UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Vinicius Steffani Schweitzer

MODELO PARA UTILIZAÇÃO DE AGENTES INTELIGENTES EM AMBIENTES VIRTUAIS MULTIAGENTE

Florianópolis

2019

Vinicius Steffani Schweitzer

MODELO PARA UTILIZAÇÃO DE AGENTES INTELIGENTES EM AMBIENTES VIRTUAIS MULTIAGENTE

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Prof. Dr. Elder Rizzon Santos

Florianópolis

2019

Vinicius Steffani Schweitzer

MODELO PARA UTILIZAÇÃO DE AGENTES INTELIGENTES EM AMBIENTES VIRTUAIS MULTIAGENTE

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciências da Computação" e aprovado em sua forma final pelo Departamento de Informática e Estatística da Universidade Federal de Santa Catarina.

Florianópolis, 05 de junho 2019.

Prof. Dr. José Francisco Danilo De Guadalupe Correa Fletes
Coordenador do Curso

Banca Examinadora:

Prof. Dr. Elder Rizzon Santos
Orientador

Prof. Dr. Maicon Rafael Zatelli

Me. Thiago Ângelo Gelaim

Aos meus falecidos avós, Alderico e Célia.

AGRADECIMENTOS

Agradeço principalmente aos meus pais, por sempre me apoiarem e me incentivarem a ir atrás daquilo que eu gostava. Também os agradeço por me darem forças nos momentos dificeis e sempre torcer por mim.

Agradeço aos dois maiores amigos que fiz na faculdade, que me acompanham desde o primeiro dia de aula e que graduarão ao meu lado, Dúnia e Fabio. Foi um prazer caminhar na mesma estrada que vocês do começo ao fim. Que nossas estradas, agora separadas, continuem se cruzando ao longo da vida.

Aos meus amigos pessoais eu agradeço também, por terem a paciência de ainda me convidar para nossos encontros mesmo eu não conseguindo ir em nenhum que ocorresse no meio do semestre.

Aos amigos que fiz nas primeiras fases do curso, agradeço a todos pela companhia e momentos de descontração em meio ao estresse das aulas. Embora alguns tenham ficado para trás, torço para que todos vocês consigam ser bem sucedidos naquilo que gostam.

Também agradeço aos amigos que fiz ao fim do curso quando ingressei no LabSEC, aprendi muito com todos tanto no nível pessoal quanto no profissional.

Por fim agradeço aos professores que me trouxeram até aqui, que me ensinaram tanto e que serão lembrados com seus trejeitos característicos e humanos.

Despite everything, it's still you.

Chara

RESUMO

Sistemas inteligentes estão tornando-se cada vez mais relevantes, tendo em vista a necessidade de resolução de problemas cada vez mais complexos. Uma das formas de modelar este tipo de sistema é através do uso de agentes inteligentes, que definem estes agentes como entidades em um ambiente com desejos próprios e que atuam nele de forma autônoma para atingir seus objetivos. Pode ser necessário modelar o comportamento humano e uma das formar de modelá-lo como um agente inteligente é através da arquitetura *belief-desire-intention* que simula o processo de decisão humano através do que se acredita e do que se deseja obter.

Diversas ferramentas de agentes inteligentes oferecem formas de executar esta arquitetura BDI, entretanto a maioria delas simula estes agentes em ambientes próprios, atrelados à ferramenta. Muitas vezes estes ambientes são complexos ou restritivos demais. Existem programas externos às ferramentas de agentes que simulam ambientes com alta fidelidade, as *game engines*. Estes programas costumam ser utilizados para o desenvolvimento de jogos e portanto são mais conhecidos e fáceis de usar para um público geral.

O modelo proposto neste trabalho estabelece um padrão de implementação que permite a utilização de agentes inteligentes em ambientes simulados em *game engines*. Este modelo abstrato especifica o protocolo de comunicação entre as ferramentas, os conteúdos das mensagens e quais comportamentos as ferramentas devem apresentar diante destas mensagens. Neste trabalho é realizada a implementação do modelo e o teste da implementação, medindo os ciclos de raciocínio por segundo por parte de ferramenta de agentes e os *frames* por segundo por parte da *game engine*, para avaliar seu desempenho.

Palavras-chave: Inteligência Artificial. Agentes inteligentes. Sistemas multiagente. BDI. Motores de jogos.

ABSTRACT

Intelligent systems are becoming more relevant than ever, given the need to solve more complex problems. One way to model this type os systems is using intelligent agents, which define those agents as entities in an environment, with their own desires and that act on it in an autonomous way to achieve their objectives. It might be necessary to model a human-like behaviour and one way to do so with intelligent agents is by using the belief-desire-intention architecture that simulates the thought process of a human based on what is believed and on what it desires to achieve.

Many tools for intelligent agents offer ways to use the BDI architecture, but most of them simulate those agents in their own environment, bound to the agent tool. A lot of those environments are either too complex or too restrictive. There are other applications not related to the agent tool that simulate environments with high fidelity, the game engines. Those applications are ussually used in game development and as a result are more known and easier to use for the general public.

The model proposed in this thesis estabilishes a implementation standard that allows the usage of intelligent agents in environments simulated in game engines. This abstract model specifies the communication protocol between the tools, the content of the messages and what behaviour the tools need to show based on the messages. In this thesis this model is implemented and this implementation is tested, measuring the reasoning cycles per second in the agent tool and the frames per second in the game engine, to evaluate it's performance.

Keywords: Artificial Intelligence. Intelligent agents. Multi-agent systems. BDI. Game engines.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 INTRODUÇÃO

Este capítulo introduz o trabalho e o que ele abordará. Na seção 1.1 é apresentada a motivação para o desenvolvimento deste trabalho. Na seção 1.2 é delineado o objetivo geral deste trabalho e também uma enumeração dos objetivos específicos que são atingidos. Por fim, a seção 1.3 descreve o que é produzido para atingir os objetivos.

1.1 MOTIVAÇÃO

O estudo de agentes inteligentes dentro da área de Inteligência Artificial possibilita a simulação de raciocínio complexo através de conjunturas simples, permitindo a simulação de modelos complexos sem grande esforço de implementação.

Uma das formas de modelar o pensamento de um agente é a de *belief*, *desire*, *intention* que modela símbolos mentais em três principais categorias: crenças, desejos e intenções. A vantagem dessa abordagem é a facilidade de se descrever comportamentos mais complexos com poucas regras, sem a necessidade de explicitamente descrever grandes árvores de decisão, que descrevem os comportamentos a serem tomados verificando diversas condições explicitamente.

Agentes inteligentes por definição se situam em um ambiente. As linguagens de agente atualmente disponíveis permitem a criação de ambientes em aplicações externas, permitindo uma grande flexibilidade quanto a sua implementação. Estes ambientes podem ser classificados em várias categorias, das quais algumas podem ser encontradas em motores de jogos. Implementá-los neste tipo de ferramenta pode ser útil para se obter uma melhor visualização dos agentes no ambiente, mas também é útil pois elas oferecem grande flexibilização em relação às diversas categorias de ambiente.

Diante disso, este trabalho se motiva a desenvolver e implementar um modelo que integra linguagens de agentes inteligentes com um ambiente simulado em um motor de jogos.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Possibilitar que agentes inteligentes modelados em linguagens especializadas atuem em ambientes virtuais desenvolvidos em um motor de jogos.

1.2.2 Objetivos Específicos

- Fazer um levantamento do estado da arte nas áreas de agentes inteligentes, sistemas multiagente e utilização de agentes em ambientes virtuais 3D;
- Analisar os requisitos do modelo de integração entre a linguagem de agentes e o ambiente virtual escolhido;
- Construir um protótipo da solução proposta;
- Testar a solução através de um estudo de caso que a utilize;
- Avaliar e analisar os resultados coletados no estudo de caso.

1.3 MÉTODO DE PESQUISA

A pesquisa se dará em quatro principais etapas:

1. Levantamento do estado da arte, através de uma exploração dos artigos e trabalhos relevantes diante do conteúdo prático deste trabalho;
2. Desenvolvimento do modelo que integra sistemas de agentes inteligentes com ambientes virtuais. Esta etapa é dividida em duas fases:
    - Enumeração e discussão dos problemas envolvidos na elaboração deste modelo;
    - Síntese de um modelo final tendo em vista os pontos levantados na enumeração dos problemas.

    Duas funcionalidades principais estão presentes neste modelo:

- Suporte para múltiplos agentes atuando e interagindo com um mesmo ambiente virtual;
- Suporte a sensores e atuadores individuais, de forma que cada agente tenha suas próprias formas de interagir com o ambiente.

3. Implementação do modelo proposto utilizando ferramentas conhecidas tanto na área de agentes inteligentes quanto na de motores de jogos;

4. Avaliação da solução desenvolvida criando um estudo de caso descrito por:

   Um ambiente simples que utilize as características de agentes inteligentes, com o propósito de avaliar a viabilidade do uso dessa arquitetura na implementação de inteligência artificial em ambientes virtuais, tendo em vista que:

   - Por parte do ambiente virtual, essa avaliação leva em conta o desempenho do sistema em relação à quantidade de quadros por segundo (FPS) que o sistema consegue proporcionar;
   - No sistema de agentes o critério de avaliação será a quantidade de ciclos por segundo que o motor de raciocínio consegue realizar;
   - A principal variável nessa análise é a quantidade de agentes no ambiente, de forma a verificar quão significativa é a queda de desempenho quando a quantidade de agentes for muito elevada.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta as definições teóricas presentes na literatura e que embasam este trabalho. As seções 2.1 até 2.3 introduzem os principais conceitos utilizados neste trabalho. Enquanto que a seção 2.4 apresenta implementações de alguns conceitos dos tópicos anteriores e alguns conceitos associados.

2.1 AGENTES INTELIGENTES

Esta área da Inteligência Artificial estuda o conceito de agentes inteligentes que, de acordo com Wooldridge (2002), são sistemas computacionais que apresentam as seguintes capacidades:

- Reatividade:

  Capacidade de observar o ambiente à sua volta e tomar ações de acordo com o que foi observado, com o propósito de atingir seus objetivos;

- Proatividade:

  Capacidade de ter objetivos e de tomar iniciativa para atingi-los;

- Habilidade social:

  Capacidade de interagir com outros agentes para atingir seus objetivos.

Já Dorri, Kanhere e Jurdak (2018) apresentam a seguinte definição:

> Agente: Uma **entidade** que se localiza em um **ambiente** e observa diferentes **parâmetros** que são utilizados para tomar decisões baseadas no objetivo da entidade. A entidade realiza a **ação** necessária no ambiente baseada nesta decisão.[1]

Esta definição apresenta quatro conceitos chave:

- **Entidade**:

  Algo que representa o agente no ambiente, como um robô, um software ou um objeto em um simulador;

[1] Tradução livre da definição apresentada em Dorri, Kanhere e Jurdak (2018).

- **Ambiente**:

  Todo agente está localizado em algum ambiente. Mais detalhes na subseção 2.1.1;

- **Parâmetros**:

  São informações observadas pelo agente sobre o estado do ambiente. Estas observações são feitas por **sensores** do agente;

- **Ação**:

  Todo agente pode realizar ações para interagir e alterar o ambiente. Ele as realiza através de **atuadores**.

As duas definições de agente inteligente não precisam ser vistas como concorrentes e são consideradas complementares para a construção deste conceito neste trabalho.

2.1.1 Ambiente

Como citado anteriormente, agentes inteligentes se situam em um ambiente (WOOLDRIDGE, 2002; RUSSELL; NORVIG, 2010; DORRI; KANHERE; JURDAK, 2018) e podem ser classificados por diversos atributos (WOOLDRIDGE, 2002):

- Acessível × Inacessível[2]:

  Um ambiente acessível é um ambiente cujas informações a respeito do estado do ambiente podem ser coletadas sem nenhum tipo de imperfeição, caso contrário o ambiente é classificado como inacessível;

- Determinístico × Não determinístico:

  Um ambiente é determinístico quando uma ação tem um efeito esperado garantido, sem nenhuma incerteza sobre como uma determinada ação altera o ambiente. Já um ambiente não determinístico é aquele onde uma ação pode ter efeitos além do previsto;

- Estático × Dinâmico:

  Um ambiente é estático quando seu estado permanece inalterado até que o agente atue sobre ele. Já ambientes dinâmicos podem sofrer alterações sem a influência direta do agente;

[2]Esta definição também é encontrada na literatura com os nomes de "completamente observável" e "parcialmente observável".

- Discreto × Contínuo:

  Ambientes discretos são aqueles que possuem um número fixo e finito de estados. Caso contrário, são considerados ambientes contínuos.

Russell e Norvig (2010) apresenta mais três classificações para ambientes:

- Agente único × Multiagente:

  Quando existe apenas um agente operando no ambiente, este é considerado um ambiente de agente único. Quando existem mais de um o ambiente é considerado multiagente;

- Episódico × Sequencial:

  Esta classificação leva em consideração as ações que os agentes realizam ao longo do tempo. Caso eles repitam ciclos independentes de ações passadas, o ambiente é considerado episódico. Enquanto que quando as ações dos agentes dependem das ações passadas o ambiente é considerado sequencial;

- Conhecido × Desconhecido:

  Ambientes conhecidos são aqueles onde os agentes conhecem o resultado de suas ações no ambiente. Caso contrário o ambiente é considerado desconhecido.

2.1.2 Arquitetura BDI

A arquitetura *belief-desire-intention* (BDI) é uma das formas de se implementar agentes inteligentes. De acordo com Bordini, Hübner e Wooldridge (2007) esta arquitetura se baseia em um modelo de raciocínio humano e modela três categorias de estados mentais:

- Crenças:

  Representam as informações que um agente tem sobre o ambiente. Estas informações podem estar incompletas ou até incorretas;

- Desejos:

  Representam os estados do ambiente que o agente gostaria que fossem atingidos efetivamente no ambiente;

- Intenções:

  Representam desejos que o agente se comprometeu a realizar, uma vez que nem todos os desejos podem ser realizados simultaneamente e que inclusive podem ser conflitantes.

O modelo utilizado para tomada de decisão diante destes três contextos mentais é chamado de **raciocínio prático**, que ainda segundo Bordini, Hübner e Wooldridge (2007), é o raciocínio direcionado para ações. Este raciocínio é construído através de duas atividades, a deliberação e o planejamento:

- Deliberação:

  O processo de deliberação consiste em escolher quais desejos se tornarão intenções.

- Planejamento:

  Este processo consiste em analisar as intenções do agente, suas crenças e suas ações possíveis, para que ao final se obtenha um **plano**. Um plano consiste em uma série de ações que ao serem executadas resultam na concretização de determinadas intenções.

Este modelo de raciocínio prático pode ser generalizado para outras arquiteturas, permitindo que estas também se baseiem nas atividades de deliberação e planejamento.

2.2 SISTEMAS MULTIAGENTE

Como citado anteriormente, sistemas multiagente são aqueles onde mais de um agente se encontra em um mesmo ambiente (RUSSELL; NORVIG, 2010). De acordo com Dorri, Kanhere e Jurdak (2018) existem oito categorias para se observar sistemas multiagente, cada uma com suas classificações:

- Liderança (*Sem líder* × *Leader-follow*):

  Os SMAs são considerados sem líder quando os agentes atuam de forma autônoma tomando suas próprias decisões. Enquanto que no segundo caso existe um agente, ou grupo de agentes, que lidera os demais, enviando instruções para os seus agentes subordinados executarem;

- Função de decisão (*Linear* × *Não linear*):

Em um sistema linear as funções de decisão dos agentes são lineares aos dados de entrada coletados do ambiente, enquanto que em sistemas não lineares existem funções não lineares às entradas;

- Heterogeneidade (*Homogêneos* × *Heterogêneos*):

  Sistemas homogêneos são aqueles em que todos os agentes presentes no ambiente apresentam as mesmas características e funcionalidades, enquanto que em um sistema heterogêneo os agentes podem ter características e implementações diferentes;

- Parâmetros de acordo (*Primeira ordem* × *Segunda ordem* × *Alta ordem*):

  Em algumas aplicações pode ser necessário que todos os agentes entrem em acordo sobre o valor de um parâmetro. Quando há apenas um parâmetro este sistema é conhecido como um sistema de primeira ordem, quando há dois parâmetros ele é considerado um sistema de segunda ordem e quando há mais parâmetros o sistema é considerado de alta ordem. Devido a dificuldade de entrar em acordo quanto a muitos parâmetros, em sistemas de alta ordem o acordo dos múltiplos parâmetros é atingido quando um ou dois desses parâmetros estão acordados entre os agentes e os demais parâmetros podem ser obtidos através de derivações destes um ou dois parâmetros já estabelecidos;

- Consideração de atraso (*Com atraso* × *Sem atraso*):

  Agentes podem sofrer atrasos quando estiverem executando suas ações. Sistemas com atraso são chamados assim pois consideram os atrasos quando realizam seu processamento. Enquanto que sistemas sem atraso simplesmente ignoram os eventuais atrasos que possam ocorrer;

- Topologia (*Estáticos* × *Dinâmicos*):

  SMAs com topologia estática são aqueles em que a posição relativa entre os agentes não se altera ao longo do tempo, enquanto que nos sistemas de topologia dinâmica estas posições podem se alterar;

- Frequência de transmissão de dados (*Time-triggered* × *Event-triggered*):

  Quando a transmissão de dados é *time-triggered*, as coletas e compartilhamento de dados dos sensores ocorrem em períodos conhecidos e predefinidos. Já em sistemas *event-triggered* estas coletas

e compartilhamento ocorrem apenas quando eventos específicos ocorrem;

- Mobilidade (*Estáticos* × *Móveis*):

  A mobilidade de um SMA é considerada estática quando os agentes presentes nela não mudam de posição, sendo considerados móveis caso contrário. Esta definição pode ser avaliada tanto do ponto de vista da posição espacial do agente no ambiente quanto de sua posição de atuação, como por exemplo em um sistema de segurança onde o agente pode estar atuando cada vez em um servidor diferente para detectar ataques.

Por mais que estas categorias sejam apresentadas por Dorri, Kanhere e Jurdak (2018) no contexto de sistemas multiagente, as categorias de função de decisão, consideração de atraso, frequência de transmissão de dados e mobilidade também podem ser consideradas em sistemas de agente único com a mesma semântica.

2.3 SIMULAÇÕES COMPUTACIONAIS

Simulações têm o propósito de replicar e executar modelos em um ambiente controlado, para se extrair mais informações do mesmo (KELTON; SADOWSKI; SADOWSKI, 1997). O método de simulações computacionais avalia a realização de simulações em computadores. A vantagem de usar simulações no lugar de modelos matemáticos é a sua capacidade de representar sistemas complexos.

Kelton, Sadowski e Sadowski (1997) apresentam algumas das formas de classificar simulações:

- *Estática* × *Dinâmica*:

  Simulações estáticas não consideram a passagem do tempo, enquanto que as dinâmicas consideram. Um exemplo de simulação estática pode ser a obtenção da distribuição de probabilidade ao se jogar dados, gerando valores aleatórios, esta simulação não leva em consideração a passagem do tempo. Um exemplo para simulações dinâmicas pode ser a simulação de um cruzamento, onde carros chegam e saem ao longo do tempo.

- *Discreta* × *Contínua*:

  Simulações discretas são aquelas que as mudanças ocorrem em pontos específicos ao longo do tempo, enquanto que as contínuas

se alteram com a passagem do tempo. Um jogo de xadrez é um exemplo de simulação discreta, enquanto que a simulação do fluxo de um rio é contínuo.

- *Determinística* × *Estocástica*:

  Simulações determinísticas são aqueles que não possuem entradas aleatórias, enquanto que as estocásticas apresentam este tipo de valor. Por consequência, as simulações determinísticas apresentarão sempre os mesmos resultados, enquanto que as estocásticas apresentam resultados distintos. Entretanto, por mais que os resultados sejam distintos, ainda existem técnicas para que estas simulações sejam reprodutíveis.

É possível traçar um paralelo entre ambientes de agentes (seção 2.1.1) e simulações, este paralelo é onde se encontra o domínio deste trabalho.

2.4 FERRAMENTAS

Com os conceitos vistos é possível então apresentar ferramentas que implementam algumas das teorias citadas, de forma à auxiliar no objetivo deste trabalho.

2.4.1 Linguagens de agentes

A implementação de linguagens de agentes ainda está em seus primeiros passos e por isso muitas novas linguagens vêm surgindo como pode ser observado no trabalho de Kravari e Bassiliades (2015), que realiza um levantamento e comparação entre vinte e quatro linguagens de agente, das quais foi escolhida a linguagem JASON para ser utilizada.

2.4.1.1 JASON

As principais características da ferramenta JASON que resultaram em sua escolha, enumeradas no trabalho de Kravari e Bassiliades (2015), são a simplicidade, facilidade de aprendizado, bom desempenho, suporte a sistemas multiagente, documentação extensa por Bordini, Hübner e Wooldridge (2007), e principalmente seu foco em aplicações de propósito geral. A ferramenta apresenta outro ponto positivo, que

é a modularidade de componentes, de forma que é possível criar não apenas os agentes como também os componentes que realizam os processos da ferramenta, como por exemplo o escalonador de ações ou as classes que administram os ciclos de raciocínio dos múltiplos agentes sendo executados.

2.4.2 Motores de jogos

Diversas ferramentas existem para modelar e implementar simulações. Dentre elas estão os motores de jogos (*game engines*), ferramentas utilizadas para o desenvolvimento de jogos de computador que, segundo Gregory (2014), também podem ser consideradas como simulações de tempo-real, dinâmicas e interativas. Estes sistemas executam através de um *game loop*, que é um ciclo que se repete durante toda a execução da simulação e que executa algumas operações a cada rodada do ciclo, estas operações consistem em: obter as teclas pressionadas pelo jogador, atualizar o estado da simulação e então desenhar a imagem atual do mundo na tela do computador. Uma visualização do fluxo de execução do *game loop* encontra-se a seguir:

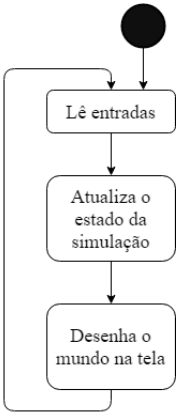

Figura 1 – Fluxo de execução do *game loop*.

2.4.2.1 Unity

Um motor de jogos muito utilizado atualmente é o **Unity** (UNITY TECHNOLOGIES, 2019b), uma ferramenta gratuita, com suporte a criação de jogos tanto 2D quanto 3D, além de possuir suporte a múltiplas plataformas. Esta é a ferramenta que será utilizada para atuar como ambiente virtual.

A ferramenta apresenta alguns conceitos que são utilizados neste trabalho: corotinas e reflexão.

