 3.5. Dentro de cada pasta os arquivos deve estar no formato apropriado.

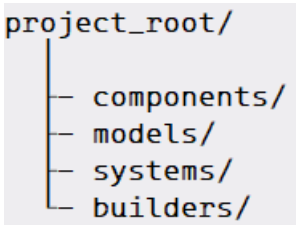


Figura 3.5: Estrutura básica de projetos que usam HMR Sim

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 3.2.4.

Após adicionar os sistemas, a simulação pode ser executada utilizando o método `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 ver logs da simulação durante a execução. A visualização gráfica da simulação é implementada através do sistema Seer, discutido na Seção 3.2.5.

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 3.2.5, e incluem sistema de navegação, movimentação, colisão, controle de robôs, sensores, e visualização.

3.2.2 `builders` e `models`

Os arquivos XML que podem ser usados como 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:

- uma função `from_mxCell`, que recebe uma `<mxCell>` e retorna uma lista de componentes;
- uma constante `MODEL` com o nome da forma que esse model traduz.

`builders` são semelhantes a `models`, mas fazem a tradução do XML contido em `<object>`. Qualquer forma (`<mxCell>`) que tenha anotações (como as mostradas na Figura 3.3) 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:

- uma função `build_object`, que transforma o XML do objeto;
- 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.

3.2.3 Entidades e Componentes

HMR Sim utiliza a biblioteca `esper` [18] (ver detalhes na Seção 2.3) 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).

3.2.4 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: um sistema compatível com sistemas da biblioteca `esper`, referidos como *sistemas esper* ou *sistemas de passos* 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 3.2.4 e 3.2.4 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 é adequado as 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.

Sistemas compatíveis com `esper`

Os sistemas esper são aqueles que ficam armazenados dentro do `World` do `esper`. Eles são definidos como classes que estendem 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 utilizá-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 sistemas 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 na Listagem 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
            # ...
```

Listagem 3.1: Formato básico de um sistema esper

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 sistema 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 assíncrona. 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 do sistema é de responsabilidade dele.

Durante a execução, o 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, estendendo o simulador. A Listagem 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
    # ...
```

Listagem 3.2: Formato básico de um sistema DES

3.2.5 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 foco 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 3.2.5 até 3.2.5) foram testados com simulações que estão disponíveis nos exemplos do projeto (`https://github.com/lesunb/HMRSsim`).

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 4.6). O sistema de movimentação também é responsável 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 utilizadas, 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.

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 robô 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 3.6, 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 com identificador '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 3.6, 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`.

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`

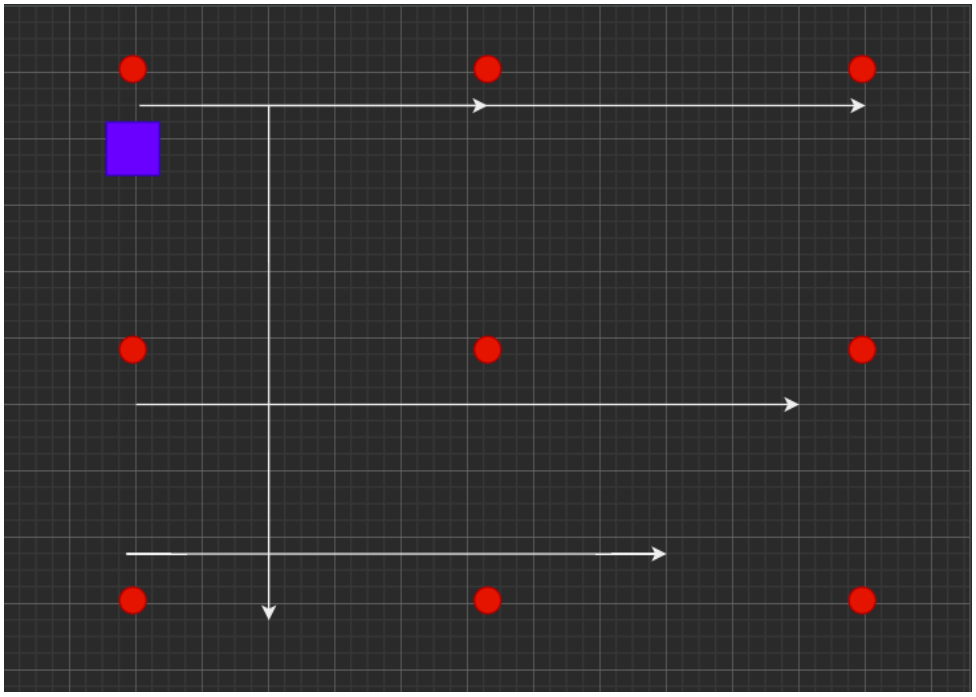


Figura 3.6: Mapa representando POIs (pontos vermelhos) e `map-paths` (setas) para o sistema de navegação

(`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 3.6) 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.

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 3.3, 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 3.3 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 3.2.5).

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.

Sensores e Atuadores

Sistemas que representam sensores e atuadores conferem diferentes habilidades a entidades da simulação. Atuadores podem abstrair componentes inteiros do robô (e.g. garras, plataformas, pinças, etc) e podem ser modulados através de sistemas DES reativos, que reagem a comandos. Sensores por outro lado por terem um comportamento mais constante podem ser também aproximados por sistemas esper. A construção desses sistemas é feita de acordo com requisitos das habilidades do robô, portanto os sistemas implementados e testados são mais como provas de conceito.

Um sistema genérico de sensores foi desenvolvido (`SensorSystem`). Ele pode ser inicializado com uma frequência e um tipo de sensor (um componente) que tenha uma área de atuação (`sensor_range`) e um canal para receber eventos (`reply_channel`, um `simpy.Store`). A ideia é que para cada tipo de sensor dentro da simulação, uma instância desse sistema seria adicionada ao ambiente de simulação, e seria responsável por capturar todas as entidades dentro da área do sensor e enviá-las através de um evento para o sensor de cada entidade através do canal do sensor. Isso abstrai a fase de captura de dados. Um sistema extra que implemente o sensor em si - processamento dos dados e envio de informações - ainda deve ser construído.

Atuadores são mais específicos. A simulação `hospital_scenario` implementa um atuador garra, capaz de pegar objetos iterativos e deixá-los em algum outro lugar. Objetos são marcados como interativos com uma anotação no mapa. Para auxiliar o processo de pegar e deixar objetos, que nessa simulação efetivamente remove a entidade interativa da

simulação e depois a reconstroi em outro local do mapa, o sistema de gerenciamento de objetos pode ser usado (`ManageObjects`). O sistema da garra foi usado também para testar a criação de instruções para o sistema de controle, exportando duas instruções.

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.

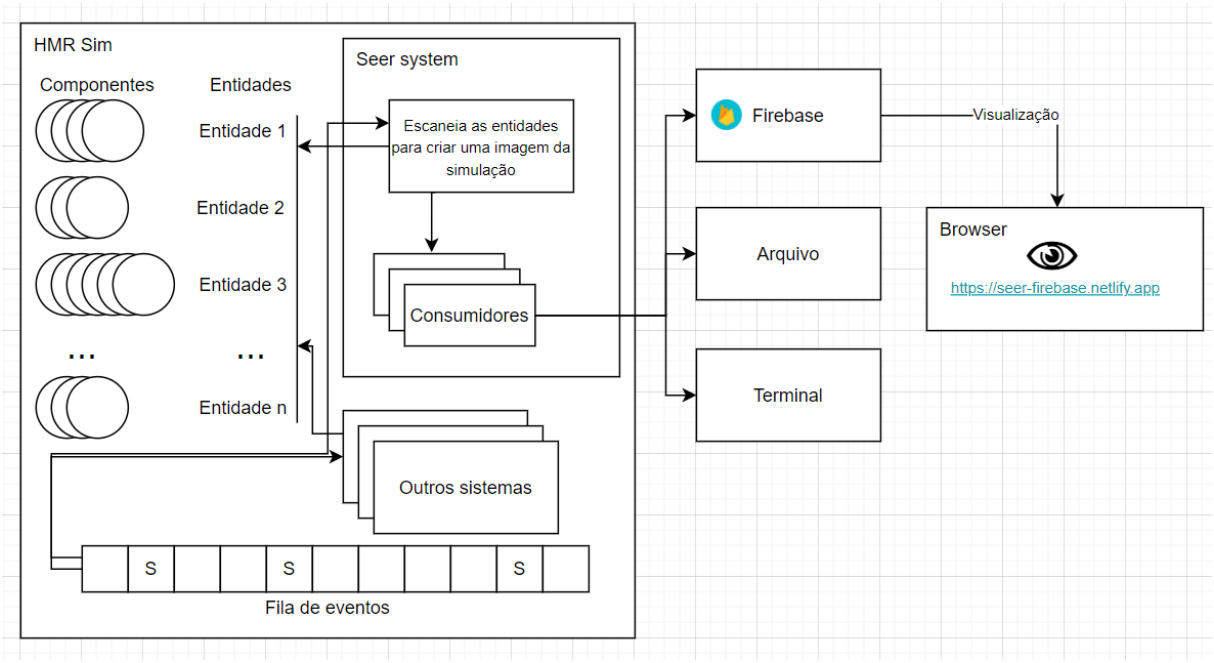


Figura 3.7: Esquema da arquitetura do sistema Seer

Seer é um sistema opcional como qualquer outro. A Figura 3.7 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 3.7 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

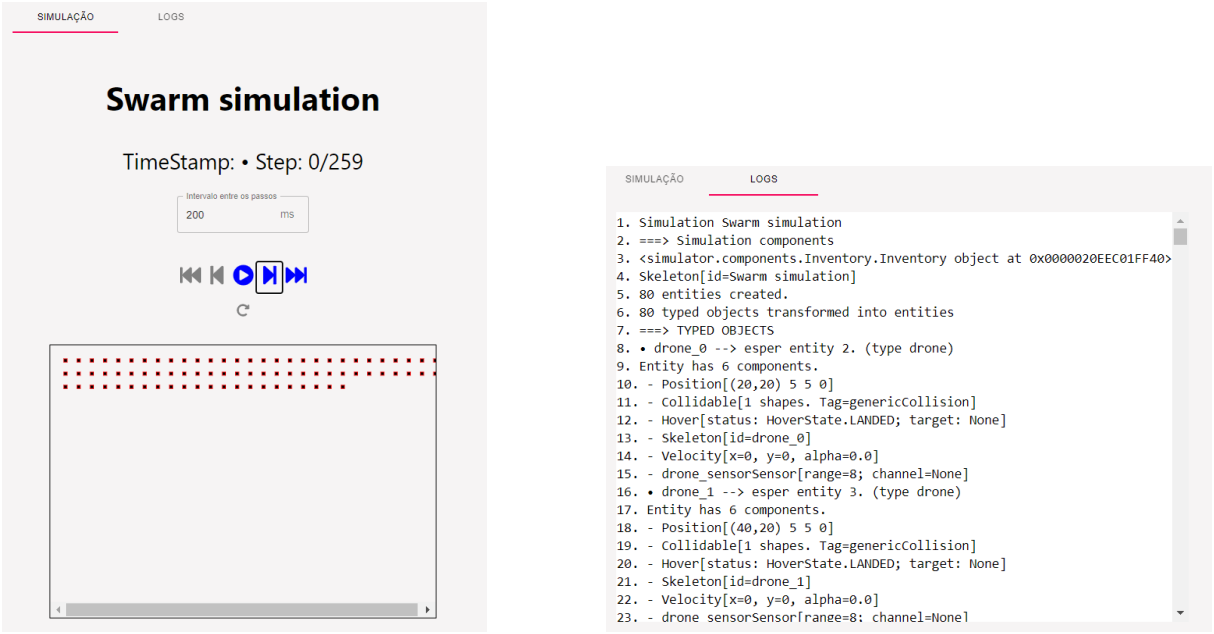


Figura 3.8: Visualizador do Seer disponível em `https://seer-firebase.netlify.app`

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 enviá-las para outros locais de armazenamento, como mostrado na Figura 3.7. 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 3.7 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 3.8). 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.

Capítulo 4

Testes

Os testes no contexto do simulador HMR Sim possuem duas finalidades principais.

A primeira finalidade é testar o simulador em si. Sendo que os principais objetivos em testar o simulador são:

- Verificar se as funcionalidades básicas do simulador estão funcionando. Se é possível instanciar uma simulação, configurar o cenário dela (adicionar sistemas, componentes, entidades, etc), executar e ter o resultado esperado. No contexto do simulador HMR Sim, foram criados cenários de testes para as principais funcionalidades, ou seja, para os principais sistemas;
- Identificar erros, auxiliando as demais pessoas desenvolvedoras. Ao longo do desenvolvimento, muitos erros de implementação foram identificados com o auxílio dos testes, por exemplo, erros na leitura do xml (diagrams.net), erros nos `imports`, erro no sistema de sensor após uma mudança no código (identificado após falha no teste), entre outros;
- Verificar se mudanças feitas no código não vão causar falhas em outras partes do simulador. Para isso, é necessário executar todos os testes e ver se eles estão passando antes de dar um `push`.

