UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO ACADÊMICO DE INFORMÁTICA

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ALEXIS VAN HAARE HEIJMEIJER

INTERLIGAÇÃO ENTRE A FERRAMENTA DE SIMULAÇÃO SUMO E O PROJETO MAPS

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2016

ALEXIS VAN HAARE HEIJMEIJER

INTERLIGAÇÃO ENTRE A FERRAMENTA DE SIMULAÇÃO SUMO E O PROJETO MAPS

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Gleifer Vaz Alves

PONTA GROSSA

2016

TERMO DE APROVAÇÃO

INTERLIGAÇÃO ENTRE A FERRAMENTA DE SIMULAÇÃO SUMO E O PROJETO MAPS

por

ALEXIS VAN HAARE HEIJMEIJER

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 27 de outubro de 2016 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Prof. Dr. Gleifer Vaz Alves
Orientador

_____
Prof. Dr. André Pinz Borges
Membro titular

_____
Prof. Dr. Augusto Foronda
Membro titular

_____
Prof. MsC. Carlos Eduardo Pantoja
Membro titular

_____
Prof. Dr. Augusto Foronda
Responsável pelo Trabalho de Conclusão de Curso

_____
Prof. Dr. Erikson Freitas de Morais
Coordenador do curso

RESUMO

HEIJMEIJER, Alexis. **Interligação entre a ferramenta de simulação SUMO e o projeto MAPS**. 2016. 80 f. Trabalho de Conclusão de Curso - Bacharelado em Ciência da Computação – Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2016.

O conceito de Cidade Inteligente utiliza a infraestrutura de uma cidade para trabalhar de maneira sincronizada com a tecnologia permitindo melhorar o monitoramento e planejamento da mesma, com o objetivo de reduzir problemas encontrados no dia a dia. Um desses problemas é a falta de vagas em estacionamentos urbanos que pode ser controlado através da implantação de sistemas de *Smart Parkings* para, por exemplo, guiar o motorista até uma vaga desocupada. Uma das maneiras de implementar um *Smart Parking* é por meio do uso de agentes que se comunicam entre si e captam informações do ambiente, como condição da vaga e lotação do estacionamento, que auxiliam os motoristas na busca pelo local ideal para estacionar. Para tentar lidar com questões de *Smart Parking*, tem-se o projeto MAPS (*MultiAgent Parking System*), que visa criar soluções para *Smart Parking* por meio de sistemas multiagentes. O projeto MAPS é implementado por meio do *framework* JaCaMo, onde os agentes são criados na linguagem Jason e os artefatos são desenvolvidos no Cartago. O uso de uma ferramenta de simulação é essencial para analisar o comportamento do ambiente simulado e obter resultados melhores e mais precisos, principalmente em ambientes relacionados à mobilidade urbana. O SUMO é uma ferramenta de simulação de trânsito que busca justamente atingir esses propósitos. O presente trabalho apresenta a interligação entre a ferramenta SUMO e o projeto MAPS, mostrando que é possível modelar um estacionamento com vagas e veículos no SUMO e definir um *Middleware* para a comunicação entre os agentes e artefatos implementados, respectivamente no Jason e Cartago, com a ferramenta de simulação de um estacionamento.

Palavras-chave: Simulação. Estacionamento Inteligente. Agentes Inteligentes. *SUMO*. *Framework* JaCaMo

ABSTRACT

HEIJMEIJER, Alexis. **Interconnection between SUMO simulation tool and MAPS Project**. 2016. 80 f. Trabalho de Conclusão de Curso - Bacharelado em Ciência da Computação – Federal University of Technology - Parana. Ponta Grossa, 2016.

Smart City is a city where its infrastructure works with technology improving the monitoring system and the city planning to reduce daily problems. The lack of parking spots is one of these problems that can be controlled by creating Smart Parking systems that can guide drivers to free spots. One way to implement a Smart Parking is through agents that communicate with each other, collect information from the environment, and send it to drivers. The project MAPS (MultiAgent Parking System) studies and develops Smart Parking solutions based on multiagent systems. MAPS is implemented using JaCaMo framework, which creates the agents using Jason programming language and develops the artifacts using Cartago. The use of simulation tools is essential to analyze the simulated environment behavior and to achieve better and more accurate results, especially when working with urban mobility. SUMO is a very powerful simulation tool that tries to obtain these results. This work presents the interconnection between SUMO and the project MAPS, proving that is possible to model a parking lot in SUMO and also to defines a middleware that connects the agents and artifacts, respectively implemented in Jason and Cartago, with the simulation tool.

Keywords: Simulation. Smart Parking. Intelligent Agents. SUMO. JaCaMo framework

LISTA DE ILUSTRAÇÕES

LISTA DE DIAGRAMAS

LISTA DE GRÁFICOS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

SUMO	*Simulation of Urban MObility*
ABGS	*Agent Based Guiding System*
MAPS	*Multi-Agent Parking System*
GPAS	Grupo de Pesquisa em Agentes de Software
JaCaMo	Jason + Cartago + Moise
Jason	*A Java-based interpreter for an extended version of AgentSpeak*
XML	*eXtensible Markup Language*
SMA	*Sistemas Multi-Agentes*
BDI	*Belief-Desire-Intention*
OSM	*OpenStreetMaps*
TraCI	*Traffic Control Interface*
UDP	*User Datagram Protocol*
TraCI4J	*Traffic Control Interface For Java*

SUMÁRIO

1 INTRODUÇÃO

Smart City (em português, cidade inteligente) mostra-se uma realidade no mundo atual devido ao rápido avanço da tecnologia. *Smart City* é definido como uma cidade onde a sua infraestrutura trabalha de maneira sincronizada com a tecnologia, permitindo aos agentes viverem e interagirem nesse ambiente contribuindo para que a cidade se torne mais eficiente (BATTY *et al.,* 2012). Cidades inteligentes contemplam diversos pontos, como segurança pública integrada e vigilância através de câmeras. Outro ponto é a Mobilidade Urbana, que corresponde às necessidades humanas de deslocamento em um ambiente de acordo com as dimensões do espaço e das atividades exercidas nele. Ainda assim, a mobilidade é um tópico muito abrangente, porém, um dos pontos mais importantes é a questão do trânsito. O trânsito envolve toda a parte de infraestrutura e serviços de transporte, sejam eles públicos ou privados, tais como estacionamento, monitoramento de transporte público, monitoramento de vias, entre outros.

Estacionamento pertence à parte de infraestrutura de trânsito de uma cidade e pode apresentar diversos problemas, como a falta de vagas e até congestionamento dentro e fora dos mesmos. Segundo pesquisa de Koster, Koch e Bazzan (2014), 40% do trânsito de Nova Iorque é causado por carros em busca de vagas de estacionamento. Segundo o Departamento Nacional de Trânsito (DENATRAN), houve um aumento de 130% na frota de veículos em circulação no Brasil entre janeiro de 2005 e janeiro de 2016. Este aumento reflete diretamente na busca de vagas. Por exemplo, em São Paulo, encontram-se 132 carros na disputa de cada vaga de estacionamento (VagaBarata 2014).

Já o projeto desenvolvido pela *Deutsche Telekom* em 2014, mostra que 30% do trânsito de grandes cidades é gerado por motoristas em busca de vagas de estacionamento. O projeto visa construir uma estrutura inteligente na cidade de Pisa (Itália), também prova que a implantação do conceito de *Smart City* garante um melhor fluxo de trânsito, reduz a emissão de gás carbônico e facilita a procura por vagas de estacionamento. Projetos como este aplicam o conceito de *Smart Parking*, o qual consiste no uso de tecnologias para automatizar, facilitar e viabilizar o melhor uso de espaços de estacionamento.

Revathi e Dhulipala (2012) descrevem diferentes tipos de *Smart Parking*, classificados conforme suas características: sistemas de pagamentos,

estacionamentos automatizados, sistemas inteligentes, sistemas de condução a vagas, entre outros. Entre eles destaca-se o *Agent Based Guiding System* (*ABGS*), em português, Sistema de Condução Baseado em Agentes, o qual caracteriza o tipo de *Smart Parking* utilizado neste trabalho. O ABGS trabalha com agentes inteligentes implementados através de sensores capazes de absorver informações do ambiente e reagir de maneira autônoma através das informações obtidas, de modo a atingir um objetivo. Além disso, os agentes podem se comunicar entre si. Um agente pode ser definido como um *software* que possui um objetivo a ser atingido em relação ao ambiente em que está atuando (WOOLDRIDGE, 2009).

Com o intuito de estudar problemas e soluções para um *Smart Parking* do tipo ABGS, criou-se o projeto *Multi-Agent Parking System* (MAPS), desenvolvido dentro da Universidade Tecnológica Federal do Paraná (UTFPR), em Ponta Grossa. Neste projeto existem diferentes linhas de estudo, como implementação de agentes, organização social, negociação, coordenação, dentre outros. Ao passo que o trabalho aqui apresentado tem como foco principal a simulação de estacionamentos controlados (ambientes que possuem controle de acesso e não estacionamentos públicos) por meio do simulador *Simulation of Urban MObility* (SUMO).

O SUMO é um simulador de tráfego urbano de código aberto, que permite a modelagem de diversos tipos de ambientes e a simulação do comportamento desses ambientes com fluxo de tráfegos variados, aproximando-se ao máximo do mundo real (BEHRISCH *et al.* 2011). A interligação entre o SUMO e o MAPS busca justamente preencher uma lacuna do projeto MAPS em relação ao uso de simuladores de trânsito que possam ajudar a obter resultados melhores e mais precisos. O uso de simuladores permite visualizar graficamente o funcionamento do ambiente e contribui para o teste de novos algoritmos de alocação de vagas implementados no projeto.

Por sua vez, o projeto MAPS é implementado utilizando o *framework* para desenvolvimento de sistemas multiagentes JaCaMo, o qual é dividido em três níveis: nível dos agentes (Jason), nível do ambiente (Cartago) e nível de organização (Moise). A interligação entre o SUMO e o MAPS dá-se por meio de um artefato, desenvolvido no nível de ambiente do projeto MAPS, e do *Middleware* MAPS-SUMO, ferramenta intermediária dessa conexão, desenvolvida por este projeto, a qual recebe protocolos do MAPS e encaminha para a simulação. O objetivo final deste trabalho é demonstrar a interligação entre o MAPS e o SUMO, bem como a

viabilidade de uma representação gráfica do uso do estacionamento por meio de uma ferramenta de simulação.

1.1 OBJETIVOS

A seguir são descritos o objetivo geral e objetivos específicos.

1.1.1 Objetivo Geral

O objetivo geral deste trabalho é interligar a ferramenta de simulação SUMO com o projeto MAPS.

1.1.2 Objetivos Específicos

Para atingir o objetivo geral, são necessários os seguintes objetivos específicos:

- Compreender o SUMO;
- Implementar modelos de fluxo de trânsito no SUMO;
- Modelar um estacionamento com alocação de vagas;
- Implementar o modelo com alocação de vagas utilizando o SUMO;
- Analisar os resultados obtidos com o modelo;
- Adaptar e estender a implementação de agentes e artefatos do MAPS;
- Interligar o SUMO com o *framework* JaCaMo por meio de um *Middleware*.

1.2 JUSTIFICATIVA

O projeto MAPS pretende trazer soluções baseadas em sistemas multiagentes para problemas de cidades inteligentes, mais especificamente, soluções para *Smart Parkings.* Como o MAPS ainda não possui um simulador gráfico específico, o uso de simuladores de trânsito pode ser útil para executar

experimentos de simulação do projeto de uma maneir robusta, com características visuais e com a possibilidade da utilização de dados reais.

A escolha da ferramenta SUMO se deve ao fato da mesma, diferente de outras ferramentas de simulação, ser disponibilizada gratuitamente além de ter o código aberto, que permite ao pesquisador utilizá-lo de maneira mais eficiente e de acordo com suas necessidades (KRAJZEWICZ *et al.* 2002). Na verdade, o SUMO não é apenas uma simples ferramenta de simulação, mas sim uma aplicação completa composta por diversos pacotes com diferentes funcionalidades. Uma delas é utilizada para importar mapas reais e convertê-los para formatos que podem ser compreendidos pelo SUMO e, assim, trazer desde simples cruzamentos até uma cidade inteira para simulação (BEHRISCH *et al.* 2011).

Além disso, como é mostrado por Vincent Baines e Julian Padget (2014), existe uma interligação da ferramenta SUMO com a plataforma de desenvolvimento de sistemas multiagentes Jason, que permite um maior controle sobre os agentes que interferem no ambiente, como os veículos. Assim, é possível desabilitar, por exemplo, o controle de velocidade pelo SUMO e assumir esse controle através da plataforma Jason. A comunicação entre a ferramenta SUMO e o Jason também é descrita por Batista Júnior e Coutinho (2013), onde os autores descrevem um sistema multiagente com fins de tornar cruzamentos de vias arteriais mais eficientes.

Os trabalhos citados afirmam a viabilidade da interligação entre a ferramenta de simulação SUMO e a linguagem de programação de agentes Jason, porém não foram encontrados trabalhos que relacionam a interligação do SUMO com as linguagens Jason e Cartago. Por isso, este trabalho busca demonstrar a interligação entre o Jason e Cartago (JaCa) com a ferramenta SUMO, apresentando conceitos teóricos e exemplos práticos que possibilitaram efetivar a comunicação.

1.3 ORGANIZAÇÃO DO TRABALHO

O restante deste documento está organizado da seguinte forma.

O capítulo 2 apresenta os trabalhos relacionados.

O capítulo 3 discorre sobre agentes, sistemas multiagentes e framework JaCaMo.

O capítulo 4 detalha o funcionamento do projeto MAPS.

O capítulo 5 aborda a ferramenta de simulação utilizada para o desenvolvimento desse trabalho, o SUMO.

O capítulo 6 demonstra as a interligação prática entre a ferramenta SUMO e o projeto MAPS.

O capítulo 7 relata sobre os experimentos realizados para verificar a validade da interligação entre o SUMO e o MAPS.

O capítulo 8 descreve as conclusões obtidas ao final deste trabalho.

2 TRABALHOS RELACIONADOS

Este capítulo tem como objetivo apresentar trabalhos que englobam as áreas de mobilidade urbana, trânsito, *Smart Parking* e SUMO. Tais áreas estão relacionadas ao desenvolvimento deste trabalho. Para contornar problemas encontrados no cotidiano, mais especificamente no trânsito e em estacionamentos, pesquisadores propuseram o conceito de *Smart Parking*, descrevendo suas ideias e classificando-os em diversas categorias. A partir disso, outros profissionais desenvolveram soluções baseadas no conceito de *Smart Parking.*

Para analisar a viabilidade das soluções propostas, faz-se necessário a utilização de ferramentas de simulação para obter resultados mais objetivos e concretos. A integração entre ferramentas de simulação e outras ferramentas, como *framework* para controlar sistemas multiagentes, permite a implementação de ambientes mais dinâmicos e realistas, como sistemas inteligentes de estacionamento.

Revathi e Dhulipala (2012) descrevem o conceito de *Smart Parking* e classificam os sistemas em categorias, de acordo com as características apresentadas. Entre as categorias descritas encontram-se sistemas de pagamento inteligente, estacionamento automatizado e sistema de reserva de estacionamento. Além desses, destaca-se o ABGS, que é um sistema de condução a vagas baseado em agentes e o mesmo utilizado pelo projeto MAPS.

Koster, Koch e Bazzan (2014) propõem uma solução para problemas de *Smart Parking* utilizando sistemas de condução a vagas por meio de informações, denominado *Parking guidance and information system*. O modelo implementa um aplicativo que permite a usuários informar e buscar vagas disponíveis, além de estudar maneiras de incentivar a população a aderir a aplicativos com esse objetivo.

A implementação de sistemas baseado em agentes é uma abordagem muito utilizada para ambientes dinâmicos, ou seja, quando diferentes agentes interagem no mesmo ambiente onde há frequentes influências externas. Visto isso, Baines e Padget (2014) descrevem o desenvolvimento de um *framework* para analisar o comportamento de veículos inteligentes. Para a simulação foi utilizado a ferramenta SUMO e para o controle de agentes foi utilizado a linguagem Jason. Em um dos cenários de teste, o Jason adiciona veículos (agentes) ao SUMO especificando

algumas características e faz verificações para detectar possíveis colisões, e então reduzir a velocidade do veículo ou trocar de faixa. Em um segundo cenário, veículos são inseridos no SUMO e detectam semáforos e seu estado (aberto, fechado). O agente recebe uma ordem para reduzir a velocidade para um certo valor e por um determinado período de tempo para chegar ao semáforo aberto, e então o controle do agente é devolvido ao SUMO.

O trabalho de Baines e Padget (2014) aborda uma forma de trabalhar com o SUMO utilizando os veículos como agentes controlados pela linguagem Jason, interligando as duas ferramentas. Porém o estudo foi feito em relação a simulação de trânsito em geral, sendo que o foco dos autores não considerava o gerenciamento de estacionamentos, como é o caso do projeto MAPS.

Outro trabalho que igualmente faz uso do SUMO é apresentado por Krajzewicz *et al.* (2012). Os autores criam um modelo denominado *CityMobil,* no SUMO, o qual simula um ambiente de estacionamento de transporte público, onde ônibus circulam em uma rota e param em alguns locais para o embarque de passageiros. Carros são gerados em fluxos e percorrem a rota até estacionarem. Porém os veículos sempre seguem uma mesma ordem na hora da escolha de vagas e não há um controle de vagas livres e ocupadas. Através da biblioteca TraCI, cria-se uma *interface* para controlar carros chegando, pessoas entrando na fila de espera e pessoas embarcando nos ônibus.

Por fim, ainda destaca-se o trabalho proposto por Batista Júnior e Coutinho (2013), onde os autores descrevem um sistema multiagente o qual atua em vias arteriais, controladas por semáforos, com o objetivo de tornar os cruzamentos mais eficientes. Cada intersecção é controlada por um agente. Os agentes são controlados através da linguagem de programação de agentes Jason e simulados na ferramenta SUMO. O artigo também demonstra o modelo de integração entra as duas ferramentas, que é feito através da API (*Application Programming Interface*) XTRACI, desenvolvida em java e baseada na biblioteca TraCI, a mesma utilizada pelo modelo *CityMobil.*

Os trabalhos previamente citados evidenciam o uso de soluções tecnológicas para *Smart Parking*, bem como o uso do simulador SUMO em conjunto com agentes inteligentes para simulações de trânsito. Contudo, pelo que se conhece da literatura atual, não foram encontrados trabalhos que demonstram a interligação entre o simulador SUMO e o *framework* JaCaMo, apenas a interligação com a

linguagem de programação Jason. Além disso, os trabalhos que mostram o uso do Jason com o SUMO têm foco na simulação de fluxo de trânsito. Ao passo que o trabalho aqui apresentado tem foco no gerenciamento de vagas de estacionamento.

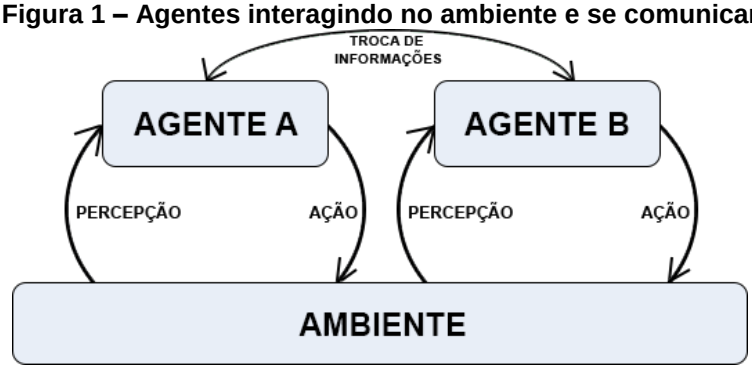
3 AGENTES E SISTEMAS MULTIAGENTES

Este capítulo descreve os agentes e sistemas multiagentes. Na seção 3.1 serão classificados os diferentes tipos de agentes. A seção 3.2 detalha o funcionamento do *framework* JaCaMo e a linguagem de programação de agentes Jason.

3.1 AGENTES

Um agente pode ser definido como um software capaz de captar informações dentro de um ambiente e reagir à elas de maneira autônoma, buscando atingir um objetivo previamente definido. O agente também pode comunicar-se com outros agentes que interagem no mesmo ambiente, como é possível observar na figura 1 (WOOLDRIDGE, 2009).

Figura 1 – Agentes interagindo no ambiente e se comunicando



Fonte: Adaptado de (WOOLDRIDGE, 2009)

As principais características de um agente são a autonomia, pró-atividade, reatividade e interatividade social que juntas trabalham para alcançar o objetivo atribuído ao agente. Devido à essas características e à capacidade de comunicação entre os agentes, estes são utilizados para resolverem problemas dinâmicos onde há interação humana no ambiente de atuação dos mesmos (WOOLDRIDGE, 2009).

Um agente autônomo tem a capacidade de agir sem a necessidade de intervenção humana ou de outros agentes, visando a execução dos objetivos à eles. Além disso, o agente é capaz de tomar decisões com base em sua própria capacidade de se adaptar e aprender com os seus comportamentos dentro do ambiente (MAES, 1995).

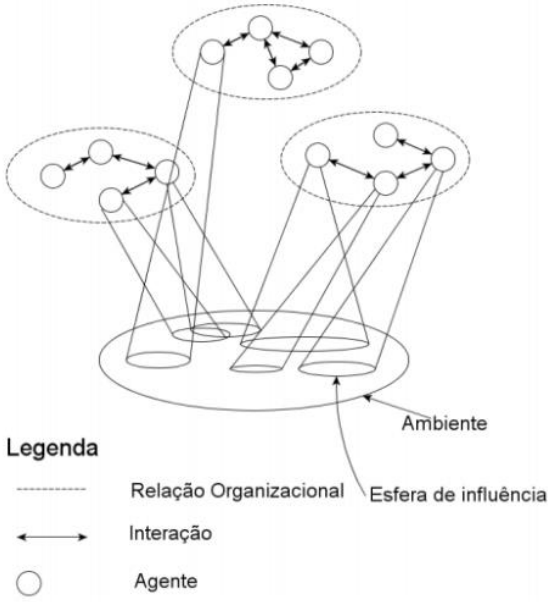
De acordo com Bordini, Hübner e Wooldridge (2007), um agente proativo é aquele que tem a capacidade de tomar a iniciativa em relação ao objetivo a ser atingido. O agente não reage simplesmente em resposta às ações do ambiente, mas sim focando em sua tarefa final.

