UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS FERNANDO SOUZA DE CASTRO

MODELAGEM E IMPLEMENTAÇÃO DE UM SISTEMA MULTIAGENTE UTILIZANDO A PLATAFORMA JACAMO PARA ALOCAÇÃO DE VAGAS EM UM ESTACIONAMENTO INTELIGENTE

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA
2015

LUCAS FERNANDO SOUZA DE CASTRO

MODELAGEM E IMPLEMENTAÇÃO DE UM SISTEMA MULTIAGENTE UTILIZANDO A PLATAFORMA JACAMO PARA ALOCAÇÃO DE VAGAS EM UM ESTACIONAMENTO INTELIGENTE

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
2015

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Ponta Grossa
Diretoria de Graduação e Educação Profissional

TERMO DE APROVAÇÃO

MODELAGEM E IMPLEMENTAÇÃO DE UM SISTEMA MULTIAGENTE UTILIZANDO A PLATAFORMA JACAMO PARA ALOCAÇÃO DE VAGAS EM UM ESTACIONAMENTO INTELIGENTE

por

LUCAS FERNANDO SOUZA DE CASTRO

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 13 de novembro de 2015 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.

_____
Professor Dr. Gleifer Vaz Alves
Prof. Orientador

_____
Professor Dr. André Pinz Borges
Membro titular

_____
Professor MsC. Dênis Lucas Silva
Membro titular

_____
Professor Dr. Ionildo José Sanches
Responsável pelos Trabalhos de Conclusão de Curso

_____
Professor Dr. Erikson Freitas de Morais
Coordenador do Curso
UTFPR – Campus Ponta Grossa

RESUMO

CASTRO, Lucas Fernando Souza de. **Modelagem e implementação de um sistema multiagente utilizando a plataforma JaCaMo para alocação de vagas em um estacionamento inteligente**. 2015. 78 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2015.

O desenvolvimento de sistemas multiagentes envolve um novo paradigma de desenvolvimento e abstração na concepção dos sistemas, visto que novos elementos agem e interferem a execução, como os agentes e suas interações entre si, o ambiente em que estão inseridos e as regras regem o ambiente em que estão inseridos. Para isso, diversas ferramentas estão sendo desenvolvidas com o intuito de proporcionar um desenvolvimento de sistemas multiagentes completo e intuitivo. Uma das ferramentas que proporciona tais capacidades é o *framework* JaCaMo. O presente trabalho apresenta o desenvolvimento de um sistema multiagente de alocação de vagas através de um agente centralizador para estacionamentos inteligentes utilizando a plataforma JaCaMo como ferramenta para o desenvolvimento do sistema multiagente.

Palavras-chave: Sistema multiagente. Agente . Framework . Jacamo. Estacionamento.

ABSTRACT

CASTRO, Lucas Fernando Souza de. **Modeling and development of a multiagent system through JaCaMo Framework to allocate spots in a smart parking system**. 2015. 78 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2015.

The development of multi-agent systems involves a new paradigm of development and abstraction in the design of systems, since new elements act and interfere with the execution, as the agents and their interactions with each other, the environment in which they live and the rules governing the environment they are inserted. To this end, various tools are being developed in order to provide a complete and simple development of multi-agent systems. One of the tools that provides such capabilities is the JaCaMo framework. This paper presents the development of a multi-agent system places allocation via a centralizing agent for intelligent parking lots using the JaCaMo platform as a tool for development of multiagent system.

Keywords: Multiagent system. Agent . Framework . Jacamo. Parking lot.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

BDI	*Belief, Desire and Intention*
Cartago	*Common ARTifact infrastructure for AGents Open environments*
dMARS	*Distributed multi-agent reasoning system*
JaCaMo	Jason + Cartago + Moise
Jason	*A Java-based interpreter for an extended version of AgentSpeak*
MAPS	*MultiAgent Parking System*
PRS	*Procedural Reasoning System*
SMA	Sistema Multiagente

SUMÁRIO

1 INTRODUÇÃO

Com o advento do homem moderno pós revolução industrial, a demanda por instrumentos capazes de facilitar as tarefas tornou-se cada vez maior. Porém, foi no fim século XXI que notou-se uma maior interação da tecnologia na vida cotidiana do homem, tornando-se essencial ou fundamental para a comunicação, informação e aprendizado. Com esta crescente demanda por tecnologia, o homem começou a buscá-la não apenas como um acessório ou uma ferramenta, mas sim um objeto inserido sempre ao seu meio de novas formas e funcionalidades.