A outra finalidade é implementar testes automatizados do comportamento de robôs em ambiente simulado. O principal objetivo de implementar esses testes automatizados é não precisar verificar o comportamento dos robôs de forma manual e repetitiva.

Com isso, ao invés de desenvolver um cenário robótico e testar toda vez de forma manual (olhando o resultado no `Seer` e observando o passo a passo do robô para verificar se tudo ocorreu conforme o esperado), é possível desenvolver testes para os cenários e a cada vez que quiser verificar o seu funcionamento, executar o `pytest`. Dessa forma os testes serão feitos de forma automatizada.

Figura 4.1: Estrutura de pasta dos testes.

Para fazer esse processo de verificação e validação do HMR Sim foi proposto utilizar princípios do BDD (*Behavior Driven Development*). Neste projeto, o BDD vai envolver: Descrever o comportamento de sistemas robóticos através de cenários básicos. Depois, essas descrições vão guiar o desenvolvimento desses sistemas e servir como documentação para futuros desenvolvedores do projeto, evitando perder informações importantes. Por fim, será validado se todos os comportamentos descritos nos cenários estão funcionando corretamente através dos testes de aceitação. Essa avaliação foi feita utilizando testes automatizados. Os cenários vão servir como casos de teste, sendo que as regras descritas pelo `Then` são os critérios de aceitação.

Para implementar testes automatizados do comportamento dos robôs, o BDD vai envolver apenas a descrição do comportamento de uma missão robótica através de cenários básicos e a validação do comportamento desses cenários de forma automatizada.

Foi escolhido utilizar o BDD, principalmente, porque é simples de descrever um cenário de missão robótica e de mapear a descrição dos cenários em casos de testes. Um outro motivo para utilizar o BDD: os cenários descritos nos arquivos `.feature` e nos seus arquivos de testes correspondentes servem como uma documentação. Assim, quem for utilizar a simulação e quiser criar novas missões, pode verificar esses arquivos e todos os métodos utilizados para executar os passos `Given`/`When`/`Then` e a partir deles, criar novos cenários.

Dessa forma, conforme mostrado no Capítulo 1 foi proposto o desenvolvimento de um *framework* para ajudar no processo de desenvolvimento dos testes e na automação deles. O *framework* é composto por um conjunto de métodos desenvolvidos para auxiliar na criação e nas assertivas dos cenários, conforme é mostrado na Seção 4.2.

Por fim, a estrutura de pastas usada para fazer os testes é descrita na Seção 4.1 e um exemplo completo da criação de uma funcionalidade utilizando princípios do BDD é descrito na Seção 4.3. No final do capítulo, são discutidos os resultados dos testes, na Seção 4.4, e as capacidades e limitações de testar, na Seção 4.5.

4.1 Estruturação dos Testes

A parte de teste é dividida em quatro pastas, conforme mostra a Figura 4.1:

- **features**: Contém a descrição dos cenários e passos de cada funcionalidade. Cada funcionalidade está descrita em um arquivo que termina com o sufixo *.feature*, que faz uso da linguagem `Gherkin`. Os arquivos dessa pasta são estruturados da seguinte forma:
  - Os passos *Given* são responsáveis por fazer a montagem do cenário, por exemplo, instanciar uma simulação, associar um mapa à simulação, adicionar sistemas, criar pontos de interesse no mapa, criar caminhos, adicionar componentes ao robô.
  - Os passos *When* são responsáveis por executar a simulação, chamando o método para executar a simulação.
  - Os passos *Then* são responsáveis por fazer as assertivas, verificando se a simulação fez o que era esperado.
- **step_defs**: Os arquivos dessa pasta possuem a montagem dos passos da simulação, a execução da simulação e as assertivas. Cada arquivo da pasta step_defs é relacionado com um arquivo `.feature` da pasta features e todos os passos presentes no `.feature` devem estar presentes também no arquivo de testes, conforme mostra a Listagem da *Feature* 4.1 e a Listagem 4.2. No início dos arquivos ocorre a instanciação da simulação e dos helpers (ScenarioCreationHelper e AssertionHelper), que serão usados nos passos *given*, *when* e *then*.
- **helpers**: A pasta helpers possui o *framework* proposto. Nela existem duas classes de métodos auxiliares. A primeira classe, a `ScenarioCreationHelper`, possui métodos para auxiliar na criação dos cenários e a segunda classe, a `AssertionHelper`, possui métodos para auxiliar nas assertivas, contendo mensagens de erros mais completas.

```
Feature: Single Static Robot

Scenario: Single static robot
    Given a map with one room
    And a robot with id 'robot' in position '1,1' inside the
    room
    When the robot stay still
    Then the robot with id 'robot' is in '1,1'
```

```
Scenario: Multi static robots
    Given a map with one room
    And a robot with id 'robot' in position '1,1' inside the
    room
    And a robot with id 'robot2' in position '25,25' inside
    the room
    And a robot with id 'robot3' in position '42,42' inside
    the room
    When the robot stay still
    Then the robot with id 'robot' is in '1,1'
    Then the robot with id 'robot2' is in '25,25'
    Then the robot with id 'robot3' is in '42,42'
```

Listagem 4.1: Arquivo com a descrição dos cenários single_static_robot.feature

```
scenarios('../features/single_static_robot.feature')

@given("a map with one room", target_fixture="simulation")
def map_with_one_room():
    config = {
        "context": "tests/data",
        "map": "one_room_map.drawio",
        "FPS": 60,
        "DLW": 10,
        "duration": 10
    }
    simulation = Simulator(config)
    return simulation

@pytest.fixture
def scenario_helper(simulation):
    return ScenarioCreationHelper(simulation)

@pytest.fixture
def assertion_helper(simulation):
    return AssertionHelper(simulation)

@given(parsers.parse(("a robot with id '{robot}' in position"
                " '{position}' inside the room")))
def a_robot_in_given_position(scenario_helper,
```

```
                                    robot, position):
    pos = position.replace(' ', '').split(',')
    x = int(pos[0])
    y = int(pos[1])
    scenario_helper.set_position(robot, x, y)

@when("the robot stay still")
def stay_still(simulation):
    simulation.run()

@then(parsers.parse(("the robot with id '{robot}' is"
                     " in '{position}'")))
def check_the_robot_is_in_position(assertion_helper,
                                   robot, position):
    pos = position.replace(' ', '').split(',')
    pos = (int(pos[0]),int(pos[1]))
    assert assertion_helper.is_in_position(robot, pos)
```

Listagem 4.2: Arquivo de teste single_static_robot.py

- **data**: Possui os mapas necessários para a criação de uma simulação. Os mapas são feitos utilizando a ferramenta `diagrams.net`.

4.2 Framework Proposto

O *framework* é composto por duas classes principais, a classe `ScenarioCreationHelper` e a `AssertionHelper`, que herdam da classe `TestHelper`. O objetivo principal é diminuir a complexidade de criar e de testar uma simulação.

A complexidade está principalmente relacionada com a necessidade de entender mais a fundo o funcionamento do simulador para poder montar e testar um cenário. Por exemplo, para adicionar um componente para uma entidade, existem diversas dificuldades, principalmente para quem não tem domínio do simulador. A primeira dificuldade é obter o identificador da entidade, pois para isso é necessário saber qual é o tipo dessa entidade no mapa (`object`, `draw2ent` ou `interactive`) e onde essa informação está guardada. A segunda dificuldade é descobrir como adicionar o componente para a entidade passando o identificador.

Outros exemplos de dificuldade são: como enviar um evento para o robô ir até uma posição, quais sistemas adicionar para que o robô possa se locomover, como adicionar

um ponto de interesse no mapa e como verificar se um robô está posicionado em uma determinada coordenada.

Uma outra complexidade são as mensagens de erros geradas pelo `AssertionError` que não são intuitivas no contexto do simulador. Por conta disso, preferimos lançar o erro através de uma `AssertionError` com uma mensagem customizada nos métodos de assertiva do `AssertionHelper`.

Os métodos auxiliares foram criados para abstrair essas complexidades e para que um desenvolvedor consiga criar e testar de forma mais rápida um cenário de missão, mesmo que ele não entenda todo o funcionamento do simulador.

```
def add_component(self, component, drawio_id):
        """
        Adds a component to the specified entity (drawio_id).

        - component: instance of a Component.
        - drawio_id: property id from an element of the drawio
        file.
        """
        entity_id = self.cast_id(drawio_id)
        self.simulation.world.add_component(entity_id, component)

def cast_id(self, drawio_id: str) -> int:
    """
    Transforms the id present in the drawio xml to the esper
    entity id.

    - In case the object is a pickable, pass the name property as
    the parameter to the drawio_id.
    """
    entity_id = self.get_object_id(drawio_id)
    if entity_id:
        return entity_id
    entity_id = self.get_pickable_id(drawio_id)
    if entity_id:
        return entity_id
    return self.get_drawio_id(drawio_id)
```

Listagem 4.3: Método do `ScenarioCreationHelper` responsável por adicionar um componente

A Listagem 4.3 mostra o método do `ScenarioCreationHelper` responsável por adicionar um componente para uma entidade.

Exemplos de métodos presentes no ScenarioCreationHelper:

- **add_component**: adiciona um componente em uma entidade;
- **create_path**: cria um caminho que parte de uma entidade e passa por diferentes pontos no mapa;
- **add_goto_position_event**: adiciona um evento na `event store` para que o robô se movimente até uma determinada posição;
- **add_ability_to_navigate**: adiciona na simulação todos os sistemas necessários para que o robô consiga se movimentar;
- **add_poi**: adiciona um ponto de interesse no mapa.

Exemplos de métodos presentes no `AssertionHelper`:

- **have_collided**: verifica se uma entidade colidiu com outra entidade;
- **is_in_poi**: verifica se o centro de uma entidade está posicionado em um ponto;
- **approximated**: verifica se um robô X se aproximou de uma entidade Y.

```
import pytest
from pytest_bdd import given
from simulator.main import Simulator
from tests.helpers.ScenarioCreationHelper import
ScenarioCreationHelper
from tests.helpers.AssertionHelper import AssertionHelper

@pytest.fixture
def scenario_helper(simulation):
    return ScenarioCreationHelper(simulation)

@pytest.fixture
def assertion_helper(simulation):
    return AssertionHelper(simulation)

@given("a map", target_fixture="simulation")
def map():
    config = {
        "context": "tests/data",
```

```
        "map": "map.drawio",
        "FPS": 60,
        "DLW": 10,
        "duration": 10
    }
    simulation = Simulator(config)
    return simulation
```

Listagem 4.4: Exemplo de instanciação das classes auxiliadoras

A Listagem 4.4 mostra como instanciar a classe de métodos auxiliares nos arquivos de teste. Pelo código é possível perceber que é necessário passar uma simulação como parâmetro.

4.3 Criando uma funcionalidade utilizando o HMR Sim e princípios do BDD

Como parte da validação do simulador, foi feito um sistema a partir do zero, desde a criação dos cenários e testes até a implementação de novos mapas, componentes e sistemas.

Utilizando o Sistema de Aproximação proposto, o robô é capaz de percorrer um caminho procurando uma pessoa, para isso utiliza um componente do tipo câmera, e se essa pessoa estiver no campo de visão da câmera, ele se aproxima dela.

4.3.1 Sistema de Detecção e de Aproximação

A lógica do sistema de aproximação foi dividida em duas partes. A primeira parte é a responsável por detectar uma entidade solicitada utilizando o `CameraProcessor`. Já a segunda parte é feita através do sistema de aproximação (`ApproximationDESProcessor`) e é a responsável por navegar até a pessoa que foi detectada.

A implementação do sistema da câmera foi feita utilizando um sistema já implementado anteriormente, o `SensorSystem`.

A cada iteração, o `SensorSystem` verifica se existem entidades próximas do sensor. Neste caso, o sensor que será passado como parâmetro é um componente do tipo câmera. Se existirem entidades próximas, o `SensorSystem` envia um evento contendo um vetor de entidades próximas para o *reply chanel* da câmera. Essa etapa é feita utilizando um sistema *Publisher/Subscriber*, no qual o `SensorSystem` (*Publisher*) envia informações para uma câmera que fica esperando por essas informações (*Subscriber*).