Um agente com características reativas tem a capacidade de interpretar o ambiente e reagir prontamente de acordo com as mudanças que ocorrem no mesmo. As respostas dos agentes podem ser reflexivas ou racionais. Agentes reflexivos simplesmente reagem a estímulos absorvidos do ambiente, enquanto os agentes racionais baseiam-se nos conhecimentos adquiridos do ambiente (BORDINI *et al.*, 2007).

Segundo Genesereth e Ketchpel (1994), a capacidade de interação social de um agente é determinada pela comunicação com outros agentes no mesmo ambiente e, possivelmente humanos, através de uma linguagem de comunicação de agentes.

Existem sistemas que são compostos por diversos agentes que interagem em um único ambiente, chamados Sistemas MultiAgentes (SMA). Basicamente, cada agente possui uma esfera de influência que é controlada total ou parcialmente por ele, conforme figura 2. Além disso, a comunicação entre os agentes permite que um agente tenha conhecimento de certas informações de outro agente (BORDINI *et al*, 2007).

Figura 2 – Exemplo da estrutura de um SMA



Fonte: Adaptado de (BORDINI *et al,* 2007)

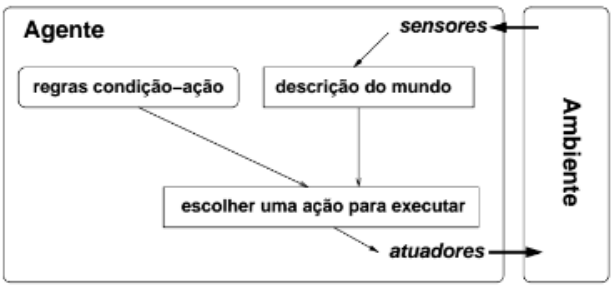
3.2 CLASSIFICAÇÃO DE AGENTES

Agentes são utilizados para resolver diversos tipos de problemas, normalmente dinâmicos, porém cada um deles possui diferentes características e requisitos, o que implica no uso de agentes específicos para cada problema. Aqui são apresentados os seguintes tipos de agentes: reativos, deliberativos, cognitivos e híbridos.

3.2.1 Agentes reativos

Agentes reativos são agentes que reagem diretamente a estímulos absorvidos do ambiente em que atuam, e não possuem conhecimento nem memória de suas ações. São agentes bem simples que baseiam as suas reações apenas de acordo com sua percepção do ambiente. Esse tipo de agente, ilustrado na figura 3, possui um conjunto de regras as quais definem qual a ação a ser realizada de acordo com as percepções do ambiente (ALVARES, SICHMAN, 1997).

Figura 3 – Arquitetura de um agente reativo



Fonte: (HÜBNER, 2003)

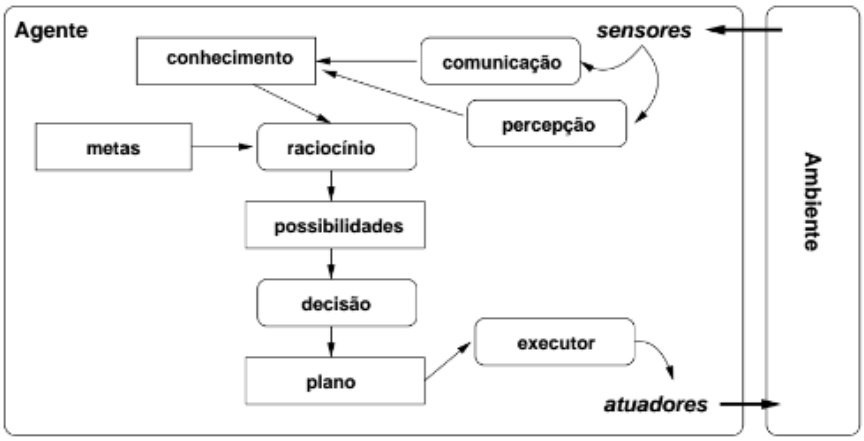
3.2.2 Agentes deliberativos

Agentes deliberativos possuem uma representação simbólica do ambiente em que estão situados. Esses agentes possuem desejos e objetivos e suas ações são baseadas em relação a essa representação própria do ambiente (ANUMBA *et al.* 2005).

3.2.3 Agentes cognitivos (ou racionais)

Agentes cognitivos possuem uma representação simbólica do ambiente em que se encontra, e suas decisões são baseadas em um conjunto de raciocínios lógicos. Este modelo, que pode ser observado na figura 4, surge do processo de decisão, momento a momento, quais ações a serem tomadas rumo a seus objetivos. Este processo envolve duas etapas: (i) decisão dos objetivos a serem atingidos e (ii) passos para alcançar estes objetivos (WOOLDRIDGE, 1999).

Figura 4 – Modelo de um agente cognitivo



Fonte: (HÜBNER, 2003)

BDI (*Belief-Desire-Intention*, ou crença-desejo-intenção) é um modelo computacional desenvolvido para trabalhar com agentes inteligentes. O conjunto crença-desejo-intenção desse modelo é considerado o estado mental do agente e suas ações são baseadas de acordo com o conteúdo desses estados (BORDINI *et al.* 2007).

As crenças são todas as informações que um agente tem sobre o ambiente, sendo elas atualizadas ou não. Os desejos são possíveis estados que o agente pretende atingir, porém não significa que os mesmos serão cumpridos. Intenções são os objetivos atribuídos a um agente, e estes serão atingidos através de desejos que podem também se tornarem intenções (BORDINI *et al.* 2007).

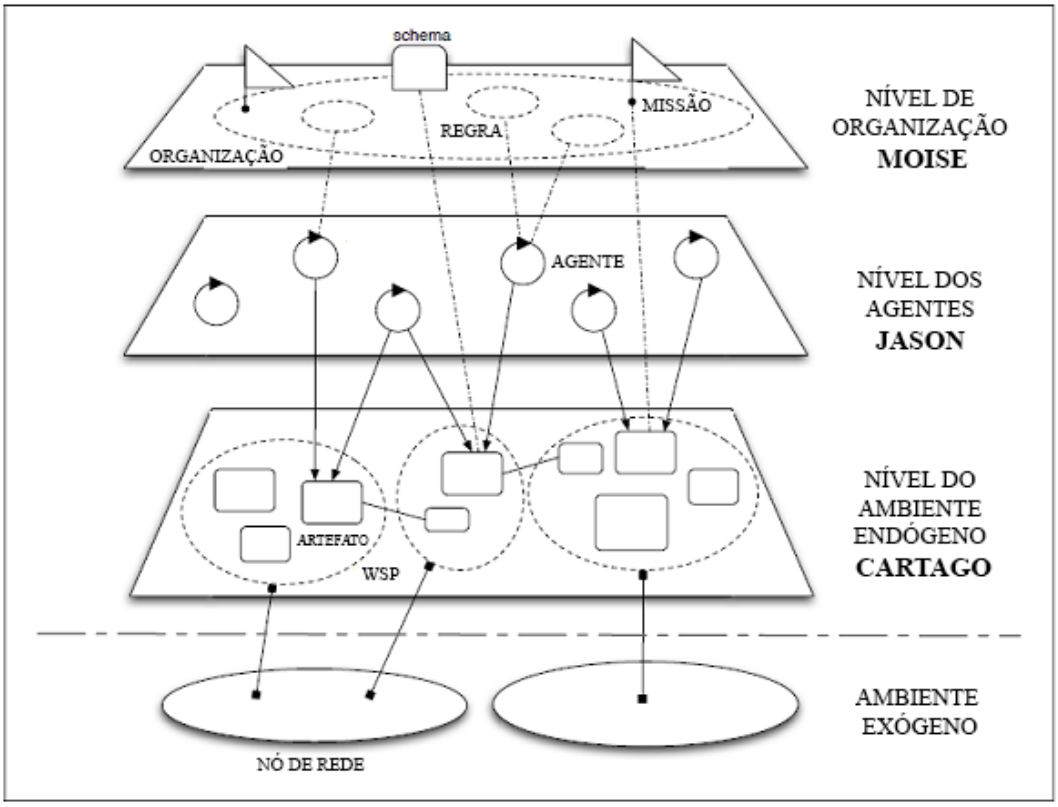
3.2.4 Agentes híbridos

Um agente híbrido é composto por características dos agentes cognitivos e reativos. Essa arquitetura foi desenvolvida com o intuito de solucionar as deficiências encontradas nos modelos cognitivos e reativos. Os agentes cognitivos possuem certa limitação quanto a velocidade de reação frente a uma situação imprevista. Os agentes reativos são incapazes de encontrar alternativas quando o estado do ambiente diverge de seus objetivos. Por isso, os agentes híbridos tentam responder rapidamente às mudanças do ambiente, levando em consideração certo conhecimento adquirido anteriormente, e possuem uma certa capacidade de armazenar informações do ambiente em que está situado (WOOLDRIDGE, 2009).

3.3 *FRAMEWORK* JACAMO

O JaCaMo é um sistema multiagente dividido em três níveis, que podem ser observados na figura 5: (i) nível de organização dos agentes autônomos BDI feita pelo *framework* de programação orientada a organização Moise; (ii) nível dos agentes, os quais são programados pela plataforma de desenvolvimento de agentes Jason; e (iii) nível do ambiente em que os agentes estão situados, onde o Cartago cria um ambiente compartilhado e distribuído baseado em artefatos (JACAMO, 2011).

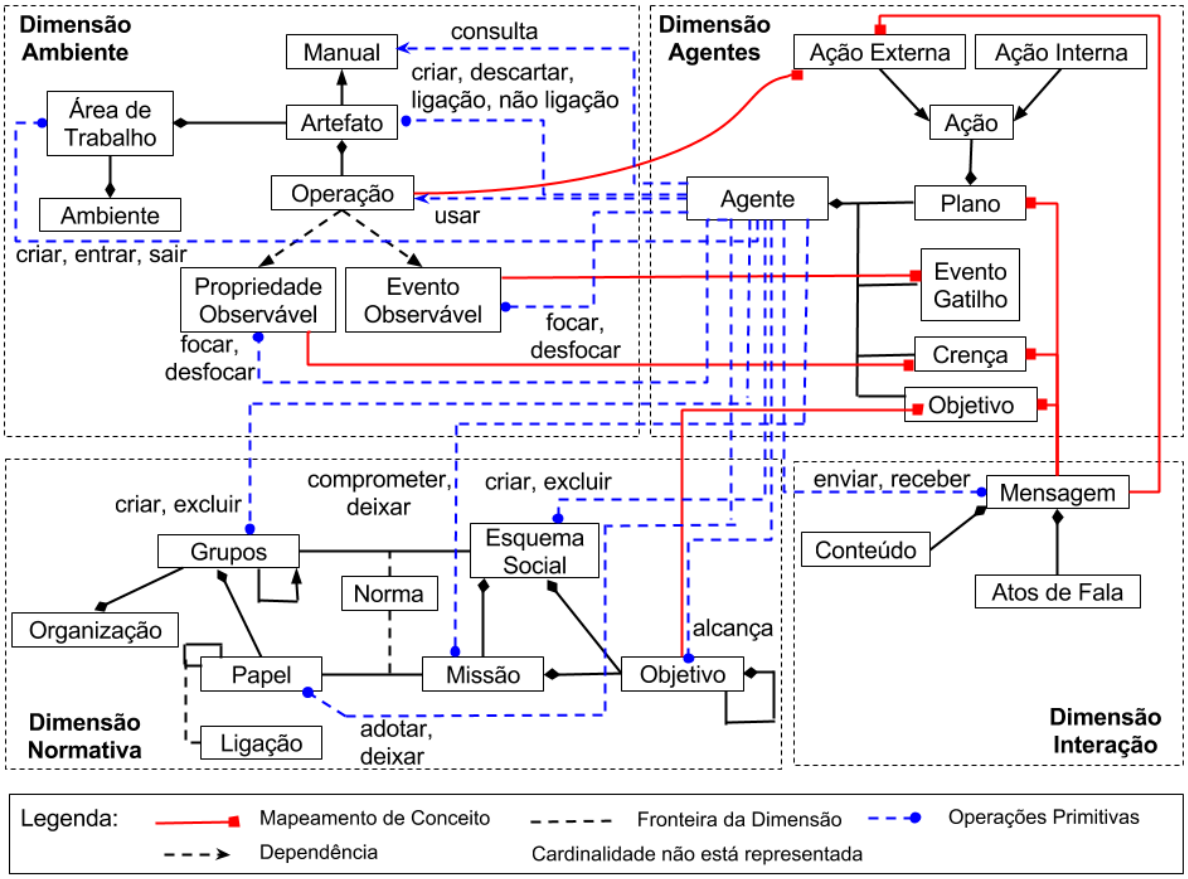
Figura 5 – Visão geral dos níveis do JaCaMo



Fonte: Adaptado de (JaCaMo Project, 2011)

Cada uma das ferramentas presentes no *framework* JaCaMo (o Jason, o Cartago e o Moise) possui um objetivo específico, bem como o modelo de abstração de seu funcionamento. Por isso, foi considerado importante a criação de um meta-modelo englobando as três ferramentas, de maneira a assegurar a integração entre elas e as dependências necessárias, como é possível observar na figura 6 (JACAMO, 2011).

Figura 6 – Meta-modelo do *framework* JaCaMo



Fonte: Adaptado de (JaCaMo Project, 2011)

As abstrações definidas pela dimensão dos agentes baseiam-se na arquitetura BDI. Um agente é composto por conjuntos de crenças, que representa o estado atual do agente e o seu conhecimento sobre o ambiente, de desejos, que contém os objetivos a serem atingidos pelo agente, e de intenções, que são as ações tomadas pelos agentes para alcançar os objetivos previamente estabelecidos. A programação dos agentes é feita pela linguagem Jason (JACAMO, 2011).

A dimensão ambiente contém diversas instâncias de ambiente Cartago, compostas por um ou mais ambientes de trabalho. Cada ambiente de trabalho é composto por um conjunto de artefatos, que são recursos e ferramentas construídos, utilizados e manipulados dinamicamente pelos agentes para realizar as atividades individuais ou coletivas (RICCI *et. al.,* 2011).

Por fim, a dimensão de organização, composta pelas dimensões normativas e interação e controlada pelo *framework* de programação orientada a organização Moise, é dividida em três especificações: (i) estrutural; (ii) funcional; e (iii) normativa (JACAMO, 2011).

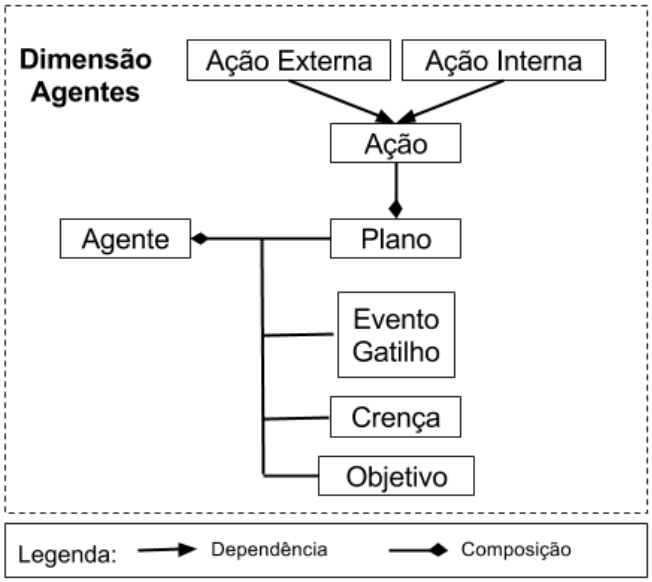
Nas seções abaixo serão detalhadas as linguagens de programação de agentes Jason e de artefatos Cartago.

3.3.1 Jason

A linguagem Jason, *Java-based interpreter for an extended version of AgentSpeak*, é uma extensão da linguagem de programação de agentes *AgentSpeak.* Esta linguagem implementa uma semântica operacional e disponibiliza uma plataforma de desenvolvimento para sistemas multiagentes customizável, além de ser uma ferramenta de código aberto distribuída sob a licença GGU LHPL (BORDINI *et al.* 2007).

Assim como o *AgentSpeak*, o modelo utilizado pelo Jason é baseado na arquitetura BDI, onde um agente possui crenças, objetivos (ou desejos) e planos (ou intenções), como é possível observar na figura 7.

Figura 7 – Dimensão agentes em destaque



Fonte: Adapatado de (JaCaMo Project, 2011)

As crenças de um agente definem o que o mesmo sabe sobre o mundo, não sendo necessariamente verdades absolutas. No código 1, é possível verificar a definição de crenças de um agente. Na linha 1 o agente crê que a quantidade máxima de vagas é 100. Já na linha 2 o agente crê que não há vagas sendo utilizadas no momento e o na linha 3 que ambiente não está cheio. A crença da linha

4 indica que uma vaga, nomeada *spotA0,* está ocupada (0 indica ocupada e 1 não ocupada), e o nome do agente a qual está estacionado, neste caso *driver07.*

Código 1 – Exemplo de definição de crenças no Jason

```
1 nSpotsMAX(100).
2 nSpotsUsed(0).
3 isFull(false).
4 spot("spotA0",1, "driver07").
```

Fonte: Autoria própria

Os objetivos de um agente em Jason definem um estado o qual o agente deve alcançar de maneira a atingir seu objetivo final. Os objetivos podem ser categorizados como objetivos a serem alcançados e objetivos de testes, precedidos em sua definição por “!” e “?”, respectivamente. No código 2 é possível observar a criação de dois objetivos: na linha 1 o objetivo *allocateSpot* para alocar uma nova vaga a um agente específico e na linha 2 o objetivo de testes *print* para visualizar a disposição das vagas.

Código 2 – Exemplo de definição de objetivos no Jason

```
1 !allocateSpot(“driver07”, 250).
2 ?print.
```

Fonte: Autoria própria

Um plano em Jason é composto em três elementos: (i) evento gatilho, (ii) contexto e (iii) corpo:

eventoGatilho : contexto <- corpo

O *evento gatilho* define as condições para que o plano seja chamado, ou a definição do cabeçalho do plano. O *contexto* informa as regras exigidas para que este plano seja executado. O *corpo* define todas as ações a serem realizadas caso este plano seja escolhido. Portanto, as ações definidas no corpo do plano serão executadas quanto o evento gatilho for acionado e as condições do contexto sejam verdadeiras.

No código 3 pode-se verificar o plano de alocação de vagas, o qual é executado quando há vagas disponíveis. Na linha 2 o agente *manager* faz a alocação da vaga e na linha 3 utiliza uma ação interna para informar ao agente *driver* a vaga alocada para o mesmo.

Código 3 – Exemplo de definição de plano no Jason

```
1  +!allocateSpot(AGENT) : isFull(CONDITION) & CONDITION = false <-
2        +spot(SPOT,1,AGENT);
3        .send(AGENT, tell, SPOT).
```

Fonte: Autoria própria

3.3.2 Cartago

O Cartago, *Common ARTifact infrastructure for Agents Open environments*, é um framework usado para o desenvolvimento de ambientes para sistemas multiagentes e que estabelece uma interface comum entre os agentes e o ambiente, portanto não é dependente de uma linguagem específica para a programação dos agentes.

Através dessa capacidade de separação entre as camadas de agente e do ambiente, o *framework* JaCaMo utiliza o Jason para a programação dos agentes e o Cartago para o desenvolvimento dos artefatos utilizados pelos agentes (CARTAGO, 2006).

O Cartago utiliza a linguagem Java para desenvolvimento dos artefatos incluindo uma biblioteca própria para realizar a comunicação com o ambiente de agentes. Além dos padrões da linguagem Java, há duas definições importantes na criação de um artefato: (i) indicação de operação e (ii) parâmetro de *feedback*.

Uma anotação de operação, definida como *@OPERATION* antes dos métodos escolhidos, permite com que estas operações estejam disponíveis para os agentes após a criação do artefato. Portanto, é possível criar operações visíveis para os agentes e operações disponíveis apenas internamente para o artefato (CARTAGO, 2006).

O parâmetro de *feedback* definido pelo Cartago permite com que um parâmetro recebido através do agente seja retornado para o mesmo com valores alterados. O código 4 demonstra a criação de um artefato contendo essas duas definições, onde o artefato recebe um parâmetro vazio o qual retorna o mesmo parâmetro contendo o agente com maior reputação (CARTAGO, 2006).

Código 4 – Definição da uma operação no Cartago

```
@OPERATION
public void freeDriver(OpFeedbackParam<Object> idDriver){
        Driver d = greatestTrust();
        idDriver.set(d.getId());
}
```

Fonte: Autoria própria

Para que um agente tenha acesso às operações disponibilizadas por um artefato, o agente deve criar este artefato, como é mostrado no código 5. Para a criação, deve-se informar o nome do artefato, sua localização (nome do pacote onde o artefato se encontra), parâmetros de inicialização e identificação do artefato para futuras referências. O comando *focus* permite com que o agente entenda que as operações do artefato compõem o seu conjunto de operações, sendo assim possível acessar as operações visíveis do artefato.