Devido a essa alta demanda por tecnologia, estabelecimentos, lares e até cidades quebraram barreiras a fim de não serem apenas um mero meio de convívio social, mas sim um local onde é possível a interação entre objetos tecnológicos e os seres humanos presentes nesse ambiente, tornando-se assim locais digitais ou inteligentes (CARAGLIU; BO; NIJKAMP, 2011).

Cidades inteligentes têm o objetivo de aprimorar os atuais recursos através do uso da tecnologia da informação aplicada dentro do conceito urbano a fim de minimizar ou otimizar o uso destes mesmos recursos para a redução de custos e/ou tempo. Há inúmeros artefatos em uma cidade que podem ser otimizados, sendo um dos principais desses, o trânsito. Grandes cidades e até as de médio porte enfrentam essa situação diariamente, gerando assim um problema que pode prejudicar a interação social, causar a poluição do meio ambiente e até mesmo afetar a economia local.

Dentro do quesito trânsito, podemos elencar outros inúmeros sub-problemas, como: otimização de tráfego e automatização de estacionamentos. Inúmeras cidades de médio e grande porte possuem um número limitado de vagas de estacionamento condizente ao número de veículos que circulam diariamente. Devido ao número reduzido de vagas, inúmeros estabelecimentos possuem estacionamentos privados a fim de proporcionar um diferencial ao seus clientes, entre os quais se destacam: estádios, teatros, shoppings centers, entre outros. Todavia, mesmo com a facilidade desses estacionamentos privados, existe o impasse de que muitas vezes este estacionamento é demasiado grande, gerando assim a necessidade dos clientes gastarem um tempo considerável a fim de localizar uma vaga adequada para seu veículo. Cidades como por exemplo São Francisco no estado da Califórnia, Estados Unidos, desenvolveram um sistema de previsão de vagas, no qual é possível verificar as condições de utilização das vagas ao longo da cidade e verificar os respectivos preços (PARKINGEDGE, 2013) e (SFPARK, 2015). Estacionamentos inteligentes, ou simplesmente *smart parking* tornaram-se uma área de pesquisa com uma grande quantidade de estudos, o qual (REVATHI; DHULIPALA, 2012) define como um estacionamento que utiliza tecnologias a fim de proporcionar a automatização das tarefas diárias, reduzindo assim o uso de recursos.. Dentre essas pesquisas é possível identificar a utilização de abordagens que aplicam Sistemas Multiagentes na tentativa de modelar e implementar todos os componentes que fazem parte de um *smart parking*.

Para (WOOLDRIDGE, 2009) define-se um sistema multiagente (SMA) como um sis-

tema composto por agentes autônomos que possuem objetivos a serem cumpridos e estão inseridos em um ambiente dinâmico. Com o intuito de tornar o desenvolvimento de um sistema multiagente capaz de empregar conceitos do raciocínio humano, é utilizado o modelo BDI (*Belief, Desire e Intention*), o qual é empregado como base para o desenvolvimento dos agentes que compõem o sistema (BORDINI; HüBNER; WOOLDRIDGE, 2007). Além disso, é de extrema importância a utilização de uma linguagem capaz de desenvolver os agentes com base no modelo BDI, bem como ferramentas que sejam capazes de modelar e implementar as partes que compõem o ambiente onde o agente está inserido e também as normas que regem este ambiente, definindo-se assim o que um agente pode ou não fazer.

Diante desses requisitos para o desenvolvimento de um sistema multiagente, foi desenvolvido o *framework* JaCaMo para a implementação de SMAs com base no modelo BDI (JACAMO, 2011) [1] Esse *framework* composto por três módulos[2]: uma linguagem para o desenvolvimento e execução dos agentes; um *framework* capaz de implementar o ambiente onde os agentes estão inseridos; uma ferramenta capaz de estabelecer normas no ambiente ou organização onde os agentes atuam (JACAMO, 2011). A linguagem responsável pela implementação dos agentes é o Jason, que é escrita em linguagem Java e é uma extensão da linguagem AgentSpeak(L) (BORDINI; HüBNER; WOOLDRIDGE, 2007). Para o desenvolvimento do ambiente, é utilizado o *framework* Cartago, sendo ele o responsável pela implementação dos artefatos em que os agentes interagem. E finalmente, a ferramenta Moise é a responsável pelo estabelecimento das normas que permitem ou requerem ações dos agentes na organização.