Nessa parte entra um segundo sistema, o `CameraProcessor`, que fica esperando receber essas informações que foram enviadas para o *reply chanel* da câmera. Esse processador

recebe como parâmetro uma câmera e o identificador da entidade que será detectada. Assim que recebe as informações de entidades próximas, o processador compara se uma dessas entidades detectadas era o alvo. Se a entidade detectada for igual ao alvo,essa informação é adicionada à lista de entidades detectadas da câmera e também é enviado um evento avisando sobre a detecção.

Para que o sistema de detecção funcione é necessário:

- Adicionar ao robô um componente do tipo Câmera; (colocar o código da função auxiliar);
- Iniciar um `SensorSystem` passando essa câmera como parâmetro e adicionar esse sensor ao conjunto de sistemas discretos da simulação;
- Adicionar um sistema para processar os eventos da câmera.

A segunda e última parte do sistema de aproximação é responsável por receber o evento do `CameraProcessor` que avisa sobre a detecção e a partir disso, o robô se locomove até a entidade detectada. Para fazer a aproximação, o `ApproximationDESSystem` envia um outro evento do tipo `GotoPosEvent` que será recebido pelo sistema de navegação, que já foi implementado anteriormente.

Para utilizar esse sistema é necessário:

- Adicionar as habilidades de navegação, de detecção de entidades e de aproximação;
- Adicionar uma câmera ao robô;
- Adicionar um processador para que o robô procure pela pessoa.

A Figura 4.2, que é inspirada na ilustração de Cristiano Ferreira et al. [7], ilustra o sistema de aproximação. Nela, o `SensorSystem` verifica se entidades que possuem os componente `Camera`, `Position`, `Collidable` e `Velocity` estão próximas de outras entidades que possuem os componentes `Position` e `Collidable`. Se tiver entidades próximas, o `SensorSystem` emite um evento chamado `SensorEvent` que é capturado pelo `CameraProcessor`. O `CameraProcessor` vai verificar se a entidade que ele quer detectar está presente na lista `close_entities` e se estiver presente, ele enviará um evento chamado `DetectedEvent` que será capturado pelo `ApproximationDESProcessor`. Após receber as informações do `CameraProcessor`, o `ApproximationDESProcessor` envia um evento contendo o identificador do robô e a posição de destino do robô para o `GotoDesProcessor`, que será o responsável por movimentar o robô do ponto inicial dele até a posição de destino.

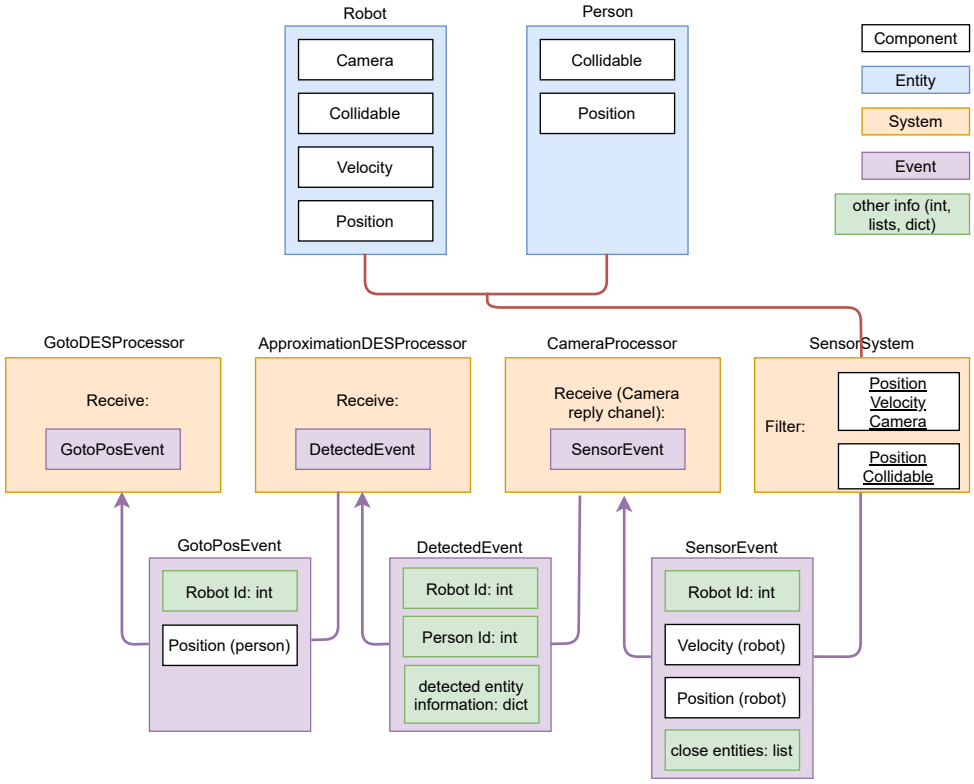


Figura 4.2: Ilustração do sistema de aproximação.

4.3.2 Criando as funcionalidades de detecção e aproximação utilizando princípios do BDD

O passo inicial antes de implementar as funcionalidades foi a realização dos testes. Primeiro foram feitos os testes e a implementação do sistema da Câmera, pois o sistema de aproximação depende dele.

Para isso, foram definidos quatro cenários no arquivo `camera.feature` para descrever o comportamento da nova funcionalidade de detecção. A Listagem 4.5 mostra a definição do primeiro cenário.

```
Feature: Camera

Scenario: The camera receives information from the entities present in its field of view
    Given a map containing robots with camera components
    And a robot with id 'robot' that has a camera component
    And all the simulation robots has detection ability
    And a camera event for robot 'robot' detect a 'person1' that is in the camera field of view
```

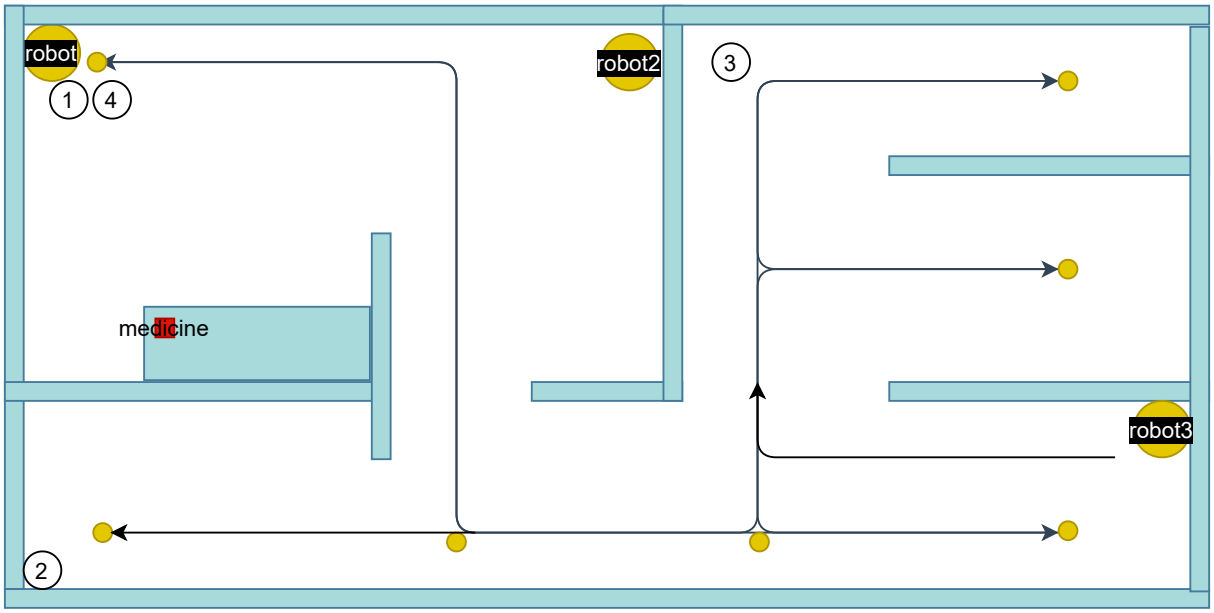


Figura 4.3: Mapa dos cenários de detecção e de aproximação.

```
    When after run simulation
    Then information about the entity 'person1' was detected by
the 'robot' robot camera
```

Listagem 4.5: Definição do primeiro cenário da feature Camera.

Depois de descrever os cenários, foi feito um mapa, utilizando o `diagrams.net`, contendo quartos de pacientes, robôs, pessoas, rotas e pontos de interesse (`POIs`) para o robô se locomover, conforme mostra a Figura 4.3.

Logo após, foi feita a criação de todos os passos necessários para implementar o cenário da nova funcionalidade e para verificar se tudo ocorreu conforme o esperado, no arquivo `test_camera.py`. Nessa etapa é instanciada a simulação e os métodos auxiliares, e também é adicionada a câmera e todos os sistemas necessários, conforme mostra a Listagem 4.6. No entanto, os testes ainda não passam nessa etapa, pois o sistema ainda não foi implementado.

```
@given("a map containing robots with camera components",
       target_fixture="simulation")
def map_robot_with_camera(config):
    config["map"] = "camera_map.drawio"
    simulation = Simulator(config)
    return simulation

@given(parsers.parse(("a robot with id '{robot}' that has a camera"
                      " component")))
```

```
def add_camera_to_robot(scenario_helper: ScenarioCreationHelper,
                        robot):
    scenario_helper.add_camera(robot)

@given("all the simulation robots has detection ability")
def recognition_ability(scenario_helper: ScenarioCreationHelper):
    scenario_helper.add_detection_ability()

@given(parsers.parse(("a camera event for robot '{robot}' detect a"
                      " '{person}' that is in the camera field of"
                      " view")))
def add_event_to_detect_person1(scenario_helper,
                                robot, person):
    scenario_helper.add_camera_detection_event(robot, person)

@given(parsers.parse(("a camera event for robot '{robot}' detect a"
                      " '{person}' that is not in the camera field"
                      " of view")))
def add_event_to_detect_person2(scenario_helper,
                                robot, person):
    scenario_helper.add_camera_detection_event(robot, person)

@when("after run simulation")
def run_simulation(simulation):
    simulation.run()

@then(parsers.parse(("information about the entity '{person}' was"
                     " detected by the '{robot}' robot camera")))
def captured_the_persons_information(assertion_helper,
                                     robot, person):
    assert assertion_helper.detected(robot, person) is True

@then(parsers.parse(("information about the entity '{person}' was"
                     " not detected by the '{robot}' robot"
                     " camera")))
def did_not_captured_the_persons_information(assertion_helper,
                                             robot, person):
    assert assertion_helper.detected(robot, person) is False
```

Listagem 4.6: Arquivo de teste `test_camera.py`

Em seguida, com o auxílio da descrição dos comportamentos presentes no arquivo camera.feature, foi feita a implementação do processador da câmera, o `CameraProcessor`, que verifica se a câmera detecta um determinado alvo, conforme descrito mais acima. Nessa etapa também é criado um componente do tipo `Camera`, que herda da classe `Component`, conforme mostra a Listagem 4.7.

```
from typehints.component_types import Component

class Camera(Component):

    def __init__(self, range=100):
        self.detected_entities = {}
        self.range = range
        self.reply_channel = None
```

Listagem 4.7: Novo componente criado: `Camera`

Retornando aos testes, foi implementado os métodos auxiliares para adicionar o componente de câmera ao robô, para iniciar e adicionar o `SensorSystem` e para adicionar um processador para receber os eventos de detecção da câmera. Também foi feito um método de assertiva para verificar se ocorreu a detecção.