Código 5 – Criação de artefato via Jason

```
makeArtifact("a_Control", "mAPS.QueueControl", ["220"], ArtId2);
focus(ArtId2).
```

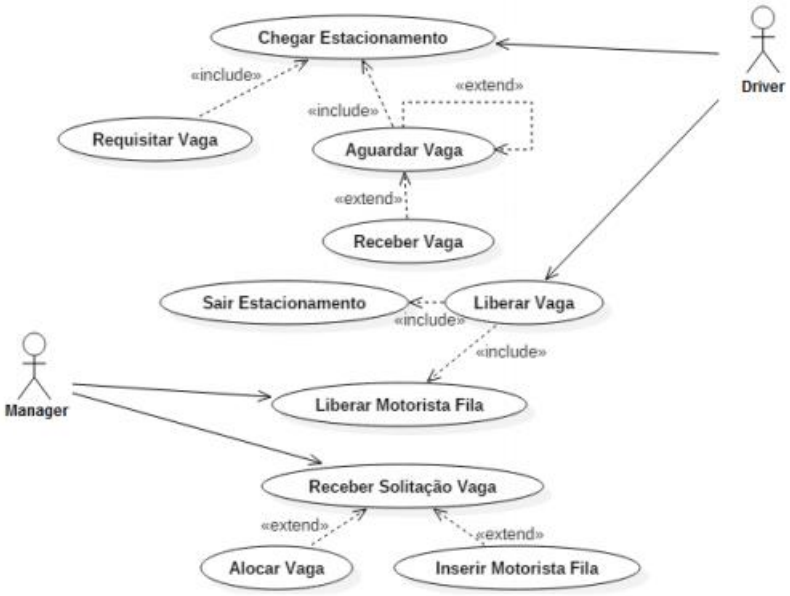
Fonte: Autoria própria

4 PROJETO MAPS

O projeto *MultiAgent Parking System* (MAPS) é desenvolvido no grupo Grupo de Pesquisa em Agentes de Software (GPAS) da UTFPR – Ponta Grossa, com o intuito de desenvolver soluções para *Smart Parking* baseadas em sistemas multiagentes, especificamente utilizando o framework JaCaMo.

No diagrama 1 é possível observar o funcionamento do modelo de alocação de vagas implementado pelo MAPS, que trabalha com dois tipos de agentes: motorista (*driver*) e um agente gerente (*manager*). O primeiro é o agente que interage no ambiente, ou seja, os motoristas que estão em busca de uma vaga para estacionar. O segundo é o agente centralizador do sistema multiagente (SMA), responsável por gerenciar as vagas do estacionamento e alocá-las aos agentes motoristas. As vagas (*spots*) são consideradas os recursos do SMA gerenciados pelo *manager* e atribuídos ao *driver* (CASTRO, 2015).

Diagrama 1 – Diagrama de caso de uso do modelo de alocação de vagas implementado pelo MAPS



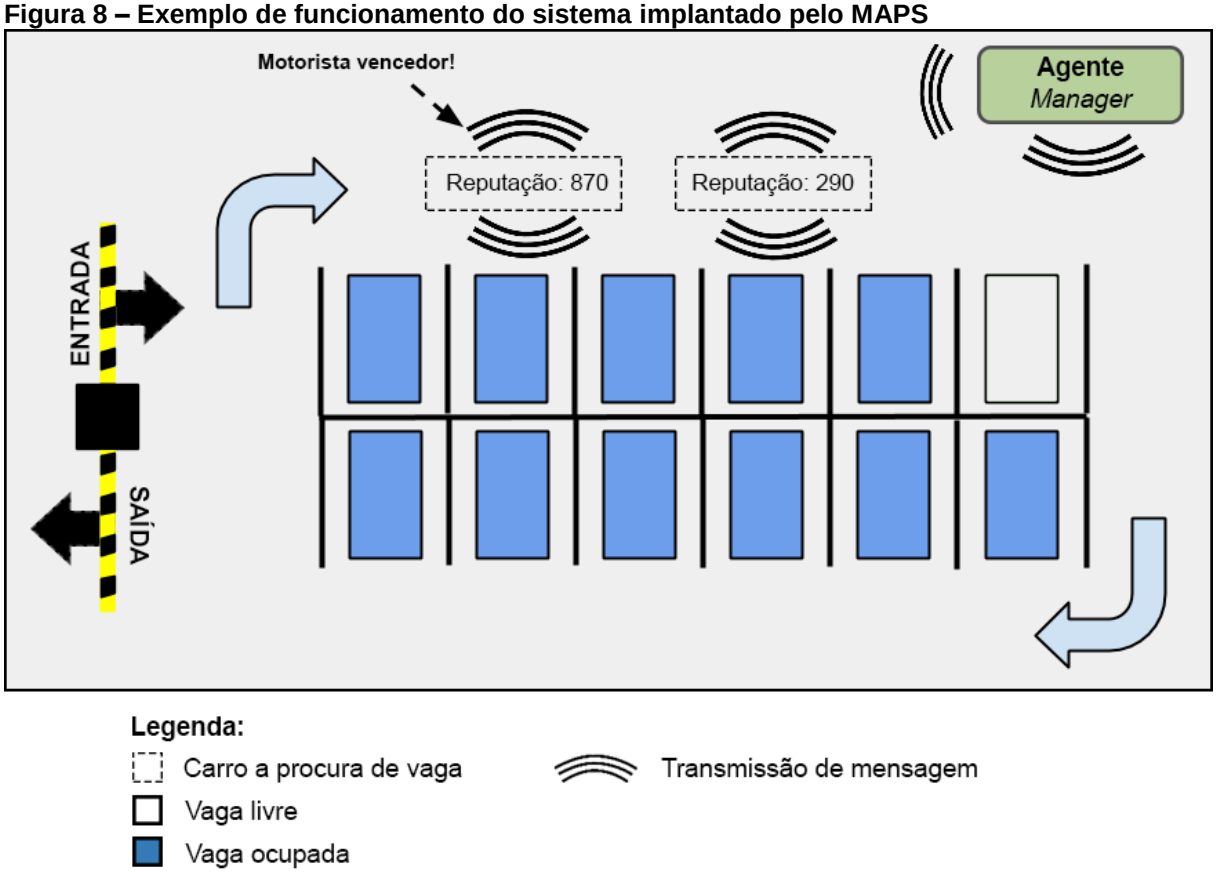
Fonte: (CASTRO, 2015)

O gerente é o responsável por manter a comunicação entre os outros agentes que estão no ambiente e também alocar as vagas para os *drivers*. Cada motorista possui uma reputação, a qual define o grau de comprometimento que o mesmo tem com o estacionamento. A reputação é calculada de acordo com a maneira com que o motorista se comporta dentro do ambiente de estacionamento.

Por exemplo, se o gerente definir uma vaga para o motorista e o mesmo não a ocupar, sua reputação será reduzida. Caso a utilize seguindo as orientações do gerente, o motorista terá sua reputação aumentada. Dessa maneira, é possível manter o ambiente melhor organizado e otimizar o funcionamento do mesmo.

A alocação de vagas é feita através do valor de reputação dos motoristas. O motorista que irá receber a vaga é aquele que possui a maior reputação entre todos os que estão na fila de espera. No momento em que o motorista deixa a vaga, o sistema retribui com uma determinada pontuação, a qual será utilizada para calcular e, consequentemente, aumentar ou reduzir a sua reputação. Na figura 8 é possível observar o funcionamento do sistema.

Figura 8 – Exemplo de funcionamento do sistema implantado pelo MAPS



Fonte: Adaptado de (CASTRO, 2015)

4.1.1 Estrutura original do projeto MAPS

O MAPS é baseado na estrutura do *framework* JaCaMo, onde os agentes são programados em Jason e os artefatos em Cartago. O estado inicial do sistema

multiagente é definido no arquivo de execução *.jcm*, onde são criados os agentes *manager* e *drivers* contendo suas crenças iniciais.

4.1.2 Agente driver

Os agentes do tipo *driver* possuem três crenças iniciais: reputação (*myTrust)*, tempo a ser gasto (*timeToSpend*) e tempo de chegada (*timeToArrive*) ao estacionamento, em milissegundos. A definição do *driver* é demonstrada no código 6.

Código 6 – Definição do agente *driver*

```
agent d1: driver.asl {
        beliefs: myTrust(87)
                 timeToSpend(12000)
                 timeToArrive(1000)
}
```

Fonte: Projeto MAPS

Este agente possui quatro planos: (i) chegar ao estacionamento, (ii) solicitar uma vaga, (iii) estacionar e (iv) deixar o estacionamento. O primeiro plano, demonstrado no código 7, é baseado no tempo de chegada (*timeToArrive)* definido em suas crenças iniciais. Portanto o agente aguarda o tempo estipulado e então faz a solicitação da vaga. A partir desse momento ele obtém a crença de que chegou ao estacionamento.

Código 7 – Plano *arriveParking* do projeto MAPS

```
+!arriveParking : timeToArrive(timeToArrive)<-
                .wait(timeToArrive);
                !requestSpot;
                +arrivalParking.
```

Fonte: Projeto MAPS

O segundo plano, descrito no código 8, é acionado pelo plano anterior e é utilizado para solicitar uma nova vaga ao agente *manager*. O motorista encaminha uma mensagem ao *manager* solicitando a alocação de uma vaga com base em sua reputação.

Código 8 – Plano *requestSpot* do projeto MAPS

```
+!requestSpot : myTrust(MT) <-
        .send(manager,achieve,requestSpot(MT)).
```

Fonte: Projeto MAPS

O terceiro plano, referente a estacionar em uma vaga definida, é acionado pelo agente *manager* quando o mesmo aloca uma vaga para o *driver*. O agente *driver* passa a acreditar que a vaga definida pertence a ele e aguarda o tempo a ser gasto no estacionamento para então deixar o ambiente. O plano *park* é demonstrado no código 9.

Código 9 – Plano *park* do projeto MAPS

```
+!park(S)[source(AGENT)] : spotOk & arrivalParking & timeToSpend(TS) <-
                +spot(S);
                .wait(TS);
                !leaveSpot.
```

Fonte: Projeto MAPS

No último plano, demonstrado no código 10, o agente *driver* envia uma mensagem ao *manager* informando-o que está deixando o estacionamento e qual vaga será liberada. Por fim, o agente desacredita que a vaga pertence a ele.

Código 10 – Plano *leaveSpot* do projeto MAPS

```
+!leaveSpot : spot(S) <-
        .send(manager,achieve,leaveSpot(S));
        -spot(S).
```

Fonte: Autoria própria

4.1.3 Agente manager

O agente *manager* é responsável pelo o controle do estacionamento, realizando a alocação e desalocação de vagas e controle da fila de espera. As crenças iniciais do *manager*, mostradas no código 11, são: quantidade de vagas total (*nSpotsMax*) e utilizadas (*nSpotsUsed*), porcentagem de uso do estacionamento (*pFull*) e informação se o estacionamento está cheio ou não (*isFull*). Além disso, o *manager* também possui uma crença para cada vaga existente, onde é armazenado o nome da vaga, um indicador de ocupação da vaga e a identificação

do *driver* que está estacionado. Caso a vaga esteja desocupada, é informado *EMPTY* no local da identificação do *driver.*

Código 11 – Exemplo de crenças iniciais do agente *manager*

```
nSpotsMAX(160).
nSpotsUsed(0).
isFull(false).
pFull(0).

spot("spotA0",0, "EMPTY").

...

spot("spotB19",0, "EMPTY").
```

Fonte: Projeto MAPS

O primeiro objetivo do *manager* é abrir o estacionamento, que consiste em criar os artefatos de controle do portão e de fila de espera. A configuração do mesmo é demonstrada no código 12. A partir deste momento o estacionamento está pronto para receber novos clientes.

Código 12 – Exemplo de configuração do estacionamento

```
+!setupParking <-
        makeArtifact("a_Gate", "mAPS.Gate", ["Starting"], ArtId);
        focus(ArtId);

        makeArtifact("a_Control", "mAPS.QueueControl", ["20"], ArtId2);
        focus(ArtId2).
```

Fonte: Projeto MAPS

Existem dois tipos de plano de solicitação de vaga: diretamente ou através da lista de espera. Ambas requerem a identificação do agente e a sua reputação. A diferença entre elas, é que a primeira é solicitada diretamente pelo *driver* e a segunda é chamada ao liberar uma vaga quando o estacionamento se encontra cheio. Como é mostrado no código 13, ambos os casos chamam o plano para alocar uma nova vaga.

Código 13 – Planos de solicitação de vaga

```
+!requestSpot(BACKGROUND)[source(AG)] <-
        .term2string(AG,AGENT);
        !allocateSpot(AGENT,BACKGROUND).

+!requestSpotQueue(AGENT,BACKGROUND) <-
        !allocateSpot(AGENT,BACKGROUND).
```

Fonte: Projeto MAPS

O plano de alocação de vaga, demonstrado no código 14, também segue dois caminhos: alocação da vaga para um agente ou inserção do agente na fila de espera. O primeiro caso encontra a vaga livre mais próxima da entrada, faz a reserva da vaga, abre o portão e encaminha uma mensagem para o *driver* solicitando que o mesmo estacione na vaga alocada. Além disso, o plano fecha o portão e atualiza as suas crenças. Já o segundo caso é chamado quando o estacionamento se encontra lotado, portanto o *manager* insere o motorista na fila de espera.

Código 14 – Plano de alocação de vagas

```
+!allocateSpot(AGENT,BACKGROUND) : nSpotsUsed(N) & nSpotsMAX(MAX) &
                                   isFull(COND) & pFull(P) & COND = false <-
        +~find;
        for(spot(S,C,A)){
                if(A = "EMPTY" & ~find & (COND = false)){
                        -spot(S,C,A); +spot(S,1,AGENT);
                        openGate;
                        .send(AGENT,tell,spotOk);
                        .send(AGENT,achieve,park(S));
                        -nSpotsUsed(N); +nSpotsUsed(N+1);
                        if((N+1) = MAX){
                                -isFull(COND);
                                +isFull(true);
                        };
                        -pFull(P); +pFull(((N+1) * 100) / MAX);
                        closeGate;
                        -~find;
                }
        };
        +~find.
+!allocateSpot(AGENT,BACKGROUND) : isFull(COND) & COND = true <-
                insertDriverQueue(AGENT,BACKGROUND).
```

Fonte: Projeto MAPS

O plano para um veículo deixar o estacionamento, demonstrado no código 15, é acionado pelo agente *driver,* que deve informar qual a vaga estava alocada. O

manager atualiza suas crenças e elimina o agente do ambiente. Então há mais uma chamada para o plano que verifica se há motoristas na fila.

Código 15 – Plano *leaveSpot*

```
+!leaveSpot(S)[source(AG)] : nSpotsUsed(N) & nSpotsMAX(MAX) &  isFull(COND)
&  pFull(P) <-

        .term2string(AG,AGENT);
        -nSpotsUsed(N);
        +nSpotsUsed(N-1);

        -isFull(COND);
        +isFull(false);

        +spot(S,0,"EMPTY");
        -spot(S,1,AGENT);

        .kill_agent(AGENT);

        !checkQueue.
```

Fonte: Autoria própria

Por fim, há o plano o qual verifica se há motoristas na fila através do artefato *QueueControl*. Este artefato controla toda a fila de espera e também retorna qual o veículo deve deixar a fila de espera, por meio do tempo de espera e da reputação. Quando há veículos na fila, o *manager* faz uma chamada ao plano de solicitar vagas através da fila de espera. Este plano é demonstrado no código 16.

Código 16 – Plano para verificar fila de espera

```
+!checkQueue : nSpotsUsed(N) & isFull(COND)<-
        isAnyone(C);
        if(C = false){
                freeDriver(AG,BG);
                !requestSpotQueue(AG,BG);
        }else{
                .print("Nobody at queue");
        }.
```

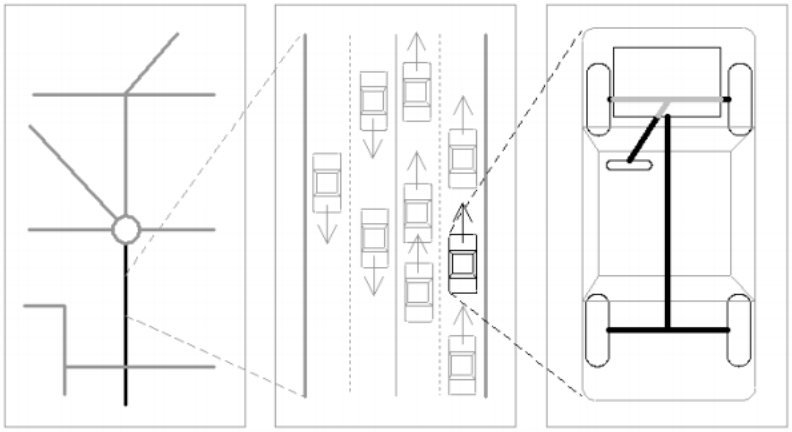
Fonte: Projeto MAPS

5 FERRAMENTAS DE SIMULAÇÃO DE TRÂNSITO

A simulação de ambientes virtuais, ou reais, é uma maneira importante de validar projetos antes de sua execução em busca de resultados mais precisos. Por isso, na área de infraestrutura de trânsito, existem diversas ferramentas, citadas neste capítulo, que possibilitam a simulação desses ambientes, para analisar o seu comportamento de acordo com agentes que interagem neste ambiente.

Há três tipos de simulação: macroscópica, microscópica e sub-microscópica (Figura 9), sendo que o objetivo deste trabalho é utilizar a simulação microscópica. A simulação microscópica foi escolhida por ter uma visão mais apropriada para a simulação de ambientes de estacionamento. O Vissim, ITSUMO, e SUMO são algumas das ferramentas microscópicas existentes no mercado de simulação, sendo que a última será utilizada neste trabalho.

Figura 9 – Tipos de simulação: (a) macroscópica (b) microscópica (c) sub-microscópica.



(a) (b) (c)

Fonte: (KRAJZEWICZ *et al.* 2002)

O Vissim foi desenvolvido pelo grupo PTV da Alemanha e, além de trabalhar com simulação microscópica, também possibilita visualizar a simulação em 3D. Porém, por ser uma ferramenta proprietária, é inviável sua utilização no projeto.

Já o ITSUMO é uma ferramenta desenvolvida no Brasil pelo laboratório de sistemas multiagentes (MASLAB) da Universidade Federal do Rio Grande do Sul. Pelo fato de ser uma ferramenta com o suporte e desenvolvimento mais restritos, o ITSUMO é limitado em relação as funcionalidades quando comparado ao SUMO (KOKKINOGENIS *et al.* 2011).

Por ser uma ferramenta de código aberto, o SUMO atraiu diversos contribuidores ao redor do mundo que auxiliaram no aprimoramento da mesma. A ferramenta se tornou repleta de recursos de modelagem que permitem ao usuário modelar ambientes personalizados, complexos e dinâmicos.

Na seção 5.1 será apresentada em detalhes a ferramenta de simulação de trânsito SUMO, adotada para o desenvolvimento do presente trabalho.

5.1 SUMO

O SUMO é uma ferramenta de simulação desenvolvida pelo Instituto de Transporte da Alemanha com o intuito de aprimorar os resultados finais de um projeto e suas análises, além de facilitar os testes de novos algoritmos que são desenvolvidos nessa área (KRAJZEWICZ *et al.* 2006). A ferramenta foi desenvolvida em 2000, porém disponibilizada desde 2001, permitindo aos usuários modelar sistemas de tráfego intermodal (tráfego que envolve diversos tipos de transporte, como pedestres, carros e bicicletas circulando no mesmo ambiente). Além de possuir suporte nativo à simulação de rodovias, transporte público, pedestres, entre outros, é possível também desenvolver modelos personalizados, principalmente por ser uma ferramenta de código livre.

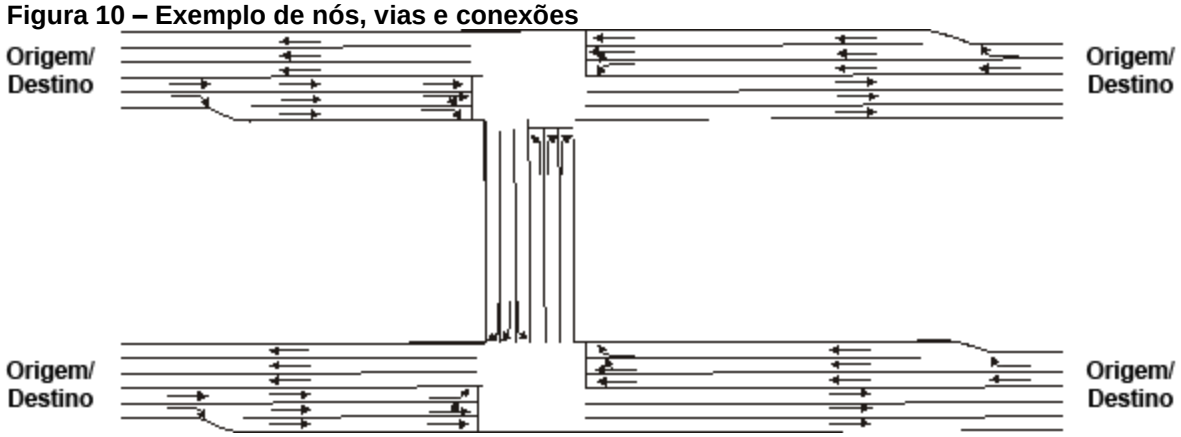
O simulador SUMO oferece diversas funcionalidades ao usuário, como simulação microscópica (veículos, pedestres, bicicletas, transporte público, por exemplo); controle de tempo de semáforos (gerados automaticamente pela ferramenta ou importado de outros modelos); importação de ambientes reais através da ferramenta NETCONVERT em conjunto com *OpenStreetMaps*; o TraCI, que permite obter valores da simulação e manipulá-los em tempo real, dentre muitas outras possibilidades.

A simulação exige como entrada as rotas (*routes*) e uma rede (*network*), que é o ambiente onde a simulação ocorre, descrita em arquivo de formato XML. Uma rede é composta por nós (*nodes* ou cruzamentos), vias (*edges*), tipos (*types*), tanto de vias como de veículos, e conexões (*connections*) entre as faixas de uma via.

Como pode-se observar no exemplo da figura 10, os nós definem os pontos de origem e/ou destino possíveis em uma rota. Essas rotas são compostas por ruas que podem ter uma ou mais faixas. O encontro entre duas ou mais ruas é controlado

por cruzamentos (ou junções). As setas ilustram o sentido permitido das vias e também as possíveis trocas de faixas ou conversões.