O presente trabalho propõe a modelagem e implementação de um sistema multiagente capaz de alocar vagas para estacionamentos privados, onde o processo de alocação de vagas torna-se responsabilidade do sistema proposto e o estacionamento caracteriza-se por ter um agente centralizador responsável pelas vagas do estabelecimento. O sistema desenvolvido utilizou o *framework* JaCaMo como plataforma para o desenvolvimento do SMA. Destaca-se ainda, que esse trabalho está inserido em um projeto de pesquisa do grupo GPAS (Grupo de Pesquisa em Agentes de Software - DAINF-PG) denonominado `MAPS (MultiAgent Parking System)` que tem como objetivo a extensão desse sistema multiagente, onde além da implementação do SMA, é proposto um mecanismo que determina o grau de confiança de cada agente que compõe o sistema (GONçALVES; ALVES, 2015). Além disso, esse projeto de pesquisa tem como objetivo a implementação das regras sociais que regem o estacionamento, bem como a simulação do sistema multiagente em diferentes ferramentas e condições.

Este trabalho subdivide-se na seguinte forma: Capítulo 2 apresenta os trabalhos relacionados, o Capítulo 3 os conceitos básicos de Agentes e Sistemas Multiagentes, Capítulo 4 a descrição do funcionamento do *framework* JaCaMo e as integrações do módulo do Jason com Cartago, Capítulo 5 a descrição detalhada do SMA desenvolvido, e por fim, o capítulo 6 com a conclusão.

---

[1] O *framework* JaCaMo foi desenvolvido por: Rafael Bordini, Jomi Hubner, Oliver Boissier, Alessandro Ricci e Andrea Santi

[2] Adicionalmente, em sua última versão possui um quarto módulo para lidar com a comunicação entre os agentes

1.1 OBJETIVOS

1.1.1 Objetivo Geral

O objetivo geral deste trabalho está centrado na modelagem e implementação de um sistema multiagente utilizando a plataforma JaCaMo a ser utilizado no projeto MAPS.

1.1.2 Objetivos Específicos

- Levantamento dos requisitos do sistema multiagente;
- Construção dos modelos do SMA por meio da metodologia Prometheus;
- Implementação dos agentes utilizando a linguagem Jason;
- Implementação dos artefatos do SMA por meio da ferramenta Cartago;
- Definição da integração dos agentes em Jason com os respectivos artefatos em Cartago;
- Executação de testes e simulações com o *framework* JaCaMo .

1.2 DELIMITAÇÕES DO TRABALHO

Esta seção descreve as delimitações e o escopo do Sistema Multiagente aqui proposto.

- Utilização do *framework* JaCaMo : O SMA desenvolvido utilizou apenas os módulos Jason (Programação dos agentes) e Cartago (Implementação dos artefatos). A etapa de definição das normas do SMA por meio do Moise será implementada por um outro integrante do Grupo de Pesquisa MAPS.
- O ambiente considerado para o SMA é um estacionamento privativo, tendo a figura de um responsável pelo estacionamento e não considerado características "abertas" de um estacionamento público;
- O agente centralizador gerente (*manager*) é o responsável pelo gerenciamento das vagas no estacionamento, sendo assim o SMA tem um ponto central de falha. Portanto, uma extensão natural é desenvolver uma versão do SMA onde as vagas são negociadas de forma distribuída;
- Não há localização geográfica das vagas no estacionamento, as vagas são apenas listadas por um identificador único.

1.3 JUSTIFICATIVA

O crescimento de tecnologias a fim de tornar as atividades rotineiras mais fáceis de serem executadas tem se tornado o alvo de diversas áreas de pesquisas, dando origem a conceitos que até então não eram citados, como no caso da Internet das Coisas, casas inteligentes e até cidades inteiras automatizadas e inteligentes. Nesse contexto de modernização desses conceitos, em específico na perspectiva dos sistemas que são necessários para o desenvolvimento de tais tecnologias "modernas", acaba surgindo uma complexidade grande para tal desenvolvimento, visto que há inúmeras variáveis a serem coordenadas e implementadas inseridas em um ambiente dinâmico. Em contrapartida, há soluções que visam justamente esse cenário, diversas variáveis e um ambiente dinâmico. Uma dessas soluções são os sistemas multiagentes.