2.4.2.1.1 Corotina

Como descrito na subseção 2.4.2 os motores de jogos funcionam através de um *game loop*. A maior parte da execução de código acontece na segunda etapa do *loop*, quando o estado da simulação deve ser atualizado. Como o *loop* ocorre múltiplas vezes por segundo certos eventos devem ocorrer gradativamente ao longo do tempo, por exemplo: o movimento de um personagem não pode ocorrer em uma única chamada de método, ele deve ocorrer através de múltiplas chamadas onde o movimento ocorre em pequenos incrementos. Uma maneira fornecida pelo Unity de facilitar a implementação de um método deste tipo são as **corotinas** (UNITY TECHNOLOGIES, 2019a) que, em termos simples, são métodos que podem ser interrompidos durante sua execução para que continuem apenas na próxima iteração do *game loop*. Um exemplo de corotina encontra-se a seguir:

```
1 IEnumerator CaminharAte(Coordenada destino) {
2     enquanto (Distancia(origem, destino) > 0) {
3         MoverAte(destino, 0.1);
4         yield return null;
5     }
6 }
```

Neste exemplo podemos ver um possível código para o caso onde é preciso mover um personagem até uma coordenada destino, nele ocorre uma repetição que executará até que a distância entre o personagem e o destino seja zero. Se este método não fosse uma corotina este ciclo de repetição aconteceria em apenas uma iteração do *game loop*, ou seja, o personagem seria desenhado na tela em um *frame* e logo no *frame* seguinte já teria se deslocado até o destino. Entretanto como o método é uma corotina (caracterizado pelo seu retorno "IEnumerator" e o comando "yield return") toda vez que o comando "yield return null" for executado o método será interrompido e continuará apenas na

próxima iteração do *game loop*, em outras palavras, a chamada de "MoverAte(destino, 0.1)" ocorrerá apenas uma vez a cada *frame* resultando em um movimento suave, tendo em vista que ele está se deslocando apenas "0.1" de distância a cada iteração.

2.4.2.1.2 Reflexão

A linguagem de programação utilizada pelo Unity é o *C#* (MICROSOFT CORPORATION, 2018), que por sua vez oferece suas próprias ferramentas, das quais encontra-se o mecanismo de **reflexão** (MICROSOFT CORPORATION, 2015). De forma simplificada este mecanismo permite referenciar atributos, métodos, tipos e classes como objetos da linguagem dinamicamente. Um de seus usos por exemplo é a capacidade de acessar métodos de uma classe dinamicamente e invocá-lo sem saber seu nome em tempo de compilação. Exestem diversos outros usos deste mecanismo, mas que encontram-se fora do escopo deste trabalho.

2.5 CONCLUSÃO

Com estes conceitos em mente é possível elaborar o modelo final. A principal utilidade dos conceitos de agentes inteligentes e de ambientes virtuais é melhor conhecer seus comportamentos e responsabilidades bem como sua abrangência quanto as formas de implementação de forma a construir um modelo que suporte múltiplos dos conceitos. Dessa forma, por exemplo, o modelo é capaz de suportar tanto um único agente quanto múltiplos agentes. Além disso, alguns dos conceitos também se aplicam na escolha das ferramentas, buscando aquelas que fornecessem uma maior flexibilidade diante da abrangência dos conceitos.

Os conceitos relacionados aos tipos de ferramentas é importante por apresentar seu papel no modelo definido, enquanto que os conceitos das ferramentas em si servem para facilitar o entendimento de suas particularidades ao longo do texto.

3 TRABALHOS CORRELATOS

Neste capítulo são apresentadas quatro publicações que se relacionam com este trabalho. A seção 3.1 apresenta uma forma de implementar agentes inteligentes em um ambiente virtual 3D. A seção 3.2 apresenta um trabalho que realiza uma forma de integração da linguagem JASON com um ambiente virtual em um motor de jogos. A seção 3.3 apresenta uma abordagem de priorização de percepções para evitar redundância. Por fim, a seção 3.4 introduz um artigo que desenvolve uma linguagem para desenvolvimento de agentes, com foco no seu uso em motores de jogos.

3.1 SPARK — A GENERIC SIMULATOR FOR PHYSICAL MULTI-AGENT SIMULATIONS

O trabalho de Obst e Rollmann (2004) introduz o simulador Spark, um simulador multiagente 3D utilizado na competição RoboCup, uma competição de robótica que ocorre desde 1996, promovendo avanços em robótica e também em Inteligência Artificial.

Os autores citam como sua motivação um dos marcos desejados da RoboCup, um cenário onde robôs completamente autônomos ganhariam do time vencedor da copa do mundo em uma partida de futebol. Para atingir esse objetivo era necessário evoluir a antiga simulação 2D da competição para um sistema 3D. Eles também deixam claro que este simulador não simula apenas partidas de futebol, sendo um sistema universal de simulações.

O Spark se baseia em uma outra implementação chamada de Zeitgeist (KÖGLER; OBST, 2004) que, de forma resumida, permite a criação de sistemas altamente modulares, tornando simples a adição de novos componentes sem que seja necessário recompilar o código original. O simulador Spark se divide em diversos desses componentes, dentre os quais estão: administrador de agentes, administrador de jogo, simulador, administrador de rede, simulador de física e administrador de cenário. Por conta da infraestrutura modular do Zeitgeist, todos esses componentes podem ser substituídos por outros conforme necessário. A funcionalidade de cada um desses componentes pode ser descrita brevemente como:

- Administrador de agentes:

Consiste em administrar e manter as representações dos agentes na simulação, estas representações são chamadas de agentes delegado (*agent proxy*), uma vez que elas estão no ambiente como representações do agente de verdade, que roda seu processamento em uma aplicação separada. Cada representação dessas possui uma entidade no ambiente físico da simulação.

- Administrador de jogo:

  Esse componente é encarregado de armazenar e administrar informações abstratas da simulação, como manter a pontuação dos dois times em uma partida de futebol ou impor as regras do jogo quando uma bola sai para fora do campo por exemplo.

- Simulador:

  Esse componente realiza a execução dos eventos do ambiente, como por exemplo as ações enviadas pelos agentes ou acontecimentos enviados pelo simulador de física. O Spark possui duas implementações deste componente. A primeira é um simulador simples que realiza as ações e percepções dos agentes assim que as recebe. A segunda opção é um sistema mais robusto chamado de SPADES (RILEY; RILEY, 2003), um *middleware* que fornece diversas funcionalidades, das quais podem ser citadas: a paralelização do processamento dos agentes sem que fatores como causalidade sejam perdidos e a possibilidade de compensação de latência na comunicação entre os agentes e o simulador.

- Administrador de rede:

  Fica encarregado de gerir a parte de comunicação pela rede, fornecendo mecanismos de troca de mensagens tanto com agentes executando em outras aplicações quanto com aplicações de monitoramento. O núcleo de *parsing* dessas mensagens também é modular, permitindo que o usuário utilize seu formato de preferência na comunicação, como JSON ou XML por exemplo.

- Simulador de física:

  O motor de física utilizado no simulador é o ODE (SMITH, 2003), uma biblioteca de código aberto escrita em C que provê funcionalidades como: objetos com massa, forças, torque, detecção de colisão e articulações.

- Administrador de cenário:

Esse componente administra o grafo de cenário que contém todas as entidades do ambiente, essas entidades podem ser qualquer objeto presente no ambiente, não apenas os agentes. Estes ambientes são construídos através de uma linguagem de descrição de cenário chamada de RubySceneGraph.

Utilizando os componentes descritos acima, pode-se observar que a implementação de agentes que atuam no Spark não faz parte do *loop* de processamento do simulador, ou seja, processar a lógica dos agentes não é responsabilidade do simulador, permitindo que os agentes possam ser implementados em qualquer linguagem ou ferramenta executando em paralelo, com a restrição de que devem se comunicar com a aplicação através do protocolo de troca de mensagens especificado.

Os autores também dedicam uma seção para citar trabalhos relacionados, demonstrando diferenças entre o Spark e esses modelos. A lista que segue é interessante por demonstrar um leque das possíveis formas de se modelar sistemas especializados em simular agentes. Segue a lista dos modelos e de suas características:

- XRaptor (BRUNS; POLANI; UTHMANN, 2001):

  Utilizado para analisar o comportamento de um grande número de agentes, tanto em 2D quanto em 3D. Entretanto, apresenta limitações quanto ao formato dos agentes no mundo, além de possuir um alto acoplamento entre o agente e o simulador, dificultando o processo de abstração do desenvolvedor do agente.

- Webots (CYBERBOTICS LTD., 2004):

  O foco deste *framework* é a modelagem precisa de sistemas de robótica existentes, resultando em um nível de abstração menor, tendo em vista que a plataforma que se deseja simular possui este mesmo nível de abstração. Por exemplo, esta ferramenta considera filtrar, classificar e interpretar dados coletados de sensores como parte do raciocínio do agente, da mesma forma que isto faz parte do raciocínio de um robô.

- Übersim (BROWNING; TRYZELAAR, 2003):

  Assim como o Spark, este simulador foi desenvolvido com a RoboCup em mente, entretanto, ele foi desenvolvido para simular uma categoria específica do campeonato, resultando em um nível de abstração específico ao desta categoria.

- M-Rose (BUCK; BEETZ; SCHMITT, 2002):

Este é um simulador 2D para simulação de robôs reais que se baseia em uma estrutura em três passos. Primeiro é utilizada uma rede neural para aprender sobre os movimentos do agente. Em seguida o que foi aprendido é utilizado para ensinar o agente à realizar as tarefas que deve fazer. Por fim o agente resultante é transferido para a controladora de um robô real para validação.

3.2 UMA ARQUITETURA PARA A ATUAÇÃO DE AGENTES INTELIGENTES

Prandi (2017) desenvolve uma solução com objetivos similares aos que são apresentados neste documento, servindo como boa referência para nortear o desenvolvimento do que é proposto por este trabalho quando levado em consideração aspectos como a abordagem escolhida por ele e desafios encontrados.

O autor realiza um levantamento teórico dos conceitos utilizados, realizando em seguida comparações entre diversas linguagens e teorias de agente, demonstrando motivações para a escolha de uma determinada abordagem perante as outras para utilizar em seu trabalho. Ao final Prandi (2017) conclui que a arquitetura BDI é a mais adequada para atingir seu objetivo por apresentar maior flexibilidade, maior facilidade de representação de comportamentos complexos e por apresentar maior embasamento teórico pois é utilizada em outros trabalhos com objetivos similares. Tendo em vista a escolha da arquitetura BDI, Prandi (2017) opta por utilizar a linguagem JASON, uma vez que esta apresenta uma maior facilidade de desenvolvimento, além de também apresentar uma maior abstração da arquitetura, permitindo que o desenvolvedor foque apenas no desenvolvimento dos agentes e, de forma similar à arquitetura BDI, é uma linguagem utilizada em trabalhos similares, resultando em um melhor embasamento teórico.

A seção principal do trabalho inicia com uma análise mais aprofundada da arquitetura JASON resultando em um dos principais pontos levantados: esta arquitetura utiliza ambientes de percepção passiva, ou seja, seus ambientes é que enviam as percepções para os agentes, o que por um ponto de vista de engenharia de software pode não ser interessante pois cria um alto acoplamento entre o ambiente e a arquitetura, além de que este tipo de funcionamento não é adequado para sistemas no mundo real, uma vez que não há como o mundo real "notificar" o agente de suas percepções. O autor então levanta outros problemas dessa utilização e em seguida propõe e implementa modificações que re-

sultam na alteração deste comportamento, ou seja, tornam o ambiente um ambiente passivo.

Após adaptar a arquitetura do JASON, Prandi (2017) realiza um teste desta modificação implementando um ambiente no motor gráfico Unity. Para realizar este teste, o autor também precisou estabelecer uma forma de transmitir as ações e percepções entre as aplicações do JASON e do Unity, optando por realizar esta tarefa através de uma comunicação *socket*, uma forma simples de troca de mensagens entre aplicações. O protocolo estabelecido para essa comunicação é bastante simples. Na parte dos atuadores o protocolo se resume à atribuir um identificador (diferente de 0) para cada ação que o agente pode realizar, feito isso basta que as mensagens enviadas do agente para o ambiente contenham apenas esse identificador, permitindo que o ambiente o receba e o traduza para a ação que deve ser realizada. A questão dos sensores parte da premissa de que existe apenas um sensor no ambiente, de forma que quando a mensagem é "0", significa que o agente está tentando observar o valor deste sensor único. O protocolo como um todo carece melhoria, mas serve como base e inspiração para o desenvolvimento deste trabalho.

3.3 UM MODELO DE RACIOCÍNIO SOBRE SENSORES PARA AGENTES SIGON EM AMBIENTES DE REALIDADE VIRTUAL

Freitas (2018) trata de realizar análises sobre estratégias de priorização de sensores de agentes inteligentes, uma vez que um agente pode perceber muitos dados em um intervalo de tempo e apenas alguns deles são de fato importantes.

O autor inicialmente apresenta as definições de agente utilizadas em seu trabalho, sendo que ele cita também uma categorização das aplicações de agentes inteligentes:

- Agentes de Informação:

  São agentes cujo objetivo é encontrar informações a respeito de seus usuários.

- Agentes de Cooperação:

  Estes agentes procuram solucionar problemas complexos através de comunicação e cooperação com outros agentes, humanos ou outros recursos.

- Agentes de Transação:

> Este tipo de agente foca no processamento e monitoramento de transações, levando em considerações aspectos como segurança e robustez.

Em seguida Freitas (2018) define o conceito de BDI utilizado em seu trabalho, uma vez que a ferramenta utilizada por ele se baseia nessa arquitetura.

A definição de percepção ativa então é delineada, realizando um comparativo com a percepção humana que pode é considerada ativa e também destacando a principal vantagem deste tipo de percepção que é a eliminação de percepções inúteis para um determinado agente. Ele também lista os cinco principais atributos de um agente de percepção ativa, descritos por Bajcsy, Aloimonos e Tsotsos (2018), sendo eles: *porque*, *o quê*, *como*, *quando* e *onde*, destacando o atributo *porquê* que é o principal fator para ajustar os demais atributos e obter suas percepções com mais precisão de acordo com suas necessidades.

O autor define também ambientes de agentes e realidade virtual, realizando a ligação entre os dois conceitos e destacando que, devido a alta complexidade dos ambientes de realidade virtual, é necessário um modelo de percepção ativa que filtre as percepções adequadamente.

A parte de agentes inteligentes do desenvolvimento deste trabalho é realizado na linguagem Sigon (GELAIM et al., 2019) que modela agentes inteligentes como *sistemas multi-contexto*. Este tipo de linguagem tem como principal vantagem sua modularização, onde cada módulo é um contexto diferente que são interligados por *regras de ponte*. Este trabalho é construído em torno do contexto de comunicação, uma vez que este é responsável por interagir com o ambiente. Neste contexto é realizada a publicação dos literais de percepção do agente e é nele que o domínio deste trabalho se encontra, propondo uma maneira de filtrar essas percepções recebidas do ambiente.

Para desenvolver seu trabalho Freitas (2018) estabelece um ambiente virtual desenvolvido no Unity, onde o agente Sigon atua como um servidor que os diversos sensores do agente se conectam como clientes. Neste ambiente os sensores estão constantemente enviando dados para o agente, dados estes muitas vezes inalterados e portanto redundantes, caracterizando exatamente o problema que este trabalho visa melhorar.

O modelo proposto por Freitas (2018) é o de priorização de sensores, onde os sensores inicialmente apresentam uma prioridade 5 e variam com o tempo em um intervalo de 1 à 10 a principio, uma vez que cada sensor pode ter seu próprio intervalo de prioridade. Esta funcionalidade é útil pois os sensores da visão próxima do agente são

mais importantes que os da visão distante, por exemplo. A partir deste modelo de prioridades foram estabelecidas duas estratégias para sua variação ao longo do tempo:

- Esta estratégia se baseia no armazenamento de um histórico de percepções de cada sensor. Onde cada nova percepção é avaliada diante do histórico, resultando na diminuição da prioridade do sensor caso a percepção seja redundante e no incremento da prioridade caso a percepção seja nova;

- A segunda estratégia se baseia na quantidade de percepções recebidas, que se resume em um aumento da prioridade quanto maior for o número de percepções com base em alguns parâmetros.

Por fim o autor realiza testes sobre estas estratégias, tendo como principal resultado uma diminuição do número de percepções recebidas pelo agente, resultado esperado da solução do problema. Um ponto relevante dos testes é a conclusão de que certas estratégias são mais adequadas para certos tipos de sensor, por exemplo o sensor de tato que se adéqua melhor à segunda estratégia pois ele não está constantemente sentindo o ambiente, porém ao perceber algo sua prioridade é elevada drasticamente.

3.4 GAME ENGINES AND MAS: BDI & ARTIFACTS IN UNITY

Em seu trabalho Poli (2018) desenvolve uma linguagem própria para a definição de agentes em um sistema multiagente. Esta linguagem tem como foco a *game engine* Unity, sendo assim uma linguagem com certo acopladamento à esse ambiente.

O autor inicia levantando outros trabalhos relevantes, dos quais podemos citar a linguagem de sistemas multiagente JASON, que serviu como base para a linguagem desenvolvida por Poli (2018) e também o trabalho de Omicini, Ricci e Viroli (2008) que introduz o modelo de artefatos, que ajuda a descrever as entidades interativas do ambiente e como os agentes podem interagir com elas.

Uma das motivações para o desenvolvimento dessa linguagem foi para suprir uma possível deficiência no desenvolvimento de jogos, pois atualmente o uso do modelo de agentes inteligentes nesse contexto é praticamente inexistente, e uma possível justificativa dada pelo autor é a inexistência de um modelo eficiente e intuitivo para criação deste tipo de inteligência artificial em jogos.

Para desenvolver a linguagem foi necessário escolher um *framework* para utilizar o paradigma lógico. O autor conclui que este *framework* não podia ser externo ao Unity uma vez que isso iria requerer comunicação entre aplicações o que poderia causar latência, algo que pode ser considerado inaceitável no desenvolvimento de jogos. Ao fim, o autor escolhe utilizar o interpretador UnityProlog (HORSWILL, 2017), desenvolvido especialmente para o Unity e com funcionalidades extras para utilizar abstrações da própria *game engine*.

Em seguida a arquitetura de agentes que será usada é definida, nela um agente é composto por: um *reasoner*, um conjunto de crenças, desejos, planos e intenções, além da capacidade de sentir e de agir. Um dos pontos dessa arquitetura é que ela é independente de plataforma, de forma que sua implementação requer poucas alterações para que seja utilizada em outros ambientes que não o Unity.

O autor então detalha sua implementação em três componentes principais: a declaração de agentes, a declaração de artefatos e a comunicação entre agentes. Parte dos textos que descrevem estes componentes tratam de descrever a sintaxe da linguagem, esta parte será abstraída neste sumário do trabalho.

Quanto à declaração dos agentes, o autor especifica que a execução de planos seja independente do *loop* de execução da *game engine*, realizando isso utilizando corotinas na parte do Unity, permitindo que planos possam ser interrompidos quando uma ação é executada, sendo retomados quando a ação tiver sido concluída. Entretanto, programar o agente na linguagem é apenas parte de seu desenvolvimento, tendo em vista que as ações que o agente executa na linguagem devem ser traduzidas para ações no contexto do ambiente em Unity. Para realizar essa integração é relativamente simples, basta que a entidade que representa o agente possua em seu código métodos com os mesmos nomes que as ações que o agente irá realizar, dessa forma as ações do agente são traduzidas diretamente para métodos em seu código que serão executados.

Outros componentes importantes são os artefatos, que são declarados de forma similar aos agentes com planos e conhecimento, com a diferença de que esses planos não podem ser ativados de forma autônoma e precisam que um agente os ative ou os utilize, essa distinção entre ativação e utilização é importante pois no primeiro caso quem executa as ações é o artefato e no segundo quem executa os planos do artefato é o agente. Artefatos também podem conter pré-condições que são verificadas pelo agente antes de serem executados. De forma análoga aos agentes, é preciso que no código das entidades que representam

esses artefatos no ambiente, existam métodos com os mesmos nomes das possíveis ações resultantes do uso do artefato.

A camada de comunicação é a mais importante quando se trata de um sistema multiagente pois ela permite a interação entre os agentes, além disso essa camada também é responsável pela comunicação entre um agente e um artefato. A comunicação agente-artefato é feita através de ações simples que permitem que o agente verifique o estado de um artefato, ative-o ou utilize-o. Quanto à comunicação agente-agente, a linguagem oferece diversos predicados para manipulação do conjunto de crenças e desejos de outros agentes, além da possibilidade de aprender os planos de outro agente.

Ao finalizar o trabalho, Poli (2018) demonstra um caso de testes contendo três agentes: um agente que recicla apenas plástico, um agente que é capaz de aprender como reciclar plástico e vidro, e um agente que recicla papel mas que apenas o faz quando instruído por outros. Nesse cenário também estão presentes artefatos representando papel, plástico e vidro, sendo que o último possui um plano que explica como reciclá-lo, permitindo que algum agente aprenda-o. O autor também implementa o mesmo cenário sem utilizar sua linguagem, mas utilizando máquinas de estados finitos, uma prática comum atualmente no Unity. Ao comparar as duas abordagens o autor demonstra como a utilização de sua linguagem torna o código mais limpo e fácil de programar e entender.

3.5 CONCLUSÃO

É possível traçar uma comparação entre os trabalhos apresentados levando em consideração as ferramentas utilizadas na integração de agentes inteligentes com o ambiente virtual:

Trabalho Correlato	Ferramenta de Agentes	Ambiente Virtual	Forma de Integração
Spark — A Generic Simulator for Physical Multi-agent Simulations	Independente de ferramenta de agentes	Spark	Troca de mensagens
Uma arquitetura para a atuação de agentes inteligentes	JASON	Unity	Troca de mensagens
Um modelo de raciocínio sobre sensores para agentes Sigon em ambientes de realidade virtual	Sigon	Unity	Troca de mensagens
Game Engines and MAS: BDI & Artifacts in Unity	Não nomeada	Unity	Implementada no ambiente virtual

Tabela 1 – Comparação entre os trabalhos correlatos

O trabalho de Obst e Rollmann (2004) descreve uma ferramenta de ambiente virtual para utilização de agentes inteligentes. Ele se relaciona com este trabalho ao apresentar uma forma de utilizar agentes inteligentes independente da ferramenta de agentes usado, utilizando comunicação entre as ferramentas como mecanismo para atingir tal resultado. No entanto seu trabalho desenvolve um ambiente virtual que utiliza este mecanismo, resultando em uma abordagem dependente desta ferramenta.

O foco do trabalho de Prandi (2017) não está na integração entre os dois tipos de ferramentas, porém para testar seu trabalho ele implementa um protocolo simples para integrar um sistema de agentes com um ambiente virtual. Este protocolo é relativamente simples e não muito flexível, servindo como inspiração para desenvolver um modelo

que permitisse uma integração mais adequada entre as duas ferramentas.

Freitas (2018) também não tem como foco a integração das duas ferramentas, no entanto para realizar seus testes ele precisa realizar o fluxo de envio de percepções do ambiente para o agente. Esse trabalho se relaciona com este por mostrar possíveis formas de realizar tal fluxo. A forma implementada por Freitas (2018) difere do que é definido no modelo proposto neste trabalho e esta diferença é discutida na subseção 4.1.4.2.

Em seu trabalho Poli (2018) aborda o problema da utilização de agentes inteligentes em *game engines* com o objetivo de averiguar a possibilidade de seu uso nesse contexto. Sua solução é a elaboração de uma linguagem própria para especificação de agentes, onde esta linguagem é implementada no próprio ambiente virtual. Esta solução é interessante principalmente quando o foco é o desempenho, tendo como desvantagem o acoplamento da linguagem ao ambiente virtual que ela é implementada.

4 DESENVOLVIMENTO

Neste capítulo encontra-se o conteúdo prático deste trabalho. A seção 4.1 apresenta o modelo proposto, primeiro apresentando as questões que devem ser respondidas no modelo e em seguida descrevendo a especificação final do modelo. A seção 4.2 descreve uma possível implementação do modelo proposto utilizando a linguagem de agentes JASON e o ambiente virtual Unity. A seção 4.3 demonstra testes realizados sobre o que é implementado na seção anterior.