Figura 10 – Exemplo de nós, vias e conexões



Fonte: Adaptado de SUMO Wiki

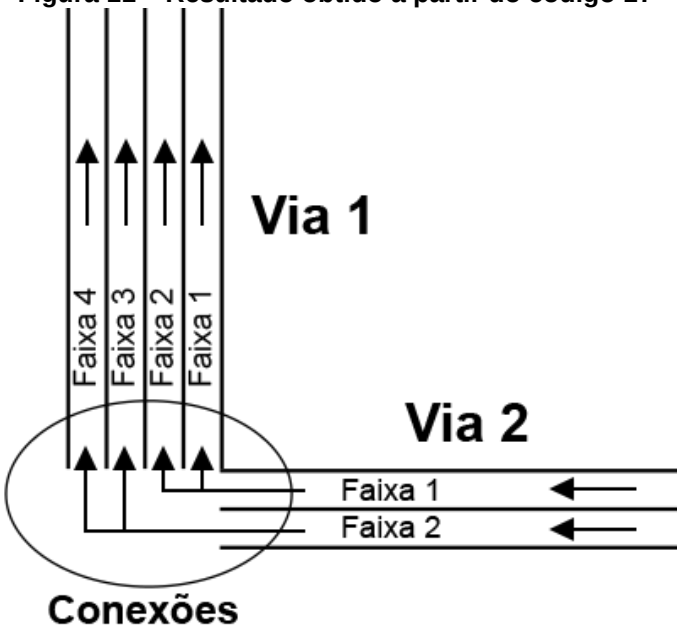
Observa-se, no código 17, a criação de nós, tipos, vias e conexões. Um nó é composto por coordenadas cartesianas. As vias possuem um nó de origem e um de destino, além de ser possível classificá-las por tipos. Esses tipos informam a quantidade de faixas, velocidade máxima e a prioridade das vias. As conexões definem possíveis trocas de faixas entre as vias. É necessário informar a via e faixa de origem e destino para defini-las. O resultado é possível observar na figura 11.

Código 17 – Exemplo de criação de nós, tipos, vias e conexões

```
<nodes>
  <node id="1" x="100.0" y="50.0" />
  <node id="2" x="100.0" y="150.0" />
  <node id="3" x="200.0" y="50.0" />
</nodes>
<types>
  <type="a" priority="3" numLanes="4" speed="70" />
  <type="b" priority="2" numLanes="2" speed="40" />
</types>
<edges>
  <edge id="via1" from="1" to="2" type="a" />
  <edge id="via2" from="3" to="1" type="b" />
</edges>
<connections>
  <connection from="via2" to="via1" fromLane="1" toLane="1" />
  <connection from="via2" to="via1" fromLane="1" toLane="2" />
  <connection from="via2" to="via1" fromLane="2" toLane="3" />
  <connection from="via2" to="via1" fromLane="2" toLane="4" />
</connections>
```

Fonte: Adaptado de SUMO Wiki

Figura 11 – Resultado obtido a partir do código 17



Fonte: Autoria própria

As rotas englobam a criação de características físicas dos veículos, as rotas que os carros podem percorrer e a criação do veículo em si. A primeira parte define diversos tipos de veículos que possam ser usados no modelo, como carros e ônibus. Esses tipos podem ser especificados pela aceleração (*accel*), desaceleração (*decel*), tamanho (*length*), velocidade máxima (*maxSpeed*), a imperfeição de condução do veículo (*sigma*), variando de 0 a 1 e a cor (*color*). Os tipos são responsáveis por definir diferentes modelos de veículos ou vias. Todos os atributos referentes a velocidade do veículo são calculados em metros por segundo (m/s).

Nó código 18 podemos observar os atributos necessários para a criação das rotas, onde deve-se informar a sequência de vias pelas quais a rota é composta (*edges*). Além disso, uma rota pode possuir um local de parada (*stop*), definido por qual faixa o veículo estacionará e até que momento ficará estacionado (*until*). O atributo *parking* define se o veículo estaciona no meio da faixa ou ao lado.

Os veículos podem ser criados individualmente ou através de um fluxo (*flow*) de carros. Os veículos individuais são compostos por um tipo (*type*), qual a rota a ser seguida (*route*), momento de partida (*depart*) e uma cor específica (*color*). O fluxo de carros deve conter o tempo em que o fluxo se inicia (*begin*), o intervalo em que os carros são inseridos no ambiente (*period*), a quantidade de veículos (*number*), a rota a qual o fluxo pertence (*route*) e a posição a qual os carros são inseridos na rota (*departPos*).

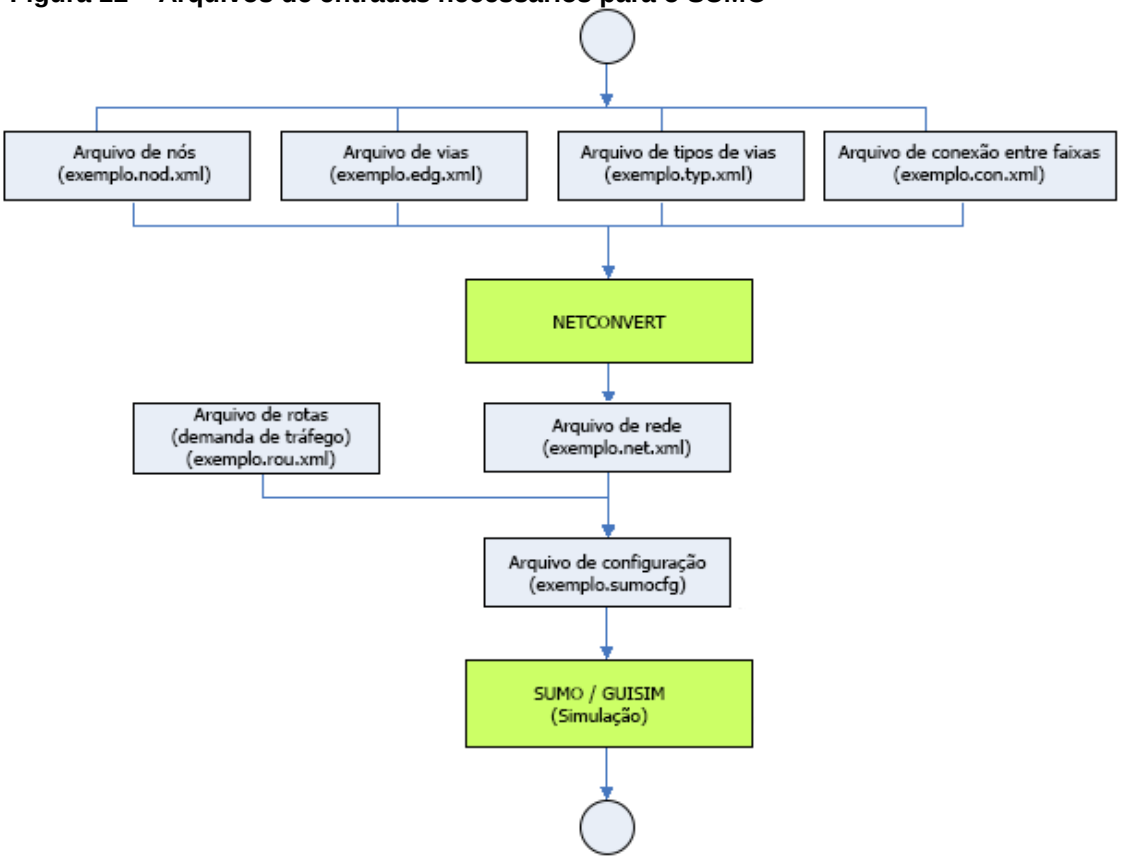
Código 18 – Exemplo de criação de rotas

```
<routes>
  <vType id="type1" accel="0.8" decel="4.5" sigma="0.5" length="5"
maxSpeed="70" color="1,1,0" />
  <route id="route0" edges="inicio meio fim" />
  <route id="route0_estacionamento" edges="inicio meio fim">
      <stop lane="meio_0" until="100" parking="false"/>
  </route>
  <vehicle id="0" type="type1" route="route0" depart="0" color="1,0,0" />
  <flow begin="0" departPos="free" id="carRight" period="1" number="70"
route="routeRight" type="car" />
</routes>
```

Fonte: Adaptado de SUMO Wiki

O SUMO possui uma ferramenta denominada NETCONVERT, a qual é usada para transformar os arquivos de nós, vias, tipos e conexões em uma rede pronta para receber fluxos de carros. A partir do arquivo de redes e de rotas é gerado um arquivo de configuração utilizado como entrada, como mostra a figura 12.

Figura 12 – Arquivos de entradas necessários para o SUMO



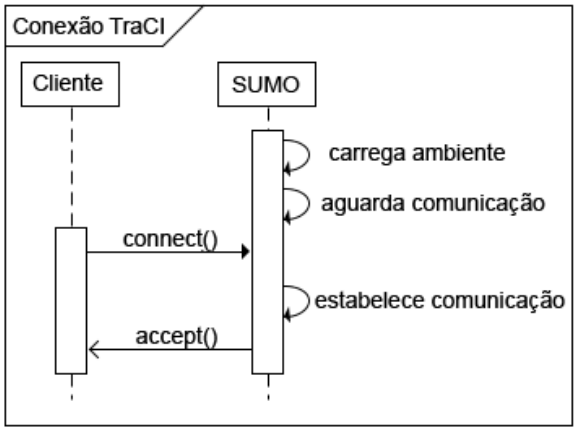
Fonte: Adaptado de SUMO wiki

Para tornar essa simulação mais precisa, a ferramenta NETCONVERT permite a conversão de redes de outras ferramentas de simulação e também do *OpenStreetMaps* (OSM), um projeto de código aberto que tem como objetivo construir um mapa mundial completo através de colaboradores do mundo todo.

5.1.1 TraCI

O *Traffic Control Interface* (TraCI) é um protocolo de comunicação que disponibiliza para qualquer ferramenta obter informações de objetos simulados no SUMO e também os manipule em tempo real. O TraCI utiliza a arquitetura TCP/IP baseada em cliente-servidor para disponibilizar acesso à simulação. O SUMO age como servidor, ativo em uma porta remota e aguardando comunicação de uma aplicação externa, como pode-se observar na figura 13 (WEGENER *et al.* 2008).

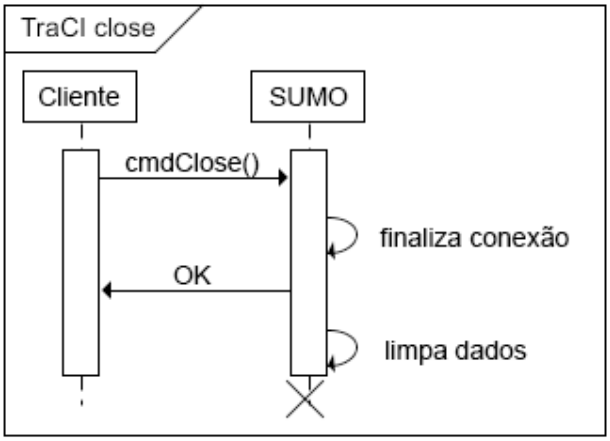
Figura 13 – Arquitetura de comunicação entre aplicação externa e SUMO via TraCI



Fonte: Adaptado de SUMO wiki

A aplicação externa envia comandos para controlar a simulação no SUMO, como alterar o comportamento de um veículo, obter informações de um veículo ou do ambiente em geral, adicionar e/ou remover veículos, entre outros. O cliente deve controlar o avanço da simulação passo a passo, caso contrário a mesma fica paralisada. Além de disparar a simulação, o cliente também é responsável por finalizar a conexão com o SUMO, como mostra a figura 14 (WEGENER *et al.* 2008).

Figura 14 – Diagrama de finalização de conexão com o SUMO



Fonte: Adaptado de SUMO wiki

A biblioteca TraCI4J, a qual será descrita na seção 6.2.2, utiliza o protocolo de comunicação TraCI para se comunicar e trocar informações com o SUMO. O próximo capítulo apresenta a interligação entre a ferramenta de simulação SUMO e o projeto MAPS, bem como a utilização da biblioteca TraCI4J.

6 INTERLIGAÇÃO ENTRE SUMO E PROJETO MAPS

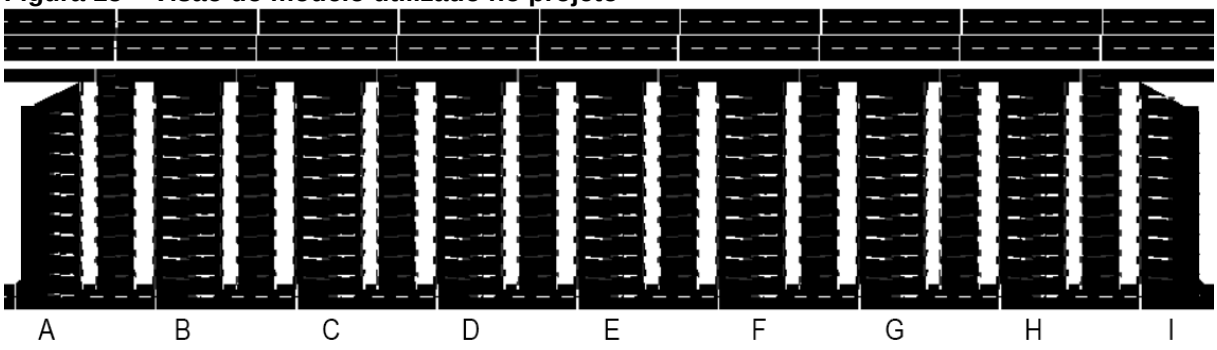
Neste capítulo serão apresentados os passos realizados para o desenvolvimento prático deste projeto, que inclui a modelagem do ambiente de estacionamento e a interligação entre a ferramenta SUMO e o projeto MAPS.

6.1 MODELO DE ESTACIONAMENTO

De modo a auxiliar no desenvolvimento do projeto foi feito um estudo de caso utilizando o modelo criado pela ferramenta *CityMobil*, a qual simula um ambiente de estacionamento semelhante a um aeroporto, com algumas adaptações (BEHRISCH *et al.*, 2011).

Faz-se necessário destacar que o modelo do *CityMobil* serve apenas para fins de testes da interligação do SUMO com o projeto MAPS. O modelo, que pode ser observado na figura 15, conta com 160 vagas divididas em 9 setores: A, B, C, D, E, F, G, H e I.

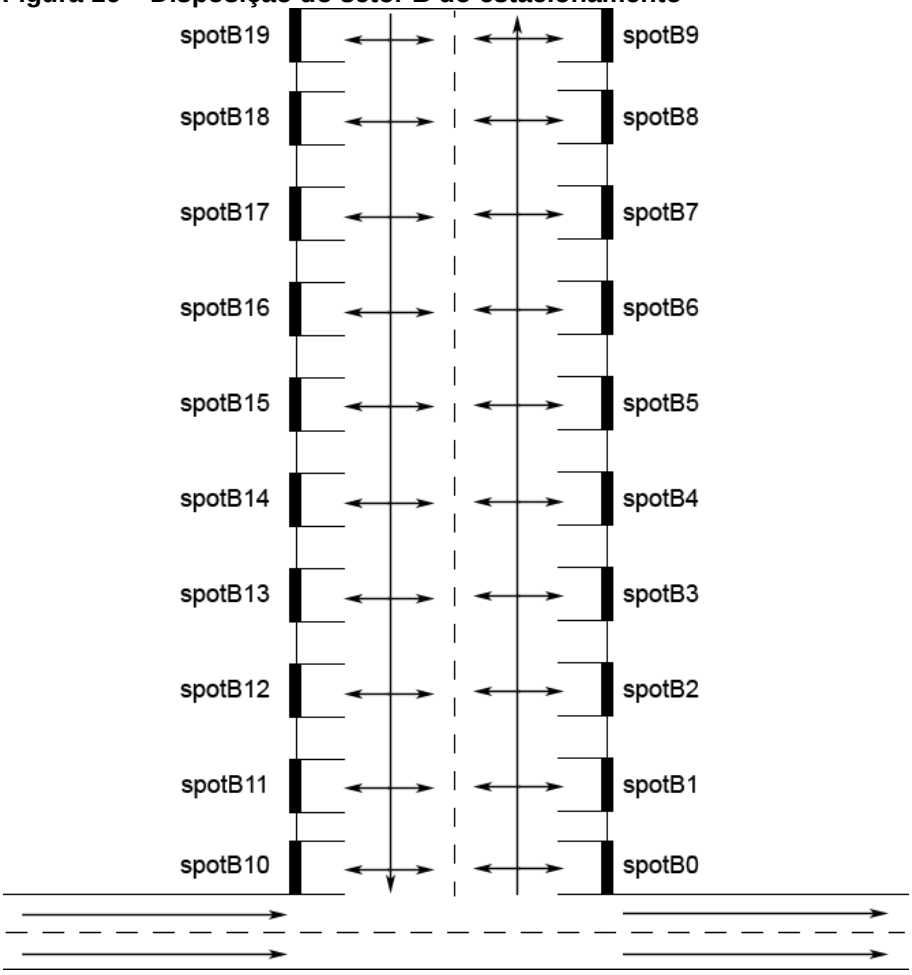
Figura 15 – Visão do modelo utilizado no projeto



Fonte: Adaptado de (*CityMobil*)

Os setores A e I possuem 10 vagas cada e os setores B, C, D, E, F, G e H possuem 20 vagas cada. As vagas são posicionadas nas extremidades esquerda e direita de cada setor e o fluxo do veículo acontece entre elas. Cada vaga é referenciada de acordo com o setor em que se encontra e o número da mesma, por exemplo: *spotB10* é a vaga número 10 do setor B. Na figura 16, é possível observar a disposição de vagas do setor B do estacionamento.

Figura 16 – Disposição do setor B do estacionamento



Fonte: Autoria própria

Todas as rotas do modelo foram definidas no arquivo de rotas do SUMO. Para cada vaga existente, há uma rota de chegada e uma de saída. As rotas de chegada são referenciadas pelos mesmos nomes das vagas, ou seja, a rota *spotB10* refere-se à rota de chegada a vaga número 10 do setor B. Deste modo, o redirecionamento do agente *driver* ao seu destino é simplificado, pois a rota de chegada já se encontra vinculada a esta vaga. As rotas de saídas são referenciadas pelos nomes das vagas seguidos pelo sufixo *_out*, indicador de saída do estacionamento, por exemplo, *spotB10_out*.

No código 19 é possível observar a rota de entrada para a vaga *spotB10.* Essa rota percorre as vias: *mainin, main0, road0-1-0* e *slot0-0r*, onde a última é a posição de parada do veículo. O atributo *until* define que o veículo ficará estacionado até o tempo 999999999, pois a simulação aguarda uma mensagem do MAPS informando que o veículo sairá do estacionamento. Ao receber essa mensagem, o veículo assume a rota *spotB10_out* e então deixa o estacionamento.

Código 19 – Rotas de entrada e saída da vaga 10 do setor B

```
<route id="spotB10" edges="mainin main0 road0-1-0 slot0-0r">
        <stop lane="slot0-0r_0" until="999999999" parking="false"/>
</route>
<route id="spotB10_out" edges="slot0-0r -slot0-0r -road0-1-0 main1 main2 main3 main4 main5 main6 main7 mainout" />
```

Fonte: Autoria própria

Como o objetivo do projeto MAPS e deste presente trabalho ainda não especificam diferentes tipos de veículos, para a simulação será utilizado apenas um tipo de veículo, definido como *car.* Este tipo de veículo possui uma aceleração de 0.8 m/s, desaceleração de 4.5 m/s, sigma 0.5, tamanho 5 e velocidade máxima de 5 m/s. A criação deste tipo é demonstrada no código 20.

Código 20 – Tipo *car* utilizado na simulação

```
<vType id="car" accel="0.8" decel="4.5" sigma="0.5" length="5" maxSpeed="5"/>
```

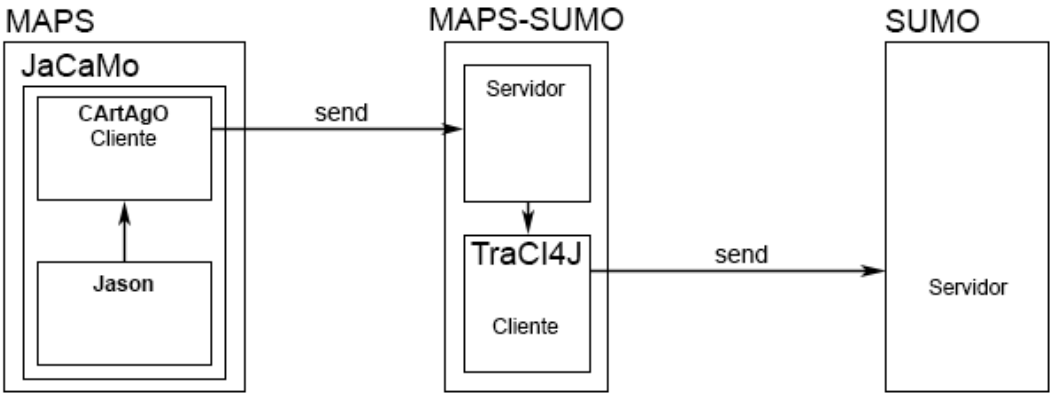
Fonte: Autoria própria

6.2 INTERLIGAÇÃO

A interligação entre o projeto MAPS e a ferramenta SUMO dá-se através da criação de uma arquitetura cliente-servidor. Para isso, foi construído o *Middleware* MAPS-SUMO, desenvolvido na linguagem de programação Java e que trabalha como intermediário entre o MAPS e o SUMO. O MAPS trabalha como cliente e o *middleware* como servidor. No *Middleware* está integrada a biblioteca TraCI4J, baseado no TraCI, responsável pela comunicação com o SUMO. No restante deste documento será utilizado o termo MAPS-SUMO para designar o *Middleware* MAPS-SUMO.

Portanto, o MAPS-SUMO funciona como servidor, aguardando mensagens enviadas pelo projeto MAPS, e também como cliente, tratando as mensagens recebidas e encaminhando para o simulador SUMO. Na figura 17 é possível observar a visão geral da comunicação entre as aplicações.