Depois de implementado o sistema de detecção, foi feito o sistema de aproximação, seguindo os mesmos passos descritos acima. Primeiro foram definidos os cenários no arquivo `approximation.feature`, conforme mostra a Listagem 4.9 e depois foi feita a implementação dos testes no arquivo `test_approximation.py`, conforme mostra a Listagem 4.8.

```
@given("a map containing robots with camera components",
       target_fixture="simulation")
def map_robot_with_camera(config):
    config["map"] = "camera_map.drawio"
    simulation = Simulator(config)
    return simulation

@given(parsers.parse(("a robot with id '{robot}' that has a"
                      " camera component")))
def add_camera_to_robot(scenario_helper, robot):
    scenario_helper.add_camera(robot)

@given("all the simulation robots has detection ability")
def recognition_ability(scenario_helper):
```

```
    scenario_helper.add_detection_ability()

@given("all the simulation robots has approximation ability")
def ability_to_approximate(scenario_helper):
    scenario_helper.add_approximation_ability()

@given("all the simulation robots has the ability to navigate")
def ability_to_navigate(scenario_helper):
    scenario_helper.add_ability_to_navigate()
    scenario_helper.add_script_ability()

@given(parsers.parse(("the '{robot}' robot pass through"
                      " POIs '{pois_tag}'")))
def pass_through_pois(scenario_helper: ScenarioCreationHelper,
                      robot, pois_tag):
    pois = pois_tag.replace(' ', '').split(',')
    for poi in pois:
        scenario_helper.add_go_command(robot, poi)

@given(parsers.parse(("a camera event for robot '{robot}' detect"
                      " a '{person}' that is in the camera field"
                      " of view")))
def add_event_to_detect_person3(scenario_helper,
                                robot, person):
    scenario_helper.add_camera_detection_event(robot, person)

@given(parsers.parse(("a camera event for robot '{robot}' detect"
                      " a '{person}' that is not in the camera"
                      " field of view")))
def add_event_to_detect_person2(scenario_helper,robot,person):
    scenario_helper.add_camera_detection_event(robot, person)

@when("after run simulation")
def run_simulation(simulation):
    simulation.run()

@then(parsers.parse("the '{robot}' approximated the '{person}'"))
```

```
def did_approximated(assertion_helper,robot, person):
    assert assertion_helper.approximated(robot, person)

@then(parsers.parse(("the '{robot}' did not approximated the"
                     " '{person}'")))
def did_not_approximated(assertion_helper, robot, person):
    assert assertion_helper.do_not_approximated(robot, person)
```

Listagem 4.8: Arquivo de teste `test_approximation.py`.

```
Feature: Approximation

Scenario: The robot searches for a person and finds them
    Given a map containing robots with camera components
    And a robot with id 'robot' that has a camera component
    And all the simulation robots has detection ability
    And all the simulation robots has approximation ability
    And all the simulation robots has the ability to navigate
    And the 'robot' robot pass through POIs 'intersection1,
    intersection2, patientRoom2, patientRoom3, patientRoom4,
    intersection2, robotHome'
    And a camera event for robot 'robot' detect a 'person3' that
    is in the camera field of view
    When after run simulation
    Then the 'robot' approximated the 'person3'
```

Listagem 4.9: Definição do primeiro cenário da feature Approximation.

O arquivo de mapa usado é o mesmo do sistema da câmera, ilustrado na Figura 4.3. Por fim, foi feita a implementação do sistema de aproximação, que recebe um evento do tipo `Detected` e envia um evento para o sistema de navegação para que o robô vá até a posição da entidade detectada. Nessa etapa, é criado um componente do tipo `ApproximationHistory` para guardar informações sobre a aproximação.

Também foram implementados métodos auxiliares para montar e validar os cenários de aproximação: um método para adicionar a habilidade de aproximação e métodos de assertiva para verificar se aproximou da pessoa.

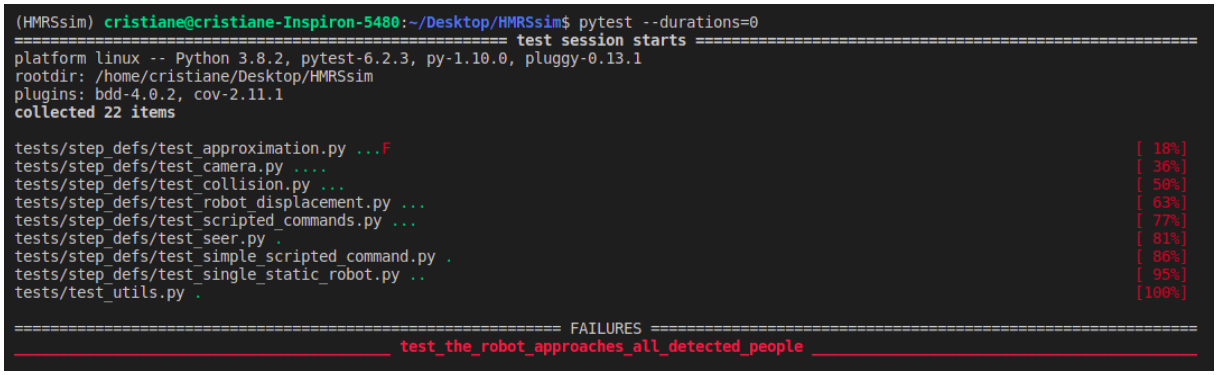


Figura 4.4: Resultados do pytest.

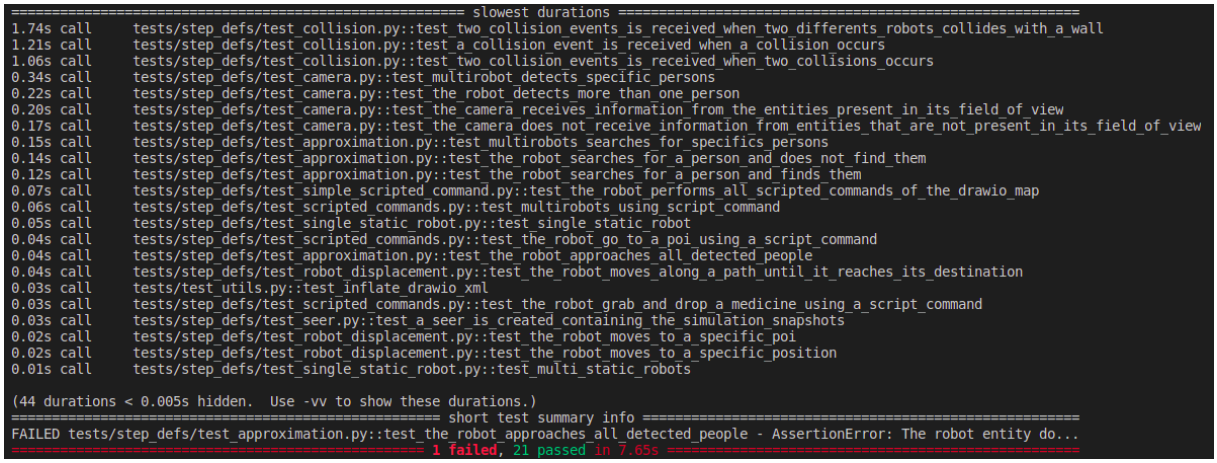


Figura 4.5: Resultados de tempo de execução do pytest.

4.4 Resultados dos Testes

Para executar os testes foi executado o comando `pytest`, para obter o tempo de execução de todos os cenários, do maior tempo para menor, foi utilizado o comando *pytest –durations=0* e para obter o relatório de cobertura em html, foi executado o comando *pytest –cov-report=html –cov=simulator tests/*.

A Figura 4.4 mostra o resultado da execução do `pytest` para todos os cenários citados abaixo. Pela imagem pode-se observar que todos os cenários estão passando, com exceção do último cenário do `test_approximation.py`. Em relação a falha desse teste, uma observação pode ser feita: é possível detectar várias pessoas, conforme mostra o último cenário da *feature* `Camera`, mas só é possível fazer uma aproximação, que será feita até a primeira entidade detectada.

Um ponto que pode ser observado é o tempo gasto para executar os testes da colisão, que é o maior de todos, conforme mostra a Figura 4.5.

Da forma que o cenário foi implementado, o robô sai de um ponto inicial e se movimenta por um caminho determinado até colidir com a parede. O robô fica parado nessa parede

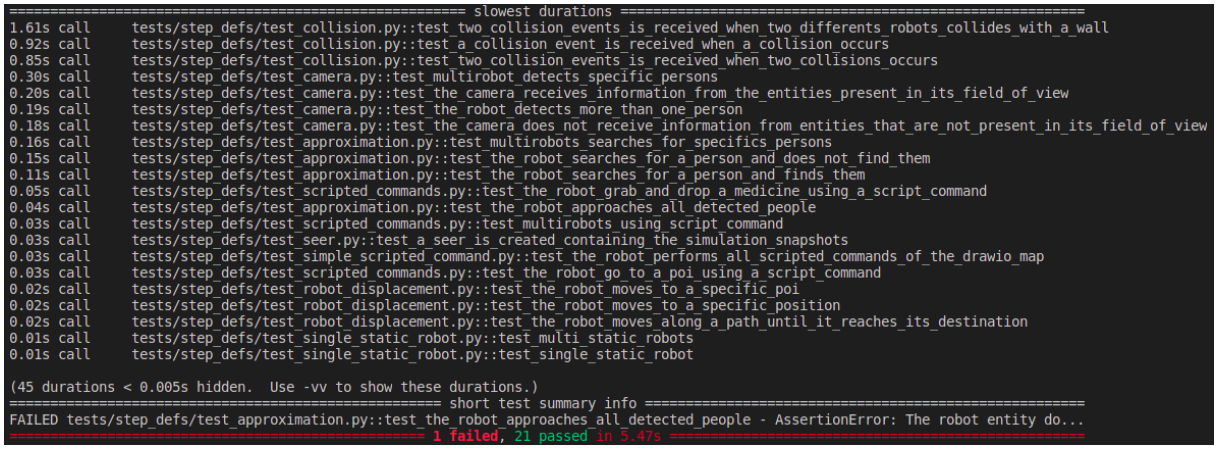


Figura 4.6: Resultados de tempo de execução após refatoração.

até o final da simulação e durante todo esse tempo são enviados eventos dizendo que está ocorrendo a colisão.

O sistema de colisão (`CollisionProcessor`) é um sistema mais custoso que fica ativo durante toda a simulação verificando se ocorreu colisões entre as entidades. Para cada entidade que possui os componentes `Collidable`, `Position`, `Velocity` é verificado se ocorreu uma colisão com cada uma das outras entidades do mapa que possuem os componentes `Collidable`, `Position`, ou seja, é uma interação dentro de outra interação. No entanto, o que está causando o gargalo não é essa interação, mas sim a função `collide`, que é responsável por verificar se uma forma está colidindo com outra. Essa função é chamada dentro de outra função, a `checkCollide` (*simulator/systems/CollisionProcessor.py*). Para diminuir o tempo de execução do sistema de colisão, algumas modificações foram feitas, sendo uma delas a inclusão de um tamanho de setor no `MovementProcessor`.

A Figura 4.6, mostra o tempo de execução após as mudanças feitas no sistema de colisão.

A cobertura total dos testes ficou em 76%, conforme mostra a Figura 4.7. O sistema `EnergyConsumptionDESProcessor` está com a cobertura em 0% porque ainda não foi terminado e não possui nenhum cenário de teste. No entanto, o sistema tem potencial e pode ser continuado em trabalhos futuros.

No geral, os cenários de teste usam todas as principais funcionalidades do sistema, conforme mostrado pela cobertura dos testes.

4.4.1 *Single Static Robot*

É composto por dois cenários. No primeiro cenário (*Single static robot*), o robô com o identificador 'robot' está em uma posição 1,1 no mapa e ao executar a simulação, esse

Coverage report: 76%

Module ↑	statements	missing	excluded	coverage
simulator/__init__.py	0	0	0	100%
simulator/builders/Debug.py	8	3	0	62%
simulator/builders/MapPath.py	52	13	0	75%
simulator/builders/POI.py	30	4	0	87%
simulator/builders/Path.py	44	34	0	23%
simulator/builders/Pickable.py	13	0	0	100%
simulator/builders/Robot.py	20	0	0	100%
simulator/components/ApproximationHistory.py	9	1	0	89%
simulator/components/BatteryComponent.py	8	4	0	50%
simulator/components/Camera.py	8	1	0	88%
simulator/components/Claw.py	7	0	0	100%
simulator/components/Collidable.py	12	0	0	100%
simulator/components/CollisionHistory.py	8	1	0	88%
simulator/components/Inventory.py	6	0	0	100%
simulator/components/Label.py	5	1	0	80%
simulator/components/Map.py	13	1	0	92%
simulator/components/Path.py	9	0	0	100%
simulator/components/Pickable.py	6	0	0	100%
simulator/components/Position.py	25	0	0	100%
simulator/components/ProximitySensor.py	9	4	0	56%
simulator/components/Renderable.py	11	7	0	36%
simulator/components/Script.py	24	0	0	100%
simulator/components/Skeleton.py	10	0	0	100%
simulator/components/Velocity.py	8	0	0	100%
simulator/components/__init__.py	0	0	0	100%
simulator/dynamic_builders.py	11	0	0	100%
simulator/dynamic_importer.py	18	1	0	94%
simulator/dynamic_models.py	9	0	0	100%
simulator/main.py	122	23	0	81%
simulator/map_parser.py	85	28	0	67%
simulator/models/Room.py	26	18	0	31%
simulator/models/Shape.py	44	0	0	100%
simulator/models/Wall.py	25	3	0	88%
simulator/models/WallCorner.py	33	26	0	21%
simulator/models/WallU.py	23	1	0	96%
simulator/mxCellDecoder.py	26	1	0	96%
simulator/primitives.py	117	79	0	32%
simulator/resources/__init__.py	0	0	0	100%
simulator/resources/load_resources.py	23	4	0	83%
simulator/systems/ApproximationDESProcessor.py	41	2	0	95%
simulator/systems/CameraProcessor.py	19	0	0	100%
simulator/systems/ClawDESProcessor.py	108	15	0	86%
simulator/systems/CollisionProcessor.py	46	4	0	91%
simulator/systems/EnergyConsumptionDESProcessor.py	30	30	0	0%
simulator/systems/GotoDESProcessor.py	83	27	0	67%
simulator/systems/ManageObjects.py	61	2	0	97%
simulator/systems/MovementProcessor.py	25	0	0	100%
simulator/systems/NavigationSystem.py	47	4	0	91%
simulator/systems/PathProcessor.py	55	0	0	100%
simulator/systems/ScriptEventsDES.py	79	23	0	71%
simulator/systems/SeerPlugin.py	70	7	0	90%
simulator/systems/SensorSystem.py	33	1	0	97%
simulator/systems/StopCollisionDESProcessor.py	44	8	0	82%
simulator/systems/__init__.py	0	0	0	100%
simulator/typehints/build_types.py	4	0	0	100%
simulator/typehints/component_types.py	7	0	0	100%
simulator/typehints/dict_types.py	24	0	0	100%
simulator/utils/Navigation.py	45	7	0	84%
simulator/utils/create_components.py	15	8	0	47%
simulator/utils/helpers.py	70	10	0	86%
simulator/utils/mxgraph.py	50	33	0	34%
Total	**1863**	**439**	**0**	**76%**

Figura 4.7: Cobertura dos testes.

robô deve permanecer parado, pois não foi enviada nenhuma instrução para o robô se movimentar. Portanto, no final, ele deve estar na posição 1,1.