4.1 MODELAGEM

De forma resumida, o problema proposto por este trabalho consiste em elaborar um modelo que pode ser implementado em sistemas de agentes inteligentes e em ambientes virtuais de forma que qualquer agente inteligente que utilize este modelo possa atuar em qualquer ambiente que também utilize-o. A imagem a seguir ilustra o modelo como um todo:

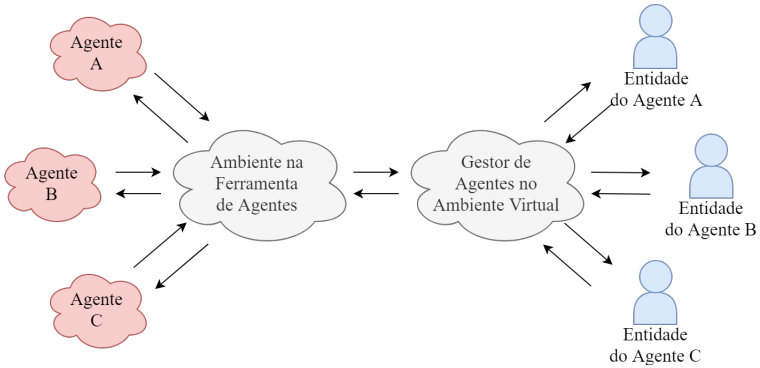

Figura 2 – Visão geral do sistema.

Nesta imagem é possível observar como existem múltiplos agentes atuando em um ambiente da ferramenta de agentes que implementa o modelo. Este ambiente por sua vez se comunica com o ambiente virtual, que também implementa o modelo. Como existem múltiplos agentes atuando na ferramenta de agentes, múltiplas entidades que re-

presentam os agentes existem no ambiente virtual. Todos os fluxos são de duas vias, pois o agente informa as ações que deseja realizar e o ambiente retorna informações resultantes da realização desta ação.

Com esta visão geral em mente, é preciso formalizar o modelo proposto. As subseções a seguir apresentam diversos problemas relacionados ao modelo, fazendo uma pequena discussão sobre as formas de abordá-lo e então tomando a decisão de qual abordagem é utilizada no modelo.

4.1.1 Responsabilidades do sistema de agentes

Analisando o sistema de agentes em relação ao ambiente, percebe-se que ele deve ser capaz de: obter percepções sobre o ambiente e de atuar sobre ele.

4.1.1.1 Percepções

Para que o agente adquira percepções sobre o ambiente deve ser possível que o ambiente lhe envie estas informações. Duas abordagens são possíveis para este problema:

- O ambiente envia percepções a cada intervalo de tempo;
- O agente por decisão própria, em seu ciclo de raciocínio, consulta o ambiente por um determinado sensor.

A primeira abordagem é conhecida como **percepção passiva**, enquanto que a outra é a **percepção ativa**.

Na percepção passiva o agente não precisa utilizar seus sensores conscientemente, apenas reagir às percepções que recebe, facilitando sua implementação. Entretanto um problema encontrado neste tipo de percepção é a sobrecarga de percepções causada pela redundância de dados, uma vez que um agente pode estar recebendo diversas vezes a mesma percepção sem que tenha ocorrido alterações no ambiente.

Já na percepção ativa há mais controle no que é percebido no ambiente, uma vez que o uso dos sensores está explicita na programação de um agente. Um ponto relevante de se levantar sobre este tipo de percepção é que é possível emular a percepção passiva fazendo com que o agente consulte por conta própria os sensores a cada intervalo de tempo.

4.1.1.2 Ações

Para atuar no ambiente o agente deve possuir uma forma de informar-lhe que deseja realizar tal ação e com quais parâmetros deseja realizá-la. Um exemplo seria "mover até 10, 12", onde "mover" é a ação e "10, 12" são coordenadas passadas como parâmetros da ação de mover-se.

A sequencialidade das ações é um ponto que deve ser levado em consideração na elaboração do modelo, uma vez que um agente pode querer realizar múltiplas ações, uma ao término da outra. Isso então adiciona um requisito na implementação do sistema de agentes, de que seja possível que os agentes aguardem o término de uma ação para prosseguir com seu plano.

4.1.2 Responsabilidades do ambiente virtual

Como diversos agentes atuam em um mesmo ambiente virtual, este deve ser capaz de administrar as diversas entidades que representam estes agentes, de forma que ao receber requisições de percepção ou de atuação de um agente ele saiba distinguir qual entidade representa o agente. As demais responsabilidades do ambiente virtual se espelham nas do sistema de agentes, ou seja, ele deve ser capaz de informar o agente de suas percepções e deve ser capaz de realizar as ações que o agente requisitar.

4.1.2.1 Percepções

O ambiente deve ser capaz de enviar para o agente as percepções e suas propriedades, como por exemplo "gosta(musica)", que informa da percepção de que o agente gosta de algo e que apresenta a propriedade "musica".

Dependendo se a percepção for passiva ou não o ambiente também deve ser capaz de enviar por conta própria as percepções para os agentes. A seção 4.1.4 elabora sobre qual tipo de percepção é esperado do modelo final.

4.1.2.2 Ações

Para que um agente realize ações no ambiente, este deve ser capaz manifestá-las e retornar para o agente as novas percepções resultantes de suas ações.

Um problema que surge durante execução de ações é a concorrência para utilizar determinados recursos. Uma demonstração desse problema seria um agente que deseja realizar estas duas ações:

- Correr para uma direção;
- Chutar uma bola.

Estas ações são concorrentes, uma vez que ambas necessitam de um mesmo recurso, as pernas do agente neste exemplo. Para este problema existem diversas soluções possíveis das quais algumas são:

- Executar o código de ambas ao mesmo tempo, mesmo que isso cause um comportamento incorreto;
- Executar uma delas enquanto a outra fica em uma fila de espera;
- Cancelar uma e executar a outra.

Mas isto nem sempre é um problema, uma vez que duas ações podem ser completamente independentes uma da outra. Como por exemplo olhar para uma direção e mover os braços, duas ações que não interferem-se.

Embora a responsabilidade de definir quais ações executar seja do raciocínio do agente, é possível que ele decida executar múltiplas ações concorrentes que o modelo pode tratar caso necessário. O comportamento do modelo diante deste problema é especificado no tópico que explicita todos os pontos do modelo final, na subseção 4.1.4.

4.1.3 Comunicação

O sistema de agentes e o ambiente virtual são, à principio, aplicações distintas. Com este ponto em mente é necessário estabelecer uma forma de comunicação entre as duas partes para que os fluxos de execução possam acontecer. Para isso podemos utilizar troca de mensagens tanto através do protocolo UDP quanto TCP. A principal diferença entre as duas abordagens é a necessidade de que se estabeleça uma conexão entre as partes no protocolo TCP, o que não ocorre no protocolo UDP.

4.1.4 Síntese do modelo

Com os pontos levantados nas subseções anteriores, o modelo final é estabelecido com as seguintes características e motivações:

4.1.4.1 Comunicação TCP

Este protocolo de comunicação foi escolhido por apresentar duas características principais:

- Retransmissão e verificação de erros:

  Permite que a comunicação com o ambiente ocorra com maior garantia, de forma que a simulação não é afetada por eventuais problemas na rede.

- Baseada em conexão:

  Permite que o fluxo de comunicação seja simplificado pois, após conectados, ambos os lados podem facilmente enviar mensagens e esperar respostas da outra aplicação. Graças à isso o problema da sequencialidade de ações é resolvido facilmente como uma espera de resposta.

4.1.4.2 Modelo cliente-servidor

Devido ao uso do protocolo TCP, deve-se estabelecer a forma que o modelo cliente-servidor deve ser utilizado. Tendo em vista que existe um ambiente que administra múltiplos agentes que se comunicam com ele, foi determinado que o ambiente atuará como servidor e os diversos agentes atuarão como clientes que se conectam nele para realizar suas ações e percepções.

Esta abordagem se opõe ao que foi feito no trabalho de Freitas (2018), onde o agente atuava como servidor e os diversos sensores se conectavam no ambiente como clientes. Esta abordagem não foi utilizada no modelo proposto por ela ser mais adequada para ambientes de percepção passiva, onde o ambiente é responsável por enviar as mensagens ao agente.

4.1.4.3 Percepção ativa

O modelo proposto por este trabalho utiliza percepção ativa por se adequar melhor no fluxo de troca da mensagens TCP, além de ser uma abordagem com melhor acoplamento, uma vez que não é papel do ambiente informar o agente de suas percepções, principalmente quando se deseja representar um agente realista, onde não faria sentido o "mundo real" enviar as percepções para o agente, o que não é o caso do modelo deste trabalho.

4.1.4.4 Administração de agentes

Como o modelo suporta múltiplos agentes é esperado que em ambas as partes, tanto no sistema de agentes quanto no ambiente, seja possível distinguir um agente inteligente através de um identificador e que exista uma forma de transformar esse identificador em uma representação textual (*string*) para ser enviada nas mensagens entre as partes. Além disso o ambiente deve estar ciente das diversas entidades que representam agentes e deve ser capaz de traduzir de um identificador de um agente para uma entidade do ambiente que representa este agente. Este identificador é utilizado pelo ambiente ao receber uma mensagem para saber qual dos diversos agentes está enviando a requisição para ele.

4.1.4.5 Fluxo de mensagens

Devido a semelhança entre a requisição de uma ação e de uma percepção, existe apenas um formato de mensagem para ambas. A principal razão desta semelhança é o fato de que "perceber um sensor" pode ser considerado uma ação como qualquer outra. A forma que o fluxo de mensagens ocorre pode ser resumido em mensagens para o ambiente no formato "faça X com parâmetros Y" e a resposta do ambiente pode ser resumida em "X ocorreu com FALHA/SUCESSO, percebendo-se Z".

A especificação da troca de mensagens é simples, existem dois tipos de mensagens: "mensagem de ação" e "mensagem de resposta de ação". No momento em que o agente deseja realizar uma ação ele envia a mensagem de ação para o ambiente, que executa a ação e retorna a mensagem de resposta para que o sistema de agentes saiba o resultado

da ação. O formato escolhido para codificar as mensagens é o formato JSON devido a sua crescente popularidade e facilidade de implementação, a especificação das mensagens neste formato encontra-se a seguir:

Mensagem de Ação: As mensagens de ação são enviadas no sentido do sistema de agentes para o ambiente e servem para manifestar ao ambiente que um determinado agente deseja atuar ou perceber algo sobre ele. Sua especificação é dada por:

Especificação:

```
{
   "type" : "ACT",
  "agent" : <nome do agente>,
 "action" : <nome da ação>,
 "params" : [
              <parâmetro 1>,
              <parâmetro 2>,
              ...
              <parâmetro n>
            ]
}
```

Exemplo:

```
{
   "type" : "ACT",
  "agent" : "cleaner_1",
 "action" : "moveTo",
 "params" : [
              "10",
              "12"
            ]
}
```

Ao enviar um comando para o ambiente, o sistema de agentes deve construir esta mensagem JSON com os campos mostrados acima. Todas as mensagens possuem um campo "type" para especificar qual o tipo de mensagem e portanto quais campos esperar dela. O próximo campo é o "agent", utilizado para especificar qual agente está executando o comando, uma vez que no ambiente virtual podem existir diversos agentes, este identificador é o identificador discutido em 4.1.4.4. O terceiro campo é o "action", que especifica qual ação deverá ser realizada por este agente e que, como dito anteriormente, também engloba percepções, uma vez que "perceber algo" pode ser tratado como uma ação. O ultimo campo é uma lista, onde cada elemento desta lista é um parâmetro para a execução da ação em questão.

O exemplo ao lado demonstra o caso onde um agente chamado "cleaner_1" deseja realizar a ação "moveTo" com os parâmetros "10" e "12". Um contexto possível para esta mensagem pode ser que o agente em questão deseja mover-se para as coordenadas 10 e 12 do ambiente.

Mensagem de Resposta: As mensagens de resposta são transmitidas no sentido do ambiente para o sistema de agentes e servem para confirmar o status do comando recebido e para repassar percepções obtidas em uma ação. Sua especificação é dada por:

Especificação:

```
{
     "type" : "ACT_RESPONSE",
   "status" : "SUCCESS"
              |"FAILURE"
              |"ERROR",
 "percepts" : [
   {
    "percept" : <percepção 1>,
    "perceptValues" : [
      <valor 1.1>,
      <valor 1.2>,
      ...
      <valor 1.n>
     ],
    "action" : "ADD"|"REMOVE"
   }, {
    "percept" : <percepção 2>,
    "perceptValues" : [
      <valor 2.1>,
      <valor 2.2>,
      ...
      <valor 2.n>
     ],
    "action" : "ADD"|"REMOVE"
   }, {
    ...
   }
  ]
}
```

Exemplo:

```
{
     "type" : "ACT_RESPONSE",
   "status" : "SUCCESS",
 "percepts" : [
   {
    "percept" : "currentPos",
    "perceptValues" : [
      "10",
      "12"
     ],
    "action" : "ADD"
   }
  ]
}
```

Assim como a outra mensagem, esta também possui um campo "type" para que o destinatário possa saber quais outros campos esperar dela. O campo seguinte é o "status" que deve ser uma das três possibilidades mostradas, servindo para responder para o sistema de agentes sobre a execução da ação onde, "SUCCESS" significa que tudo ocorreu como o esperado, "FAILURE" significa que a ação não pôde ser concluída devido a algum acontecimento no ambiente que pode ter impossibilitado sua conclusão, uma vez que ambientes podem ser dinâmicos, por fim este campo pode ser "ERROR" para representar algum erro além da execução do ambiente, como uma mensagem de comando mal formulada ou uma divisão por zero por exemplo. Esta distinção é útil para que o sistema de agentes possa reagir adequadamente, pois no caso de erro pode ser que seja necessário reenviar o comando, enquanto que no caso de falha pode ser que o agente possa utilizar esta informação. Por fim, o campo "percepts" contem uma lista de triplas, onde os elementos de cada tripla representa o nome de uma percepção, seus valores e se esta percepção deve ser adicionada ou removida.

O exemplo mostrado ao lado representa uma execução bem sucedida da ação "moveTo" utilizada de exemplo nas mensagens de co-

mando. Para representar que a ação ocorreu corretamente o campo "status" está como "SUCCESS" e, como resultado, retorna em sua lista de percepções a percepção "currentPos" com os valores "10" e "12", indicando que sua nova posição atual é a coordenada 10, 12.

4.1.4.6 Concorrência de ações

Em relação ao problema da concorrência de ações o modelo não exige nenhum comportamento específico, transferindo esta responsabilidade para a implementação do modelo. Esta solução é a mais vantajosa pois permitir ou não a execução concorrente de ações depende do contexto do problema e, portanto, não é papel do modelo limitar a execução destas ações. Ao estabelecer o modelo desta forma, ele permite que implementação do problema trate da concorrência de recursos apenas nos casos em que for necessário.

4.1.4.7 Interoperabilidade

Um requisito deste modelo é que ele seja independente de plataforma, de modo que um agente implementado em uma determinada ferramenta possa executar em um ambiente virtual de qualquer outra ferramenta que implemente este modelo. O caminho inverso também é verdade, onde um determinado ambiente deve aceitar agentes de qualquer linguagem de agentes que implementem este modelo.

Este ponto do modelo não é ativamente implementado, mas surge como resultado dos demais requisitos do modelo, que por suas vez foram construídos de forma a proporcionar esta propriedade. Este requisito serve apenas para explicitar que o modelo é independente de ferramenta e que os demais requisitos garantem isso.

4.2 IMPLEMENTAÇÃO

Com o modelo consolidado, é necessário implementá-lo para verificar sua validade. Como citado na subseção 2.4 as ferramentas utilizadas foram o JASON e o Unity como ferramentas de agentes inteligentes e ambiente virtual respectivamente. O primeiro passo é estabelecer a estrutura do *framework* em relação à implementação dos agentes por parte dos usuários, sendo estabelecida seguinte estrutura:

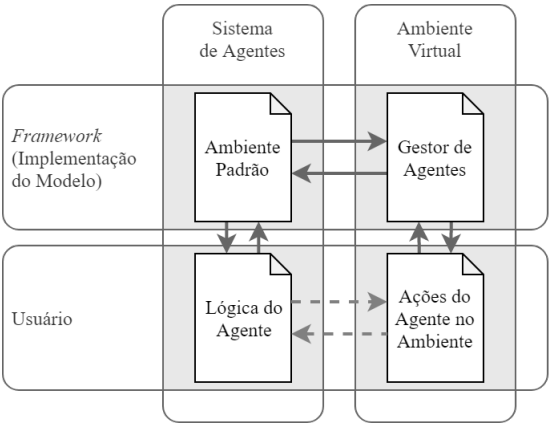

Figura 3 – Estrutura do framework.

Neste diagrama as colunas representam qual o lado do sistema a implementação se encontra, enquanto que as linhas especificam de quem é a responsabilidade de programar aquela parte do sistema. Podemos visualizar como a implementação que é realizada nesta seção se responsabiliza pela comunicação entre as ferramentas, enquanto que o usuário apenas deve programar seus agentes atuando no ambiente padrão e as manifestações de suas ações no ambiente virtual. Para isso é necessário que a implementação do modelo cubra:

- A criação de um ambiente padrão na linguagem de ambientes do sistema de agentes (que no caso do JASON é a linguagem Java), onde este ambiente padrão tem a capacidade de receber as ações do agente, transformando-as em mensagens, no formato estabelecido para isso neste trabalho, para então repassá-las por protocolo TCP e esperar uma resposta do ambiente;
- No outro lado do sistema, no ambiente virtual, devem haver receptores destas mensagens capazes de traduzir nomes de ações em ações programadas pelo usuário e responder às mensagens com as percepções retornadas pelas ações.

Esta estrutura fornece duas principais vantagens:

- Do ponto de vista do usuário as ações que o agente executa no sistema de agentes são manifestadas diretamente como suas implementações no ambiente, sem ter que se preocupar com os aspectos

de troca de mensagem que são responsabilidade do *framework*. Em outras palavras, do ponto de vista do usuário a comunicação do agente com o ambiente parece acontecer como nas setas pontilhadas, enquanto que na realidade ela ocorre pelo caminho das setas preenchidas;

- Uma vez implementada, esta estrutura resulta em um código genérico o suficiente para que usuários possam representar diversos problemas de agentes inteligentes programando apenas o agente e seu comportamento no ambiente, sem a necessidade de modificar a implementação do modelo uma vez que para implementar novos agentes e ações não é necessário modificar o código do *framework*.

A implementação desta estrutura para uso com a ferramenta JASON resulta no seguinte diagrama de sequência (simplificado para facilitar leitura):

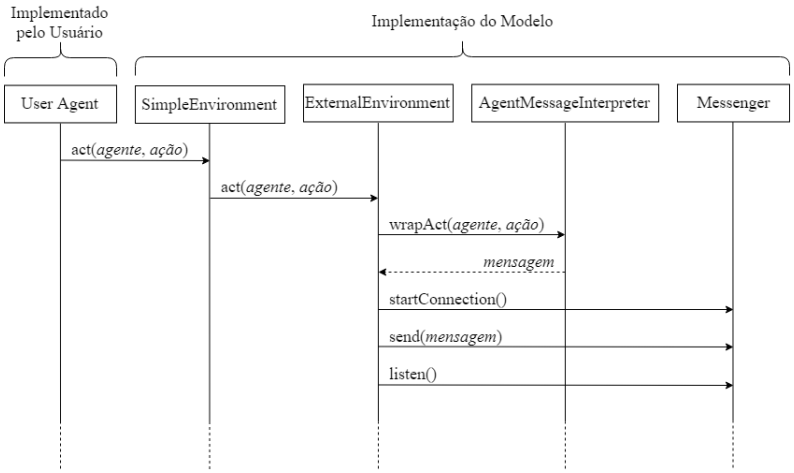

Figura 4 – Diagrama de sequência do fluxo de execução de uma ação.

A classe *SimpleEnvironment* representa um ambiente padrão da linguagem JASON que repassa a ação para o *ExternalEnvironment*, que é responsável por realizar a chamada de ações que requerem o ambiente virtual. Esta classe realiza chamadas que irão transformar a ação em uma mensagem de ação, conectar o sistema ao ambiente virtual, enviar a mensagem para ele e esperar sua resposta.

É importante ressaltar que a chamada de *listen()* irá bloquear a *thread* deste agente, fazendo com que esta ação espere a resposta do ambiente para terminar sua execução, comportamento este que é esperado da implementação do modelo para fornecer a sequencialidade das ações de um plano. Isto é possível pois na ferramenta JASON cada agente é executado por uma *thread* e portanto bloqueá-lo não impede os demais agentes de executar. Esta solução pode não ser viável para todos os sistemas de agente pois seu uso de *threads* pode variar.

O diagrama de sequência (também simplificado) na parte do ambiente virtual se encontra à seguir:

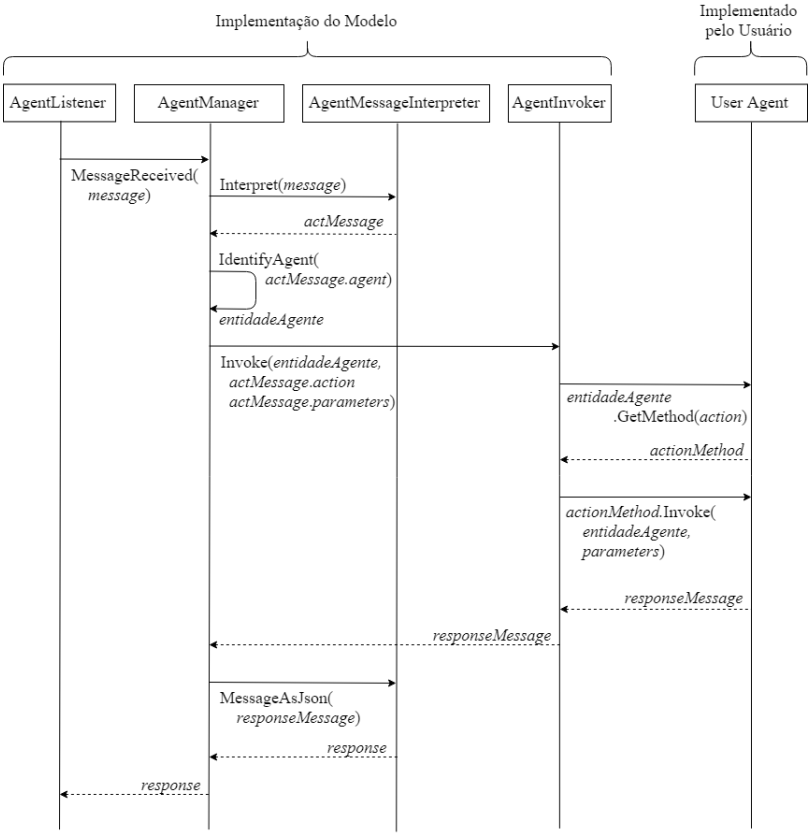

Figura 5 – Diagrama de sequência do recebimento de mensagem de ação no ambiente.

Ao receber uma mensagem o ambiente a envia para a classe *AgentManager* que é responsável por administrar todos os agentes do ambiente. Esta classe interpreta a mensagem recebida, identifica qual entidade é representada por aquele identificador de agente, invoca a ação especificada na mensagem, transforma a resposta da ação em uma mensagem no formato esperado e a retorna para o *AgentListener* que responde a mensagem TCP com ela. Este processo executa inteiramente como uma **corotina** (seção 2.4.2.1.1), permitindo que todo o processamento da ação, e por consequência a ação em si também, ocorram em múltiplos ciclos do *game loop*.