Figura 17 – Visão geral da interligação entre o projeto MAPS e o SUMO

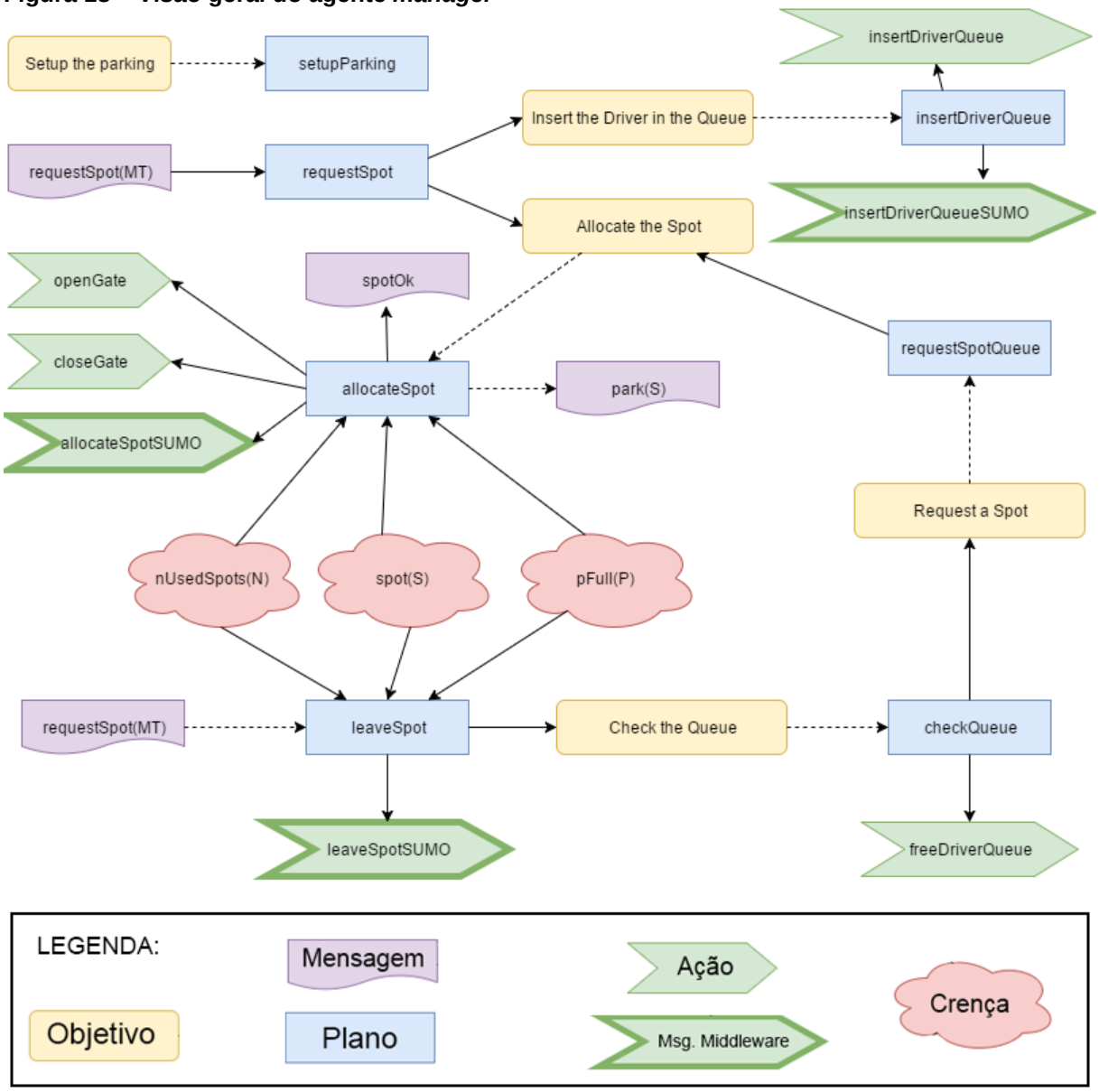


Fonte: Autoria própria

A comunicação do projeto MAPS com o MAPS-SUMO é feita por meio de um artefato em Cartago e, devido ao fato do mesmo ser programado em Java, isto facilita a comunicação entre os dois. O artefato é acessado por meio do agente *manager* criado em Jason.

Esse artefato contém as três principais operações as quais são necessárias para estabelecer a comunicação com a simulação no momento da execução: (i) alocar uma vaga (*allocateSpotSumo)*, (ii) deixar o estacionamento (*leaveSpotSumo*) e (iii) inserir motorista na fila de espera (*insertDriverQueueSumo)*. Essas operações são executadas pelo agente *manager*, o qual controla todas as requisições feitas pelos agentes *drivers*. Uma visão geral do agente *manager* pode ser observada na figura 18, com destaque para as chamadas ao artefato de integração com o MAPS-SUMO criadas por este projeto: *allocateSpotSumo, leaveSpotSumo* e *insertDriverQueueSumo.*

Figura 18 – Visão geral do agente *manager*



Fonte: Adaptado de (CASTRO, 2015)

No momento em que o agente *manager* executa uma das três operações citadas anteriormente, o mesmo dispara um chamado para o artefato criado, enviando como parâmetro a identificação do agente participante da operação, e, se necessário, a vaga a qual o agente irá ocupar ou desocupar. O artefato abre uma comunicação com o servidor MAPS-SUMO e encaminha um protocolo. Este protocolo foi desenvolvido com o intuito de padronizar e simplificar a comunicação entre as ferramentas. O protocolo pode ser definido como:

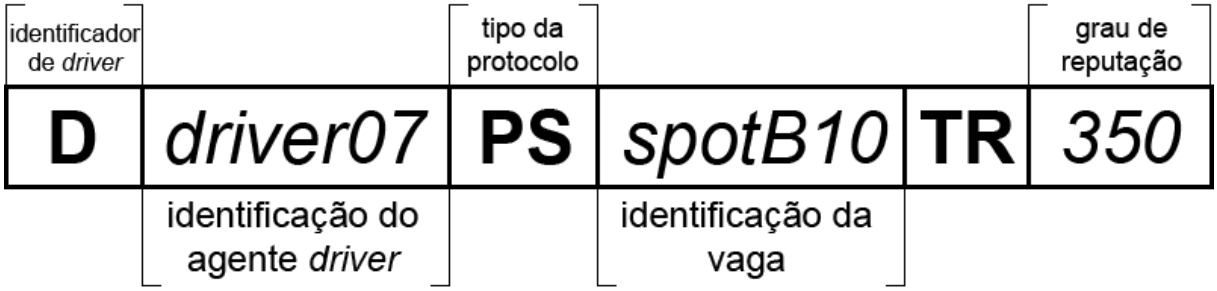
Protocolo ::= <idDriver, nomeDriver, tipoProtocolo, idSpot, trustDriver>

onde,

- *idDriver*: é um campo identificador do agente *driver*, sempre preenchido com a letra D`;
- *nomeDriver*: identificação do agente *driver;*
- tipoProtocolo: tipo do protocolo, que está relacionado com a operação em que ele foi gerado: PS (*Parking Spot*) para alocação de vaga, LS (*Leaving Spot*) para agentes deixando o estacionamento e QU (*Queue*) para inserir um motorista na fila de espera;
- *idSpot*: identificação da vaga a ser ocupada ou desocupada. Caso o protocolo seja do tipo QU, este campo é ignorado;
- *trustDriver*: grau de reputação do agente. Necessário apenas para protocolos referentes à alocação de vaga.

Na figura 19 é possível observar um exemplo de um protocolo de comunicação pronto, o qual representa a alocação da vaga *spotB10* para o agente *driver07*.

Figura 19 – Exemplo de protocolo de comunicação entre MAPS e *Middleware*



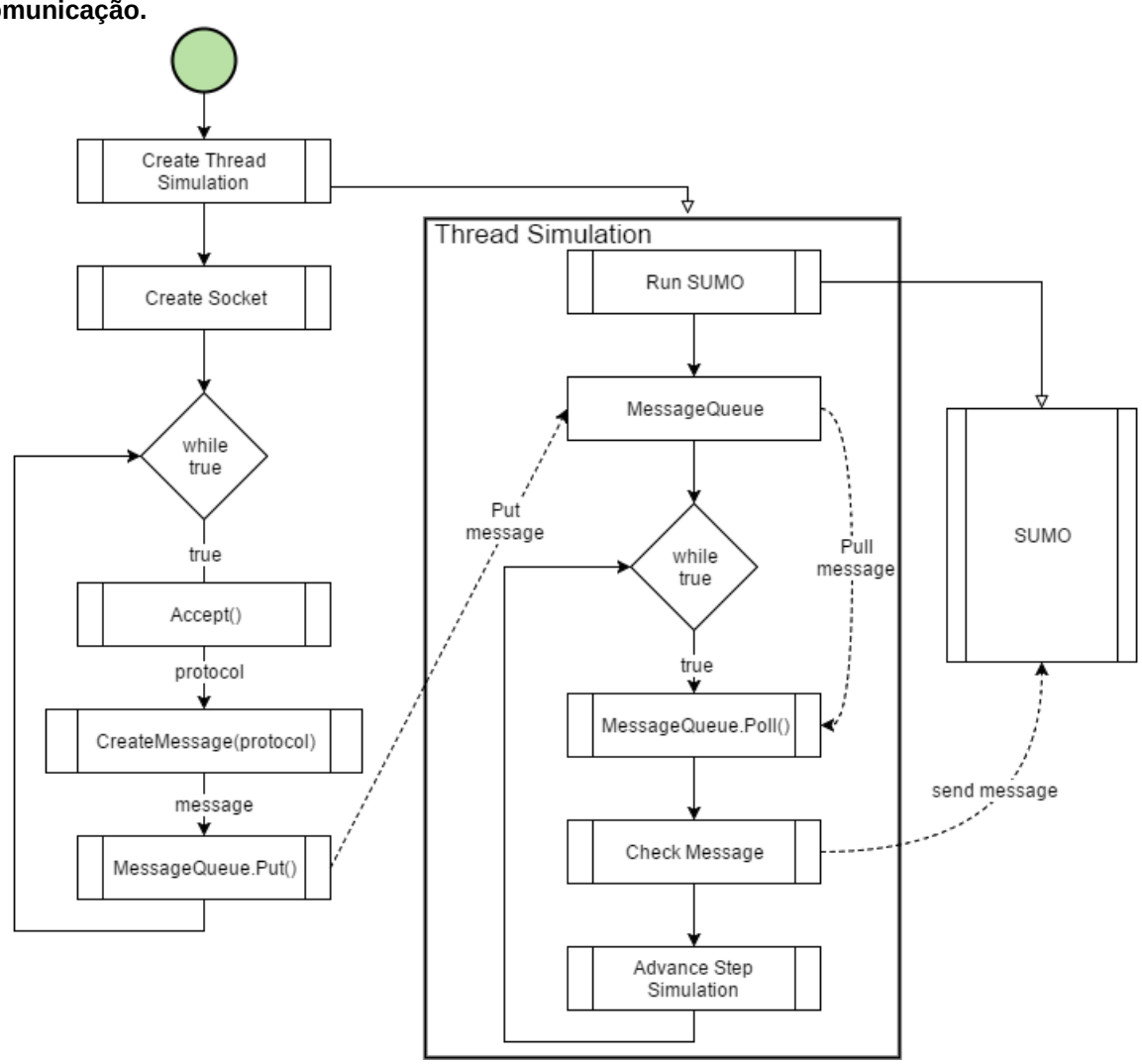
Fonte: Autoria própria

Para concretizar a comunicação entre o projeto MAPS e o SUMO foram necessárias algumas alterações de componentes no projeto, as quais são especificadas no item a seguir.

6.2.1 Modificações nos códigos Jason e Cartago do projeto MAPS

Para efetivar o envio de mensagens do projeto para o MAPS-SUMO foi desenvolvido um artefato chamado *Simulation,* responsável por criar os protocolos específicos para cada operação e encaminhá-los para a classe de comunicação *TCPConnection*. A comunicação entre o agente *manager*, o artefato e a classe de comunicação é demonstrada por meio do diagrama 2.

Diagrama 2 – Comunicação entre o agente *manager,* o artefato *Simulation* e a classe de comunicação.



Fonte: Autoria própria

A criação do artefato é feita pelo agente *manager* em seu plano *setupParking*, demonstrado no código 21.

Código 21 – Criação do artefato *Simulation*

```
+!setupParking <-
        makeArtifact("a_Simulation", "mAPS.Simulation", ["SUMO"], ArtId3);
        focus(ArtId3).
```

Fonte: Autoria própria

Após a criação do artefato, o agente tem acesso as operações disponíveis no mesmo, e podem ser chamadas a qualquer momento. O plano *allocateSpot* é responsável por alocar uma vaga a um motorista específico quando o estacionamento não está lotado. Portanto, ao enviar o protocolo, é necessário informar a identificação do motorista, da vaga e o grau de reputação do mesmo. O

código 22 mostra a chamada a operação responsável por comunicar a alocação de uma vaga nova à simulação.

Código 22 – Chamada a operação *allocateSpotSumo*

```
+!allocateSpot(AGENT,BACKGROUND) :
                                    nSpotsUsed(N) &
                                    nSpotsMAX(MAX) &
                                    isFull(COND) &
                                    pFull(P) &
                                    COND = false
                                    <-
...
        allocateSpotSumo(AGENT, S, BACKGROUND);
...
```

Fonte: Autoria própria

A operação *allocateSpotSumo(AGENT, S)* recebe como parâmetros a identificação do agente, da vaga e o grau de reputação do agente, respectivamente, e compõe o protocolo a ser enviado para o MAPS-SUMO. Essa operação é especificada no código 23.

Código 23 – Operação *allocateSpotSumo*

```
@OPERATION
public void allocateSpotSumo(Object idDriver, Object idSpot, Object trust) {

        String protocol = "D" + idDriver + "PS" + idSpot + "TRUST" + trust;

        conn.connect(protocol);
}
```

Fonte: Autoria própria

O plano *leaveSpot* é responsável por liberar a vaga de um motorista, portanto o mesmo está deixando o estacionamento. A mensagem deve informar a identificação do motorista e da vaga. O código 24 mostra a chamada da operação responsável por comunicar a saída de um veículo à simulação.

Código 24 – Chamada à operação *leaveSpotSumo*

```
+!leaveSpot(S)[source(AG)] : nSpotsUsed(N)
                           & nSpotsMAX(MAX)
                           & isFull(COND)
                           & pFull(P) <-
...
        leaveSpotSumo(AGENT, S);

...
        !checkQueue.
```

Fonte: Autoria própria

A operação *leaveSpotSumo(AGENT, S)* solicita como parâmetros a identificação do agente e da vaga, respectivamente, e monta o protocolo a ser encaminhado para o MAPS-SUMO. Essa operação é especificada no código 25.

Código 25 – Operação *leaveSpotSumo*

```
@OPERATION
public void leaveSpotSumo(Object idDriver, Object idSpot) {

        String protocol = "D" + idDriver + "LS" + idSpot;

        conn.connect(protocol);
}
```

Fonte: Autoria própria

Nos casos em que o estacionamento não possua vagas disponíveis para alocação e um novo *driver* faz a solicitação de uma vaga, o mesmo plano *allocateSpot* é utilizado pelo agente *manager*, porém com a indicação de que o estacionamento está cheio. Portanto, o motorista será adicionado à fila de espera. Este plano pode ser observado no código 26.

Código 26 – Chamada a operação *insertDriverQueueSumo*

```
+!allocateSpot(AGENT,BACKGROUND) : isFull(COND)
                                 & COND = true <-
        insertDriverQueueSumo(AGENT);
        insertDriverQueue(AGENT,BACKGROUND).
```

Fonte: Autoria própria

Para a operação *insertDriverQueue* é necessário informar apenas qual motorista está aguardando na fila, pois ainda não há uma vaga alocada para o mesmo. Esta operação é demonstrada no código 27.

Código 27 – Operação *insertDriverQueueSumo*

```
@OPERATION
public void insertDriverQueueSumo(Object idDriver) {

        String protocol = "D" + idDriver + "QU";

        conn.connect(protocol);
}
```

Fonte: Autoria própria

6.2.2 Traci4J

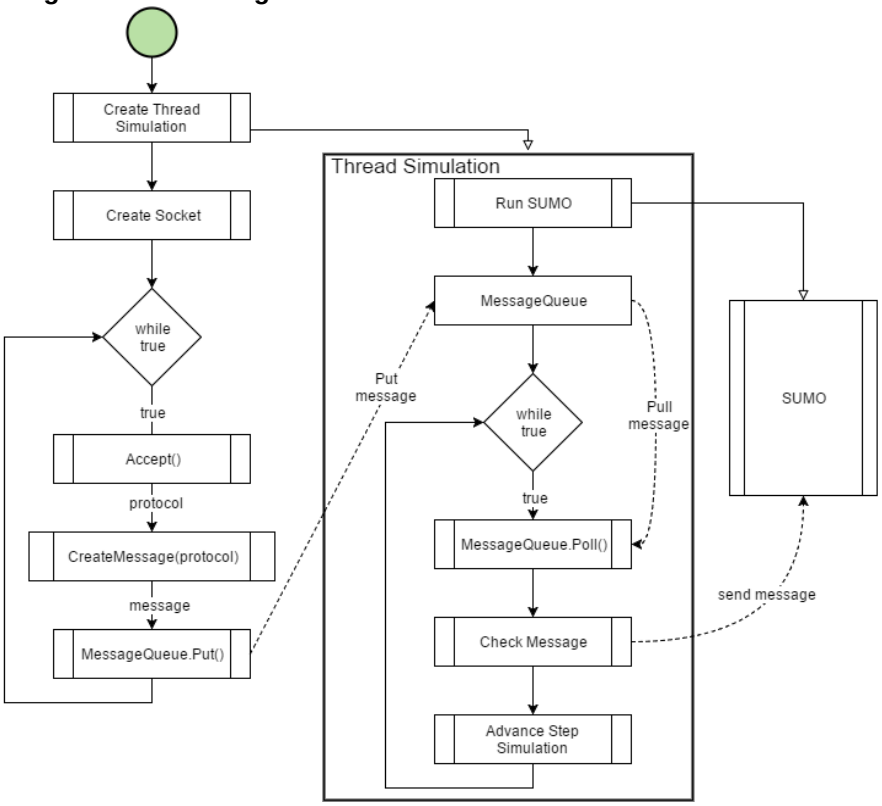
A Traci4J é uma biblioteca de alto nível, desenvolvida na linguagem Java, que cria uma interface de comunicação com o SUMO, permitindo controlar e/ou obter informações de uma simulação, utilizando o protocolo TraCI, demonstrado na seção 5.1.1. Enquanto a simulação está em execução, a biblioteca pode obter diversos tipos de informações, sejam elas estáticas (conjunto de vias, rotas, entre outros) ou dinâmicas (velocidade e posição dos veículos, por exemplo) (TraCI4J, 2011).

6.2.3 Middleware MAPS-SUMO

O MAPS-SUMO é o elemento chave da interligação entre o projeto MAPS e a ferramenta SUMO. Atua como servidor, recebendo protocolos de comunicação do projeto MAPS relacionados às decisões tomadas pelo agente *manager.* O próprio MAPS-SUMO decodifica esses protocolos e os transforma em mensagens, que serão transmitidas para o SUMO através da biblioteca TraCI4J.

O diagrama 3 demonstra uma visão geral do funcionamento do MAPS-SUMO. Primeiramente é criada uma *thread,* a qual executa a simulação gráfica, cria uma lista de mensagens e aguarda novas mensagens a serem enviadas à simulação. Todos os procedimentos de comunicação direta com o SUMO são controlados pela biblioteca Traci4J. Em paralelo, o MAPS-SUMO cria o servidor responsável por receber os protocolos enviados pelo MAPS, transformá-los em mensagens e encaminhá-las para a *thread*.

Diagrama 3 – Visão geral do funcionamento do MAPS-SUMO



Fonte: Autoria própria

O primeiro procedimento realizado pelo MAPS-SUMO ao receber um protocolo é transformá-lo em uma mensagem, a qual contém o tipo da mensagem, demonstrado na tabela 1, a identificação do motorista, a vaga, quando necessário e o grau de reputação do agent e, em casos de alocação de uma nova vaga.

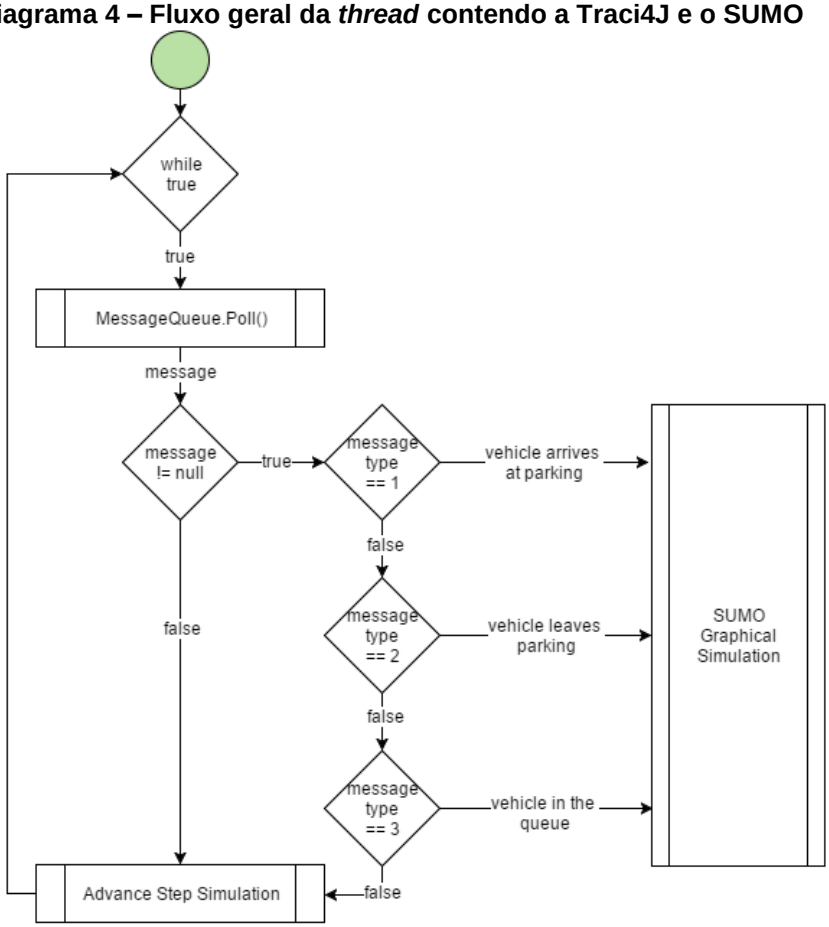
Tabela 1 – Conversão tipo do protocolo para tipo da mensagem

Tipo do Protocolo	Tipo da Mensagem
PS (*Parking Spot*)	1
LS (*Leaving Spot*)	2
QU (*Queue*)	3

Fonte: Autoria própria

A mensagem obtida por meio do protocolo é encaminhada para a *thread* onde situa-se a biblioteca Traci4J, responsável pela comunicação com o SUMO. O diagrama 4 demonstra o fluxo realizado pela *thread* para mostrar graficamente o funcionamento do estacionamento. Este fluxo segue basicamente quatro passos: (i) receber uma mensagem do MAPS-SUMO*;* (ii) verificar o tipo da mensagem; (iii) encaminhá-la para simulação; e (iv) avançar a simulação.