O segundo cenário (*Multi static robots*) conta com multi-robôs, cada um numa posição diferente e todos esses robôs devem permanecer parados até o final da execução, pois nenhuma instrução foi dada.

Todos os dois cenários passam, pois todos os robôs permanecem na mesma posição, conforme o esperado.

4.4.2 *Simple Scripted Command*

É composto por um cenário (*The robot performs all scripted commands of the drawio map*) no qual o robô lê uma lista de comandos que foram adicionados a ele pelo edit data no drawio: `[["Go medRoom", "Grab medicine", "Go patientRoom", "Drop medicine"], 0]`. É esperado que o robô vá até a sala de medicamentos (`medRoom`), pegue o medicamento 'medicine' presente nesta sala e depois vá até a o quarto do paciente (`pacientRoom`) e deixe este medicamente lá.

O teste passa para esse cenário, pois o robô segue todo o caminho esperado e deixa o medicamento no local correto.

4.4.3 *Scripted Commands*

O teste é composto por três cenários. No primeiro (*The robot Go to a POI using a script command*) é verificado se o robô foi até o quarto de medicamentos usando o comando `Go`. No segundo (*The robot grab and drop a medicine using a script command*) é verificado se o robô deixou o medicamento no quarto do paciente utilizando os comandos `Grab` e `Drop`. No terceiro cenário (*Multi-robots using script command*) é testado se dois robôs realizam determinados comandos de forma paralela, o primeiro robô deixa um medicamento no quarto do paciente e o segundo robô vai até o quarto do paciente. Os cenários descritos utilizam o sistema de comandos (`ScriptEventDES.py`) e os comandos `Go`, `Grab` e `Drop`.

Os testes estão passando para todos os cenários descritos acima, pois os robôs seguem corretamente todos os comandos solicitados.

4.4.4 *Robot Displacement*

O teste é composto por três cenários, no qual cada cenário descreve uma forma diferente de se locomover. Utilizando cada uma dessas três formas, o robô deve parar no mesmo ponto. No primeiro cenário (*The robot moves along a path until it reaches its destination*), a locomoção é feita através de uma seta (*type: Path*) que sai do robô e vai até

um determinado destino. No segundo cenário (*The robot moves to a specific Position*), o robô recebe um comando para ir até uma posição `(x, y)`. Já no terceiro cenário (*The robot moves to a specific POI*), o robô recebe um comando para se locomover até um `POI` (ponto de interesse).

Os três testes passam, pois o robô para no mesmo ponto de destino independente da forma de locomoção e da rota utilizada.

4.4.5 *Collision*

Existem três cenários para testar colisões simples. No primeiro cenário (*A collision event is received when a collision occurs*), o robô colide com uma parede e recebe uma notificação da ocorrência da colisão. No segundo cenário (*Two collision events is received when two collisions occurs*), o robô colide com duas paredes diferentes e recebe a notificação da ocorrência das duas colisões. No terceiro cenário (*Two collision events is received when two differents robots collides with a wall*), é testado se dois robôs colidem com uma parede e se recebem a notificação de colisões de forma separada.

Todos os testes passam, pois os robôs recebem a notificação da colisão com a parede.

4.4.6 *Camera*

O teste é composto por quatro cenários. O primeiro cenário (*The camera receives information from the entities present in its field of view*) é responsável por testar se a câmera detectou uma determinada entidade que estava no campo de visão dela. O segundo cenário (*The camera does not receive information from entities that are not present in its field of view*) é responsável por testar se não foi detectada uma determinada entidade que não estava no campo de visão da câmera do robô. O terceiro cenário (*Multi-robot detects specific persons*) é responsável por testar se multi-robôs detectaram determinadas entidades que estavam em seu campo de visão. Já o último cenário (*The robot detects more than one person*) é responsável por verificar se um robô é capaz de detectar mais de uma pessoa que esteja em seu campo de visão.

Todos os testes passam, pois só são detectadas as entidades que estão no campo de visão da câmera e a funcionalidade serve para um cenário com multi-robôs.

4.4.7 *Approximation*

O teste é composto por quatro cenários. No primeiro cenário (*The robot searches for a person and finds them*), o robô procura por uma pessoa usando uma câmera e assim que essa pessoa é detectada, o robô se aproxima dela. No segundo cenário (*The robot searches for a person and does not find them*), o robô procura por uma pessoa, mas não a encontra

e logo, ele não se aproxima dela. No terceiro cenário (*Multi-robots searches for specifics persons*), é testado se mais de um robô com a capacidade de detectar outras entidades, se aproximam das entidades detectadas. No último cenário (*The robot approaches all detected people*), o robô procura por duas pessoas que estão no campo de visão dele e tenta se aproximar delas.

Os três primeiros testes passam, pois os robôs só se aproximam das entidades detectadas. Já o último teste falha, pois da forma que ocorreu a implementação do *ApproximationDESProcessor* só é possível fazer uma aproximação.

4.4.8 *Seer*

Foi feito um cenário básico (*A seer is created containing the simulation snapshots*) para testar o funcionamento do sistema Seer, que é responsável por obter `snapshots` da simulação. O teste cria um cenário simples de movimentação e adiciona o `Seer` para obter os `snapshots`. Os resultados obtidos serão salvos em um arquivo `seer_report.txt` e depois será verificado se esse arquivo gerado contém os `logs`. No momento, para testar se o arquivo está correto, é verificado se a primeira linha dele possui um dicionário com uma chave chamada *timestamp*.

O teste feito passa, pois o *snapshot* é gerado corretamente pelo sistema do `Seer`.

4.5 Capacidades e Limitações

A princípio, é possível verificar:

- Posições finais de entidades;
- Informações salvas em componentes que são capturadas durante a execução dos sistemas.

Não é possível verificar:

- Informações que não fazem parte do estado final da simulação, a não ser que essa informação tenha sido gerada por um dos sistemas e tenha sido guardada em algum componente que possa ser acessado ao término da execução;
- Informações que mudam a cada vez que a simulação é executada, ou seja, não é possível testar informações que não são possíveis de se reproduzir. Um exemplo de um cenário que pode mudar a cada vez que a simulação é executada seria um cenário de aproximação no qual as pessoas se movimentam de forma aleatória pelo mapa e o robô procura por duas pessoas. Nesse caso, o robô vai se aproximar da primeira

pessoa entre essas duas que ele identificar, lembrando que só é possível fazer uma aproximação. Então dependendo do caminho dessas pessoas, o resultado final varia.

Nos testes são feitos os seguintes tipos de verificações e existem as seguintes limitações:

- ***Robot Displacement***: é verificado se um robô foi até a posição requisitada (posição final do robô), mas não verifica o caminho que ele percorreu.
- ***Scripted Commands***: são verificadas se determinadas entidades estão na posição ou ponto de interesse (*POI*) correto. Um dos cenários, testa se um medicamento (*pickable*) foi deixado em uma determinada posição. No entanto, só se verifica a posição final dele. Para verificar posições intermediárias é necessário guardar em um componente as informações dos locais que o robô deixou o medicamento.
- ***Collision***: é verificado se um robô colidiu com outra entidade, para isso é testado se o identificador da entidade colidida está presente na lista de colisões do robô, guardada no componente `CollisionHistory`. Uma limitação dessa abordagem é que só é guardada a última colisão entre o robô e uma determinada entidade.
- ***Seer***: é possível acessar as mensagens geradas pelo `Seer`, mas é necessário guardá-las em um componente ou em arquivo para depois acessar essas mensagens e testá-las. No entanto, é trabalhoso testar se todos os *logs* estão no formato esperado, e se a formatação das mensagens muda, é preciso mudar as assertivas. Por conta disso, o teste do `Seer` verifica apenas se o arquivo gerado possui pelo menos uma linha no formato *json*, e se essa primeira linha possui uma chave com o nome *"timestamp"*.
- ***Camera***: é possível verificar se uma ou mais entidades foram detectadas por um robô que possui o componente `Camera`. No entanto, uma das limitações é que a câmera guarda apenas uma detecção de um alvo (a última adicionada), por exemplo, se detectar uma pessoa duas vezes, guarda apenas a última detecção.
- ***Approximation***: é verificado se o robô se movimentou até a pessoa detectada, sendo essa informação guardada em um componente do tipo `ApproximationHistory`. No momento não é possível fazer duas aproximações porque a Câmera é removida quando a primeira aproximação ocorre e o `ApproximationHistory` só guarda informações de uma aproximação.

Para verificar passos intermediários, por exemplo, verificar os pontos que o robô parou durante a simulação, é necessário guardar a informação em uma lista para depois acessá-la na hora de fazer a verificação.

Uma outra limitação está presente nos métodos de assertivas (*tests/helpers/AssertionHelper.py*): A maioria dos métodos verifica apenas o caso verdadeiro, caso contrário

lança um erro (*raise AssertionError*). Então para verificar os casos falsos seria necessário fazer modificações nos métodos ou criar novos.

Um dos problemas relatados pelos desenvolvedores que utilizam simuladores é a mudança constante nas interfaces de programação e a falta de documentação, principalmente em relação a essas mudanças. Este problema também é uma limitação atual do HMR Sim e foi observado durante o desenvolvimento dos testes. Essa limitação geralmente ocorria da seguinte forma: Era gasto um certo tempo para entender como funcionavam todos os parâmetros, sistemas e componentes necessários para fazer funcionar um determinado sistema e depois de entender o funcionamento, eram feitos os testes e eles passavam. No entanto, como a ferramenta estava em constante mudanças, quase toda vez que ocorria um *pull* da *branch dev*, todos os testes quebravam por conta de mudanças na forma que eram chamados determinados métodos (mudanças na *API*). Depois de quebrar os testes, era necessário procurar essas mudanças e os erros que estavam acontecendo para refazer os testes.

As mudanças constantes nas interfaces de programação causaram bastante retrabalho. O que ajudou bastante nessa parte foi que o desenvolvedor da ferramenta sempre ajudava quando entrava em contato com ele.

O que teria ajudado e o que pode ajudar em trabalhos futuros:

- Ter uma interface fixa ou ter uma comunicação mais clara da ocorrência de mudanças na interface.
- Executar os testes antes de dar um *push* para verificar erros nas interfaces de programação.
- Uma documentação mais completa e atualizada com todas as mudanças feitas na interface.

4.6 Testes de Desempenho

Os testes descritos a seguir, diferente dos outros testes abordados no capítulo, são testes de desempenho do simulador. Eles não seguiram a técnica do BDD nem usaram o *framework* descrito, pois seu objetivo é ter uma ideia da capacidade do simulador em lidar com simulações maiores.

Um dos objetivos desse projeto é ter um simulador capaz de simular grandes times de robôs (50-200) de maneira eficiente. Nesse sentido, duas simulações foram criadas para analisar diferentes aspectos do desempenho do simulador. A primeira - enxame de drones - descrita na Seção 4.6.1 analisa o desempenho do simulador para simulações com uma grande quantidade de entidades. A segunda simulação - restaurante de Poke -

apresentada na Seção 4.6.2 analisa o desempenho do simulador quando muitos eventos e muitos processos simpy são utilizados. Ambas simulações estão disponíveis no repositório do projeto, nos exemplos.