A classe *AgentInvoker* utiliza o conceito de **reflexão** (visto na seção 2.4.2.1.2) para obter um método do código do agente através de seu nome (chamada "GetMethod(*action*)" no diagrama acima), seu uso é vantajoso nesse momento para que não seja necessário que o usuário mapeie o nome das ações no sistema de agente para as ações no ambiente virtual, este processo é automático, bastando que o nome das ações executadas nos planos do agente sejam as mesmas da implementação no ambiente.

Após o ambiente responder a mensagem, a execução do sistema de agentes prossegue de sua espera de acordo com o seguinte diagrama de sequência:

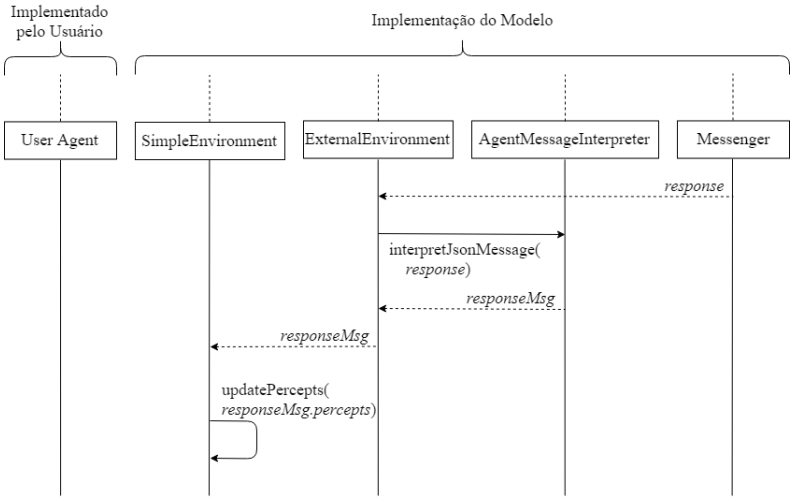

Figura 6 – Diagrama de sequência do fluxo de execução de uma ação após a resposta.

A finalização do processo é simples, bastando interpretar a resposta recebida para tratar as percepções retornadas pela ação, podendo elas serem percepções que devem ser adicionadas ou removidas. Note que o agente implementado pelo usuário não precisa implementar nenhum tratamento sobre as percepções recebidas para recebê-las adequadamente, pois estas são processadas pelo *framework*.

Um ponto importante de ressaltar nesta implementação é a utilização de *threads* disponibilizada pela ferramenta JASON. A implementação demonstrada acima faz com que a execução do plano ponha a *thread* para dormir, o que pode ser um problema caso a ferramenta esteja configurada para executar com apenas uma *thread* por exemplo, nesse caso a ferramenta pararía de responder até que a mensagem de resposta fosse obtida. Tendo isso em mente fica claro que para utilizar o *framework* é necessário configurar o sistema adequadamente conforme as necessidades do problema. A configuração mais simples, e que é utilizada na avaliação desta implementação, é a criação de uma *thread* para cada agente, de forma que quando um agente deseja realizar uma ação sempre haverá uma disponível para que ele use-a, esta solução no entanto pode se tornar um problema caso muitos agentes existam pois ela causaria um *overhead* pois o sistema operacional teria que administrar muitas *threads*.

4.2.1 Especificação para usuários

Para desenvolver um agente na linguagem de agentes basta desenvolver o código em *AgentSpeak* e executá-lo no ambiente *SimpleEnvironment*. O nome das ações executadas nos planos devem ser os mesmos de suas respectivas implementações no código do agente no ambiente.

No ambiente virtual o código de um agente precisa ser uma classe que herda de *IntelligentAgent*, esta herança irá permitir que o administrador de agentes encontre as instâncias de seus agentes, além de disponibilizar um atributo *agentIdentifier* que serve para identificar os agentes instanciados de sua classe. A classe resultante será um "componente" no unity, significando que suas instâncias podem ser atribuidas a objetos do mundo virtual. A implementação de cada ação do agente pode ocorrer em uma única chamada ou através de varias iterações do *game loop*. Para implementar uma ação de chamada única basta que ela retorne *void* ou um objeto *ActResponseMessage*. Já para implementar uma ação de múltiplas iterações é necessário implementá-la como

uma corotina, retornando um *IEnumerator*. Caso deseje que a ação retorne percepções basta que ao final de sua execução haja o retorno de um objeto *ActResponseMessage*. Além disso todas as ações que podem ser invocadas pelo agente devem estar demarcadas com o decorador *AgentAction*.

4.3 AVALIAÇÃO

4.3.1 Ambiente de testes

Para realizar a validação e avaliação da implementação do modelo é necessário criar um ambiente virtual que utilize-a. O ambiente escolhido é um exemplo de ambiente *Cleaning Robots* onde, há uma lata de lixo e lixo surge com o passar do tempo. O comportamento do agente deste ambiente se resme a:

- O agente sempre deseja que o ambiente esteja limpo;
- Se ele conhece a posição de algum lixo ele irá se deslocar até ele e irá pegá-lo;
- Ao pegar um lixo ele se deslocará até a lata de lixo e descartará ele;
- Caso o agente não conheça a posição de algum lixo ele irá escanear a região até que encontre mais lixo.

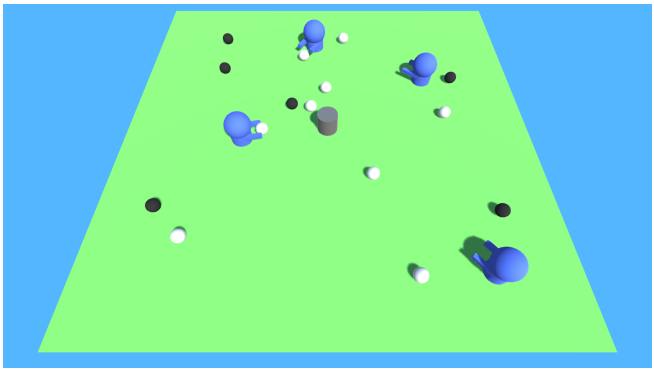

Figura 7 – Ambiente de exemplo no Unity.

A figura 7 é uma demonstração do ambiente em funcionamento[1]. As bolinhas brancas e pretas são lixo, sendo que as pretas surgiram depois do escaneamento de um agente, de forma que elas não constam nas percepções de nenhum deles por enquanto.

A implementação do agente em *AgentSpeak* encontra-se à seguir:

```
// Desejos iniciais
    !findTrashCan.
    !findTrash.

// Planos
+carryingTrash <-
     !dispose.

+!findTrashCan <-
     locateTrashCan.

+!clean(X,Y) : not carryingTrash <-
     .broadcast(achieve, forget(X,Y));
     moveTo(X,Y);
     pickupTrashAt(X,Y);
     if (not carryingTrash) {
          !findTrash
     }.

+!forget(X,Y) <-
 .abolish(trash(X,Y)).

+!dispose : carryingTrash & trashCan(X,Y) <-
     moveTo(X,Y);
     disposeTrash;
     !findTrash.

+!findTrash : trash(X,Y) <-
     !clean(X,Y).
+!findTrash : not trash(_,_) <-
     scanSurroundings;
     !findTrash.
```

Inicialmente o agente deseja encontrar a lata de lixo e também seu primeiro lixo. Em seguida temos os seguintes planos em ordem de sua declaração no código:

- Ao perceber que está carregando um lixo o agente desejará descartar ele;
- Quando desejar encontrar a lata de lixo o agente executará a ação **locateTrashCan**;
- Ao desejar limpar um lixo em uma coordenada do ambiente e não estiver carregando outro lixo (*not carryingTrash*) o agente irá limpá-lo. Neste plano o agente primeiro avisa os demais agente

[1] Uma demonstração em vídeo do ambiente em questão pode ser acessada neste endereço: youtu.be/XBO5itsxsck

para que esqueçam deste lixo, pois ele já irá limpá-lo. Em seguida ele executa as ações **moveTo(X,Y)** e **pickupTrashAt(X,Y)** que são ações do ambiente para se deslocar até uma posição e para pegar um lixo localizado nesta posição respectivamente. É importante observar como a ação de pegar o lixo só executará depois que a ação de deslocamento ocorrer completamente. Por fim, caso após realizar a ação de pegar o lixo ele não esteja carregando um lixo, ele desejará encontrar outro, isto pode ocorrer caso outro agente já tenha pego aquele lixo;

- O plano para esquecer de um lixo é simples, uma vez que ele apenas remove a percepção da existência de um lixo na posição especificada.

- O desejo de descartar um lixo só é executado caso o agente esteja carregando um e saiba a localização da lata de lixo. Este plano é simples, bastando mover-se para a posição da lata de lixo para em seguida descartá-lo e desejar procurar mais lixo;

- Encontrar mais lixo pode ser executado de duas maneiras. A primeira ocorre caso o agente já conheça a posição de algum lixo, nesse caso ele desejará limpá-lo. Caso o agente deseje encontrar o próximo lixo mas não tenha conhecimento de nenhum o segundo plano ocorre, onde ele executará a ação **scanSurroundings** que irá escanear a região por mais lixo.

Como a implementação das ações no ambiente é extensa, é apresentado à seguir apenas a implementação de uma ação. O código completo encontra-se no apêndice A. Segue a implementação da ação **locateTrashCan**:

```
1  [AgentAction]
2  public ActResponseMessage LocateTrashCan() {
3       float trashCanX = trashCan.position.x;
4       float trashCanY = trashCan.position.z;
5       return new ActResponseMessage(
6            new List<Percept> {
7                 new Percept("trashCan",
8                         new List<string> {
9                                                trashCanX.ToString(),
10                                               trashCanY.ToString()
11                                          },
12                        PerceptAction.ADD)
13           }
14      );
15 }
```

Esta é uma ação simples que obtém as coordenadas da lata de lixo e retorna um novo objeto *ActResponseMessage* contendo a percep-

ção “trashCan” que deve ser adicionada com as posições X e Y da lata de lixo.

4.3.2 Resultados

Os testes foram realizados em um computador com as seguintes especificações:

- Processador Intel Core i5-7400 @3.00GHz;
- Placa de vídeo NVIDIA GeForce GTX 1060 6GB;
- 2×8GB de memória RAM DDR4.

4.3.2.1 Avaliação do ambiente

O primeiro resultado analisado é a quantidade de quadros por segundo (FPS) que o ambiente proporcionou conforme o número de agentes aumentava, com o objetivo de avaliar qual se há impacto no desempenho de um ambiente contendo muitos agentes. Os testes avaliam uma execução normal do programa durante um período de um minuto. Os resultados são apresentados à seguir:

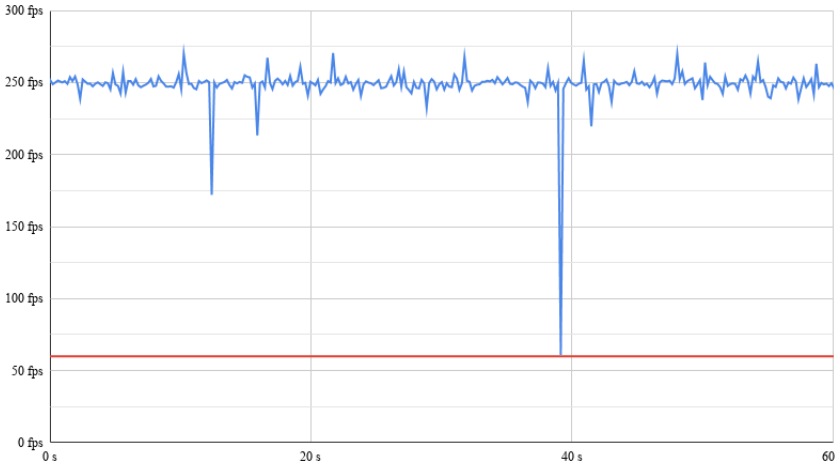

Figura 8 – FPS ao longo do tempo com 10 agentes.

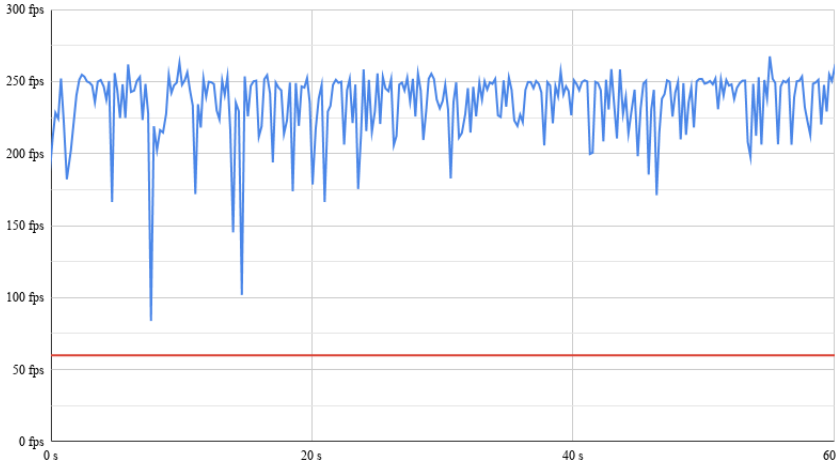

Figura 9 – FPS ao longo do tempo com 25 agentes.

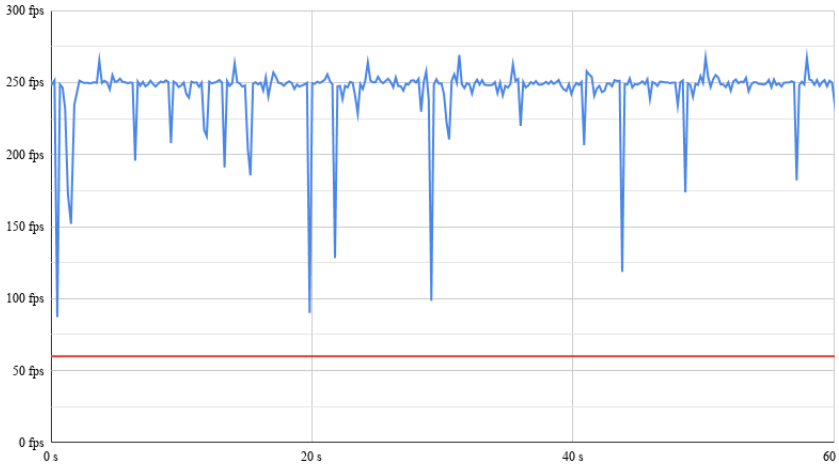

Figura 10 – FPS ao longo do tempo com 75 agentes.

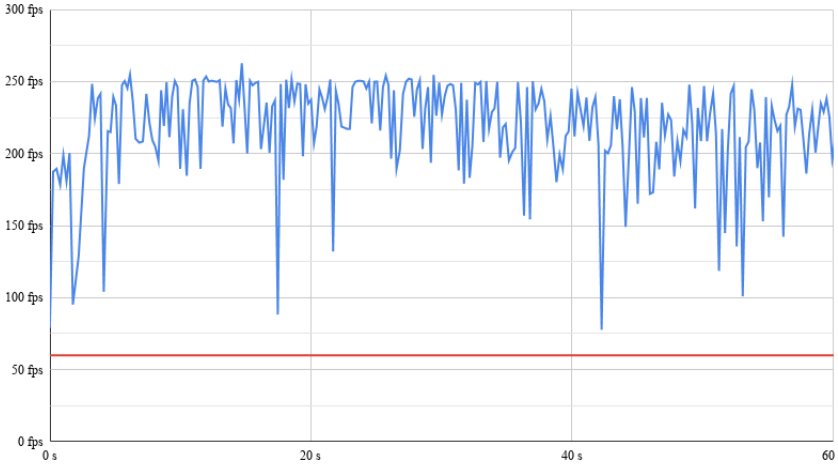

Figura 11 – FPS ao longo do tempo com 100 agentes.

Os gráficos acima exibem as estatísticas coletadas em azul, enquanto que a linha vermelha demarca a faixa de 60 quadros por segundo, o que é considerado uma taxa de atualização ótima para jogos eletrônicos.

É possível observar principalmente que em nenhum momento a taxa de atualização cai para baixo da faixa vermelha, significando que o impacto no desempenho não é facilmente perceptível. Outra característica dos dados é o aumento da variância conforme o número de agentes aumenta, provavelmente causado pela maior quantidade de troca de mensagens que deve ser realizada.

Em alguns casos o processamento das mensagens acaba demorando mais tempo, este fenômeno pode ser percebido nas quedas muito grandes, como por exemplo na figura 8. A motivação para estas quedas não é clara, uma vez que uma mesma mensagem pode ser processada rapidamente em um momento e causar uma grande queda em outro. Estas quedas não são perceptíveis no teste escolhido pois o sistema se recupera rapidamente. Em um sistema mais complexo é possível que a média de *frames* por segundo seja menor, fazendo com que estas quedas se tornem perceptíveis, este problema no entanto não está relacionado ao modelo e sim com a complexidade do cenário na *game engine* em si.

A grande variação de FPS pode ser um problema em casos onde medidas precisas são necessárias, uma vez que a ferramenta executa códigos por *frame*, o que resulta em execuções cujo intervalo entre cada

iteração não é constante. A ferramenta Unity apresenta recursos para melhor gerenciar este comportamento, como por exemplo a possibilidade de compensar ações com base no tempo desde a ultima execução do *game loop* ou a possibilidade de realizar a execução em um *game loop* paralelo ao principal, onde este segundo itera em intervalos fixos de tempo. Apesar destes recursos ainda assim é provável que certo ruído surja nas medidas, embora menos significativo. Além disso, caso o modelo seja implementado em outros ambientes virtuais pode ser que eles não apresentem estes recursos e a variação na taxa de *frames* se torne um problema.

4.3.2.2 Avaliação da ferramenta de agentes

Além do ambiente, também é preciso avaliar o desempenho no sistema de agentes. Uma das principais métricas para avaliar um sistema desenvolvido em JASON é a quantidade de ciclos de raciocínio executados por segundo. As médias e desvios padrão coletados nos testes dessa métrica encontram-se na tabela a seguir:

Num. de Agentes	Média	Desvio Padrão
10	$\approx$ 1199,455	$\approx$ 824,235
25	$\approx$ 1323,655	$\approx$ 536,523
75	$\approx$ 754,200	$\approx$ 159,534
100	$\approx$ 602,062	$\approx$ 141,735

Tabela 2 – Resultados coletados sobre a quantidade de ciclos por segundo do sistema de agentes.

É possível perceber com os dados coletados que conforme o número de agentes aumenta a quantidade de ciclos de raciocínio por segundo diminui, resultado já esperado, uma vez que mais agentes implicam em um maior uso do processador. O desvio padrão apresentou um comportamento interessante, onde ele reduzia conforme o número de agentes crescia, ou seja, o sistema se tornava mais consistente em relação a quantidade de ciclos de raciocínio. A motivação para este comportamento não é clara, entretanto a hipótese é que a ferramenta JASON esteja limitando a execução do próximo ciclo de raciocínio de cada agente para que eles estejam em sincronia com os demais, isto causaria um efeito onde agentes que levam menos tempo em seus ciclos deveriam esperar os demais, nivelando o tempo por ciclo e consequen-

temente diminuindo o desvio padrão. Este comportamento também contribuiria para a redução da média de ciclos por segundo junto ao simples fato de existirem mais agentes para serem administrados.

Mesmo com a queda de desempenho não foi possível observa-la visivelmente no ambiente de teste, de forma que para aplicações menos precisas a solução é facilmente viável. No entanto para aplicações que precisam de um melhor desempenho é possível que precauções devam ser tomadas para melhor configurar as ferramentas e extrair um melhor desempenho.

5 CONCLUSÃO

A solução proposta é satisfatória diante dos resultados obtidos. O modelo atende aos requisitos propostos no início do trabalho e o *framework* implementa os requisitos do modelo corretamente.

Os testes também foram bem sucedidos. O ambiente virtual não manifestou nenhuma queda de desempenho drástica, sempre se mantendo acima da faixa de 60 quadros por segundo. O sistema de agentes também obteve bons resultados, uma vez que que os agentes se manifestaram da maneira correta no ambiente sem apresentar erros ou atrasos na tomada de decisões dos agente. Em ambos os casos os valores dos critérios de avaliação diminuíram conforme o número de agentes aumentava, o que já era esperado. Entretanto, em nenhum dos casos os sistemas apresentaram grandes perdas de desempenho, indicando que os objetivos deste trabalho foram cumpridos.

5.1 TRABALHOS FUTUROS

A partir do que foi desenvolvido neste trabalho é possível desenvolver novos trabalhos relacionados a este:

- Estabelecer um novo tipo de mensagem para comunicação entre agentes:

  Atualmente não é possível, através do ambiente, que um agente envie uma requisição para outro agente pois não há suporte para este fluxo de mensagens. Um possível trabalho futuro estabeleceria o formato deste tipo de mensagem, analisando aspectos como a necessidade ou não do uso do protocolo TCP neste tipo de mensagem por exemplo. Este fluxo poderia ser utilizado para um agente requisitar a percepção de outro, ou informar outros de suas próprias percepções. No caso deste trabalho a linguagem JASON suporta a comunicação entre agentes na ferramenta, sem interação com o ambiente, de forma que não é possível tornar a comunicação parte do ambiente, como por exemplo simulando falhas de transmissão, algo que seria responsabilidade do ambiente. Além disso esta implementação permitiria que ferramentas sem suporte a comunicação entre agentes pudessem realizá-las.

  Este novo fluxo faria com que a conexão ocorresse no sentido do ambiente se conectando a um agente, o que é interessante também

para realizar percepção passiva, uma vez que se o ambiente é capaz de se comunicar com seus agentes ativamente, ele é capaz se informar à eles suas percepções sem que haja uma requisição para isso.

- Adicionar suporte para o modelo de artefatos:

  O modelo de artefatos apresenta uma forma de criar objetos interativos no mundo. Neste trabalho para que objetos pudessem ser manipulados eles eram identificados por coordenadas, o que não é uma abordagem muito adequada pois nada garante que dois objetos não possam estar na mesma coordenada. Com um suporte a artefatos seria possível estabelecer identificadores para os artefatos de forma que todos os agentes pudessem definir unicamente cada artefato, além de permitir a execução de ações descritas pelos próprios artefatos, de forma análoga ao que é realizado por Poli (2018) em seu trabalho apresentado na subseção 3.4.

- Expandir o *framework* para outras linguagens:

  A utilização de JASON e Unity neste trabalho apresentou diversos desafios com soluções interessantes, como o uso de corotinas e reflexão na parte do Unity por exemplo. Expandir o *framework* para outras *game engines*, como a *Unreal Engine*, ou para outras linguagens de agente, como o Sigon (utilizado no trabalho correlato da subseção 3.3), pode apresentar outros desafios específicos destas ferramentas e que resultariam em um trabalho interessante.

REFERÊNCIAS

BAJCSY, R.; ALOIMONOS, Y.; TSOTSOS, J. K. Revisiting active perception. *Autonomous Robots*, 2018. <https://doi.org/10.1007/s10514-017-9615-3>.

BORDINI, R. H.; HÜBNER, J. F.; WOOLDRIDGE, M. *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*. USA: John Wiley & Sons, 2007.

BROWNING, B.; TRYZELAAR, E. Übersim: a multi-robot simulator for robot soccer. In: *The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, July 14-18, 2003, Melbourne, Victoria, Australia, Proceedings*. [S.l.: s.n.], 2003.

BRUNS, G.; POLANI, D.; UTHMANN, T. Eine virtuelle kontinuierliche welt als testbett für ki-modelle. *KI*, v. 15, n. 1, 2001.