Diagrama 4 – Fluxo geral da *thread* contendo a Traci4J e o SUMO



Fonte: Autoria própria

A *thread* mantém duas listas de veículos, atualizadas a cada passo: (i) *driverQueue,* o qual armazena os motoristas que estão aguardando na fila de espera, e (ii) *driverList,* que contém a lista de veículos atualmente no estacionamento. Além dessas listas locais, a *thread* também tem conhecimento sobre os repositórios de tipos de veículos, de rotas e de faixas existentes em todo o ambiente. O código 28 especifica a criação das listas e dos repositórios em Java, linguagem utilizada para desenvolvimento do MAPS-SUMO.

Código 28 – Listas e repositórios criados pelo *Middleware*

```
List<String> driverQueue = new ArrayList<String>();
List<String> driverList = new ArrayList<String>();
Repository<VehicleType> types = conn.getVehicleTypeRepository();
Repository<Route> routes = conn.getRouteRepository();
Repository<Lane> lanes = conn.getLaneRepository();
```

Fonte: Autoria própria

Conforme descrito na tabela 1, o MAPS-SUMO trabalha com três mensagens, relacionadas ao protocolo recebido, a saber: mensagens do tipo 1, tipo 2 e tipo 3.

A mensagem tipo 1 (PS) é responsável por adicionar um novo veículo à vaga especificada. Este procedimento é o mesmo utilizado para veículos contemplados com a vaga no momento da chegada ao estacionamento e para veículos os quais aguardam a alocação da vaga na fila de espera. Para a última situação, o veículo é inserido na simulação e retirado da fila de espera.

Os dados do veículo a ser adicionado na simulação são definidos pela TraCI4J através da função *queryAddVehicle* e do método *setVehicleData*. O veículo é identificado pelo nome definido no protocolo e pelo tipo *car.* A rota a qual o veículo percorrerá será definida pela vaga alocada. Como o modelo de estacionamento utilizado para a simulação possui apenas uma entrada, a faixa onde os veículos são posicionados sempre será a mesma: *mainin_0*, a qual indica a faixa 0 da via de entrada do estacionamento. O veículo é adicionado no tempo atual da simulação, também obtido através da biblioteca Traci4J, na posição 0 e com uma velocidade de 0 m/s.

O veículo então é adicionado a simulação ao executar a *query* definida anteriormente. O mesmo veículo também é adicionado na lista de veículos presentes no estacionamento. O código 29 demonstra a sintaxe do código para adicionar um novo veículo à simulação.

Código 29 – Comandos para adicionar um novo veículo à simulação

```
conn.queryAddVehicle().setVehicleData(message.getDriver(),
                                      types.getByID("car"),
                                      routes.getByID(message.getSpot()),
                                      lanes.getByID("mainin_0"),
                                      conn.getCurrentSimTime(),
                                     0,
                                     0);
conn.queryAddVehicle().run();
driverList.add(message.getDriver());

conn.getVehicleRepository().getByID(message.getDriver()).queryChangeColor().setValu
e(message.getColor());
conn.getVehicleRepository().getByID(message.getDriver()).queryChangeColor().run();
```

Fonte: Autoria própria

A simulação também utiliza do grau de confiança dos agentes para exibí-los em cores diferenciadas, de acordo com um intervalo pré-definido na tabela 2.

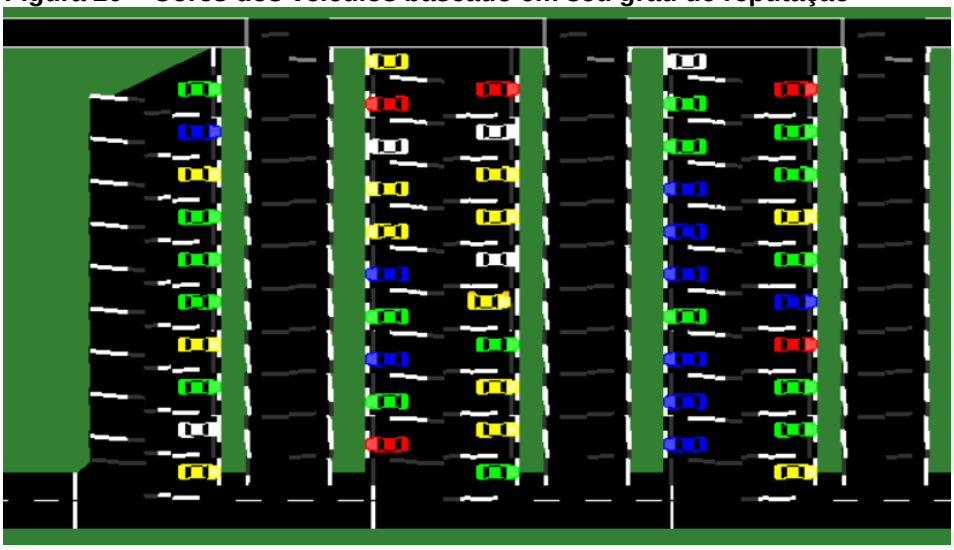
Tabela 2 – Graus de reputação dos agentes e suas respectivas cores na simulação

Grau de reputação	Cor
0 a 100	Vermelho
101 a 200	Verde
201 a 300	Azul
301 a 400	Amarelo
até 500	Branco

Fonte: Autoria própria

O resultado da aplicação de cores na simulação de acordo com o grau de reputação do agente é demonstrado na figura 20.

Figura 20 – Cores dos veículos baseado em seu grau de reputação



Fonte: Autoria própria

A mensagem tipo 2 (LS) identifica que um veículo está deixando o estacionamento. Aqui tem-se uma das limitações da integração entre a biblioteca Traci4J e o SUMO. Quando definido um tempo de parada em uma rota, seja ele por duração ou por tempo máximo, a função *queryChangeRoute(),* relacionada a um veículo, não o redireciona para a rota de saída no mesmo momento em que é solicitada. Ao contrário, o veículo só trocará de rota no momento em que o tempo definido anteriormente se esgotar.

Para contornar essa situação, o veículo destinado a deixar o estacionamento é removido da simulação e uma cópia do mesmo veículo é adicionado em seu lugar,

na mesma posição e com as mesmas características, porém identificado com o sufixo _*out* para evitar duplicidade de veículos. Como a simulação é avançada apenas ao final da execução de todos os comandos, visualmente não é possível identificar essa adaptação. Essa alteração também não reflete no funcionamento do projeto MAPS.

Os dados do veículo original são obtidos da simulação, para então removê-lo da lista de veículos presentes no estacionamento. Para adicionar um novo veículo em seu lugar é necessário obter a faixa referente à vaga a qual o veículo se encontra. A classe *mapsLane* é responsável por retornar a faixa de acordo com o nome da vaga. O veículo então é definido com a mesma identificação seguida de _*out* e redirecionado à rota de saída referente a sua vaga. O código 30 demonstra os passos realizados para essa mensagem.

Código 30 – Comandos para remover um veículo da simulação

```
Vehicle vehicle = conn.getVehicleRepository().getByID(message.getDriver());

conn.queryRemoveVehicle().setVehicleData(vehicle, 1);
conn.queryRemoveVehicle().run();

driverList.remove(message.getDriver());

String lane = mapsLanes.getLane(message.getSpot());

conn.queryAddVehicle().setVehicleData(message.getDriver() + "_out",
                                      types.getByID("car"),
                                      routes.getByID(message.getSpot() + "_out"),
                                      lanes.getByID(lane),
                                      conn.getCurrentSimTime(),
                                      0,
                                      0);
conn.queryAddVehicle().run();
```

Fonte: Autoria própria

A mensagem tipo 3 (QU) é responsável por adicionar um veículo à fila de espera de alocação de vagas. A fila de espera ainda não é demonstrada graficamente, porém é gravada em um controle interno, através da lista *driverList* criada anteriormente. A cada chamada a esta mensagem um novo veículo é inserido na *driverList,* como mostra o código 31.

Código 31 – Comando para adicionar um veículo a fila de espera

```
driverQueue.add(message.getDriver());
```

Fonte: Autoria própria

Por fim, a ferramenta SUMO é responsável por receber comandos encaminhados pelo MAPS-SUMO e demonstrar de maneira gráfica o funcionamento do ambiente de estacionamento em tempo real. O MAPS-SUMO é responsável por encaminhar as mensagens ao SUMO, através da biblioteca Traci4J. O próprio SUMO trabalha como servidor, rodando paralelamente ao MAPS-SUMO e aguardando mensagens para exibir na simulação.

7 EXPERIMENTOS

Neste capítulo será mostrado como foram feitos os experimentos práticos desse projeto, os quais visam aplicar a ferramenta SUMO ao projeto MAPS. O restante deste capítulo é organizado da seguinte forma: (i) preparação do ambiente, (ii) elaboração dos cenários, (iii) resultados obtidos, (iv) comparação entre o projeto MAPS original e o projeto integrado ao SUMO e (v) dificuldades encontradas.

7.1 CONFIGURAÇÃO DAS FERRAMENTAS E AMBIENTES

Para realizar os experimentos práticos é necessário a configuração do ambiente de simulação e dos agentes. A simulação engloba o equipamento físico utilizado, a ferramenta de simulação e os arquivos utilizados para efetuar a simulação. O ambiente dos agentes engloba a criação e configuração dos agentes que irão atuar na simulação.

7.1.1 Ambiente de Simulação

Os experimentos foram realizados utilizando as seguintes configurações de *hardware*:

- CPU: Intel® Core™ i7-4700MQ @ 2.4GHz;
- Memória: 8 GB RAM;
- GPU: 2x GeForce GT 755M

Além disso, as ferramentas utilizadas no desenvolvimento do projeto são destacadas a seguir:

- Windows 10 64bits;
- SUMO 0.27.1-win64 (http://sumo.dlr.de/wiki/Installing);
- TraCI4J (https://github.com/egueli/TraCI4J);
- Eclipse Java Neon 64bits (https://eclipse.org/downloads/);
- Projeto MAPS (https://github.com/MAPS-UTFPR/MAPS);
- Framework JaCaMo 0.6-SNAPSHOT (https://sourceforge.net/projects/jacamo/files/);

Como é especificado na seção 5.1, o SUMO necessita como entrada um arquivo contendo o modelo do ambiente; um arquivo o qual armazena todas as rotas possíveis dentro do mesmo ambiente; E ainda um terceiro arquivo, o qual é utilizado pela biblioteca TraCI4J para iniciar a simulação, deve ser criado contendo a referência aos arquivos da rede e do ambiente e a configuração da porta remota a qual o SUMO receberá conexões. O arquivo de configuração utilizado por este projeto, de extensão *.sumocfg* é demonstrado no código 32.

Código 32 – Exemplo de um arquivo de configuração

```
<configuration>
   <input>
      <net-file value="parking.net.xml"/>
      <route-files value="parking.rou.xml"/>
      <remote-port value="8883"/>
   </input>
</configuration>
```

Fonte: Autoria própria

7.1.2 Ambiente dos Agentes

A criação dos agentes *drivers* em Jason segue um modelo específico, demonstrado na seção 4.1.2, o qual requer a definição de alguns elementos: identificação, reputação, tempo a ser gasto e tempo de chegada ao estacionamento. Para abranger diferentes tipos de cenários, criou-se uma classe em Java para gerar uma quantidade pré-definida de agentes, baseados em variáveis aleatórias: reputação, tempo a ser gasto e tempo de chegada ao estacionamento.

Portanto, é possível definir a quantidade de agentes (*nAgents*) a serem criados e o intervalo de chegada entre cada agente (TA). O tempo a ser gasto por cada agente no estacionamento (TS) é escolhido aleatoriamente em um vetor que contém diversos tempos de estadia. Além disso, a classe também atribui uma reputação (*trust*) aleatória para cada um dos agentes, baseado em um intervalo também pré-definido.

- *nAgents:* define a quantidade de agentes a serem criados. Não há um limite máximo;
- TA: intervalo de chegada entre um agente e outro. Definido em 0.25 segundos, porém permite alteração;
- TS: tempo em segundos a ser gasto no estacionamento, escolhido aleatoriamente entre o vetor {20, 20, 25, 30, 35, 60, 60, 120, 120};
- *trust*: reputação do motorista, escolhido um valor aleatoriamente entre 50 e 500.

Dessa maneira foi possível diversificar os motoristas que participam do ambiente de testes, permitindo uma análise ampla da simulação implementada.

7.2 ELABORAÇÃO DOS CENÁRIOS

Utilizando a combinação do modelo demonstrado na figura 13 com os agentes criados através do método da seção 7.1, foi possível elaborar diversos tipos de cenários a serem testados por esse projeto. Especificamente foram criados 6 cenários.

O modelo foi distribuído em três diferentes tipos de estacionamento (considerando a quantidade de vagas disponíveis): (1) 10 vagas, (2) 100 vagas e (3) 160 vagas. Cada um desses ambientes engloba algumas áreas do modelo previamente estabelecido no SUMO e são especificados na tabela 3. As áreas A, B, C, D, E, F, G, H e I referem-se às áreas do modelo ilustrado na figura 13.

Tabela 3 – Distribuição de vagas dos ambientes estabelecidos.

Áreas / **Ambientes**	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**Total Vagas**
1	X									10
2	X	X		X		X		X	X	100
3	X	X	X	X	X	X	X	X	X	160

Fonte: Autoria própria

Para a definição dos conjuntos de agentes *drivers,* utilizou-se uma previsão de ocupação dos ambientes, e então criaram-se os conjuntos: (v1) 50 veículos, (v2) 100 veículos, (v3) 250 veículos e (v4) 500 veículos. Cada veículo obteve um valor de reputação aleatório entre 50 e 500. O tempo a ser gasto foi sorteado dentre um vetor

contendo: 20, 20, 25, 30, 35, 60, 60, 120, 120 segundos. O tempo de chegada entre um *driver* e outro foi definido em 0.25 segundos. Por exemplo, para dois *drivers* quaisquer, do conjunto total de *drivers*, tem-se que o tempo de chegada entre eles é 0.25 segundos.

A tabela 4 demonstra os ambientes os quais cada conjunto de agente irá atuar, totalizando seis diferentes tipos de cenários

Tabela 4 – Ambientes de atuação dos conjuntos de agentes

Conjuntos / Ambientes	v1	v2	v3	v4
1	X			
2		X	X	X
3			X	X

Fonte: Autoria própria

Nos gráficos gerados por meio dos resultados obtidos na próxima seção, a porcentagem de uso refere-se à porcentagem de ocupação total do ambiente do estacionamento. A fila de espera demonstra a quantidade de agentes aguardando uma alocação de vaga na fila de espera. As medidas estão em relação à unidade de tempo do SUMO (denominada aqui como *uts*), medida de tempo calculada internamente pelo simulador SUMO.

7.3 RESULTADOS OBTIDOS

Para analisar a validade da integração entre o projeto MAPS e a ferramenta da simulação SUMO, cada cenário será analisado dentro de alguns quesitos, sendo eles: (i) porcentagem de uso do estacionamento; (ii) tamanho da fila de espera; e (iii) consistência da comunicação entre o MAPS e SUMO. Os três tipos de análises, em conjunto com a simulação gráfica, permitem verificar a consistência da interligação entre os agentes Jason e o simulador SUMO.

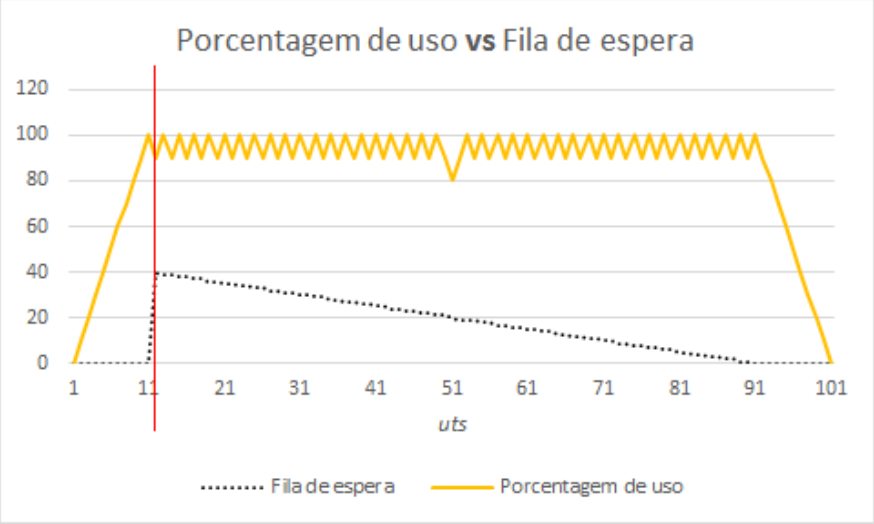
O primeiro e segundo quesitos são adquiridos no ambiente do projeto MAPS e comparados à simulação gráfica do SUMO. Para analisar a consistência de comunicação, os protocolos são armazenados em três momentos: ao serem criados no projeto MAPS, ao serem recebidos pelo MAPS-SUMO e ao serem transmitidos ao SUMO pela biblioteca TraCI4J. Desta maneira é possível obter informações sobre a consistência da interligação das ferramentas.

Na sequência são descritos os experimentos, os quais estão organizados conforme os três ambientes de estacionamento definidos na tabela 3.

- Ambiente 1:

Por possuir quantidade de vagas limitadas, esse ambiente foi testado apenas com o conjunto de agentes v1. O gráfico 1 mostra a porcentagem de uso do ambiente comparado com o tamanho da fila de espera. Na *uts* 12, destacada no gráfico 1 pela linha vermelha, é possível verificar que há 40 veículos na fila de espera e o estacionamento está com aproximadamente 90% do seu espaço ocupado.

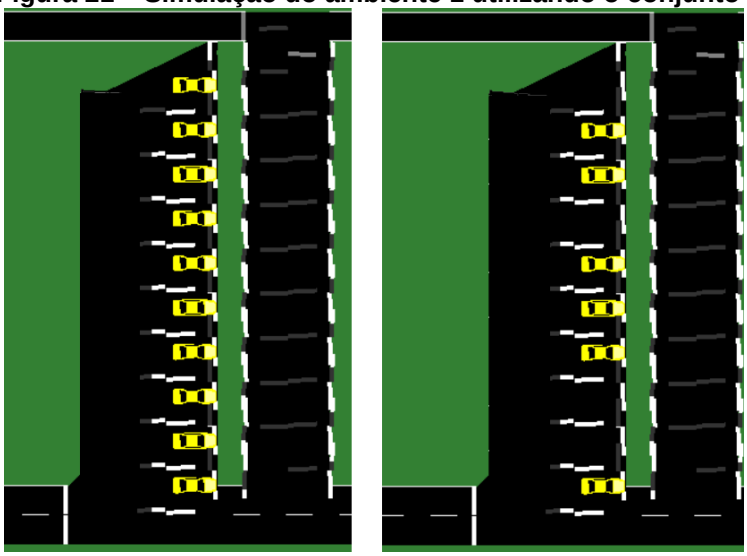
Gráfico 1 – Porcentagem de uso e fila de espera do ambiente 1 com 50 agentes



Fonte: Autoria própria

Na figura 21 é possível verificar dois momentos distintos da simulação. A figura da esquerda mostra o momento em que o estacionamento encontra-se cheio, enquanto a figura da direita mostra o exato momento o qual há 60% do estacionamento ocupado. No gráfico, esses momentos são representados pelos valores de *uts* 92 e 96, respectivamente.

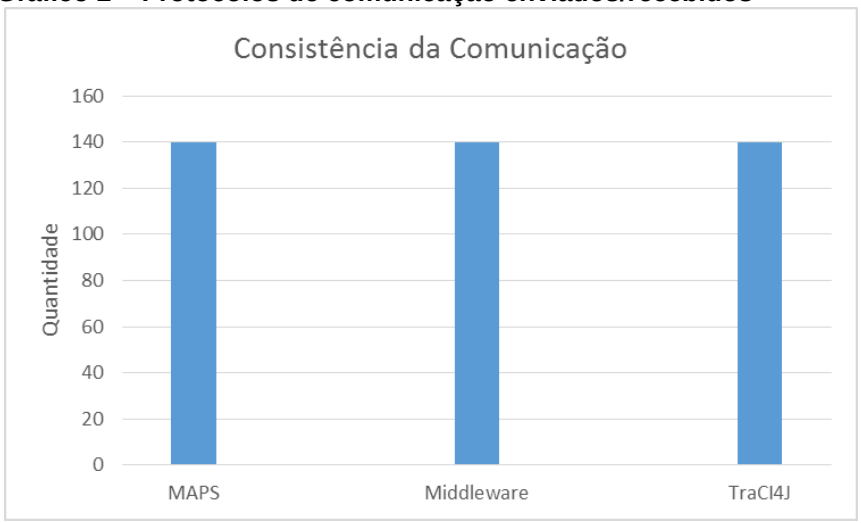
Figura 21 – Simulação do ambiente 1 utilizando o conjunto de agentes v1



Fonte: Autoria própria

O gráfico 2 demonstra a quantidade de protocolos enviados pelo projeto MAPS e recebidos pelo MAPS-SUMO e pela TraCI4J. Podemos observar que não houve falha na comunicação entre as ferramentas, totalizando a transmissão de 140 protocolos sem perdas.