Todas as simulações foram executadas em ambiente com sistema operacional Windows 10 Home 2004, com processador Intel Core i7-7500U, e 16GB de memória RAM. A versão Python utilizada foi 3.8.5. A versão do simpy foi 4.0.1 e do esper 1.3.

4.6.1 Simulação de Enxame de Drones

A simulação de enxame de drones busca verificar o desempenho do simulador aumentando a quantidade de drones de 20 até 200, e marcando quanto tempo demorou o tempo de processamento de 1 segundo da simulação ao longo da simulação.

A simulação pode ser descrita da seguinte maneira: “Um controlador que conhece configuração para os drones tomarem. Temos uma quantidade grande de drones disponíveis. O controlador decide qual posição na configuração cada drone deve tomar e informa o drone qual sua posição. Cada drone deve tentar movimentar-se até a posição alvo sem bater nos outros drones. Ao chegar em sua posição, deve informar ao controlador e manter sua posição. Se ele bater em outro drone no caminho deve informar ao controlador. O controlador espera a resposta de todos os drones, ou um tempo máximo de 15s antes de encerrar a simulação”.

A Figura 4.8, que é inspirada na ilustração de Cristiano Ferreira et al. [7], ilustra as entidades, componentes e sistemas utilizados na simulação. A entidade *simulator* se refere ao cenário em si, que possui o componente de controle. Cada sistema possui uma descrição breve de suas principais funções. A comunicação entre os sistemas *ControlSystem* e *HoverSystem* é feita indiretamente pela alteração do componente *Hover*; na direção contrária, como indicado na figura, é feita explicitamente através de eventos do tipo *ControlResponse*. O sistema de *HoverDisturbance* afeta somente os drones pairando, causando distúrbios em sua velocidade que devem ser corrigidos pelo *HoverSystem* para que o drone se mantenha na mesma posição.

Essa simulação foi construída programaticamente. A posição inicial e a posição final dos drones na simulação com 80 drones é mostrada na Figura 4.9. Essa simulação pode ser vista pelo visualizador do Seer no usuário "simulator".

A Tabela 4.1 mostra um resumo do desempenho do simulador com quantidades diferentes de drones. Tempo simulado é o tempo total da simulação, tempo mínimo e máximo são o menor e maior tempo do relógio necessário para computar 1s de simulação. Média é a média total em segundos do tempo do relógio para se computar 1s de simulação. A diferença no tempo necessário para computar um segundo de simulação está relacionado

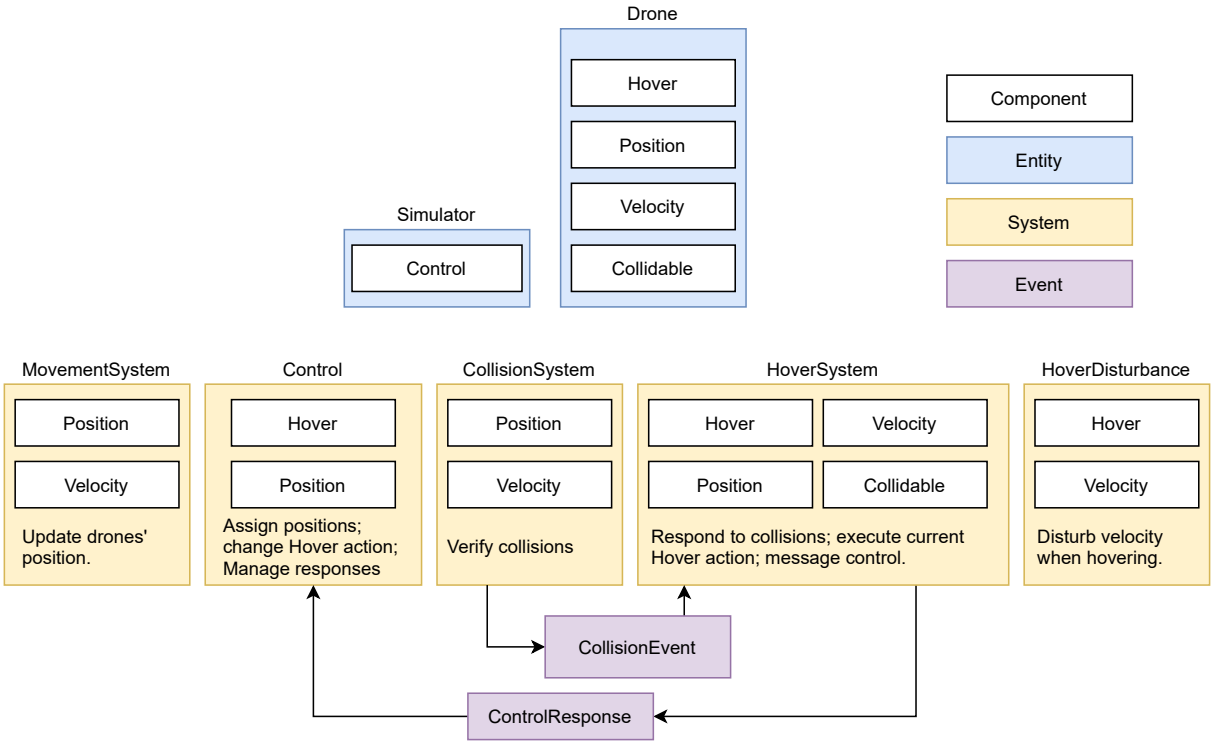


Figura 4.8: Ilustração da simulação de enxame de drones

(a) Configuração inicial

(b) Configuração final

Figura 4.9: Configurações inicial e final do enxame com 80 drones

Nº de Drones	Tempo Simulado	Min. tempo 1s	Max. tempo 1s	Média
20	8s	0.0979s	0.1778s	0.1414s
40	10s	0.2700s	0.3969s	0.3458s
60	20s	0.4404s	0.8829s	0.6042s
80	23s	0.6978s	1.3350s	0.9698s
100	22s	0.9491s	2.3663s	1.4146s
150	25s	1.7435s	3.0516s	2.3176s
200	32s	3.1433s	4.4711s	3.7685s

Tabela 4.1: Desempenho do simulador na simulação de enxame de drones por número de drones

à posição dos drones ao longo da simulação. Nessa simulação, com 80 drones o tempo da simulação está aproximadamente sincronizado com o tempo do relógio.

A Figura 4.10 mostra amédia de tempo de relógio necessário para computar 1s de simulação em função da quantidade de drones simulados. Fazendo a aproximação da função do gráfico com regressão quadrática obtemos a função $0.0001x^2+0.0087x-0.0641$, com média de erro relativo de 4.57%. Usando regressão de expoente o resultado é $0.0018x^{1.4322}$, com erro relativo de 4.23%. Isso indica que o aumento de entidades nessa simulação acarreta um aumento do tempo da simulação de forma polinomial, com um coeficiente próximo de 2.

A Figura 4.11 mostra como a média de tempo de relógio necessário para computar 1s de simulação muda ao longo da simulação, com números diferentes de drones. É possível observar que independente da natureza dos drones, existe uma tendência de aumento do tempo médio até um pico e depois uma queda até o final da simulação. Isso acontece devido à natureza da simulação. Inicialmente todos os drones estão a uma distância constante (ver Figura 4.9), conforme o controlador assinala posições aos drones eles começam a se movimentar na direção do centro do mapa e ficam mais próximos uns dos outros (ver figura 4.12), causando um aumento no tempo de processamento (principalmente devido ao sistema de colisão). Conforme os drones chegam nas suas posições (ver Figura 4.9) o número de entidades ativamente se movendo diminui, e com isso o tempo de processamento médio também. Isso indica que a natureza da simulação afeta diretamente o desempenho do simulador.

4.6.2 Simulação de Restaurante de Poke

Para analisar o desempenho do simulador em simulações com grande quantidade de eventos e de processos sendo executados no ambiente do simpy (i.e. `Simulator.ENV`) a simulação do restaurante de Poke foi criada. Essa simulação utiliza apenas sistemas DES.

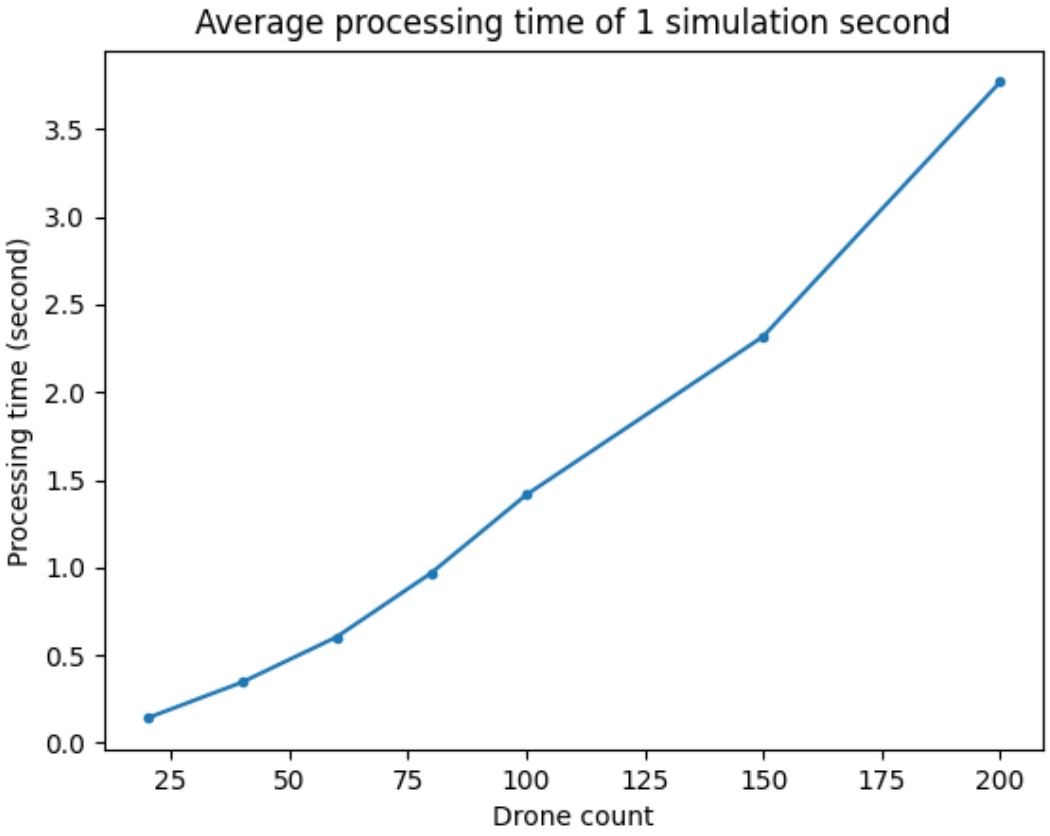
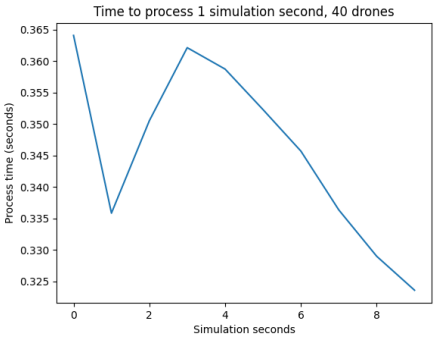


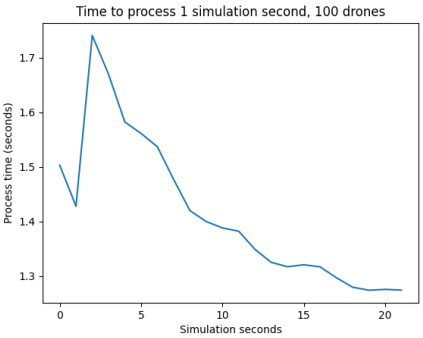
Figura 4.10: Tempo para processar 1s de simulação por número de drones

A simulação pode ser descrita da seguinte maneira: “Clientes fazem pedidos a um restaurante de Poke que devem ser processados e realizados pela cozinha. A gerência da cozinha recebe os pedidos dos clientes e os passa aos cozinheiros, que são responsáveis por realizar os pedidos. Para realizar um pedido, o cozinheiro pega os ingredientes e monta os Pokes nas mesas, utilizando o fogão conforme necessário. As mesas e fogão têm um limite de ocupação.”. A Figura 4.13 mostra uma esquematização do funcionamento da simulação.