BUCK, S.; BEETZ, M.; SCHMITT, T. M-rose: A multi robot simulation environment for learning cooperative behavior. In: ASAMA, H. et al. (Ed.). *Distributed Autonomous Robotic Systems 5*. Tokyo: Springer Japan, 2002.

CYBERBOTICS LTD. *Webots User Guide*. [S.l.], Abril 2004.

DORRI, A.; KANHERE, S. S.; JURDAK, R. Multi-agent systems: A survey. *IEEE Access*, v. 6, 2018.

FREITAS, G. S. d. Um modelo de raciocínio sobre sensores para agentes sigon em ambientes de realidade virtual. 2018. <https://repositorio.ufsc.br/handle/123456789/192166>.

GELAIM, T. Ângelo et al. Sigon: A multi-context system framework for intelligent agents. *Expert Systems with Applications*, v. 119, 2019.

GREGORY, J. *Game Engine Architecture, Second Edition*. 2nd. ed. Natick, MA, USA: A. K. Peters, Ltd., 2014.

HORSWILL, I. *UnityProlog*. 2017. <https://github.com/ianhorswill/UnityProlog>. Acessado em 27/11/2018.

KELTON, D. W.; SADOWSKI, R. P.; SADOWSKI, D. A. *Simulation with Arena*. 1st. ed. New York, NY, USA: McGraw-Hill, Inc., 1997. ISBN 0070275092.

KÖGLER, M.; OBST, O. Simulation league: The next generation. In: POLANI, D. et al. (Ed.). *RoboCup 2003: Robot Soccer World Cup VII*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

KRAVARI, K.; BASSILIADES, N. A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, v. 18, n. 1, 2015.

MICROSOFT CORPORATION. *Reflexão (C#)*. [S.l.], 2015. <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/reflection>. Acessado em 05/06/2019.

MICROSOFT CORPORATION. *Guia de C#*. [S.l.], 2018. <https://docs.microsoft.com/pt-br/dotnet/csharp/>. Acessado em 05/06/2019.

OBST, O.; ROLLMANN, M. Spark – a generic simulator for physical multi-agent simulations. In: LINDEMANN, G. et al. (Ed.). *Multiagent System Technologies*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

OMICINI, A.; RICCI, A.; VIROLI, M. Artifacts in the a&a meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, v. 17, n. 3, 2008.

POLI, N. *Game Engines and MAS: BDI & Artifacts in Unity*. Dissertação (Mestrado) — University of Bologna, 2018. <http://amslaurea.unibo.it/15657/>.

PRANDI, H. Uma arquitetura para a atuação de agentes inteligentes. 2017. <https://repositorio.ufsc.br/handle/123456789/182208>.

RILEY, P. F.; RILEY, G. F. Spades — a distributed agent simulation environment with software-in-the-loop execution. In: *Proceedings of the 35th Conference on Winter Simulation: Driving Innovation*. [S.l.]: Winter Simulation Conference, 2003. (WSC '03).

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. Third. Upper Saddle River, NJ: Prentice Hall, 2010. (Series in Artificial Intelligence).

SMITH, R. *Open Dynamics Engine (ODE) User Guide*. [S.l.], Maio 2003.

UNITY TECHNOLOGIES. *Coroutines*. [S.l.], 2019. <https://docs.unity3d.com/Manual/Coroutines.html>. Acessado em 05/06/2019.

UNITY TECHNOLOGIES. *Unity User Manual (2019.1)*. [S.l.], 2019. <https://docs.unity3d.com/Manual/index.html>. Acessado em 05/06/2019.

WOOLDRIDGE, M. *Introduction to MultiAgent Systems*. [S.l.]: John Wiley & Sons, 2002.

APÊNDICE A – Framework JASON

O código para a implementação do framework na parte que utiliza o JASON encontra-se no repositório público localizado no seguinte endereço:

- https://github.com/vsSchweitzer/jason-external-environment

Este repositório também contém os códigos do agente utilizado no exemplo *Cleaning Robots*.

A.1 CÓDIGO FONTE

Segue a transcrição do código deste repositório, onde cada seção é um arquivo:

A.1.1 cleaner_agent.asl

```
1  // It starts wanting to know the location of the trash can
2  !findTrashCan.
3  !findTrash.
4
5  // Whenever it believes that it is carrying trash, it will desire to
       dispose of it
6  +carryingTrash <-
7   !dispose.
8
9  // Plan to find the trash can
10 +!findTrashCan <-
11  locateTrashCan.
12
13 // Plan to pickup trash
14 +!clean(X,Y) : not carryingTrash <-
15  .broadcast(achieve, forget(X,Y));
16  moveTo(X,Y);
17  pickupTrashAt(X,Y);
18  if (not carryingTrash) {
19   !findTrash
20  }.
21
22 // Plan to forget a trash because another agent already cleaned it
23 +!forget(X,Y) <-
24  .abolish(trash(X,Y)).
25
26 // Plan to dispose of trash
27 +!dispose : carryingTrash & trashCan(X,Y) <-
28  moveTo(X,Y);
29  disposeTrash;
30  !findTrash.
31
32 // Plan to find trash
33 +!findTrash : trash(X,Y) <-
34  !clean(X,Y).
35 +!findTrash : not trash(_,_) <-
```

```
36  scanSurroundings;
37  !findTrash.
```

A.1.2 exceptions/MalformedConfigurationsException.java

```
1  package br.ufsc.vsschweitzer.thesis.configuration.exceptions;
2
3  public class MalformedConfigurationsException extends Exception {
4
5   private static final long serialVersionUID = 1L;
6
7   public enum ErrorField {
8    IpAddress,
9    Port,
10   Other
11  }
12
13  ErrorField erroField = ErrorField.Other;
14
15     public MalformedConfigurationsException(ErrorField errorField) {
16         super();
17         setField(errorField);
18     }
19
20     public MalformedConfigurationsException(String message,
       ErrorField errorField) {
21         super(message);
22         setField(errorField);
23     }
24
25     public MalformedConfigurationsException(Throwable cause,
       ErrorField errorField) {
26         super(cause);
27         setField(errorField);
28     }
29
30     public MalformedConfigurationsException(String message,
       Throwable cause, ErrorField errorField) {
31         super(message, cause);
32         setField(errorField);
33     }
34
35  public ErrorField getField() {
36   return erroField;
37  }
38
39  private void setField(ErrorField field) {
40   this.erroField = field;
41  }
42
43 }
```

A.1.3 EnvironmentConfiguration.java

```
1  package br.ufsc.vsschweitzer.thesis.configuration;
2
3  import java.util.regex.Matcher;
4  import java.util.regex.Pattern;
```

```
5
6  import br.ufsc.vsschweitzer.thesis.configuration.exceptions.
       MalformedConfigurationsException;
7  import br.ufsc.vsschweitzer.thesis.configuration.exceptions.
       MalformedConfigurationsException.ErrorField;
8
9  public class EnvironmentConfiguration {
10
11  private static final String DEFAULT_IP_ADDRESS = "127.0.0.1";
12  private static final String DEFAULT_PORT = "12345";
13  private static final String IP_ADDRESS_REGEX = "\\d{1,3}\\.\\d
       {1,3}\\.\\d{1,3}\\.\\d{1,3}";
14  private static final String PORT_REGEX = "\\d{1,5}";
15
16  private String ipAddress;
17  private int port;
18
19  public EnvironmentConfiguration() {
20   try {
21    this.setIpAddress(DEFAULT_IP_ADDRESS);
22    this.setPort(DEFAULT_PORT);
23   } catch (Exception e) {
24    // Never happens
25   }
26  }
27
28  public EnvironmentConfiguration(String ipAddress, String port)
       throws MalformedConfigurationsException {
29   this.setIpAddress(ipAddress);
30   this.setPort(port);
31  }
32
33  public String getIpAddress() {
34   return ipAddress;
35  }
36
37  private void setIpAddress(String ipAddress) throws
       MalformedConfigurationsException {
38   verifyIpAddress(ipAddress);
39   this.ipAddress = ipAddress;
40  }
41
42  public int getPort() {
43   return port;
44  }
45
46  private void setPort(String port) throws
       MalformedConfigurationsException {
47   verifyPort(port);
48   this.port = Integer.parseInt(port);
49  }
50
51  private void verifyIpAddress(String ipAddress) throws
       MalformedConfigurationsException {
52   if (ipAddress != "localhost") {
53    boolean valid = verifyPattern(IP_ADDRESS_REGEX, ipAddress);
54    if (!valid) {
55     throw new MalformedConfigurationsException(ErrorField.IpAddress)
       ;
56    }
57    String[] octets = ipAddress.split(".");
```

```
58    for (String octet : octets) {
59     int octetAsInt = Integer.parseInt(octet);
60     if (octetAsInt < 0 || octetAsInt > 255) {
61      throw new MalformedConfigurationsException(ErrorField.IpAddress
       );
62     }
63    }
64   }
65  }
66
67  private void verifyPort(String port) throws
        MalformedConfigurationsException {
68   boolean valid = verifyPattern(PORT_REGEX, port);
69   if (!valid) {
70    throw new MalformedConfigurationsException(ErrorField.Port);
71   }
72   int portAsInt = Integer.parseInt(port);
73   if (portAsInt < 1 || portAsInt > 65535) {
74    throw new MalformedConfigurationsException(ErrorField.Port);
75   }
76  }
77
78  private boolean verifyPattern(String regex, String data) {
79   Pattern pattern = Pattern.compile(regex);
80
81   Matcher matcher = pattern.matcher(data);
82   boolean correct = matcher.matches();
83
84   return correct;
85  }
86
87 }
```

A.1.4 FailedActionException.java

```
1  package br.ufsc.vsschweitzer.thesis.environment.exceptions;
2
3  public class FailedActionException extends Exception {
4
5   private static final long serialVersionUID = 1L;
6
7   public FailedActionException() {
8      }
9
10     public FailedActionException(String message) {
11         super(message);
12     }
13
14     public FailedActionException(Throwable cause) {
15         super(cause);
16     }
17
18     public FailedActionException(String message, Throwable cause) {
19         super(message, cause);
20     }
21
22 }
```

A.1.5 ExternalEnvironment.java

```
1  package br.ufsc.vsschweitzer.thesis.environment;
2
3  import java.io.IOException;
4  import java.util.List;
5
6  import br.ufsc.vsschweitzer.thesis.configuration.
       EnvironmentConfiguration;
7  import br.ufsc.vsschweitzer.thesis.configuration.exceptions.
       MalformedConfigurationsException;
8  import br.ufsc.vsschweitzer.thesis.environment.exceptions.
       FailedActionException;
9  import br.ufsc.vsschweitzer.thesis.exceptions.
       ConnectionNotOpenException;
10 import br.ufsc.vsschweitzer.thesis.messaging.AgentMessageInterpreter
       ;
11 import br.ufsc.vsschweitzer.thesis.messaging.Messenger;
12 import br.ufsc.vsschweitzer.thesis.messaging.messages.ActMessage;
13 import br.ufsc.vsschweitzer.thesis.messaging.messages.
       ActResponseMessage;
14 import br.ufsc.vsschweitzer.thesis.messaging.messages.Percept;
15 import jason.asSyntax.Structure;
16
17 public class ExternalEnvironment {
18
19  private EnvironmentConfiguration configuration;
20
21  public ExternalEnvironment(String ipAddress, String port) throws
       MalformedConfigurationsException {
22   configuration = new EnvironmentConfiguration(ipAddress, port);
23  }
24
25  public List<Percept> act(String agent, Structure action) throws
       FailedActionException {
26   Messenger messenger = new Messenger(configuration.getIpAddress(),
       configuration.getPort());
27   try {
28    ActMessage message = AgentMessageInterpreter.wrapAct(agent,
       action);
29    String messageAsJson = AgentMessageInterpreter.messageToJson(
       message);
30
31    messenger.open();
32    messenger.send(messageAsJson);
33    String response = messenger.listen();
34    ActResponseMessage responseMsg = (ActResponseMessage)
       AgentMessageInterpreter.interpretJsonMessage(response);
35    return responseMsg.getPercepts();
36   } catch (IOException | ConnectionNotOpenException e) {
37    throw new FailedActionException(e);
38   } finally {
39    messenger.close();
40   }
41  }
42 }
```

A.1.6 ConnectionNotOpenException.java

```
1  package br.ufsc.vsschweitzer.thesis.exceptions;
2
3  public class ConnectionNotOpenException extends Exception {
4
5   private static final long serialVersionUID = 1L;
6
7   static final String MESSAGE = "Cannot send a message without
        starting the connection first";
8
9   public ConnectionNotOpenException() {
10   super(MESSAGE);
11  }
12
13 }
```

A.1.7 ActResponseStatus.java

```
1  package br.ufsc.vsschweitzer.thesis.messaging.messages.enums;
2
3  public enum ActResponseStatus {
4   SUCCESS,
5   FAILURE,
6   ERROR;
7  }
```

A.1.8 MessageType.java

```
1  package br.ufsc.vsschweitzer.thesis.messaging.messages.enums;
2
3  import br.ufsc.vsschweitzer.thesis.messaging.messages.ActMessage;
4  import br.ufsc.vsschweitzer.thesis.messaging.messages.
       ActResponseMessage;
5  import br.ufsc.vsschweitzer.thesis.messaging.messages.BaseMessage;
6
7  public enum MessageType {
8   ACT(ActMessage.class),
9   ACT_RESPONSE(ActResponseMessage.class);
10
11  Class<? extends BaseMessage> respectiveClass;
12
13  MessageType(Class<? extends BaseMessage> respectiveClass) {
14   this.respectiveClass = respectiveClass;
15  }
16
17  public Class<? extends BaseMessage> getRespectiveClass() {
18   return respectiveClass;
19  }
20
21 }
```

A.1.9 PerceptAction.java

```
package br.ufsc.vsschweitzer.thesis.messaging.messages.enums;

public enum PerceptAction {
 ADD,
 REMOVE;
}
```

A.1.10 ActMessage.java

```
package br.ufsc.vsschweitzer.thesis.messaging.messages;

import java.util.List;

import br.ufsc.vsschweitzer.thesis.messaging.messages.enums.
    MessageType;

public class ActMessage extends BaseMessage {

 String agent;
 String action;
 List<String> parameters;

 public ActMessage() {
  super(MessageType.ACT);
 }

 public ActMessage(String agent, String action, List<String>
     parameters) {
  super(MessageType.ACT);
  setAgent(agent);
  setAction(action);
  setParameters(parameters);
 }

 public String getAgent() {
  return agent;
 }

 private void setAgent(String agent) {
  this.agent = agent;
 }

 public String getAction() {
  return action;
 }

 private void setAction(String action) {
  this.action = action;
 }

 public List<String> getParameters() {
  return parameters;
 }

 private void setParameters(List<String> parameters) {
  this.parameters = parameters;
 }

}
```

A.1.11 ActResponseMessage.java

```
1  package br.ufsc.vsschweitzer.thesis.messaging.messages;
2
3  import java.util.List;
4
5  import br.ufsc.vsschweitzer.thesis.messaging.messages.enums.
       ActResponseStatus;
6  import br.ufsc.vsschweitzer.thesis.messaging.messages.enums.
       MessageType;
7
8  public class ActResponseMessage extends BaseMessage {
9
10  ActResponseStatus status;
11  List<Percept> percepts;
12
13  public ActResponseMessage() {
14   super(MessageType.ACT_RESPONSE);
15  }
16
17  public ActResponseMessage(ActResponseStatus status, List<Percept>
       percepts) {
18   super(MessageType.ACT_RESPONSE);
19   setStatus(status);
20   setPercepts(percepts);
21  }
22
23  public ActResponseStatus getStatus() {
24   return status;
25  }
26
27  private void setStatus(ActResponseStatus status) {
28   this.status = status;
29  }
30
31  public List<Percept> getPercepts() {
32   return percepts;
33  }
34
35  private void setPercepts(List<Percept> percepts) {
36   this.percepts = percepts;
37  }
38
39 }
```

A.1.12 BaseMessage.java

```
1  package br.ufsc.vsschweitzer.thesis.messaging.messages;
2
3  import br.ufsc.vsschweitzer.thesis.messaging.messages.enums.
       MessageType;
4
5  public abstract class BaseMessage {
6
7  MessageType type;
8
9  public BaseMessage(MessageType type) {
10  this.type = type;
```

```
 }

 public MessageType getType() {
  return type;
 }

 public void setType(MessageType type) {
  this.type = type;
 }

}
```

A.1.13 Percept.java

```
package br.ufsc.vsschweitzer.thesis.messaging.messages;

import java.util.List;
import java.util.stream.Collectors;

import br.ufsc.vsschweitzer.thesis.messaging.messages.enums.
    PerceptAction;
import jason.asSyntax.Literal;

public class Percept {

 String percept;
 List<String> perceptValues;
 PerceptAction action;

 public Percept() {}

 public Percept(String percept, List<String> perceptValues,
     PerceptAction action) {
  setPercept(percept);
  setPerceptValues(perceptValues);
  setAction(action);
 }

 public String getPercept() {
  return percept;
 }

 public void setPercept(String percept) {
  this.percept = percept;
 }

 public List<String> getPerceptValues() {
  return perceptValues;
 }

 public void setPerceptValues(List<String> perceptValues) {
  this.perceptValues = perceptValues;
 }

 public PerceptAction getAction() {
  return action;
 }

 public void setAction(PerceptAction action) {
  this.action = action;
```

```
 }

 public Literal asLiteral() {
  String literalToParse = getPercept();
  if (getPerceptValues() != null
    && getPerceptValues().size() > 0) {
   literalToParse += getPerceptValues().stream()
        .map(n -> String.valueOf(n))
        .collect(Collectors.joining(",", "(", ")"));
  }
  return Literal.parseLiteral(literalToParse);
 }

}
```

A.1.14 AgentMessageInterpreter.java

```
package br.ufsc.vsschweitzer.thesis.messaging;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;

import br.ufsc.vsschweitzer.thesis.messaging.messages.ActMessage;
import br.ufsc.vsschweitzer.thesis.messaging.messages.BaseMessage;
import br.ufsc.vsschweitzer.thesis.messaging.messages.enums.
    MessageType;
import jason.asSyntax.Structure;
import jason.asSyntax.Term;

public class AgentMessageInterpreter {

 private static final String TYPE_FIELD = "type";

 static ObjectMapper objectMapper = new ObjectMapper();

 public static String messageToJson(BaseMessage message) throws
    IOException {
  return objectMapper.writeValueAsString(message);
 }

 public static BaseMessage interpretJsonMessage(String message)
    throws IOException {
  ObjectNode genericNode = objectMapper.readValue(message,
    ObjectNode.class);
  MessageType type = MessageType.valueOf(genericNode.get(TYPE_FIELD)
    .asText());

  return objectMapper.convertValue(genericNode, type.
    getRespectiveClass());
 }

 public static ActMessage wrapAct(String agent, Structure action) {
  List<String> passedParameters = new ArrayList<String>();
  if (action.getTerms() != null) {
   for (Term term : action.getTerms()) {
    passedParameters.add(term.toString());
```

```
38     }
39    }
40    ActMessage actMessage = new ActMessage(agent, action.getFunctor(),
          passedParameters);
41
42    return actMessage;
43   }
44
45 }
```

A.1.15 Messenger.java

```
1  package br.ufsc.vsschweitzer.thesis.messaging;
2
3  import java.io.Closeable;
4  import java.io.IOException;
5  import java.io.InputStream;
6  import java.io.OutputStream;
7  import java.net.Socket;
8  import java.net.UnknownHostException;
9
10 import br.ufsc.vsschweitzer.thesis.exceptions.
       ConnectionNotOpenException;
11
12 /**
13  * Class responsible for connecting to a TCP server and sending
        messages.
14  */
15 public class Messenger implements Closeable {
16
17  boolean isOpen = false;
18  Socket socket;
19
20  String ip;
21  int port;
22
23  /**
24   * Constructor for a messenger that can stabilish connections via
         TCP.
25   *
26   * @param ip   ip address of the TCP server.
27   * @param port the port of the TCP server.
28   */
29  public Messenger(String ip, int port) {
30   this.ip = ip;
31   this.port = port;
32  }
33
34  /**
35   * Opens a connection with the server.
36   */
37  public void open() throws UnknownHostException, IOException {
38   socket = new Socket(ip, port);
39   isOpen = true;
40  }
41
42  /**
43   * Sends a message to the connected server.
44   *
45   * @param message the message to send.
```

```
46   * @throws ConnectionNotOpenException when the connection is not
        open.
47   * @throws IOException                if there was a problem
        sending the message
48   *                                    to the server.
49   */
50  public void send(String message) throws ConnectionNotOpenException,
        IOException {
51   if (isOpen) {
52    OutputStream output = socket.getOutputStream();
53    byte[] toSend = message.getBytes();
54    output.write(toSend);
55   } else {
56    throw new ConnectionNotOpenException();
57   }
58  }
59
60  /**
61   * Waits foe a message from the connected server.
62   *
63   * @return the message received.
64   * @throws ConnectionNotOpenException when the connection is not
        open.
65   * @throws IOException                if there was a problem
        sending the message
66   *                                    to the server.
67   */
68  public String listen() throws ConnectionNotOpenException,
        IOException {
69   if (isOpen) {
70    InputStream input = socket.getInputStream();
71
72    // Blocks thread until end of stream
73    byte[] receivedBytes = input.readAllBytes();
74    String receivedMsg = new String(receivedBytes);
75
76    return receivedMsg;
77   } else {
78    throw new ConnectionNotOpenException();
79   }
80  }
81
82  /**
83   * Closes the connection with the server.
84   */
85  public void close() {
86   if (isOpen) {
87    try {
88     socket.close();
89     isOpen = false;
90    } catch (IOException e) {
91    }
92   }
93  }
94
95 }
```

A.1.16 CommucationTest.java

```
1 package br.ufsc.vsschweitzer.thesis.tests;
```

```
2
3  import java.util.List;
4
5  import br.ufsc.vsschweitzer.thesis.environment.ExternalEnvironment;
6  import br.ufsc.vsschweitzer.thesis.messaging.messages.Percept;
7  import jason.asSyntax.Structure;
8  import jason.asSyntax.VarTerm;
9
10 public class CommucationTest {
11
12  static String ip = "127.0.0.1";
13  static int port = 10000;
14
15  static String testAgentName = "Ag1";
16
17  public static void main(String args[]) throws Exception {
18
19   ExternalEnvironment env = new ExternalEnvironment(ip, String.
       valueOf(port));
20
21   Structure action = new Structure("VoidPublic");
22   List<Percept> percepts = env.act(testAgentName, action);
23   printReceivedMessage(percepts);
24
25   action = new Structure("VoidPrivate");
26   percepts = env.act(testAgentName, action);
27   printReceivedMessage(percepts);
28
29   action = new Structure("List");
30   percepts = env.act(testAgentName, action);
31   printReceivedMessage(percepts);
32
33   action = new Structure("Parameterized");
34   action.addTerm(new VarTerm("A"));
35   action.addTerm(new VarTerm("B"));
36   action.addTerm(new VarTerm("3"));
37   percepts = env.act(testAgentName, action);
38   printReceivedMessage(percepts);
39
40   action = new Structure("TenSecondsWait");
41   percepts = env.act(testAgentName, action);
42   printReceivedMessage(percepts);
43
44   action = new Structure("MultipleTenSecondsWait");
45   percepts = env.act(testAgentName, action);
46   printReceivedMessage(percepts);
47  }
48
49  public static void printReceivedMessage(List<Percept> percepts) {
50   System.out.println("Percepts : {");
51   for (Percept percept : percepts) {
52    String toPrint = "    " + percept.getAction().toString() + " " +
       percept.getPercept() + ": [";
53    if (percept.getPerceptValues() != null) {
54     for (String term : percept.getPerceptValues()) {
55      toPrint += term + ", ";
56     }
57     toPrint = toPrint.substring(0, toPrint.length() - 2);
58    }
59    toPrint += "]";
60    System.out.println(toPrint);
```

```
  }
  System.out.println("}");
 }

}
```