Gráfico 2 – Protocolos de comunicação enviados/recebidos



Fonte: Autoria própria

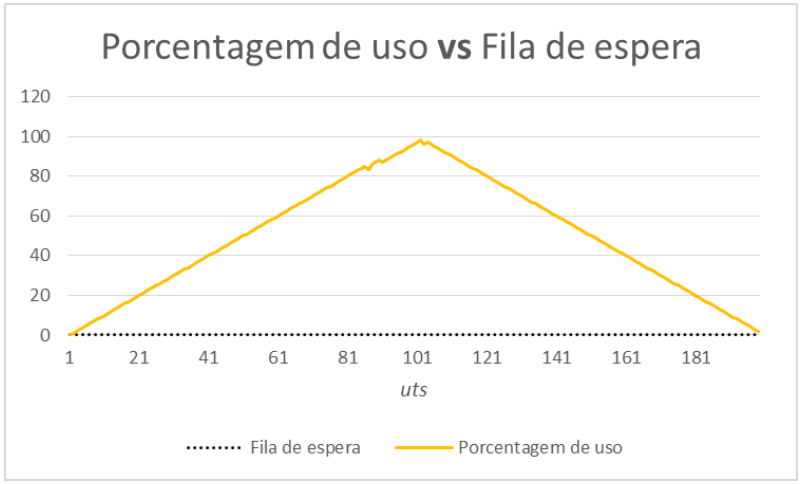
- Ambiente 2:

Este cenário utiliza as áreas A, B, D, F, H e I do modelo inicial, totalizando 100 vagas. Neste cenário foram testados os conjuntos v2, v3 e v4, contendo 100,

250 e 500 veículos, respectivamente, resultando em ambientes sem fila de espera e com baixa e alta variação na fila de espera.

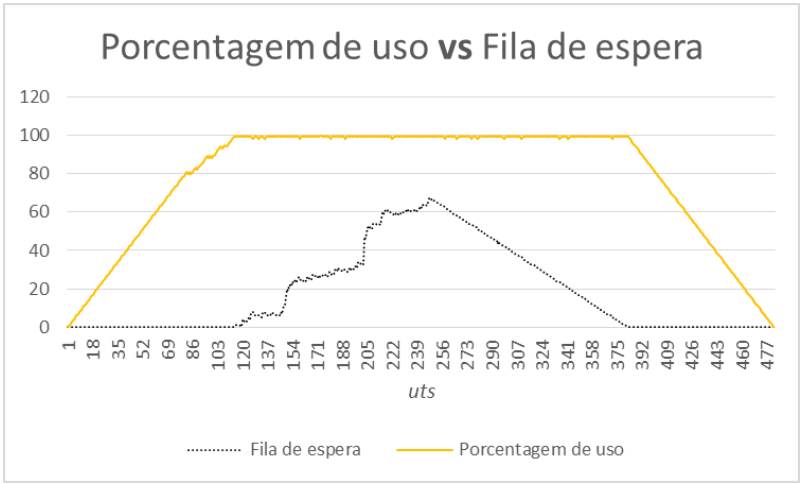
É possível observar no gráfico 3 que, devido ao baixo fluxo de carros do conjunto v2, não foi possível a obtenção da fila de espera. Já no gráfico 4 houve uma fila de espera com poucas variações e que se esgotou rapidamente. Por fim, no gráfico 5, houve a criação de duas filas e que estiveram ativas durante quase todo o tempo em que o ambiente esteve em funcionamento. É possível observar também que a criação da fila de espera acontece no momento em que o estacionamento fica cheio, e toda alteração na fila acontece quando há uma variação na lotação do ambiente.

Gráfico 3 – Porcentagem de uso e fila de espera do ambiente 2 com 100 agentes



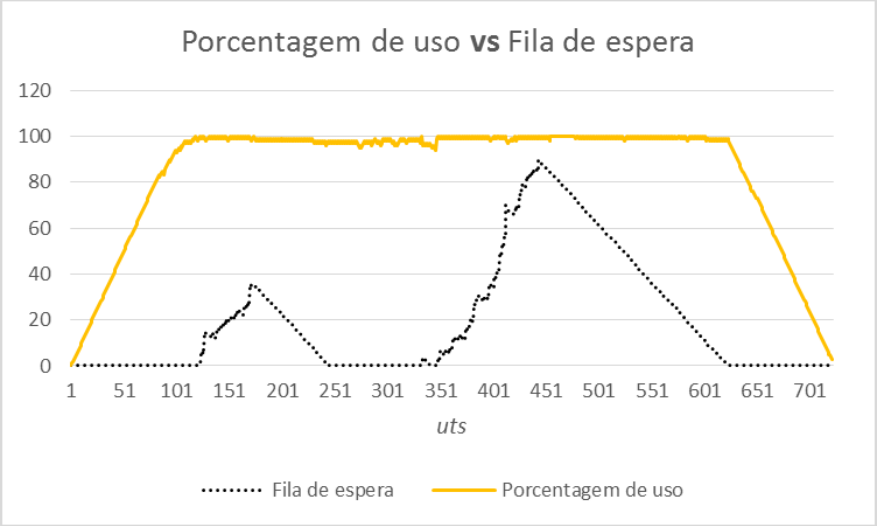
Fonte: Autoria própria

Gráfico 4 – Porcentagem de uso e fila de espera do ambiente 2 com 250 agentes



Fonte: Autoria própria

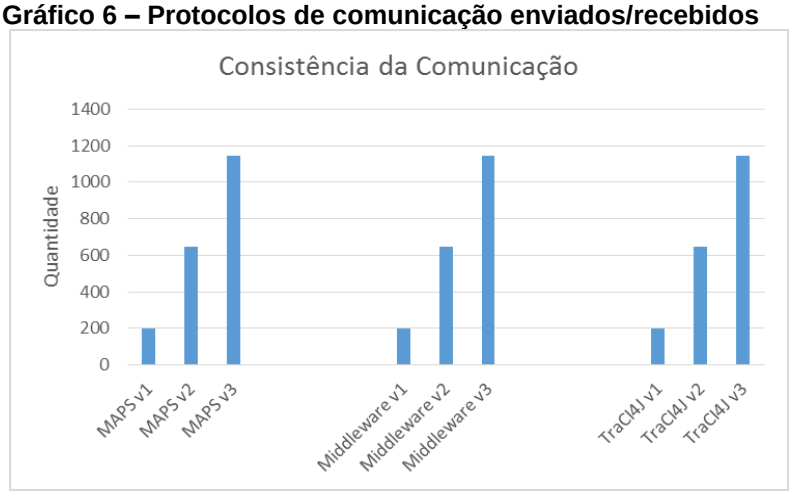
Gráfico 5 – Porcentagem de uso e fila de espera do ambiente 2 com 500 agentes



Fonte: Autoria própria

O gráfico 6 demonstra a consistência da comunicação nos três casos previamente citados, não apresentando perdas de protocolos. Nota-se que, devido a variação da quantidade de veículos, a quantidade de protocolos também sofre variações.

Gráfico 6 – Protocolos de comunicação enviados/recebidos



Fonte: Autoria própria

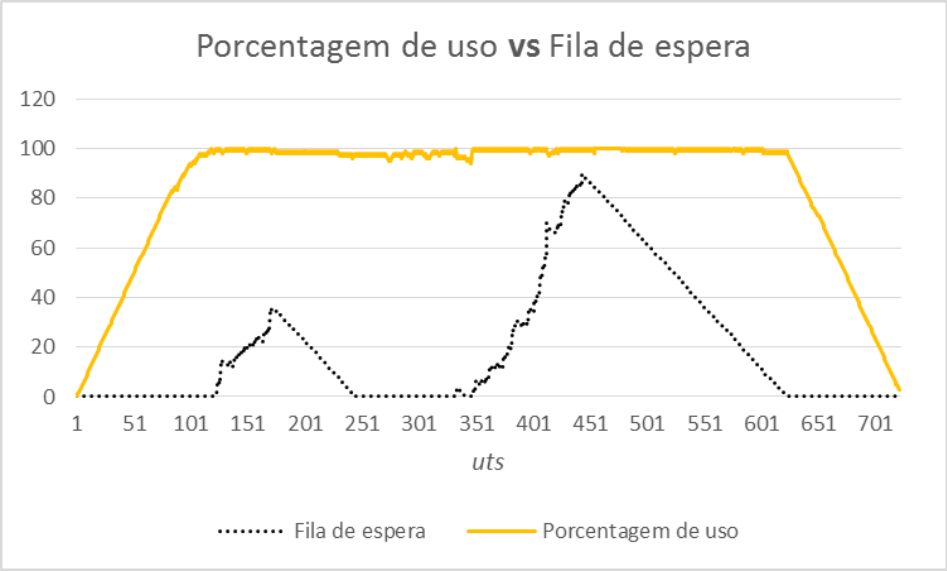
- Ambiente 3:

O ambiente 3 engloba todos os setores do modelo principal, utilizando o total de 160 vagas. Os testes foram feitos utilizando os conjuntos de *drivers* v3 e v4, com 250 e 500 agentes, respectivamente.

Nos gráficos 7 e 8 é possível observar e comparar a porcentagem de uso do ambiente e o tamanho da fila de agentes em espera para se obter uma vaga, para os conjuntos v3 e v4, respectivamente. É possível notar que no segundo caso há

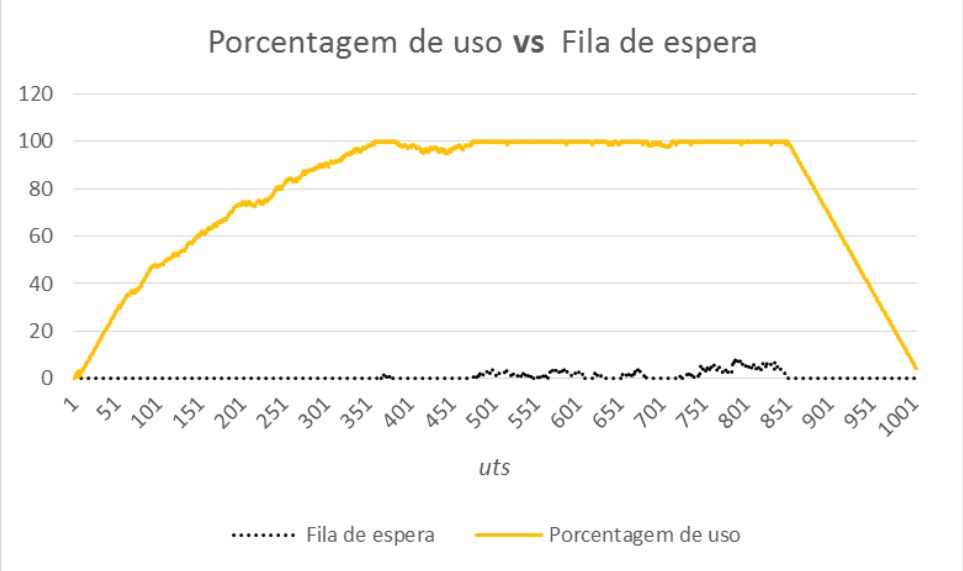
uma maior variação na fila de espera, devido ao fluxo de veículos ser mais alto. Também é possível notar que o tamanho máximo da fila de espera é maior no conjunto v3 e isso deve ao fato dos conjuntos serem criados aleatoriamente. Portanto, no momento o qual o estacionamento ficou completamente ocupado, os agentes do conjunto v3 permaneceram durante um maior período de tempo estacionados, acumulando agentes na fila de espera.

Gráfico 7 – Porcentagem de uso e fila de espera do ambiente 3 com 250 agentes



Fonte: Autoria própria

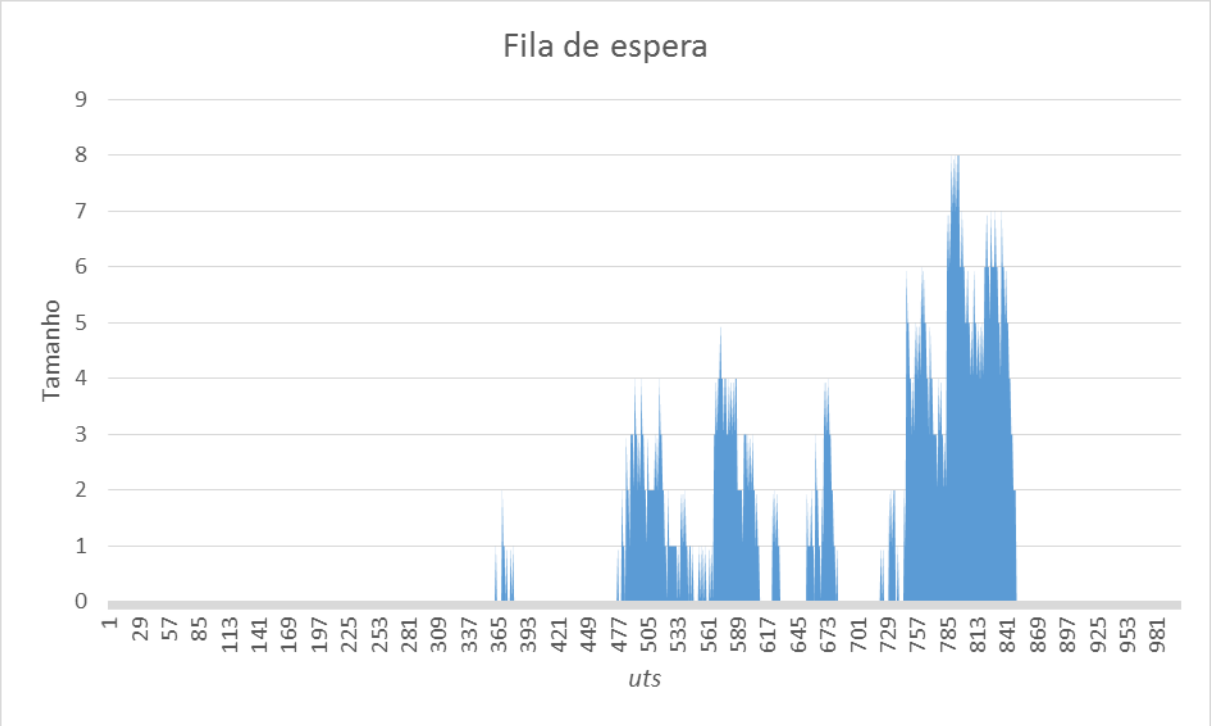
Gráfico 8 – Porcentagem de uso e fila de espera do ambiente 3 com 500 agentes



Fonte: Autoria própria

O gráfico 8 apresenta os resultados para uma quantidade significativa de agentes (500, ao total). Portanto, a ilustração das variações na fila de espera são dificilmente perceptíveis. Assim, tem-se o gráfico 9 para ilustrar em detalhes (com a devida escala) tais variações na fila de espera.

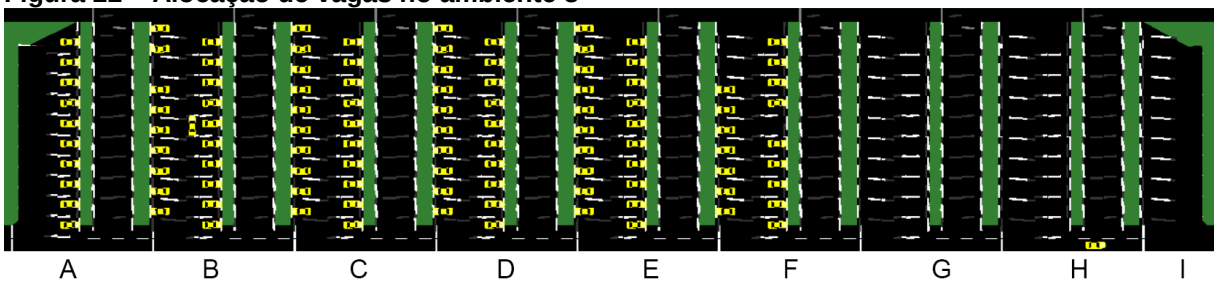
Gráfico 9 – Comportamento da fila de espera do ambiente 3 com 500 agentes



Fonte: Autoria própria

A figura 22 permite a visualização do ambiente em andamento, onde há carros estacionados, chegando às vagas e deixando o estacionamento. Este estacionamento está com 105 vagas alocadas de um total de 160, o que corresponde à 65.62% do ambiente ocupado.

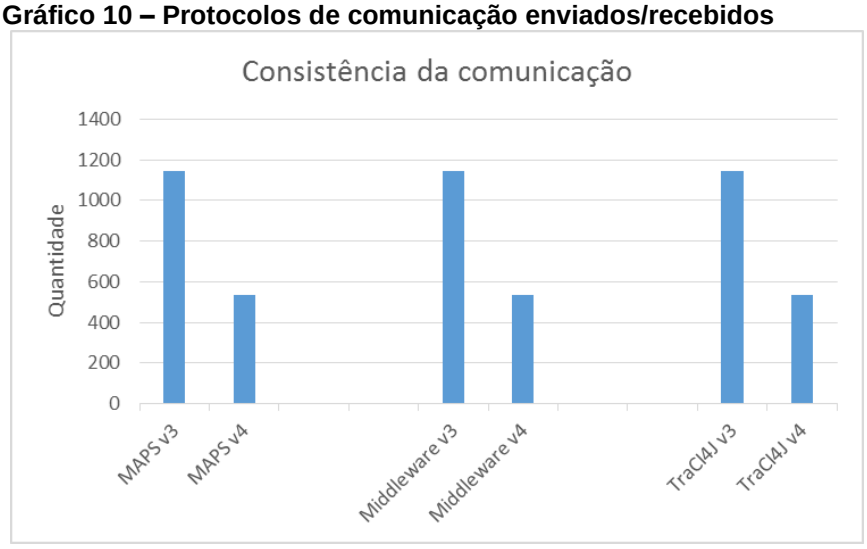
Figura 22 – Alocação de vagas no ambiente 3



Fonte: Autoria própria

Nota-se que no gráfico 10 a consistência da comunicação entre o projeto MAPS, o MAPS-SUMO e a TraCI4J, quando testados os conjuntos v3 e v4 no cenário 4.

Gráfico 10 – Protocolos de comunicação enviados/recebidos



Fonte: Autoria própria

Por meio dos resultados obtidos é possível constatar que o tamanho da fila de espera está diretamente associado à porcentagem de uso do estacionamento, onde a mesma aumenta seu tamanho quando o ambiente está lotado e diminui caso contrário. Além disso, o SUMO mostrou-se adequado na simulação gráfica dos cenários testados, de modo que facilita a análise do comportamento do ambiente e os agentes que nele atuam. Por fim, a consistência da comunicação entre o projeto MAPS, o MAPS-SUMO e ao SUMO se mostrou efetiva, não havendo perdas de protocolos entre o percurso da comunicação.

A simulação com o SUMO pode ajudar a verificar diretamente como está a ocupação das vagas no estacionamento. Por exemplo, na figura 22, é possível

identificar que há uma quantidade significativa de vagas não ocupadas nos setores F, G, H e I.

7.4 COMPARAÇÃO: MAPS X MAPS-SUMO

O objetivo deste trabalho é efetivar a interligação entre a ferramenta de simulação SUMO e o projeto MAPS, sem que haja alteração nos resultados finais obtidos. Todo o controle de alocação de vagas é realizado pelo agente *manager* no projeto MAPS, portanto o SUMO é responsável apenas por disponibilizar graficamente o uso do estacionamento, de acordo com os dados enviados pelo *manager*.

Os testes de comparação entre o projeto original e o MAPS integrado com o SUMO foram realizados de acordo com os cenários utilizados por Castro (2015) para o desenvolvimento do inicial do projeto. No restante desta seção os gráficos apresentados ilustram os resultados obtidos no MAPS original (ou apenas MAPS) e no MAPS integrado com a ferramenta SUMO (ou apenas MAPS-SUMO).

Os cenários utilizados para essa comparação são demonstrados na tabela 5.