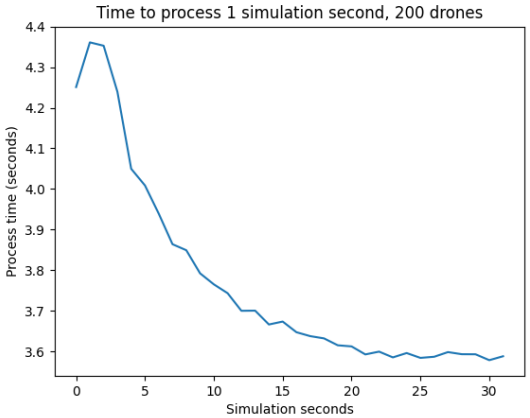
A quantidade de pedidos que chega na cozinha é definida previamente e são sempre mais do que a cozinha consegue processar, para que a quantidade de eventos na fila de eventos cresça ao longo da simulação. A simulação acontece por 1000 segundos de simulação. Para essa simulação, cada segundo de simulação representa um minuto do tempo real, logo são simulados 16 horas e 40 minutos do funcionamento do restaurante. As simulações com 10 e 20 cozinheiros utilizam a mesma quantidade de pedidos. As simulações com 30, 50 e 70 cozinheiros também utilizam a mesma quantidade de pedidos, sendo esta maior que das duas primeiras simulações. A simulação com 100 pedidos utiliza um conjunto ainda maior de pedidos. Cada cozinheiro é modelado por um processo no



(a) 40 drones



(b) 100 drones



(c) 200 drones

Figura 4.11: Tempo médio para computar 1s de simulação ao longo da simulação com 40, 100 e 200 drones

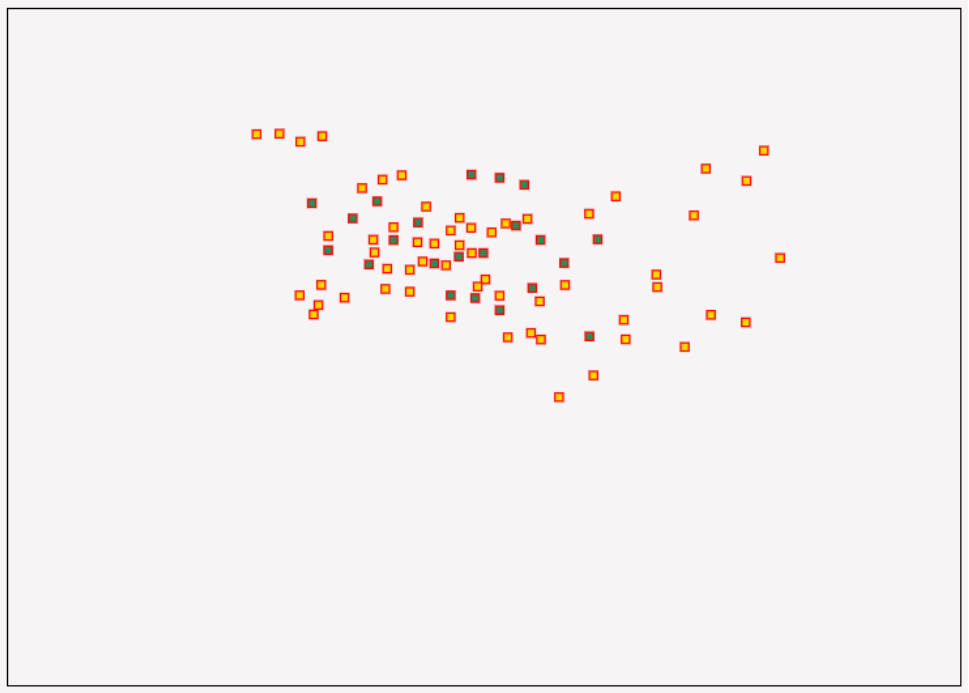


Figura 4.12: Drones próximos uns dos outros ao longo da simulação. Pontos verdes são drones pairando em sua posição. Pontos amarelos são drones movendo-se para sua posição

ambiente do simpy, e o número de cozinheiros cresce de 10 até 100 entre as simulações realizadas.

A Tabela 4.2 mostra um resumo dos dados coletados nas execuções. Tempo total é o tempo total em que a simulação foi executada. Média 1s é o tempo médio necessário para computar 1 segundo de simulação. Delta 1s simulação é a diferença entre o segundo de simulação que mais demorou para ser computado e o que menos demorou. Delta Nº de eventos é a diferença entre a maior e menor quantidade de eventos na fila de eventos ao longo da simulação.

A Figura 4.16 mostra o tempo médio de processamento de 1 segundo de simulação ao longo das diferentes simulações. Nota-se que o tempo médio cresce ao longo da simulação. Nota-se também que o tempo médio cresce mais rápido para uma quantidade maior de cozinheiros. Para analisar melhor essa última observação, a Figura 4.15 mostra as mesmas informações que a Figura 4.16, mas considerando apenas as simulações com 30, 50 e 70 cozinheiros. Essas três simulações usam o mesmo conjunto de pedidos, mudando apenas o número de cozinheiros. Nota-se claramente que a simulação com 70 cozinheiros começa com uma média maior e que cresce mais rapidamente que as outras duas.

A Figura 4.16 mostra o tempo médio para computar 1s de simulação em conjunto com o tamanho da fila de eventos, para diferentes simulações. Nota-se que em todos os casos o tempo médio cresce conforme a fila de eventos cresce. Logo o tamanho da fila de eventos

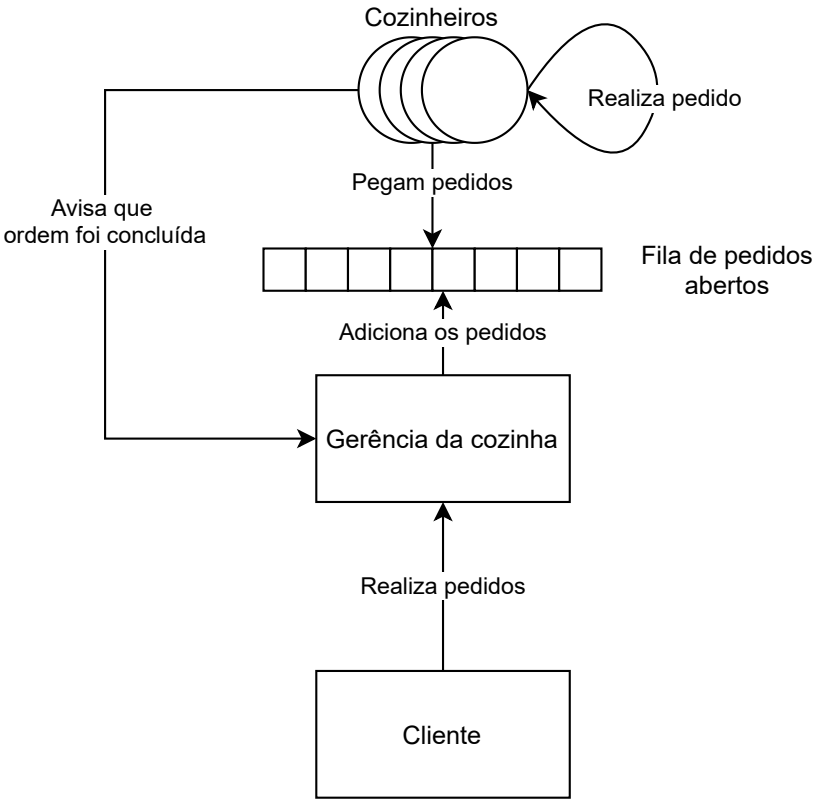


Figura 4.13: Esquema da simulação de restaurante de Poke

é um fator importante para o desempenho da simulação. Cabe observar que todos os eventos dessa simulação foram colocados na `EVENT_STORE` do simulador. É possível que um desempenho melhor fosse conseguido se parte desses eventos (especificamente a fila de pedidos abertos, ver Figura 4.13) fosse movida para uma outra fila de eventos, mas isso não foi testado.

Pode ainda ser percebido que o crescimento do tempo médio de processamento não é constante, mas acontece em "saltos". Esse crescimento acompanha o crescimento da fila de eventos, como mostra a Figura 4.16.

Os tempos de processamento desta simulação foram de maneira geral muito menores do que da simulação de drones. Isso é devido ao grau muito menor de realismo da simulação do restaurante. Todos os sistemas utilizados foram sistemas DES, e o comportamento dos cozinheiros (parte mais pesada da simulação) foi abstraído pelo tempo de preparo da receita, com a única restrição sendo o limite de uso de algumas estações de trabalho.

Cozinheiros	Tempo total	Média 1s	Delta 1s simulação	Delta Nº de eventos
10	00:00.407	0.000250	0.011007	176
20	00:01.580	0.000959	0.043236	235
30	00:16.703	0.008137	0.581195	971
50	00:39.784	0.020657	1.197616	987
70	01:12.934	0.036506	2.341889	994
100	04:50.547	0.136498	11.271133	2641

Tabela 4.2: Informações da simulação do restaurante de Poke

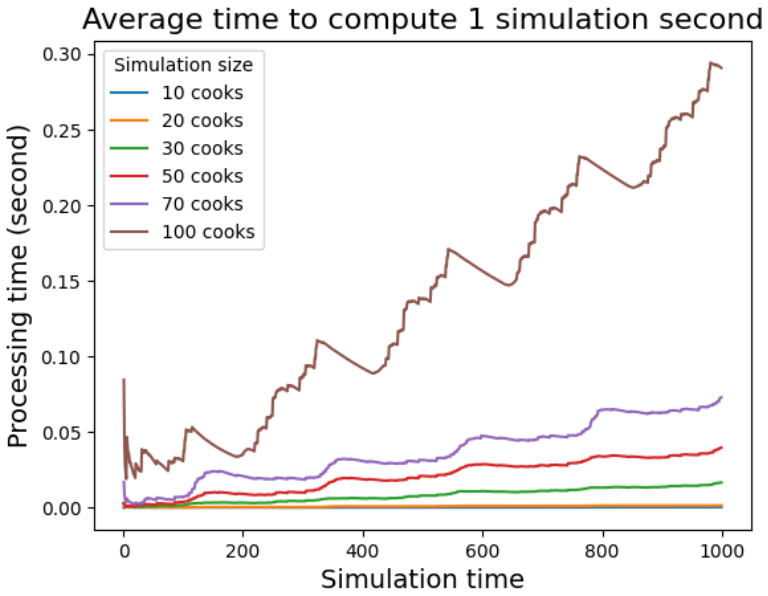


Figura 4.14: Tempo médio para computar 1s de simulação na simulação do restaurante de Poke

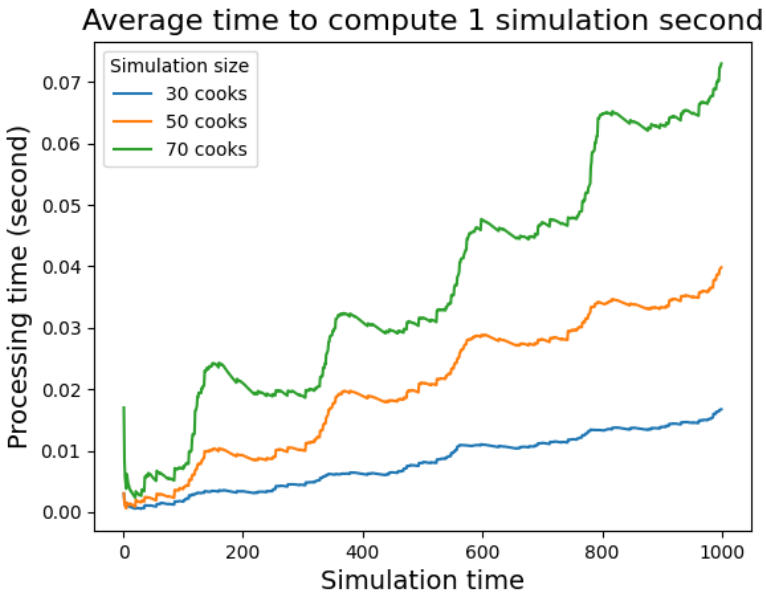
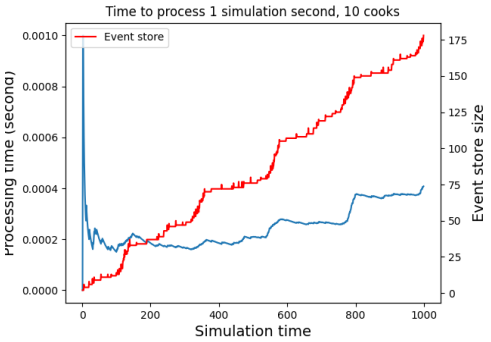
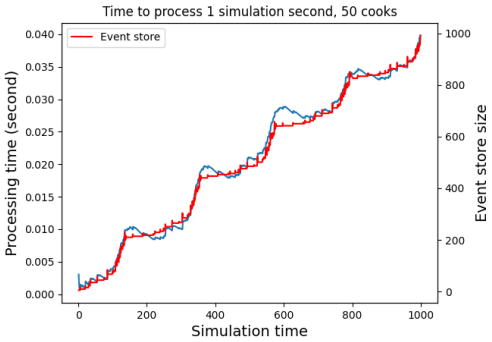


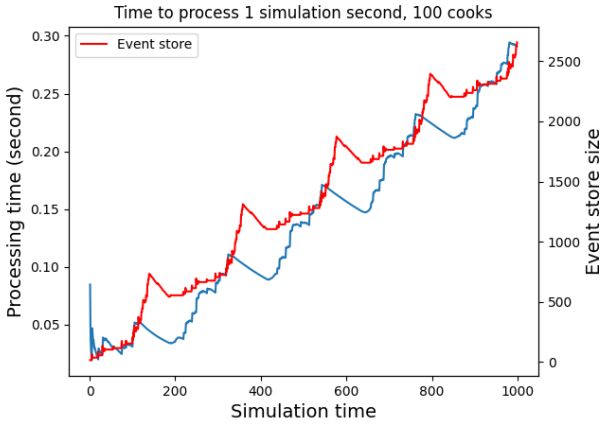
Figura 4.15: Tempo médio para computar 1s de simulação, considerando apenas 30, 50 e 70 cozinheiros



(a) 10 cozinheiros



(b) 50 cozinheiros



(c) 100 cozinheiros

Figura 4.16: Tempo médio para computar 1s de simulação ao longo da simulação e tamanho da fila de eventos.