A.1.17 CleaningAgent.jada

```
import java.util.Iterator;
import java.util.Queue;
import jason.asSemantics.Agent;
import jason.asSemantics.Event;
import jason.asSyntax.Trigger;

// This class complements the Cleaning Robots agent.
//  It's main change is giving higher priority to "forget" actions so that it doesn't goes to clean trash.
public class CleaningAgent extends Agent {

 static Trigger focus = Trigger.parseTrigger("+forget(_,_)");

 @Override
 public Event selectEvent(Queue<Event> events) {
  Iterator<Event> i = events.iterator();
  while (i.hasNext()) {
   Event e = i.next();
   if (isFocus(e)) {
    i.remove();
    return e;
   }
  }
  return super.selectEvent(events);
 }

 private boolean isFocus(Event e) {
  return e.getTrigger().getLiteral().getFunctor().equals(focus.getLiteral().getFunctor());
 }

}
```

A.1.18 SimpleEnvironment.java

```
import jason.asSyntax.*;
import jason.environment.*;

import java.util.List;
import java.util.concurrent.Executors;
import java.util.logging.*;
import br.ufsc.vsschweitzer.thesis.configuration.exceptions.MalformedConfigurationsException;
import br.ufsc.vsschweitzer.thesis.environment.ExternalEnvironment;
import br.ufsc.vsschweitzer.thesis.environment.exceptions.FailedActionException;
import br.ufsc.vsschweitzer.thesis.messaging.messages.Percept;

public class SimpleEnvironment extends Environment {

```

```
14  private Logger logger = Logger.getLogger("tcpagent.mas2j." +
        SimpleEnvironment.class.getName());
15
16  ExternalEnvironment extEnvironment;
17
18  /** Called before the MAS execution with the args informed in .
        mas2j */
19  /* The first arg is the ipAddress of the environment server
20   * The second arg is the port of the environment server
21   * The third arg is the number of threads to execute the agents.
22   *  It is recommended that there is a thread for every agent.
23   *  * Note: This value is limited to 1000 by JASON itself.
24   *
25   */
26  @Override
27  public void init(String[] args) {
28   try {
29    extEnvironment = new ExternalEnvironment(args[0], args[1]);
30       executor = Executors.newFixedThreadPool(Integer.parseInt(args
        [2]));
31   } catch (MalformedConfigurationsException e) {
32    e.printStackTrace();
33   }
34  }
35
36  @Override
37  public boolean executeAction(String agName, Structure action) {
38   logger.info("Agent " + agName + " executing: " + action);
39   try {
40    List<Percept> percepts = extEnvironment.act(agName, action);
41    updatePercepts(agName, percepts);
42    logger.info("[SUCCESS] " + agName + " executed " + action);
43    return true;
44   } catch (FailedActionException e) {
45    logger.info(" [FAILED] " + agName + " failed to execute " +
        action);
46    return false;
47   }
48  }
49
50  private void updatePercepts(String agName, List<Percept> percepts)
        {
51   for (Percept percept: percepts) {
52    switch (percept.getAction()) {
53     case ADD:
54      addPercept(agName, percept.asLiteral());
55      break;
56     case REMOVE:
57      removePerceptsByUnif(agName, percept.asLiteral());
58      break;
59    }
60   }
61  }
62
63  /** Called before the end of MAS execution */
64  @Override
65  public void stop() {
66   super.stop();
67  }
68 }
```

A.1.19 pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
     apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>br.ufsc.vsschweitzer</groupId>
 <artifactId>tcp-agent</artifactId>
 <version>0.1</version>
 <properties>
  <jackson.version>2.9.8</jackson.version>
 </properties>
 <build>
  <sourceDirectory>src/main/asl</sourceDirectory>
  <resources>
   <resource>
    <directory>src/main/asl</directory>
    <excludes>
     <exclude>**/*.java</exclude>
    </excludes>
   </resource>
   <resource>
    <directory>src/main/resources</directory>
    <excludes>
     <exclude>**/*.java</exclude>
    </excludes>
   </resource>
  </resources>
  <plugins>
   <plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.0</version>
    <configuration>
     <source>1.8</source>
     <target>1.8</target>
    </configuration>
   </plugin>
   <plugin>
    <artifactId>maven-dependency-plugin</artifactId>
    <executions>
     <execution>
      <phase>install</phase>
      <goals>
       <goal>copy-dependencies</goal>
      </goals>
      <configuration>
       <outputDirectory>${project.build.directory}/../lib</
    outputDirectory>
      </configuration>
     </execution>
    </executions>
   </plugin>
  </plugins>
 </build>
 <dependencies>
  <dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-annotations</artifactId>
   <version>${jackson.version}</version>
```

```
  </dependency>
  <dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-core</artifactId>
   <version>${jackson.version}</version>
  </dependency>
  <dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
   <version>${jackson.version}</version>
  </dependency>
 </dependencies>
</project>
```

A.1.20 tcpagent.mas2j

```
MAS tcpagent {

 infrastructure: Centralised

 environment: SimpleEnvironment("127.0.0.1", 10000, 10)

 agents:
  cleaner cleaner_agent agentClass CleaningAgent #10;

 aslSourcePath:
  "src/asl";
}
```

APÊNDICE B – Framework Unity

O código para a implementação do framework na parte que utiliza o Unity encontra-se no repositório público localizado no seguinte endereço:

- https://github.com/vsSchweitzer/UnityIntelligentAgents

Este repositório também contém os códigos do ambiente do exemplo *Cleaning Robots*.

B.1 CÓDIGO FONTE

Segue a transcrição do código deste repositório, onde cada seção é um arquivo:

B.1.1 AgentAction.cs

```
using System;

[AttributeUsage(AttributeTargets.Method)]
public class AgentAction : Attribute {}
```

B.1.2 AgentInvoker.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.Globalization;
using System.Reflection;
using UnityEngine;

public static class AgentInvoker {

 private static readonly CultureInfo dotSeparatedFloat = CultureInfo
     .CreateSpecificCulture("en-US");

 public static IEnumerator Invoke(IntelligentAgent agent, string
     action, List<string> parameters) {
  MethodInfo actionMethod = agent.GetType().GetMethod(action,
     BindingFlags.NonPublic | BindingFlags.Public | BindingFlags.
     Instance | BindingFlags.IgnoreCase);
  List<object> genericParameters = ConvertParameterList(actionMethod
     , parameters);
  if (actionMethod != null) {
   if (actionMethod.GetCustomAttribute(typeof(AgentAction), true) !=
      null) {
    if (actionMethod.ReturnType == typeof(void)) {
     actionMethod.Invoke(agent, genericParameters.ToArray());
     yield return new ActResponseMessage();
```

```
21      } else if (actionMethod.ReturnType == typeof(ActResponseMessage)
        ) {
22       yield return (ActResponseMessage)actionMethod.Invoke(agent,
        genericParameters.ToArray());
23      } else if (actionMethod.ReturnType == typeof(IEnumerator)) {
24       CoroutineWithData<ActResponseMessage> invocationCoroutine = new
          CoroutineWithData<ActResponseMessage>(agent, (IEnumerator)
        actionMethod.Invoke(agent, genericParameters.ToArray()));
25       yield return invocationCoroutine.coroutine;
26       ActResponseMessage responseMessage;
27       try {
28        responseMessage = invocationCoroutine.GetResult();
29        if (responseMessage == null) {
30         responseMessage = new ActResponseMessage();
31        }
32       } catch {
33        responseMessage = new ActResponseMessage();
34       }
35       yield return responseMessage;
36      } else {
37       Debug.LogError("Action \"" + action + "\" on agent \"" + agent.
        getAgentIdentifier() + "\" should return one of the following:
        \"ActResponseMessage\", \"IEnumerator\" or \"void\".");
38       throw new Exception();
39      }
40     } else {
41      Debug.LogError("Action \"" + action + "\" on agent \"" + agent.
        getAgentIdentifier() + "\" is not tagged with \"[AgentAction]\"
        attribute.");
42      throw new Exception();
43     }
44    } else {
45     Debug.LogError("Action \"" + action + "\" on agent \"" + agent.
        getAgentIdentifier() + "\" was not found.");
46     throw new Exception();
47    }
48   }
49
50   private static List<object> ConvertParameterList(MethodInfo method,
         List<string> parameters) {
51    List<object> genericParameterList = new List<object>();
52    ParameterInfo[] methodParameters = method.GetParameters();
53    if (methodParameters.Length == parameters.Count) {
54     int i = 0;
55     foreach (ParameterInfo paramInfo in methodParameters) {
56      try {
57       TypeConverter converter = TypeDescriptor.GetConverter(paramInfo
        .ParameterType);
58       genericParameterList.Add(converter.ConvertFromInvariantString(
        parameters[i]));
59      } catch {
60       Debug.LogError("Method " + method.Name + " has a parameter that
          can't be converted from a string: " + paramInfo.ParameterType +
          " " + paramInfo.Name);
61       throw new Exception();
62      }
63      i++;
64     }
65    } else if (methodParameters.Length > parameters.Count) {
66     Debug.Log("Method " + method.Name + " was called with less
        parameters than it has");
```

```
67     throw new Exception();
68    } else if (methodParameters.Length < parameters.Count) {
69     Debug.Log("Method " + method.Name + " was called with more
         parameters than it has");
70     throw new Exception();
71    }
72
73    return genericParameterList;
74   }
75
76  }
```

B.1.3 AgentMessageInterpreter.cs

```
1  using System.Collections.Generic;
2  using System.IO;
3  using UnityEngine;
4
5  public static class AgentMessageInterpreter {
6
7   public static BaseMessage Interpret(string message) {
8    BaseMessage baseMessage = JsonUtility.FromJson<BaseMessage>(
       message);
9    switch (baseMessage.GetTypeEnum()) {
10    case MessageType.ACT:
11     return JsonUtility.FromJson<ActMessage>(message);
12    default:
13     throw new IOException();
14    }
15   }
16
17   public static string MessageAsJson(BaseMessage message) {
18    return JsonUtility.ToJson(message);
19   }
20
21  }
```

B.1.4 ActResponseStatus.cs

```
1
2  public enum ActResponseStatus {
3   SUCCESS,
4   FAILURE,
5   ERROR
6  }
```

B.1.5 MessageType.cs

```
1
2  [System.Serializable]
3  public enum MessageType {
4   ACT,
5   ACT_RESPONSE
6  }
```

B.1.6 PerceptAction.cs

```
1
2 [System.Serializable]
3 public enum PerceptAction {
4  ADD,
5  REMOVE
6 }
```

B.1.7 ActMessage.cs

```
1 using System.Collections.Generic;
2
3 [System.Serializable]
4 public class ActMessage : BaseMessage {
5
6  public string agent;
7  public string action;
8  public List<string> parameters;
9
10  public ActMessage() {
11   SetType(MessageType.ACT);
12  }
13
14 }
```

B.1.8 ActResponseMessage.cs

```
1 using System;
2 using System.Collections.Generic;
3
4 [System.Serializable]
5 public class ActResponseMessage : BaseMessage {
6
7  public string status;
8  public List<Percept> percepts;
9
10  public ActResponseMessage() {
11   SetType(MessageType.ACT_RESPONSE);
12   SetStatus(ActResponseStatus.SUCCESS);
13   percepts = new List<Percept>();
14  }
15
16  public ActResponseMessage(List<Percept> percepts) {
17   SetType(MessageType.ACT_RESPONSE);
18   SetStatus(ActResponseStatus.SUCCESS);
19   this.percepts = percepts;
20  }
21
22  public ActResponseMessage(ActResponseStatus status) {
23   SetType(MessageType.ACT_RESPONSE);
24   SetStatus(status);
25   percepts = new List<Percept>();
26  }
27
28  public ActResponseStatus GetStatusEnum() {
```

```
29   return (ActResponseStatus)Enum.Parse(typeof(ActResponseStatus),
        status);
30  }
31
32  public void SetStatus(ActResponseStatus status) {
33   this.status = status.ToString().ToUpper();
34  }
35
36 }
```

B.1.9 BaseMessage.cs

```
1
2 using System;
3
4 [System.Serializable]
5 public class BaseMessage {
6
7  public string type;
8
9  public MessageType GetTypeEnum() {
10  return (MessageType)Enum.Parse(typeof(MessageType), type);
11 }
12
13 public void SetType(MessageType type) {
14  this.type = type.ToString().ToUpper();
15 }
16
17 }
```

B.1.10 Percept.cs

```
1  using System;
2  using System.Collections.Generic;
3
4  [System.Serializable]
5  public class Percept {
6
7   public string percept;
8   public List<string> perceptValues;
9   public string action;
10
11  public Percept(string percept) {
12   this.percept = percept;
13   perceptValues = new List<string>();
14   SetAction(PerceptAction.ADD);
15  }
16
17  public Percept(string percept, PerceptAction action) {
18   this.percept = percept;
19   perceptValues = new List<string>();
20   SetAction(action);
21  }
22
23  public Percept(string percept, List<string> perceptValues) {
24   this.percept = percept;
25   this.perceptValues = perceptValues;
26   SetAction(PerceptAction.ADD);
```

```
27  }
28
29  public Percept(string percept, List<string> perceptValues,
        PerceptAction action) {
30   this.percept = percept;
31   this.perceptValues = perceptValues;
32   SetAction(action);
33  }
34
35  public PerceptAction GetActionEnum() {
36   return (PerceptAction)Enum.Parse(typeof(PerceptAction), action);
37  }
38
39  public void SetAction(PerceptAction action) {
40   this.action = action.ToString().ToUpper();
41  }
42
43 }
```

B.1.11 AgentListener.cs

```
1  using System.Text;
2  using System.Net;
3  using System.Net.Sockets;
4  using UnityEngine;
5  using System.Collections;
6  using System;
7
8  public class AgentListener : MonoBehaviour {
9
10  #region Singleton
11  private static AgentListener instance;
12
13  private void SingletonCheck() {
14   if (instance != null && instance != this) {
15    Destroy(this);
16    Debug.LogError("There were multiple AgentListeners in the scene
        .");
17   } else {
18    instance = this;
19   }
20  }
21
22  public static AgentListener Instance() {
23   return instance;
24  }
25  #endregion
26
27  void Awake() {
28   SingletonCheck();
29  }
30
31  private string ipAddress = "127.0.0.1";
32  public int port = 10000;
33
34  private TcpListener tcpListener;
35  private byte[] buffer = new byte[2048];
36
37  Func<string, IEnumerator> messageReceivedAction;
38
```

```
39  void Start() {
40   tcpListener = new TcpListener(IPAddress.Parse(ipAddress), port);
41   tcpListener.Start();
42  }
43
44  void Update() {
45   if (tcpListener.Pending()) {
46    StartCoroutine(ProcessMessage());
47   }
48  }
49
50  IEnumerator ProcessMessage() {
51   using (TcpClient connectedTcpClient = tcpListener.AcceptTcpClient
        ()) {
52    using (NetworkStream stream = connectedTcpClient.GetStream()) {
53     while (!stream.DataAvailable) {
54      yield return null;
55     }
56
57     string clientMessage = "";
58     while (stream.DataAvailable) {
59      int length = stream.Read(buffer, 0, buffer.Length);
60      var incommingData = new byte[length];
61      Array.Copy(buffer, 0, incommingData, 0, length);
62      clientMessage += Encoding.ASCII.GetString(incommingData);
63     }
64
65     if (messageReceivedAction != null) {
66      CoroutineWithData<string> messageProcessorCoroutine = new
        CoroutineWithData<string>(this, messageReceivedAction(
        clientMessage));
67      yield return messageProcessorCoroutine.coroutine;
68
69      string response = messageProcessorCoroutine.GetResult();
70      byte[] data = Encoding.ASCII.GetBytes(response);
71      stream.Write(data, 0, data.Length);
72     } else {
73      Debug.LogError("A message was received but there wasn't an
        action assigned to process it.");
74     }
75    }
76   }
77   yield return null;
78  }
79
80  public void SetMessageReceivedAction(Func<string, IEnumerator>
        actionToExecute) {
81   messageReceivedAction = actionToExecute;
82  }
83
84 }
```

B.1.12 AgentManager.cs

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class AgentManager : MonoBehaviour {
6
```

```
7  private Dictionary<string, IntelligentAgent> agentsInScene;
8  private AgentListener myAgentListener;
9
10 void Start() {
11  PopulateAgentDictionary();
12  SetupListener();
13 }
14
15 void PopulateAgentDictionary() {
16  agentsInScene = new Dictionary<string, IntelligentAgent>();
17  IntelligentAgent[] foundAgents = FindObjectsOfType<
       IntelligentAgent>();
18  foreach (IntelligentAgent intelligentAgent in foundAgents) {
19   string identifier = intelligentAgent.getAgentIdentifier();
20   try {
21    agentsInScene.Add(identifier, intelligentAgent);
22   } catch (System.ArgumentException) {
23    Debug.LogError("There are multiple agents with the same
       identifier: " + identifier);
24   }
25  }
26 }
27
28 void SetupListener() {
29  myAgentListener = FindObjectOfType<AgentListener>();
30  myAgentListener.SetMessageReceivedAction(MessageReceivedCallback);
31 }
32
33 IntelligentAgent IdentifyAgent(string identifier) {
34  return agentsInScene[identifier];
35 }
36
37 public IEnumerator MessageReceivedCallback(string message) {
38  BaseMessage baseMessage = AgentMessageInterpreter.Interpret(
       message);
39  switch (baseMessage.GetTypeEnum()) {
40   case MessageType.ACT:
41    ActMessage actMessage = (ActMessage) baseMessage;
42    IntelligentAgent agent = IdentifyAgent(actMessage.agent);
43    CoroutineWithData<string> methodCoroutine = new
       CoroutineWithData<string>(this, ExecuteAction(agent, actMessage.
       action, actMessage.parameters));
44    yield return methodCoroutine.coroutine;
45    yield return methodCoroutine.GetResult();
46    break;
47   default:
48    // Currently there are only ACT messages
49    yield return null;
50    break;
51  }
52 }
53
54 IEnumerator ExecuteAction(IntelligentAgent agent, string action,
       List<string> parameters) {
55  CoroutineWithData<ActResponseMessage> invokerCoroutine = new
       CoroutineWithData<ActResponseMessage>(this, AgentInvoker.Invoke(
       agent, action, parameters));
56  yield return invokerCoroutine.coroutine;
57  ActResponseMessage responseMessage = invokerCoroutine.GetResult();
58
59  string responseJsonMessage = AgentMessageInterpreter.MessageAsJson
```

```
    (responseMessage);
  yield return responseJsonMessage;
 }

}
```

B.1.13 IntelligentAgent.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class IntelligentAgent : MonoBehaviour {

 [SerializeField]
 protected string agentIdentifier = "Agent-0";

 public string getAgentIdentifier() {
  return agentIdentifier;
 }

}
```

B.1.14 AgentModelController.cs

```
using System;
using System.Collections;
using UnityEngine;

public class AgentModelController : MonoBehaviour {

 public Transform armsPivot;
 public Transform rArmPivot;
 public Transform lArmPivot;
 public Transform heldItemPivot;

 public float startFrontAngle = 0f;
 public float finalFrontAngle = 15f;

 public float normalSideAngle = 0f;
 public float openSideAngle = 15f;

 public event Action PickupEvent;

 Vector3 originalRotationArms;
 Vector3 originalRotationLArm;
 Vector3 originalRotationRArm;

 void Start() {
  originalRotationArms = armsPivot.localEulerAngles;
  originalRotationLArm = lArmPivot.localEulerAngles;
  originalRotationRArm = rArmPivot.localEulerAngles;
 }

 public IEnumerator AnimatePickup(float animationDuration) {
  float downFraction = 0.6f; // What fraction of the duration is
     dedicated to making the down animation;

```

```
33    float frontSpeedDown = Mathf.Abs(normalSideAngle - finalFrontAngle
        ) / (animationDuration * downFraction);
34    float frontSpeedUp = -(Mathf.Abs(normalSideAngle - finalFrontAngle
        ) / (animationDuration * (1- downFraction) ));
35    float sideSpeed = Mathf.Abs(normalSideAngle - openSideAngle) / (
        animationDuration * downFraction);
36
37    float startTime = Time.time;
38    float finishTime = startTime + animationDuration;
39    float elapsedTime = 0f;
40    bool eventTriggered = false;
41    while (Time.time < finishTime) {
42     if (elapsedTime < animationDuration * downFraction) {
43      armsPivot.Rotate(Vector3.right, frontSpeedDown * Time.deltaTime,
         Space.Self);
44     } else {
45      if (!eventTriggered) {
46       PickupEvent?.Invoke();
47       eventTriggered = true;
48      }
49      armsPivot.Rotate(Vector3.right, frontSpeedUp * Time.deltaTime,
        Space.Self);
50     }
51
52     if (elapsedTime < animationDuration * downFraction / 2) {
53      rArmPivot.Rotate(Vector3.up, sideSpeed * Time.deltaTime, Space.
        Self);
54      lArmPivot.Rotate(Vector3.up, -sideSpeed * Time.deltaTime, Space.
        Self);
55     } else if (elapsedTime >= animationDuration * downFraction / 2
56        && elapsedTime < animationDuration * downFraction) {
57      rArmPivot.Rotate(Vector3.up, -sideSpeed * Time.deltaTime, Space.
        Self);
58      lArmPivot.Rotate(Vector3.up, sideSpeed * Time.deltaTime, Space.
        Self);
59     }
60
61     yield return null;
62     elapsedTime = Time.time - startTime;
63    }
64    returnArmsPosition();
65    yield return null;
66   }
67
68   private void returnArmsPosition() {
69    armsPivot.localEulerAngles = originalRotationArms;
70    lArmPivot.localEulerAngles = originalRotationLArm;
71    rArmPivot.localEulerAngles = originalRotationRArm;
72   }
73
74   public void SetToHand(GameObject trash) {
75    trash.transform.SetParent(heldItemPivot, true);
76   }
77  }
```