Tabela 5 – Cenários utilizados para comparação

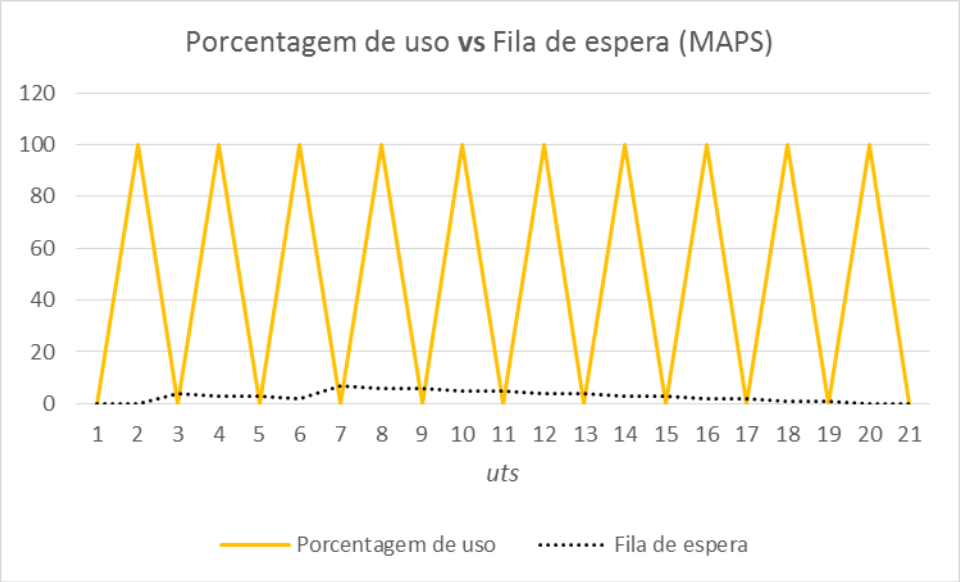
Cenário	Quantidade de *drivers*	Quantidade de Vagas
1	10	1
2	10	5

Fonte: Autoria própria

- Cenário 1:

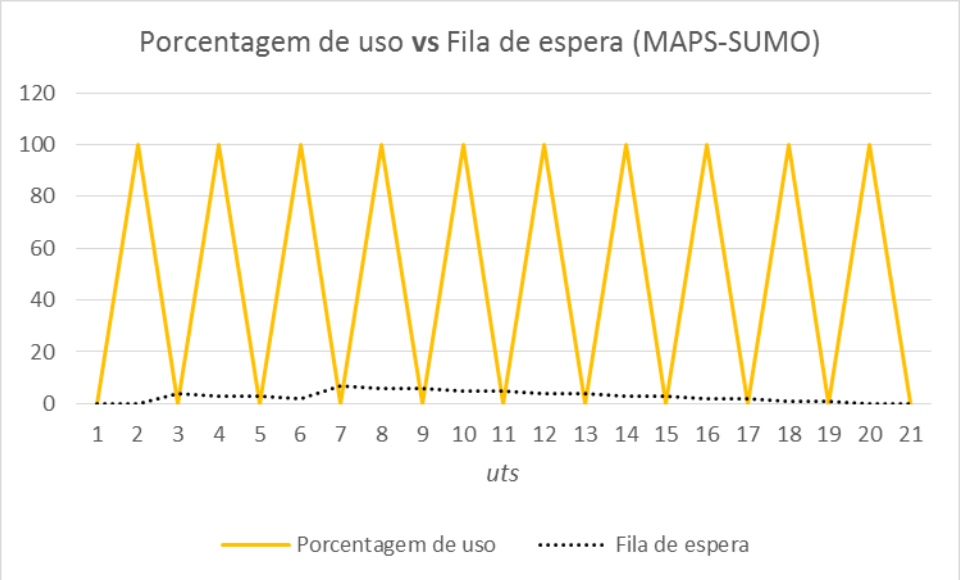
Neste cenário foram utilizados 10 agentes com apenas 1 vaga de estacionamento gerenciada pelo *manager*, visando analisar o comportamento da fila de espera em um ambiente com poucas vagas. Nos gráficos 11 e 12, é possível verificar os resultados obtidos através dos cenários testados no projeto MAPS original e no projeto integrado à simulação, respectivamente.

Gráfico 11 – Porcentagem de uso vs Fila de espera (MAPS)



Fonte: Autoria própria

Gráfico 12 – Porcentagem de uso vs Fila de espera (MAPS-SUMO)



Fonte: Autoria própria

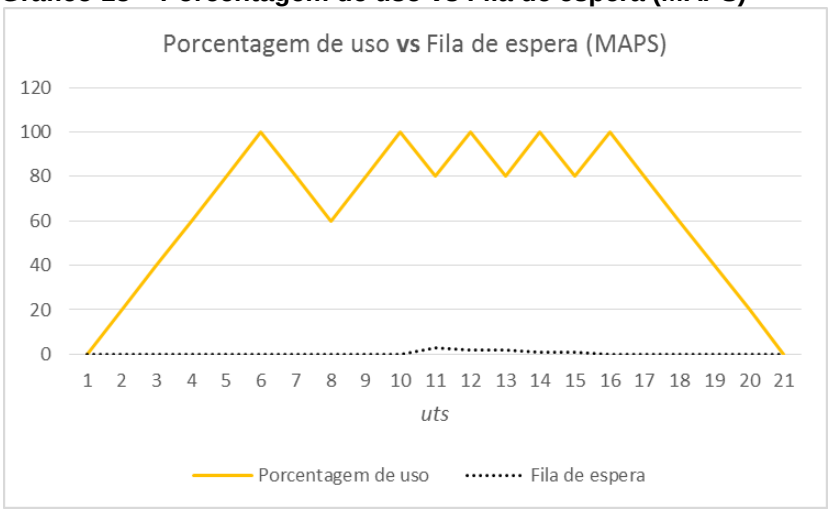
Nota-se que não houve alterações nos resultados obtidos, portanto, neste cenário, a interligação se demonstrou adequada.

- Cenário 2:

Neste cenário também foram utilizados 10 agentes *drivers*, porém com 5 vagas disponíveis no estacionamento. Os gráficos 13 e 14 demonstram os

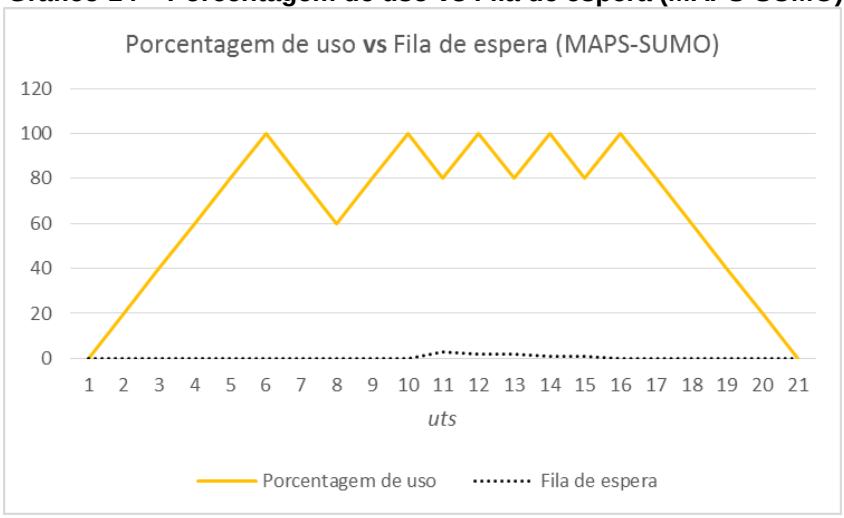
resultados obtidos nos testes realizados no MAPS original e no MAPS-SUMO, respectivamente.

Gráfico 13 – Porcentagem de uso vs Fila de espera (MAPS)



Fonte: Autoria própria

Gráfico 14 – Porcentagem de uso vs Fila de espera (MAPS-SUMO)



Fonte: Autoria própria

Nota-se que, neste cenário, também não houve alteração nos resultados obtidos. Vale ressaltar também que o crescimento na fila de espera ocorre quando não há mais vagas disponíveis no estacionamento e, quando um veículo deixa a vaga, a fila de espera diminui.

7.5 DIFICULDADES ENCONTRADAS

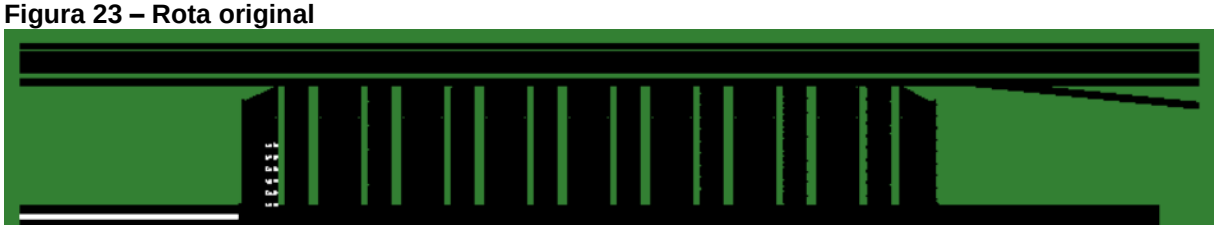
Para a execução deste trabalho foram encontradas algumas dificuldades em relação a simulação gráfica do funcionamento do projeto MAPS. Na sequência serão discutidas as dificuldades, envolvendo a troca de rotas.

- Troca de rotas

O trabalho possui duas rotas configuradas para cada vaga: rota de chegada e rota de saída. A rota de chegada possui um atributo de parada, o qual define o local de parada do veículo e o tempo máximo ou duração da parada, conforme especificado na seção 6.1. A ferramenta SUMO não permite alterar certos componentes após a sua criação, como é o caso das rotas.

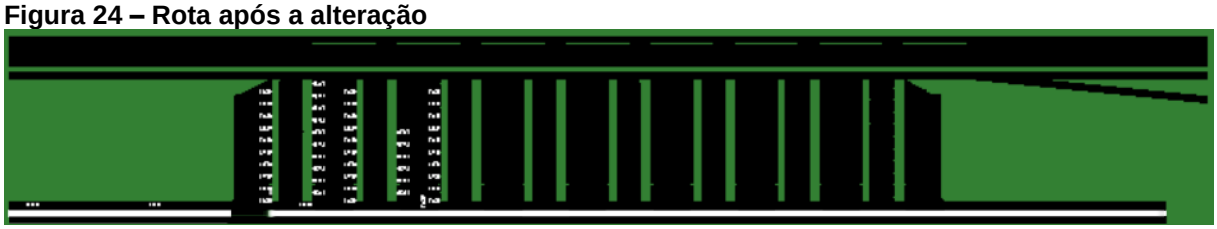
Para cada veículo na simulação, é possível fazer algumas alterações, como mudança de cor e de velocidade. Entre elas, a biblioteca Traci4J também disponibiliza a função *queryChangeRoute()*, responsável por alterar a rota do veículo indicado. A nova rota é enviada para o veículo, porém o mesmo não deixa a vaga até o tempo definido anteriormente se esgotar. Como podemos observar nas figuras 23 e 24, temos a rota original de chegada e a rota após a alteração (junção da rota original com a rota de saída), respectivamente. As rotas são as faixas destacadas na via inferior do modelo.

Figura 23 – Rota original



Fonte: Autoria própria

Figura 24 – Rota após a alteração



Fonte: Autoria própria

A rota é recalculada de acordo com os novos dados informados, porém o veículo continua estacionado na vaga original. Portanto o próximo veículo alocado

para a mesma vaga, não conseguiria estacionar e teria que aguardar. Para contornar essa situação, o veículo primeiramente é removido do ambiente e um novo veículo é inserido com a nova rota. Visualmente não é possível identificar essa adaptação e o simulador exibe o resultado esperado.

Futuramente pretende-se estudar maneiras de alterar a rota do veículo quando o mesmo se encontra estacionado. A biblioteca Traci4J possui código aberto, portanto será possível trabalhar diretamente no código para encontrar soluções para esse problema encontrado.

8 CONCLUSÃO

O objetivo principal deste trabalho é apresentar a interligação entre o *framework* JaCaMo, com foco no JaCa (Jason + Cartago), e a ferramenta de simulação SUMO. Para atingir este objetivo foi realizado um estudo detalhado do SUMO bem como do *framework* JaCaMo e do projeto MAPS. O projeto MAPS foi utilizado como um exemplo de aplicação prática do *framework* JaCaMo para demonstrar a viabilidade da interligação entre as ferramentas. Por meio do desenvolvimento deste trabalho, os seguintes resultados foram obtidos:

- Representação gráfica da simulação do uso de um estacionamento;
- Criação de um protocolo de comunicação entre JaCaMo e SUMO.

A interligação entre as ferramentas foi possível através da criação de um protocolo de comunicação, responsável por transmitir informações entre os agentes e a simulação gráfica. Esse protocolo contém a informação do agente responsável pela ação, o tipo da ação a ser exibida na simulação e, quando necessário, a vaga do estacionamento. Os protocolos foram criados em um artefato situado no ambiente Cartago, e são chamados pelos agentes, programados em Jason. Além de montar os protocolos, o artefato também foi responsável por transmitir as mensagens para o MAPS-SUMO, ferramenta desenvolvida para este projeto e responsável por intermediar a comunicação entre o JaCa e o SUMO.

A aplicação do projeto MAPS neste projeto permitiu a obtenção de resultados interessantes, além de beneficiar outras linhas do projeto. Uma das vertentes do projeto visa estudar diferentes tipos de algoritmos de alocação e otimização de vagas, portanto, com a visualização gráfica, será possível fazer uma análise mais clara e simples dos algoritmos implementados.

A criação do protocolo de comunicação pode também viabilizar a definição de uma futura comunicação genérica entre o JaCaMo e o SUMO. Pretende-se utilizar o Cartago apenas para efetivar a comunicação entre o JaCaMo e o SUMO. As classes *allocateSpotSumo, leaveSpotSumo* e *insertDriverQueueSumo* seriam removidas por serem referentes ao projeto MAPS, e o protocolo seria um possível parâmetro recebido pela classe de comunicação definida no Cartago.

O próximo passo para o projeto será a modelagem de um ambiente personalizado baseado em dados reais. Isso permite com que se obtenha dados

mais concretos da integração entre o projeto MAPS com a visualização gráfica do estado do estacionamento. Também será possível desenvolver ambientes específicos para estacionamentos particulares, permitindo assim a implantação prática do projeto. Além disso, pretende-se estudar maneiras de alterar as rotas dos veículos e implementar graficamente as filas de espera.

Por meio dos experimentos realizados (apresentados no capítulo 7), foi possível concluir e afirmar a viabilidade da integração entre o *framework* JaCaMo e a ferramenta de simulação SUMO, além de exibir uma aplicação prática do projeto MAPS. Apesar do desenvolvimento de um MAPS-SUMO de intermediação entre o JaCa e o SUMO, a comunicação se demonstrou consistente, não apresentando perdas nas transmissões de protocolos.

REFERÊNCIAS

ALVARES, L. e SICHMAN, J. Introdução aos Sistemas Multiagentes. **XVII Congresso da SBC.** Brasília – Brasil, p. 2-8, ago. 1997.

ANUMBA, C. UGWU, O. e REN, Z. **Agents and multi-agent systems in construction.** Taylor & Francis, 2005.

BAINES, V. e PADGET, J. A situational awareness approach to intelligent vehicle agents. **SUMO2014 – Modeling Mobitlity with Open Data**. Berlim - Alemanha, v. 24, p. 1-17, maio 2014.

BATISTA JÚNIOR, A. A. e COUTINHO, L. R. Incorporating explicit coordination mechanisms by agents to obtain green waves. **Advanced Methods and Technologies for Agent and Multi-Agent Systems.** v. 252, p. 137-145, 2013.

BATTY, M, AXHAUSEN, K., *et al*. Smart Cities of the Futures. **UCL Working Papers Series**. Londres - Inglaterra, v. 188, out. 2012.

BEHRISCH, M., BIEKER, L., ERDMANN, J., e KRAJZEWICZ, D. SUMO – Simulation of Urban Mobility. **SIMUL 2011 – The Third International Conference on Advances in System Simulation**. Barcelona – Espanha, 2011.

BORDINI, R., HÜBNER, J. e WOOLDRIDGE, M. Programming Multi-Agent systems in AgentSpeak using Jason. **Wiley Series in Agent Technology.** John Wiley & Sons, 2007.

CArtAgO. 2006. Disponível em <http://cartago.sourceforge.net>. Acesso em: 12 abril 2016.

CASTRO, L. Modelagem e implementação de um sistema multiagente utilizando a plataforma JaCaMo para alocação de vagas em um estacionamento inteligente. Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2015.

DENATRAN. Departamento Nacional de Trânsito. Disponível em < http://www.denatran.gov.br/frota2016.htm>. Acesso em: 10 maio 2016.

Deutsche Telekom. Deutsche Telekom and Pisa start Smart City project. **Mobile World Congress.** Barcelona – Espanha, 2014.

GENESERETH, M. e KETCHPEL, M. Software agents. **Communications of the ACM.** v. 37, n. 7, p. 48-53, jul. 1994.

HÜBNER, J. **Um Modelo de Reorganização de Sistemas Multiagentes.** 2003. 224 f. Tese (Doutorado em Engenharia Elétrica) – Escola Politécnica da USP. Universidade de São Paulo, São Paulo.

JACAMO. **The JaCaMo Project**. Multi-Agent Programming Framework. 2011. Disponível em <http://jacamo.sourceforge.net/>. Acesso em: 15 março 2016.

KOKKINOGENIS, Z., PASSOS, L. e ROSSETTI, R. Towards the next-generation traffic simulation tools: a first appraisal. **6ª Conferência Ibérica de Sistemas e Tecnologias de Informação (CISTI).** Chaves – Portugal, jun. 2011.

KOSTER, A., KOCH, F. e BAZZAN, A. Incentivising Crowdsourced Parking Solutions. **Citizen in Sensor Networks. CitSens.** Barcelona – Espanha, v. 8313, set. 2014.

KRAJZEWICZ, D., BONERT, M. e WAGNER, P. The open source traffic simulation package SUMO. **RoboCup 2006 Infraestructure Simulation Competition.** Berlim – Alemanha, 2006.

KRAJZEWICZ, D., ERDMANN, J., BEHRISCH, M. e BIEKER, L. Recent development and applications of SUMO – Simulation of Urban MObility. **International Journal On Advances in Systems and Measurements.** Berlim – Alemanha, v. 5, n. 3 & 4, p. 128-138, 2012.

KRAJZEWICZ, D., HERTKORN, G., WAGNER, P. e ROSSEL, C. SUMO (Simulation of Urban Mobility). An open-source traffic simulation. **Proceeding of the 4th Middle East Symposium on Simulation and Modeling.** Sharjah – Emirados Árabes, p. 183-187, 2002.

MAES, P. Artificial Life Meets Entertainment: Lifelike Autonomous Agents. **Communications of the ACM**. v. 138, n. 11, p. 108-114, 1995.

RICCI, A., PUNTI, M., e VIROLI, M. Environment programming in multiagent systems – an artifact-based perspective. **Autonomous Agents and Multi-Agent Systems.** Special Issue on Programming Multi-Agent Systems. v. 33, n. 2, p. 158-192, set. 2011.

REVATHI, G. e DHULIPALA, V. Smart parking Systems and Sensors: A Survey. **Computing, Communication and Applications.** Dindigul – India, p. 1-5, fev. 2012.

SUMO Wiki. Disponível em <http://sumo.dlr.de/wiki>. Acesso em: 10 fevereiro 2016.

TraCI4J GitHub repository. 2011. Disponível em <https://github.com/egueli/TraCI4J>. Acesso em: 15 julho 2016.

VagaBarata (2014). Soluções para quem busca estacionamentos em São Paulo. Disponínel em <http://www.vagabarata.com.br/>. Acesso em: 10 maio 2016.

WEGENER, A., PIÓRKOWSKI, M., RAYA, M., HELLBRÜCK, H., FISCHER, S. E HUBAUX, J. TraCI: An Interface for Coupling Road Traffic and Network Simulators. **11th Communications and Networking Simulation Symposium.** Ottawa, Canadá. p. 155-163, 2008.

WOOLDRIDGE, M. Intelligent Agents. **Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence.** MIT Press, 1999.

WOOLDRIDGE, M. **An Introduction to MultiAgent Systems.** 2nd. Ed. [S.1.]: Wiley Publishing, 2009.

APÊNDICE A – INSTALAÇÃO DO SUMO

Este trabalho utilizou a versão 0.27.1 da ferramenta SUMO. A ferramenta pode ser obtida através dos links abaixo (utilize a versão referente ao seu sistema operacional):

- 32 bits: http://prdownloads.sourceforge.net/sumo/sumo-win32-0.27.1.zip
- 64 bits: http://prdownloads.sourceforge.net/sumo/sumo-win64-0.27.1.zip

As versões anteriores e os códigos fonte da ferramenta podem ser acessados através do endereço:

- http://sumo.dlr.de/wiki/Downloads

Após efetuar o download da ferramenta, o arquivo *.zip* deve ser extraído em um diretório qualquer, por exemplo em C:\. O arquivo pode ser extraído utilizando a ferramenta 7zip (http://www.7-zip.org/) ou outra similar. O pacote contém os seguintes arquivos:

- Binários: arquivos executáveis, devem ser acessados via linha de comando, com exceção do *sumo-gui* que executa a simulação gráfica;
- DLLs: bibliotecas necessárias para a ferramenta;
- Exemplos: arquivos contendo exemplos básicos, avançados e tutoriais relacionas à simulação;
- Ferramentas extras, como o TraCI;
- Documentação em formato HTML.

Para a chamada da simulação e outras ferramentas relacionadas via linha de comando, é necessário adicionar a variável SUMO_HOME às variáveis de ambiente do sistema. Qualquer linha de comando aberta deve ser finalizada para que as atualizações sejam aplicadas com sucesso.

Os seguintes passos (referente ao Windows 10) devem ser seguidos:

1. Abrir o “Painel de Controles”;
2. Acessar “Sistema e Segurança”;
3. Acessar “Sistema”;
4. Ao lado esquerdo, acessar “Configurações avançadas do sistema”;

5. Na aba “Avançado”, acessar “Variáveis de Ambiente...”;
6. Clicar em “Novo...”:
    a. Nome da variável: SUMO_HOME
    b. Valor da variável: local de instalação do SUMO, por exemplo: C:\sumo\sumo-0.27.1
7. Selecionar “Ok”;
8. Na variável “*Path*”, clicar em “Editar...”;
9. Adicionar “Novo...”:
    a. Referenciar o mesmo endereço adicionado em SUMO_HOME;
10. Selecionar “Ok”, “Ok” e “Ok”.

A partir deste momento, qualquer executável relacionado ao SUMO pode ser acessado via linha de comando. Considere que o diretório atual na linha de comando seja a pasta de instalação do SUMO. No código X é possível verificar uma chamada à simulação gráfica do SUMO, utilizando um exemplo básico incluso na ferramenta, onde “--net-file” indica o arquivo da rede e “--route-files” indica os arquivos de rotas.

Código 33 – Exemplo de chamada à simulação gráfica via linha de comando

```
sumo-gui --net-file docs\examples\sumo\vehicle_stops\net.net.xml --route-files docs\examples\sumo\vehicle_stops\input_routes.rou.xml
```

Fonte: Autoria própria

O código 34 demonstra o comando necessário para executar a simulação utilizando os arquivos usados por este projeto.

Código 34 – Execução da simulação gráfica utilizando arquivos deste trabalho

```
sumo-gui --net-file parking.net.xml --route-files parking.rou.xml
```

Fonte: Autoria própria