Capítulo 5

Conclusão e Trabalhos Futuros

Como descrito no Capítulo 1, o projeto realizado teve dois objetivos principais, sendo eles o desenvolvimento de uma ferramenta para simulação de sistemas multi-robôs e a realização de testes automatizados de missões de uma simulação robótica.

Para cumprir com a primeira proposta foi desenvolvido o simulador HMR Sim[1], um simulador para SMR com alto nível de abstração. Em sua construção foi usada a arquitetura ECS, através da biblioteca `esper` e a técnica de simulação DES, através da biblioteca `simpy`. Diversos sistemas considerados comuns em diversos SMRs foram desenvolvidos e testados para validação das capacidades e desempenho do simulador.

Em relação aos objetivos específicos do simulador listados na Seção 1.1, o primeiro foi alcançado através da possibilidade de utilizar mapas definidos com JGraph para representar o cenário da simulação (paredes, caminhos, entidades estáticas) combinado com definições programáticas das entidades dinâmicas. Isso permite que um cenário possa ser utilizado em múltiplas simulações sem alteração.

Analisando o segundo objetivo específico, devido à grande modularidade proporcionada pelo simulador, as capacidades de qualquer entidade que se queira simular podem ser modeladas e adicionadas com relativa facilidade às simulações.

Finalmente, como apresentado na Seção 4.6, o simulador é capaz de lidar com cenários de múltiplos robôs. Na prova de conceito elaborada, 80 robôs puderam ser simulados próximo do tempo real, superando o objetivo inicial.

Ainda assim, o projeto se encontra nos estágios iniciais de desenvolvimento. Até o momento as simulações criadas foram provas de conceito para validar as capacidades e escolhas arquiteturais feitas, espera-se futuramente fazer um estudo de caso mais detalhado e robusto utilizando o simulador. Alguns pontos a serem melhor analisados sobre o simulador e seu desempenho incluem:

- Estudo de usabilidade do simulador dentro da sua proposta;

[1] https://github.com/lesunb/HMRSsim

- Testar o desempenho do simulador na sua fase de carregamento de maneira mais extensiva; e
- Estudo comparativo entre o HMR Sim e outros simuladores apresentados.

Além disso uma quantidade maior de componentes, sistemas e `models` poderia ser desenvolvida e disponibilizada, facilitando ainda mais a criação de simulações. Particularmente implementar suporte para *behaviour trees* diretamente no simulador e implementar suporte para comunicação com ROS, facilitando o uso de robôs já existentes dentro das simulações.

Para cumprir com a segunda proposta, foram feitos testes de aceitação utilizando a técnica do tipo caixa-preta (testes funcionais). Para isso foi utilizada uma das técnicas do desenvolvimento ágil, o BDD, que é o desenvolvimento orientado a comportamento. O BDD foi utilizado majoritariamente para descrever os casos de teste, que são compostos por entradas, um conjunto de passos e as saídas para validar.

Os testes automatizados do simulador HMR Sim foram implementados utilizando a linguagem `Python` e o *framework* `pytest` em conjunto com a biblioteca `pytest-bdd`, que implementa partes da especificação da linguagem `Gherkin`. O resultado final da cobertura de testes ficou em 76%, sendo que foram feitos o máximo de cenários possíveis levando em consideração o tempo previsto. No entanto, essa cobertura de testes pode ser incrementada em trabalhos futuros e o framework desenvolvido facilita a criação de novos cenários de testes, ajudando a aumentar essa cobertura.

Os testes implementados mostraram que é possível compor diferentes sistemas e componentes para montar diversos cenários de missões robóticas e testá-los de forma automatizada. Assim se mostraram úteis para identificar erros, para testar os principais sistemas construídos e para testar o comportamento do robô sem ser necessário intervenção manual e repetitiva.

Várias das dificuldades relatadas por desenvolvedores no Capítulo 1 ao implementar a automação dos testes em um ambiente de simulação foram levadas em consideração. Uma dessas dificuldades é não ser possível executar os testes sem desabilitar a Interface Gráfica. Sendo que, no início do desenvolvimento do HMR Sim, também havia um forte acoplamento entre a parte gráfica e a simulação, tornando inviável desabilitar a interface gráfica e consequentemente, demandando grande esforço para construir os testes. Outros problemas apontados eram saber a hora de término da execução da simulação e a dificuldade de executar a simulação sem intervenção manual. Esses problemas foram discutidos e a arquitetura do simulador modificada para resolvê-los, através do sistema Seer.

Os testes começaram a serem feitos com mais facilidade quando ocorreu essa separação da parte gráfica do restante do código, assim foi possível verificar de forma mais simples

diferentes variáveis e informações da simulação somente pelo código. Se não tivesse ocorrido essa separação, não teria sido viável fazer a automação dos testes.

Um outro problema apontado é a falta de documentação. No desenvolvimento do projeto, tentamos aplicar princípios do BDD, sendo que uma das etapas propostas é descrever o comportamento das funcionalidades em arquivos que sejam acessível e entendível por todas as pessoas envolvidas com o desenvolvimento do produto. Os arquivos que descrevem o comportamento das missões robóticas e dos sistemas desenvolvidos tem como um dos propósitos servir como uma forma de documentação e, portanto, contribuem para resolver o problema da falta de documentação.

A dificuldade na construção de cenários e ambientes também foi um problema relatado. Para contornar essa dificuldade foi proposto e desenvolvido um *framework* para auxiliar na construção dos cenários e na realização das assertivas, conforme relatado na Seção 4.2.

O *framework* de teste desenvolvido facilitou o processo de criação de cenários e o BDD foi uma escolha satisfatória para fazer os testes, pois é simples de descrever cenários de missões robóticas e de mapeá-los a casos de teste e também serve como parte da documentação.

No entanto, uma parte essencial do BDD é a interação entre a equipe no processo de desenvolvimento do produto, desde o levantamento dos requisitos e escrita das *features* até a parte de validação do comportamento dessas *features* pelos testes automatizados. Uma das ideias é que a especificação dos cenários guie o desenvolvimento, ajudando a não perder nenhum requisito. Faltou uma interação maior em relação à etapa de levantamento de requisitos e escrita dos cenários de forma conjunta, principalmente porque alguns dos sistemas testados já tinham sido desenvolvidos.

Para validar melhor o desempenho do BDD em ambiente de simulação robótica, foram feitos dois sistemas a partir do zero. Primeiro foi decidido o que cada um deles tinha que fazer e seus requisitos. Depois foi feita a especificação do comportamento deles nos arquivos `.feature` e os casos de teste. Em seguida, com base na descrição dos cenários, foi implementado os dois sistemas e por fim, foi validado através da automação dos testes o comportamento deles. Várias refatorações e mudanças nas especificações foram feitas até ter o resultado esperado.

Para trabalhos futuros na parte de testes, pretendemos implementar novos métodos para o *framework* de testes para ajudar na criação de novos tipos de cenários. Também pretendemos complementar os testes utilizando o TDD na criação de testes unitários. Pontuando que neste projeto não foram desenvolvidos testes unitários.

Uma outra proposta de trabalho futuro é verificar a viabilidade de utilizar o BDD para implementar testes automatizados do comportamento do robô em outros tipos de simuladores.

Referências

[1] Afsoon Afzal, Deborah S. Katz, Claire Le Goues, and Christopher S. Timperley. A study on the challenges of using robotics simulators for testing, 2020. 1, 2, 3

[2] Rafael Fazzolino P. Barbosa. *Feature-Trace: An Approach to Generate Operational Profile and to Support Regression Testing from BDD Features*. PhD thesis, Universidade de Brasília, 2020. 16

[3] Jean Bélanger, P Venne, and Jean-Nicolas Paquin. The what, where and why of real-time simulation. *Planet Rt*, 1(1):25–29, 2010. 13, 14

[4] Brian. Inheritance vs. composition. `https://medium.com/better-programming/inheritance-vs-composition-2fa0cdd2f939#:~:text=Compositionisincontrastto,thatimplementthedesiredfunctionality`, 2020. [Online; acessado em 24 de Maio 2021]. 8, 9, 10

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

[6] Gilberto Echeverria, Nicolas Lassabe, Arnaud Degroote, and Séverin Lemaignan. Modular open robots simulation engine: Morse. In *2011 IEEE International Conference on Robotics and Automation*, pages 46–51. IEEE, 2011. 1, 2, 5

[7] Cristiano Ferreira and Mike Geig. Entity component system with jobs diagram. `https://software.intel.com/content/www/us/en/develop/articles/get-started-with-the-unity-entity-component-system-ecs-c-sharp-job-system-and-burst-compiler.html`, 2018. [Online; acessado em 24 de Maio 2021]. 11, 45, 60

[8] gyuta Legoist. Lego computer bricintosh '3 in 1'. `https://ideas.lego.com/projects/3ed27a21-f2ac-4f7d-ad85-dd2621b2123e#&gid=1&pid=7`, 2020. [Online; acessado em 21 de Outubro 2020]. 10

[9] Louis Hugues and Nicolas Bredeche. Simbad: an autonomous robot simulation package for education and research. In *International Conference on Simulation of Adaptive Behavior*, pages 831–842. Springer, 2006. 2, 5

[10] Toni Härkönen. Advantages and implementation of entity-component-systems, 2019. 8, 9, 10, 11

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

[12] JGraph. https://github.com/jgraph. 21

[13] JGraph. drawio. `https://app.diagrams.net`. Acessado 08 de Maio de 2020. 22

[14] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004. 2, 5

[15] Ontje Lünsdorf and Stefan Scherfke. Simpy. `https://gitlab.com/team-simpy/simpy/-/commit/b29e72af9975aae6cc0f2ffeaff0c5c876c9e644`, 04 2020. 15

[16] Paulo Henrique Maia, Lucas Vieira, Matheus Chagas, Yijun Yu, Andrea Zisman, and Bashar Nuseibeh. Dragonfly: a tool for simulating self-adaptive drone behaviours. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 107–113. IEEE, 2019. 2, 5

[17] Norm Matloff. 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. 14

[18] Benjamin Moran. Esper. `https://github.com/benmoran56/esper/commit/850ee6365dcdc6c36c7b02053bef121a98042849`, 05 2020. 13, 26

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

[20] Carlo Pinciroli, 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

[21] Eric Rohmer, Surya PN Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326. IEEE, 2013. 2, 5

[22] Vittorio Romeo. *Analysis of entity encoding techniques, design and implementation of a multithreaded compile-time Entity-Component-System C++14 library.* PhD thesis, Università degli Studi di Messina, 07 2016. 9, 11

[23] John Ferguson Smart. *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle.* Manning Publications, 1 edition, 2014. 15, 16

[24] Daniel Sykes. *Autonomous architectural assembly and adaptation.* PhD thesis, Citeseer, 2010. 1

[25] Dennis Wiebusch and Marc Erich Latoschik. Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems. In *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 25–32. IEEE, 2015. 11, 13