B.1.15 AgentSpawner.cs

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
```

```
4
5  public class AgentSpawner : MonoBehaviour
6  {
7   [SerializeField]
8   string baseName;
9
10  [SerializeField]
11  GameObject agentPrefab;
12
13  [SerializeField]
14  Transform trashCan;
15
16  [SerializeField]
17  int agentCount;
18
19  [SerializeField]
20  float spawnRange;
21
22  void Start() {
23   for (int i = 0; i < agentCount; i++) {
24    float x = Random.Range(-spawnRange, spawnRange);
25    float z = Random.Range(-spawnRange, spawnRange);
26    GameObject spawnedTrash = Instantiate(agentPrefab, new Vector3(x,
        0, z), Quaternion.identity, transform);
27    spawnedTrash.name = "Cleaner";
28    CleaningRobotAgent agent = spawnedTrash.GetComponent<
        CleaningRobotAgent>();
29    agent.SetTrashCan(trashCan);
30    agent.SetIdentifier(baseName + (i+1));
31   }
32  }
33 }
```

B.1.16 CleaningRobotAgent.cs

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Globalization;
5  using UnityEngine;
6
7  public class CleaningRobotAgent : IntelligentAgent {
8
9   private static readonly string agentTrashPercept = "trash";
10  private static readonly string agentTrashCanPercept = "trashCan";
11  private static readonly string agentCarryingPercept = "
        carryingTrash";
12
13  private static readonly CultureInfo dotSeparatedFloat = CultureInfo
        .CreateSpecificCulture("en-US");
14
15  [Space(10)]
16
17  [SerializeField]
18  float moveSpeed = 5f;
19  [SerializeField]
20  float moveStopDistance = 1f;
21
22  [SerializeField]
23  float scanDuration = 2f;
```

```
24 [SerializeField]
25 float scanRadius = 10f;
26
27 [SerializeField]
28 float rotateSpeed = 250f;
29
30 [SerializeField]
31 float snapThreshold = 0.01f;
32
33 [SerializeField]
34 float pickupDistance = 1.2f;
35 [SerializeField]
36 float pickupCheckRadius = 0.01f;
37
38 [SerializeField]
39 float pickupDuration = 1f;
40
41 GameObject heldTrash;
42
43 [SerializeField]
44 GameObject scanPrefab;
45
46 [SerializeField]
47 Transform trashCan;
48
49 [SerializeField]
50 Material scannedMaterial;
51
52 AgentModelController myModel;
53
54 void Start() {
55  myModel = GetComponentInChildren<AgentModelController>();
56 }
57
58 public void SetTrashCan(Transform traschCanTransform) {
59  trashCan = traschCanTransform;
60 }
61
62 public void SetIdentifier(string newIdentifier) {
63  agentIdentifier = newIdentifier;
64 }
65
66 public IEnumerator TurnTowards(Vector3 targetPosition) {
67  Vector3 targetDirection = targetPosition - transform.position;
68  float angleToTarget = Vector3.SignedAngle(transform.forward,
       targetDirection, Vector3.up);
69  while (Math.Abs(angleToTarget) > snapThreshold) {
70   float amountToRotate = rotateSpeed * Time.deltaTime;
71   if (angleToTarget > 0) {
72    amountToRotate = Math.Min(amountToRotate, angleToTarget);
73   } else {
74    amountToRotate = Math.Max(-amountToRotate, angleToTarget);
75   }
76   transform.Rotate(Vector3.up, amountToRotate, Space.Self);
77   yield return null;
78   angleToTarget = Vector3.SignedAngle(transform.forward,
       targetDirection, Vector3.up);
79  }
80  transform.Rotate(Vector3.up, angleToTarget, Space.Self);
81 }
82
```

```
[AgentAction]
public IEnumerator ScanSurroundings() {
 List<Percept> trashPercepts = new List<Percept>();

 GameObject instatiatedEffect = Instantiate(scanPrefab, transform.
    position, Quaternion.identity, transform);
 yield return instatiatedEffect.GetComponent<ScannerField>().
    ApplyEffect(scanDuration, scanRadius);
 Destroy(instatiatedEffect);

 Collider[] trashFound = Physics.OverlapSphere(transform.position,
    scanRadius, LayerMask.GetMask("Trash"));
 foreach (Collider trashCollider in trashFound) {
  trashCollider.gameObject.GetComponentInChildren<MeshRenderer>().
    material = scannedMaterial;
  string x = trashCollider.transform.position.x.ToString(
    dotSeparatedFloat);
  string z = trashCollider.transform.position.z.ToString(
    dotSeparatedFloat);
  Percept trashPercept = new Percept(agentTrashPercept, new List<
    string> { x, z });

  trashPercepts.Add(trashPercept);
 }

 yield return new ActResponseMessage(
  trashPercepts
 );

}

[AgentAction]
public IEnumerator MoveTo(float x, float z) {
 Vector3 destination = new Vector3(x, 0, z);
 yield return TurnTowards(destination);

 while (Vector3.Distance(transform.position, destination) >
    moveStopDistance) {
  transform.position = Vector3.MoveTowards(transform.position,
    destination, moveSpeed * Time.deltaTime);
  yield return null;
 }
}

[AgentAction]
public IEnumerator PickupTrashAt(float x, float z) {
 Vector3 trashPosition = new Vector3(x, 0, z);
 if (Vector3.Distance(transform.position, trashPosition) <=
    pickupDistance) {
  Collider[] trashFound = Physics.OverlapSphere(trashPosition,
    pickupCheckRadius, LayerMask.GetMask("Trash"));
  if (trashFound.Length > 0) {
   float closestTrashDistance = Mathf.Infinity;
   GameObject trash = trashFound[0].gameObject;
   foreach (Collider trashCollider in trashFound) {
    float distanceToCollider = Vector3.Distance(trashPosition,
    trashCollider.transform.position);
    if (distanceToCollider < closestTrashDistance) {
     trash = trashCollider.gameObject;
     closestTrashDistance = distanceToCollider;
    }
```

```
132     }
133     trash.layer = 0;
134     Action OnPickupAction = () => {
135      myModel.SetToHand(trash);
136      heldTrash = trash;
137     };
138     myModel.PickupEvent += OnPickupAction;
139     yield return myModel.AnimatePickup(pickupDuration);
140     myModel.PickupEvent -= OnPickupAction;
141     yield return new ActResponseMessage(
142      new List<Percept> {
143       new Percept(agentTrashPercept, new List<string> { x.ToString(
        dotSeparatedFloat), z.ToString(dotSeparatedFloat) },
        PerceptAction.REMOVE),
144       new Percept(agentCarryingPercept)
145      }
146     );
147    } else {
148     Debug.Log("There was no trash in that position");
149     yield return new ActResponseMessage(
150      new List<Percept> {
151       new Percept(agentTrashPercept, new List<string> { x.ToString(
        dotSeparatedFloat), z.ToString(dotSeparatedFloat) },
        PerceptAction.REMOVE)
152      }
153     );
154    }
155   } else {
156    Debug.Log("Can't pickup trash, too far away");
157   }
158  }
159
160  [AgentAction]
161  public IEnumerator DisposeTrash() {
162   if (Vector3.Distance(transform.position, trashCan.position) <=
       pickupDistance
163    && heldTrash != null) {
164    Destroy(heldTrash);
165    yield return new ActResponseMessage(
166     new List<Percept> {
167      new Percept(agentCarryingPercept, PerceptAction.REMOVE)
168     }
169    );
170   } else {
171    Debug.Log("Trash can too far away");
172   }
173  }
174
175  [AgentAction]
176  public ActResponseMessage LocateTrashCan() {
177   float trashCanX = trashCan.position.x;
178   float trashCanZ = trashCan.position.z;
179   return new ActResponseMessage(
180    new List<Percept> {
181     new Percept(agentTrashCanPercept, new List<string> { trashCanX.
        ToString(dotSeparatedFloat), trashCanZ.ToString(
        dotSeparatedFloat) })
182    }
183   );
184  }
185
```

```
}
```

B.1.17 DebugShortcuts.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class DebugShortcuts : MonoBehaviour {

 [SerializeField]
 TrashSpawner spawner;

    void Update() {
  if (Input.GetKeyDown(KeyCode.Q)) { // Start trash spawning
   spawner.spawnEnabled = !spawner.spawnEnabled;
  } else if (Input.GetKeyDown(KeyCode.W)) { // Reload scene
   Scene scene = SceneManager.GetActiveScene();
   SceneManager.LoadScene(scene.name);
  } else if (Input.GetKeyDown(KeyCode.Escape)) { // Exit program
   Application.Quit();
  }

 }
}
```

B.1.18 ScannerField.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ScannerField : MonoBehaviour {

 float diameter = 0f;
 public float initialAlpha = 1f;
 public float finalAlpha = 0f;

 public IEnumerator ApplyEffect(float duration, float radius) {
  MeshRenderer myRenderer = GetComponent<MeshRenderer>();
  float startTime = Time.time;
  float finalDiamater = 2 * radius;
  float speed = finalDiamater / duration;
  Color originalColor = myRenderer.material.color;
  while (diameter <= finalDiamater) {
   float elapsedTime = Time.time - startTime;
   float alphaToApply = Mathf.Lerp(initialAlpha, finalAlpha,
     elapsedTime / duration);
   myRenderer.material.color = new Color(originalColor.r,
     originalColor.g, originalColor.b, alphaToApply);
   diameter += speed * Time.deltaTime;
   transform.localScale = Vector3.one * diameter;
   yield return null;
  }
 }
}
```

B.1.19 TrashSpawner.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TrashSpawner : MonoBehaviour {

 public GameObject trashPrefab;
 public float secondsBetweenSpawns;
 public int trashPerSpawn;

 float lastSpawn = Mathf.NegativeInfinity;

 public float spawnRange;

 public bool spawnEnabled = false;

 void Update() {
  if (spawnEnabled && lastSpawn + secondsBetweenSpawns < Time.time)
     {
   lastSpawn = Time.time;
   for (int i = 0; i < trashPerSpawn; i++) {
    float x = Random.Range(-spawnRange, spawnRange);
    float z = Random.Range(-spawnRange, spawnRange);
    GameObject spawnedTrash = Instantiate(trashPrefab, new Vector3(x
     , 0, z), Quaternion.identity, transform);
    spawnedTrash.name = "Trash";
   }
  }
 }
}
```

B.1.20 CoroutineWithData.cs

```
using System.Collections;
using UnityEngine;

// Code by Ted-Bigham
// from: https://answers.unity.com/questions/24640/how-do-i-return-a
    -value-from-a-coroutine.html
// last accessed in: 2019-05-19
public class CoroutineWithData<T> {

 public Coroutine coroutine { get; private set; }
 private object result;
 private IEnumerator target;

 public CoroutineWithData(MonoBehaviour owner, IEnumerator target) {
  this.target = target;
  coroutine = owner.StartCoroutine(Run());
 }

 private IEnumerator Run() {
  while (target.MoveNext()) {
   result = target.Current;
   yield return result;
  }
```

```
 }

 public T GetResult() {
  return (T)result;
 }
}
```

APÊNDICE C – Artigo

Modelo para utilização de agentes inteligentes em ambientes virtuais multiagente

Vinicius Steffani Schweitzer[1], **Elder Rizzon Santos**[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 – 88.040-900 – Santa Catarina – SC – Brazil

vinicius.ss@grad.ufsc.br, elder.santos@ufsc.br

Abstract. *Intelligent systems are becoming more relevant than ever, given the need to solve more complex problems. One way to model this type os systems is using intelligent agents. Many tools for intelligent agents simulate those agents in their own environment, bound to the agent tool. These environments usually are either too complex or too restrictive. There are other applications that simulate virtual environments with high fidelity, the game engines. The model proposed in this thesis estabilishes an implementation standard that allows the usage of intelligent agents in environments simulated in game engines.*

Resumo. *Sistemas inteligentes estão tornando-se cada vez mais relevantes, tendo em vista a necessidade de resolução de problemas mais complexos. Uma das formas de modelar este tipo de sistema é através do uso de agentes inteligentes. Diversas ferramentas de agentes inteligentes simulam estes agentes em ambientes próprios, atrelados à ferramenta. Estes ambientes costumam ser complexos ou restritivos demais. Existem programas que simulam ambientes virtuais com alta fidelidade, as game engines. O modelo proposto neste trabalho estabelece um padrão de implementação que permite a utilização de agentes inteligentes em ambientes simulados em game engines.*

1. Introdução

O estudo de agentes inteligentes dentro da área de Inteligência Artificial possibilita a simulação de raciocínio complexo através de conjunturas simples, permitindo a simulação de modelos complexos sem grande esforço de implementação.

Uma das formas de modelar o pensamento de um agente é a de *belief*, *desire*, *intention* que modela símbolos mentais em três principais categorias: crenças, desejos e intenções. A vantagem dessa abordagem é a facilidade de se descrever comportamentos mais complexos com poucas regras, sem a necessidade de explicitamente descrever grandes árvores de decisão, que descrevem os comportamentos a serem tomados verificando diversas condições explicitamente.

Agentes inteligentes por definição se situam em um ambiente. As linguagens de agente atualmente disponíveis permitem a criação de ambientes em aplicações externas, permitindo uma grande flexibilidade quanto a sua implementação. Estes ambientes podem ser classificados em várias categorias, das quais algumas podem ser encontradas em motores de jogos. Implementá-los neste tipo de ferramenta pode ser útil para se obter

uma melhor visualização dos agentes no ambiente, mas também é útil pois elas oferecem grande flexibilização em relação às diversas categorias de ambiente.

Diante disso, este trabalho se motiva a desenvolver e implementar um modelo que integra linguagens de agentes inteligentes com um ambiente simulado em um motor de jogos.

2. Fundamentação Teórica

2.1. Agentes Inteligentes

Esta área da Inteligência Artificial estuda o conceito de agentes inteligentes que, de acordo com [Wooldridge 2002], são sistemas computacionais que apresentam as seguintes capacidades:

- Reatividade:
  Capacidade de observar o ambiente à sua volta e tomar ações de acordo com o que foi observado, com o propósito de atingir seus objetivos;
- Proatividade:
  Capacidade de ter objetivos e de tomar iniciativa para atingi-los;
- Habilidade social:
  Capacidade de interagir com outros agentes para atingir seus objetivos.

Além disso, agentes se situam em um **ambiente** [Wooldridge 2002, Russell and Norvig 2010, Dorri et al. 2018]. Caso existam múltiplos agentes em um mesmo ambiente, este é chamado de ambiente multiagente [Russell and Norvig 2010].

2.2. Simulações Computacionais

Simulações têm o propósito de replicar e executar modelos em um ambiente controlado, para se extrair mais informações do mesmo [Kelton et al. 1997]. O método de simulações computacionais avalia a realização de simulações em computadores. A vantagem de usar simulações no lugar de modelos matemáticos é a sua capacidade de representar sistemas complexos.

É possível traçar um paralelo entre ambientes de agentes e simulações, este paralelo é onde se encontra o domínio deste trabalho.

2.3. Ferramentas

2.3.1. JASON

A ferramenta escolhida como sistema de agentes para realizar os testes foi o JASON. As principais características que resultaram em sua escolha são a simplicidade, facilidade de aprendizado, bom desempenho, suporte a sistemas multiagente, documentação extensa por [Bordini et al. 2007], e principalmente seu foco em aplicações de propósito geral. A ferramenta apresenta outro ponto positivo, que é a modularidade de componentes, de forma que é possível criar não apenas os agentes como também os componentes que realizam os processos da ferramenta, como por exemplo o escalonador de ações ou as classes que administram os ciclos de raciocínio dos múltiplos agentes sendo executados.

2.4. Unity

Diversas ferramentas existem para modelar e implementar simulações. Dentre elas estão os motores de jogos (*game engines*), ferramentas utilizadas para o desenvolvimento de jogos de computador que, segundo [Gregory 2014], também podem ser consideradas como simulações de tempo-real, dinâmicas e interativas. Estes sistemas executam através de um *game loop*, que é um ciclo que se repete durante toda a execução da simulação e que executa algumas operações a cada rodada do ciclo.

Um motor de jogos muito utilizado atualmente é o **Unity** [Unity 2019], uma ferramenta gratuita, com suporte a criação de jogos tanto 2D quanto 3D, além de possuir suporte a múltiplas plataformas. Esta é a ferramenta que será utilizada para atuar como ambiente virtual. A ferramenta apresenta alguns conceitos que são utilizados neste trabalho: corotinas e reflexão.

3. Modelagem

De forma resumida, o problema proposto por este trabalho consiste em elaborar um modelo que pode ser implementado em sistemas de agentes inteligentes e em ambientes virtuais de forma que qualquer agente inteligente que utilize este modelo possa atuar em qualquer ambiente que também utilize-o. A imagem a seguir ilustra o modelo como um todo:

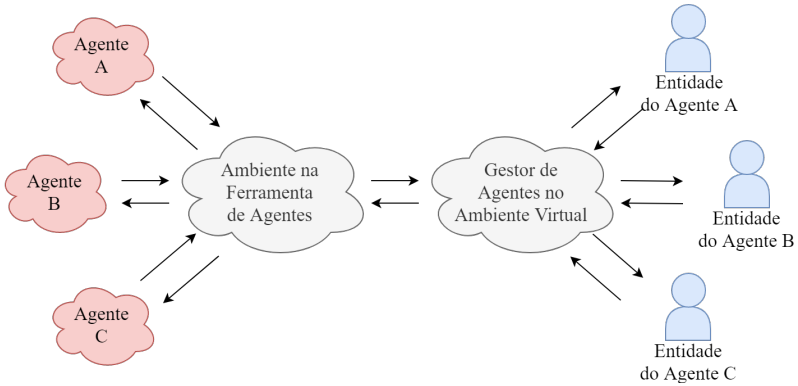

Figure 1. Visão geral do sistema.

Nesta imagem é possível observar como existem múltiplos agentes atuando em um ambiente da ferramenta de agentes que implementa o modelo. Este ambiente por sua vez se comunica com o ambiente virtual, que também implementa o modelo. Como existem múltiplos agentes atuando na ferramenta de agentes, múltiplas entidades que representam os agentes existem no ambiente virtual. Todos os fluxos são de duas vias, pois o agente informa as ações que deseja realizar e o ambiente retorna informações resultantes da realização desta ação.

Com esta visão geral em mente, é preciso formalizar o modelo proposto. As subseções a seguir apresentam diversos problemas relacionados ao modelo, fazendo uma

pequena discussão sobre as formas de abordá-lo e então tomando a decisão de qual abordagem é utilizada no modelo.

3.1. Responsabilidades do Sistema de Agentes

Analisando o sistema de agentes em relação ao ambiente, percebe-se que ele deve ser capaz de: obter percepções sobre o ambiente e de atuar sobre ele.

3.1.1. Percepções

Para que o agente adquira percepções sobre o ambiente deve ser possível que o ambiente lhe envie estas informações. Duas abordagens são possíveis para este problema:

- O ambiente envia percepções a cada intervalo de tempo;
- O agente por decisão própria, em seu ciclo de raciocínio, consulta o ambiente por um determinado sensor.

A primeira abordagem é conhecida como **percepção passiva**, enquanto que a outra é a **percepção ativa**.

Na percepção passiva o agente não precisa utilizar seus sensores conscientemente, apenas reagir às percepções que recebe, facilitando sua implementação. Entretanto um problema encontrado neste tipo de percepção é a sobrecarga de percepções causada pela redundância de dados, uma vez que um agente pode estar recebendo diversas vezes a mesma percepção sem que tenha ocorrido alterações no ambiente.

Já na percepção ativa há mais controle no que é percebido no ambiente, uma vez que o uso dos sensores está explicita na programação de um agente. Um ponto relevante de se levantar sobre este tipo de percepção é que é possível emular a percepção passiva fazendo com que o agente consulte por conta própria os sensores a cada intervalo de tempo.

3.1.2. Ações

Para atuar no ambiente o agente deve possuir uma forma de informar-lhe que deseja realizar tal ação e com quais parâmetros deseja realizá-la. Um exemplo seria “mover até 10, 12”, onde “mover” é a ação e “10, 12” são coordenadas passadas como parâmetros da ação de mover-se.

A sequencialidade das ações é um ponto que deve ser levado em consideração na elaboração do modelo, uma vez que um agente pode querer realizar múltiplas ações, uma ao término da outra. Isso então adiciona um requisito na implementação do sistema de agentes, de que seja possível que os agentes aguardem o término de uma ação para prosseguir com seu plano.

3.2. Responsabilidades do Ambiente Virtual

Como diversos agentes atuam em um mesmo ambiente virtual, este deve ser capaz de administrar as diversas entidades que representam estes agentes, de forma que ao receber requisições de percepção ou de atuação de um agente ele saiba distinguir qual entidade

representa o agente. As demais responsabilidades do ambiente virtual se espelham nas do sistema de agentes, ou seja, ele deve ser capaz de informar o agente de suas percepções e deve ser capaz de realizar as ações que o agente requisitar.

3.2.1. Percepções

O ambiente deve ser capaz de enviar para o agente as percepções e suas propriedades, como por exemplo "gosta(musica)", que informa da percepção de que o agente gosta de algo e que apresenta a propriedade "musica".

Dependendo se a percepção for passiva ou não o ambiente também deve ser capaz de enviar por conta própria as percepções para os agentes. A seção 3.4 elabora sobre qual tipo de percepção é esperado do modelo final.

3.2.2. Ações

Para que um agente realize ações no ambiente, este deve ser capaz manifestá-las e retornar para o agente as novas percepções resultantes de suas ações.

Um problema que surge durante execução de ações é a concorrência para utilizar determinados recursos. Uma demonstração desse problema seria um agente que deseja realizar estas duas ações:

- Correr para uma direção;
- Chutar uma bola.

Estas ações são concorrentes, uma vez que ambas necessitam de um mesmo recurso, as pernas do agente neste exemplo. Para este problema existem diversas soluções possíveis das quais algumas são:

- Executar o código de ambas ao mesmo tempo, mesmo que isso cause um comportamento incorreto;
- Executar uma delas enquanto a outra fica em uma fila de espera;
- Cancelar uma e executar a outra.

Mas isto nem sempre é um problema, uma vez que duas ações podem ser completamente independentes uma da outra. Como por exemplo olhar para uma direção e mover os braços, duas ações que não interferem-se.

Embora a responsabilidade de definir quais ações executar seja do raciocínio do agente, é possível que ele decida executar múltiplas ações concorrentes que o modelo pode tratar caso necessário. O comportamento do modelo diante deste problema é especificado no tópico que explicita todos os pontos do modelo final, na subseção 3.4.

3.3. Comunicação

O sistema de agentes e o ambiente virtual são, à principio, aplicações distintas. Com este ponto em mente é necessário estabelecer uma forma de comunicação entre as duas partes para que os fluxos de execução possam acontecer. Para isso podemos utilizar troca de mensagens tanto através do protocolo UDP quanto TCP. A principal diferença entre as duas abordagens é a necessidade de que se estabeleça uma conexão entre as partes no protocolo TCP, o que não ocorre no protocolo UDP.

3.4. Síntese do Modelo

Com os pontos levantados nas subseções anteriores, o modelo final é estabelecido com as seguintes características e motivações:

3.4.1. Comunicação TCP

Este protocolo de comunicação foi escolhido por apresentar duas características principais:

- Retransmissão e verificação de erros:
  Permite que a comunicação com o ambiente ocorra com maior garantia, de forma que a simulação não é afetada por eventuais problemas na rede.
- Baseada em conexão:
  Permite que o fluxo de comunicação seja simplificado pois, após conectados, ambos os lados podem facilmente enviar mensagens e esperar respostas da outra aplicação. Graças à isso o problema da sequencialidade de ações é resolvido facilmente como uma espera de resposta.

3.4.2. Modelo Cliente-servidor

Devido ao uso do protocolo TCP, deve-se estabelecer a forma que o modelo cliente-servidor deve ser utilizado. Tendo em vista que existe um ambiente que administra múltiplos agentes que se comunicam com ele, foi determinado que o ambiente atuará como servidor e os diversos agentes atuarão como clientes que se conectam nele para realizar suas ações e percepções.

3.4.3. Percepção Ativa

O modelo proposto por este trabalho utiliza percepção ativa por se adequar melhor no fluxo de troca da mensagens TCP, além de ser uma abordagem com melhor acoplamento, uma vez que não é papel do ambiente informar o agente de suas percepções, principalmente quando se deseja representar um agente realista, onde não faria sentido o "mundo real" enviar as percepções para o agente, o que não é o caso do modelo deste trabalho.

3.4.4. Administração de Agentes

Como o modelo suporta múltiplos agentes é esperado que em ambas as partes, tanto no sistema de agentes quanto no ambiente, seja possível distinguir um agente inteligente através de um identificador e que exista uma forma de transformar esse identificador em uma representação textual (*string*) para ser enviada nas mensagens entre as partes. Além disso o ambiente deve estar ciente das diversas entidades que representam agentes e deve ser capaz de traduzir de um identificador de um agente para uma entidade do ambiente que representa este agente. Este identificador é utilizado pelo ambiente ao receber uma mensagem para saber qual dos diversos agentes está enviando a requisição para ele.

3.4.5. Fluxo de Mensagens

Devido a semelhança entre a requisição de uma ação e de uma percepção, existe apenas um formato de mensagem para ambas. A principal razão desta semelhança é o fato de que "perceber um sensor" pode ser considerado uma ação como qualquer outra. A forma que o fluxo de mensagens ocorre pode ser resumido em mensagens para o ambiente no formato "faça X com parâmetros Y" e a resposta do ambiente pode ser resumida em "X ocorreu com FALHA/SUCESSO, percebendo-se Z".

A especificação da troca de mensagens é simples, existem dois tipos de mensagens: "mensagem de ação" e "mensagem de resposta de ação". No momento em que o agente deseja realizar uma ação ele envia a mensagem de ação para o ambiente, que executa a ação e retorna a mensagem de resposta para que o sistema de agentes saiba o resultado da ação. O formato escolhido para codificar as mensagens é o formato JSON devido a sua crescente popularidade e facilidade de implementação, a especificação das mensagens neste formato encontra-se a seguir:

Mensagem de Ação: As mensagens de ação são enviadas no sentido do sistema de agentes para o ambiente e servem para manifestar ao ambiente que um determinado agente deseja atuar ou perceber algo sobre ele. Sua especificação é dada por:

Especificação:

```
{
   "type" : "ACT",
  "agent" : <nome do agente>,
 "action" : <nome da ação>,
 "params" : [
             <parâmetro 1>,
             <parâmetro 2>,
             ...
             <parâmetro n>
            ]
}
```

Exemplo:

```
{
   "type" : "ACT",
  "agent" : "cleaner_1",
 "action" : "moveTo",
 "params" : [
             "10",
             "12"
            ]
}
```

Ao enviar um comando para o ambiente, o sistema de agentes deve construir esta mensagem JSON com os campos mostrados acima. Todas as mensagens possuem um campo "type" para especificar qual o tipo de mensagem e portanto quais campos esperar dela. O próximo campo é o "agent", utilizado para especificar qual agente está executando o comando, uma vez que no ambiente virtual podem existir diversos agentes, este identificador é o identificador discutido em 3.4.4. O terceiro campo é o "action", que especifica qual ação deverá ser realizada por este agente e que, como dito anteriormente, também engloba percepções, uma vez que "perceber algo" pode ser tratado como uma ação. O ultimo campo é uma lista, onde cada elemento desta lista é um parâmetro para a execução da ação em questão.

O exemplo ao lado demonstra o caso onde um agente chamado "cleaner_1" deseja realizar a ação "moveTo" com os parâmetros "10" e "12". Um contexto possível para esta mensagem pode ser que o agente em questão deseja mover-se para as coordenadas 10 e 12 do ambiente.

Mensagem de Resposta: As mensagens de resposta são transmitidas no sentido do ambiente para o sistema de agentes e servem para confirmar o status do comando

recebido e para repassar percepções obtidas em uma ação. Sua especificação é dada por:

Especificação:

```
{
     "type" : "ACT_RESPONSE",
   "status" : "SUCCESS"
              |"FAILURE"
              |"ERROR",
 "percepts" : [
   {
    "percept" : <percepção 1>,
    "perceptValues" : [
      <valor 1.1>,
      <valor 1.2>,
      ...
      <valor 1.n>
     ],
    "action" : "ADD"|"REMOVE"
   }, {
    "percept" : <percepção 2>,
    "perceptValues" : [
      <valor 2.1>,
      <valor 2.2>,
      ...
      <valor 2.n>
     ],
    "action" : "ADD"|"REMOVE"
   }, {
    ...
   }
  ]
}
```

Exemplo:

```
{
     "type" : "ACT_RESPONSE",
   "status" : "SUCCESS",
 "percepts" : [
   {
    "percept" : "currentPos",
    "perceptValues" : [
      "10",
      "12"
     ],
    "action" : "ADD"
   }
  ]
}
```

Assim como a outra mensagem, esta também possui um campo “type” para que o destinatário possa saber quais outros campos esperar dela. O campo seguinte é o “status” que deve ser uma das três possibilidades mostradas, servindo para responder para o sistema de agentes sobre a execução da ação onde, “SUCCESS” significa que tudo ocorreu como o esperado, “FAILURE” significa que a ação não pôde ser concluída devido a algum acontecimento no ambiente que pode ter impossibilitado sua conclusão, uma vez que ambientes podem ser dinâmicos, por fim este campo pode ser “ERROR” para representar algum erro além da execução do ambiente, como uma mensagem de comando mal formulada ou uma divisão por zero por exemplo. Esta distinção é útil para que o sistema de agentes possa reagir adequadamente, pois no caso de erro pode ser que seja necessário reenviar o comando, enquanto que no caso de falha pode ser que o agente possa utilizar esta informação. Por fim, o campo “percepts” contem uma lista de triplas, onde os elementos de cada tripla representa o nome de uma percepção, seus valores e se esta percepção deve ser adicionada ou removida.

O exemplo mostrado ao lado representa uma execução bem sucedida da ação “moveTo” utilizada de exemplo nas mensagens de comando. Para representar que a ação ocorreu corretamente o campo “status” está como “SUCCESS” e, como resultado, retorna em sua lista de percepções a percepção “currentPos” com os valores “10” e “12”, indicando que sua nova posição atual é a coordenada 10, 12.

3.4.6. Concorrência de Ações

Em relação ao problema da concorrência de ações o modelo não exige nenhum comportamento específico, transferindo esta responsabilidade para a implementação do modelo. Esta solução é a mais vantajosa pois permitir ou não a execução concorrente de ações depende do contexto do problema e, portanto, não é papel do modelo limitar a execução destas ações. Ao estabelecer o modelo desta forma, ele permite que implementação do problema trate da concorrência de recursos apenas nos casos em que for necessário.

3.4.7. Interoperabilidade

Um requisito deste modelo é que ele seja independente de plataforma, de modo que um agente implementado em uma determinada ferramenta possa executar em um ambiente virtual de qualquer outra ferramenta que implemente este modelo. O caminho inverso também é verdade, onde um determinado ambiente deve aceitar agentes de qualquer linguagem de agentes que implementem este modelo.

Este ponto do modelo não é ativamente implementado, mas surge como resultado dos demais requisitos do modelo, que por suas vez foram construídos de forma a proporcionar esta propriedade. Este requisito serve apenas para explicitar que o modelo é independente de ferramenta e que os demais requisitos garantem isso.

4. Implementação

Com o modelo consolidado, é necessário implementá-lo para verificar sua validade. Como citado na subseção 2.3 as ferramentas utilizadas foram o JASON e o Unity como ferramentas de agentes inteligentes e ambiente virtual respectivamente. O primeiro passo é estabelecer a estrutura do *framework* em relação à implementação dos agentes por parte dos usuários, sendo estabelecida seguinte estrutura:

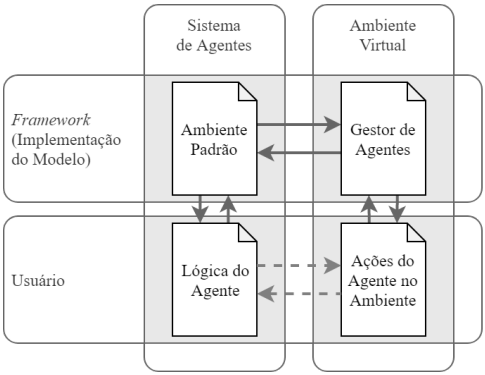

Figure 2. Estrutura do framework.

Neste diagrama as colunas representam qual o lado do sistema a implementação se

encontra, enquanto que as linhas especificam de quem é a responsabilidade de programar aquela parte do sistema. Podemos visualizar como a implementação que é realizada nesta seção se responsabiliza pela comunicação entre as ferramentas, enquanto que o usuário apenas deve programar seus agentes atuando no ambiente padrão e as manifestações de suas ações no ambiente virtual. Para isso é necessário que a implementação do modelo cubra:

- A criação de um ambiente padrão na linguagem de ambientes do sistema de agentes (que no caso do JASON é a linguagem Java), onde este ambiente padrão tem a capacidade de receber as ações do agente, transformando-as em mensagens, no formato estabelecido para isso neste trabalho, para então repassá-las por protocolo TCP e esperar uma resposta do ambiente;
- No outro lado do sistema, no ambiente virtual, devem haver receptores destas mensagens capazes de traduzir nomes de ações em ações programadas pelo usuário e responder às mensagens com as percepções retornadas pelas ações.

  Esta estrutura fornece duas principais vantagens:

- Do ponto de vista do usuário as ações que o agente executa no sistema de agentes são manifestadas diretamente como suas implementações no ambiente, sem ter que se preocupar com os aspectos de troca de mensagem que são responsabilidade do *framework*. Em outras palavras, do ponto de vista do usuário a comunicação do agente com o ambiente parece acontecer como nas setas pontilhadas, enquanto que na realidade ela ocorre pelo caminho das setas preenchidas;
- Uma vez implementada, esta estrutura resulta em um código genérico o suficiente para que usuários possam representar diversos problemas de agentes inteligentes programando apenas o agente e seu comportamento no ambiente, sem a necessidade de modificar a implementação do modelo uma vez que para implementar novos agentes e ações não é necessário modificar o código do *framework*.

5. Avaliação

5.1. Ambiente de Testes

Para realizar a validação e avaliação da implementação do modelo é necessário criar um ambiente virtual que utilize-a. O ambiente escolhido é um exemplo de ambiente *Cleaning Robots* onde, há uma lata de lixo e lixo surge com o passar do tempo. O comportamento do agente deste ambiente se resme a:

- O agente sempre deseja que o ambiente esteja limpo;
- Se ele conhece a posição de algum lixo ele irá se deslocar até ele e irá pegá-lo;
- Ao pegar um lixo ele se deslocará até a lata de lixo e descartará ele;
- Caso o agente não conheça a posição de algum lixo ele irá escanear a região até que encontre mais lixo.

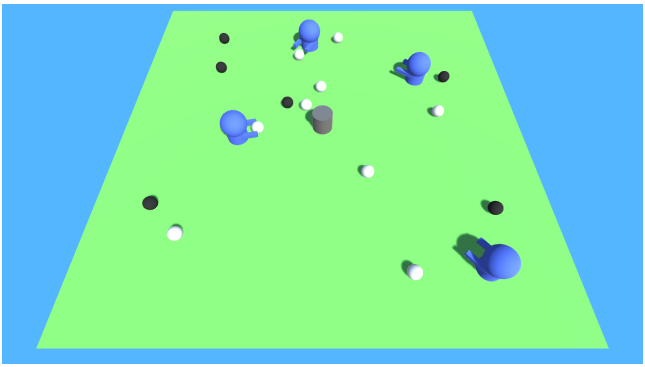

Figure 3. Ambiente de exemplo no Unity.

A figura 3 é uma demonstração do ambiente em funcionamento[1]. As bolinhas brancas e pretas são lixo, sendo que as pretas surgiram depois do escaneamento de um agente, de forma que elas não constam nas percepções de nenhum deles por enquanto.

5.2. Resultados

Os testes foram realizados em um computador com as seguintes especificações:

- Processador Intel Core i5-7400 @3.00GHz;
- Placa de vídeo NVIDIA GeForce GTX 1060 6GB;
- 2×8GB de memória RAM DDR4.

5.2.1. Avaliação do Ambiente

O primeiro resultado analisado é a quantidade de quadros por segundo (FPS) que o ambiente proporcionou conforme o número de agentes aumentava, com o objetivo de avaliar qual se há impacto no desempenho de um ambiente contendo muitos agentes. Os testes avaliam uma execução normal do programa durante um período de um minuto. Os resultados são apresentados à seguir:

[1]Uma demonstração em vídeo do ambiente em questão pode ser acessada neste endereço: youtu.be/XBO5itsxsck

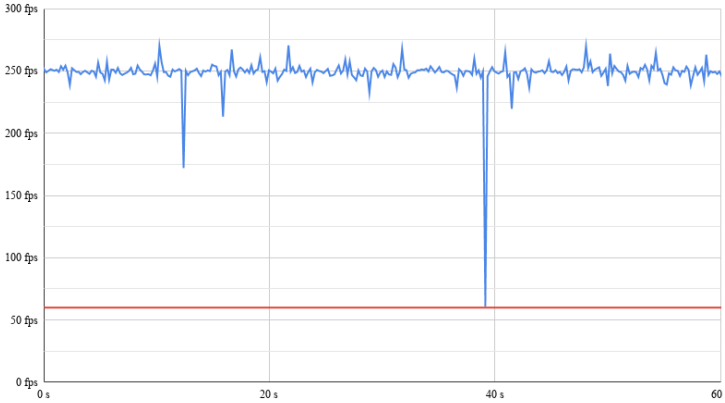

Figure 4. FPS ao longo do tempo com 10 agentes.

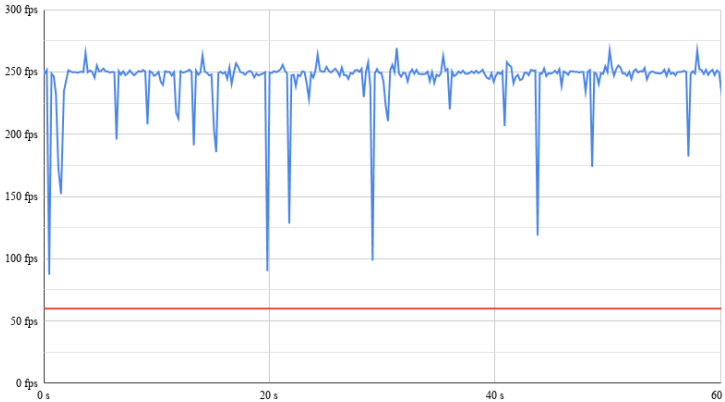

Figure 5. FPS ao longo do tempo com 25 agentes.

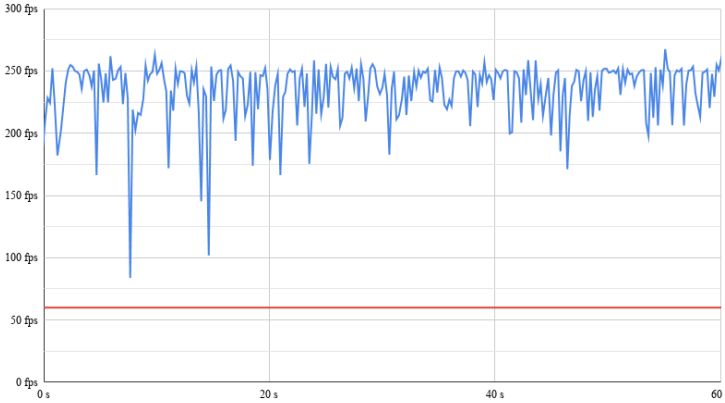

Figure 6. FPS ao longo do tempo com 75 agentes.

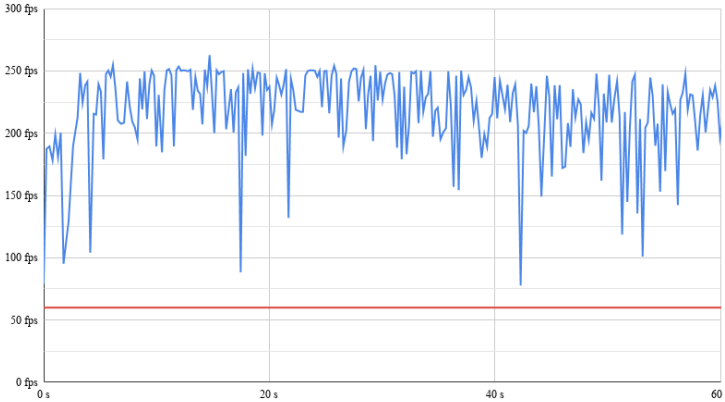

Figure 7. FPS ao longo do tempo com 100 agentes.

Os gráficos acima exibem as estatísticas coletadas em azul, enquanto que a linha vermelha demarca a faixa de 60 quadros por segundo, o que é considerado uma taxa de atualização ótima para jogos eletrônicos.

É possível observar principalmente que em nenhum momento a taxa de atualização cai para baixo da faixa vermelha, significando que o impacto no desempenho não é facilmente perceptível. Outra característica dos dados é o aumento da variância conforme o número de agentes aumenta, provavelmente causado pela maior quantidade de troca de mensagens que deve ser realizada.

Em alguns casos o processamento das mensagens acaba demorando mais tempo, este fenômeno pode ser percebido nas quedas muito grandes, como por exemplo na figura

4. A motivação para estas quedas não é clara, uma vez que uma mesma mensagem pode ser processada rapidamente em um momento e causar uma grande queda em outro. Estas quedas não são perceptíveis no teste escolhido pois o sistema se recupera rapidamente. Em um sistema mais complexo é possível que a média de *frames* por segundo seja menor, fazendo com que estas quedas se tornem perceptíveis, este problema no entanto não está relacionado ao modelo e sim com a complexidade do cenário na *game engine* em si.

A grande variação de FPS pode ser um problema em casos onde medidas precisas são necessárias, uma vez que a ferramenta executa códigos por *frame*, o que resulta em execuções cujo intervalo entre cada iteração não é constante. A ferramenta Unity apresenta recursos para melhor gerenciar este comportamento, como por exemplo a possibilidade de compensar ações com base no tempo desde a ultima execução do *game loop* ou a possibilidade de realizar a execução em um *game loop* paralelo ao principal, onde este segundo itera em intervalos fixos de tempo. Apesar destes recursos ainda assim é provável que certo ruído surja nas medidas, embora menos significativo. Além disso, caso o modelo seja implementado em outros ambientes virtuais pode ser que eles não apresentem estes recursos e a variação na taxa de *frames* se torne um problema.

5.2.2. Avaliação da Ferramenta de Agentes

Além do ambiente, também é preciso avaliar o desempenho no sistema de agentes. Uma das principais métricas para avaliar um sistema desenvolvido em JASON é a quantidade de ciclos de raciocínio executados por segundo. As médias e desvios padrão coletados nos testes dessa métrica encontram-se na tabela a seguir:

Table 1. Resultados coletados sobre a quantidade de ciclos por segundo do sistema de agentes.

Num. de Agentes	Média	Desvio Padrão
10	≈ 1199	≈ 824
25	≈ 1324	≈ 537
75	≈ 754	≈ 160
100	≈ 602	≈ 143

É possível perceber com os dados coletados que conforme o número de agentes aumenta a quantidade de ciclos de raciocínio por segundo diminui, resultado já esperado, uma vez que mais agentes implicam em um maior uso do processador. O desvio padrão apresentou um comportamento interessante, onde ele reduzia conforme o número de agentes crescia, ou seja, o sistema se tornava mais consistente em relação a quantidade de ciclos de raciocínio. A motivação para este comportamento não é clara, entretanto a hipótese é que a ferramenta JASON esteja limitando a execução do próximo ciclo de raciocínio de cada agente para que eles estejam em sincronia com os demais, isto causaria um efeito onde agentes que levam menos tempo em seus ciclos deveriam esperar os demais, nivelando o tempo por ciclo e consequentemente diminuindo o desvio padrão. Este comportamento também contribuiria para a redução da média de ciclos por segundo junto ao simples fato de existirem mais agentes para serem administrados.

Mesmo com a queda de desempenho não foi possível observa-la visivelmente no ambiente de teste, de forma que para aplicações menos precisas a solução é facilmente viável. No entanto para aplicações que precisam de um melhor desempenho é possível que precauções devam ser tomadas para melhor configurar as ferramentas e extrair um melhor desempenho.

6. Conclusão

A solução proposta é satisfatória diante dos resultados obtidos. O modelo atende aos requisitos propostos no início do trabalho e o *framework* implementa os requisitos do modelo corretamente.

Os testes também foram bem sucedidos. O ambiente virtual não manifestou nenhuma queda de desempenho drástica, sempre se mantendo acima da faixa de 60 quadros por segundo. O sistema de agentes também obteve bons resultados, uma vez que que os agentes se manifestaram da maneira correta no ambiente sem apresentar erros ou atrasos na tomada de decisões dos agente. Em ambos os casos os valores dos critérios de avaliação diminuíram conforme o número de agentes aumentava, o que já era esperado. Entretanto, em nenhum dos casos os sistemas apresentaram grandes perdas de desempenho, indicando que os objetivos deste trabalho foram cumpridos.

6.1. Trabalhos Futuros

A partir do que foi desenvolvido neste trabalho é possível desenvolver novos trabalhos relacionados a este:

- **Estabelecer um novo tipo de mensagem para comunicação entre agentes:**
  Atualmente não é possível, através do ambiente, que um agente envie uma requisição para outro agente pois não há suporte para este fluxo de mensagens. Um possível trabalho futuro estabeleceria o formato deste tipo de mensagem, analisando aspectos como a necessidade ou não do uso do protocolo TCP neste tipo de mensagem por exemplo. Este fluxo poderia ser utilizado para um agente requisitar a percepção de outro, ou informar outros de suas próprias percepções. No caso deste trabalho a linguagem JASON suporta a comunicação entre agentes na ferramenta, sem interação com o ambiente, de forma que não é possível tornar a comunicação parte do ambiente, como por exemplo simulando falhas de transmissão, algo que seria responsabilidade do ambiente. Além disso esta implementação permitiria que ferramentas sem suporte a comunicação entre agentes pudessem realizá-las.
  Este novo fluxo faria com que a conexão ocorresse no sentido do ambiente se conectando a um agente, o que é interessante também para realizar percepção passiva, uma vez que se o ambiente é capaz de se comunicar com seus agentes ativamente, ele é capaz se informar à eles suas percepções sem que haja uma requisição para isso.
- **Adicionar suporte para o modelo de artefatos:**
  O modelo de artefatos apresenta uma forma de criar objetos interativos no mundo. Neste trabalho para que objetos pudessem ser manipulados eles eram identificados por coordenadas, o que não é uma abordagem muito adequada pois nada garante que dois objetos não possam estar na mesma coordenada. Com um suporte a

artefatos seria possível estabelecer identificadores para os artefatos de forma que todos os agentes pudessem definir unicamente cada artefato, além de permitir a execução de ações descritas pelos próprios artefatos, de forma análoga ao que é realizado por [Poli 2018] em seu trabalho.

- **Expandir o *framework* para outras linguagens:**
  A utilização de JASON e Unity neste trabalho apresentou diversos desafios com soluções interessantes, como o uso de corotinas e reflexão na parte do Unity por exemplo. Expandir o *framework* para outras *game engines*, como a *Unreal Engine*, ou para outras linguagens de agente, como o Sigon, pode apresentar outros desafios específicos destas ferramentas e que resultariam em um trabalho interessante.

References

Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, USA.

Dorri, A., Kanhere, S. S., and Jurdak, R. (2018). Multi-agent systems: A survey. *IEEE Access*, 6.

Gregory, J. (2014). *Game Engine Architecture, Second Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition.

Kelton, D. W., Sadowski, R. P., and Sadowski, D. A. (1997). *Simulation with Arena*. McGraw-Hill, Inc., New York, NY, USA, 1st edition.

Poli, N. (2018). Game engines and mas: Bdi & artifacts in unity. Master's thesis, University of Bologna.

Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Series in Artificial Intelligence. Prentice Hall, Upper Saddle River, NJ, third edition.

Unity (2019). *Unity User Manual (2019.1)*. Unity Technologies.

Wooldridge, M. (2002). *Introduction to MultiAgent Systems*. John Wiley & Sons.