O desenvolvimento de sistemas multiagentes possui como objetivo modelar e implementar múltiplos agentes em um ambiente dinâmico, o qual os agentes interagem em prol de atingir um objetivo mútuo (WOOLDRIDGE, 2009). Com base nisso, sistemas multiagentes vêm sendo utilizados em diversos cenários, sendo um destes cenários o de cidades inteligentes. O desenvolvimento de sistemas multiagentes relacionados a cidades inteligentes deve-se ao fato das cidades comportarem vários elementos que interagem simultâneamente (agentes que podem representar motoristas, veículos, dispositivos eletrônicos, sinalização, entre outros) em um ambiente dinâmico e complexo (ambiente que representa a cidade em si).

Dentro do contexto das cidades inteligentes, o trânsito e os estacionamentos mostram ser uma sub-área que necessita atenção, visto o grande crescimento de veículos que circulam diariamente nas cidades. Em (NAPOLI; NOCERA; ROSSI, 2014b), Nápoles na Itália foi o ambiente de um sistema multiagente a fim de otimizar as vagas de estacionamento com o objetivo de evitar congestionamentos no centro da cidade e oferecer aos motoristas vagas de estacionamento de acordo com suas necessidades (distância, tempo de permanência, preço), e de acordo com as exigências do motorista um agente gerente envia ao motorista uma oferta de vaga, podendo o motorista aceitá-la ou fazer uma nova requisição de vaga até que sua exigência seja atendida. Além da questão de alocação de vagas, há a otimização de como esta alocação é realizada, visto que um sistema multiagente pode utilizar um método não eficiente para atribuir as vagas, sendo assim, em (ZHAO; ZHAO; HAI, 2014) os autores desenvolveram um algoritmo de alocação de vagas com base na distância do motorista até a vaga e suas necessidades.

O desenvolvimento do sistema multiagente do atual trabalho destina-se a alocação de vagas para estacionamentos com a utilização de um agente centralizador para atribuição das vagas aos agentes motoristas, e além disso verificando as condições que o motorista possui para a vaga, dando preferência a motoristas que contribuem com o estacionamento através de um grau de confiança (*trust*). Para o desenvolvimento de tal sistema foi necessário analisar soluções existentes que pudessem fornecer ferramentas para implementar os agentes e o ambiente que os rege. Com esse objetivo, o sistema desenvolvido baseou-se no *framework* JaCaMo , o qual é

uma ferramenta que proporciona um alto grau de abstração para os agentes, visto que através desse *framework* é possivel realizar a programação dos agentes, criação de artefatos, gerenciar a comunicação dos agentes. Além disso, o JaCaMo é composto por três principais componentes, sendo eles:

1. Programação dos agentes baseada na linguagem Jason com a utilização da arquitetura BDI;
2. Implementação do ambiente e seus artefatos com a utilização da ferramenta CartAgO;
3. Modelagem e implementação das normas da organização utilizando a ferramenta Moise+. O atual trabalho não empregou as normas sociais oferecidas pelo Moise+, ficando essas regras a serem implementadas futuramente por um outro membro do grupo de pesquisa MAPS.

2 TRABALHOS RELACIONADOS

Este capítulo possui um breve levantamento sobre os trabalhos relacionados com a área de sistemas multiagentes e suas ferramentas, bem como a respeito de cidades inteligentes.

O levantamento bibliográfico, bem como os trabalhos relacionados, foram baseados utilizando os seguintes repositórios:

- IEEEXplore - http://ieeexplore.ieee.org
- Sciello - http://www.scielo.org
- ACM - http://www.dl.acm.org
- Google Scholar - http://www.scholar.google.com
- Research Gate - http://www.researchgate.com
- Períodicos de eventos da área, tais como:
  - AAMAS (*International Conference on Autonomous Agents and Multiagent Systems*)
  - WESAAC (Workshop-Escola de Sistemas de Agentes, seus Ambientes e apliCações)
- Repositórios de universidades brasileiras, tais como:
  - Universidade Federal de Santa Catarina
  - Universidade Tecnológica Federal do Paraná

Especificamente foi utilizado uma expressão lógica a fim de filtrar os resultados dos repositórios acima, pois os mesmos possuem a opção de busca avançada, sendo possível a utilização de palavras chaves com operadores lógicos.

$$(((multiagent \wedge agent) \wedge (BDI \vee jason \vee cartago \vee moise \vee JaCaMo \vee smartcity \vee smartparking \vee parking \vee city \vee traffic) \neg fuzzy) \quad (2.1)$$

Com base na definição mostrada em (2.1) os resultados foram divididos em três grandes categorias: Cidades inteligentes e trânsito, desenvolvimento de sistemas multiagentes utilizando a plataforma JaCaMo: e soluções para alocações de vagas em estacionamentos.

1. Cidades inteligentes e trânsito:

   O termo cidade inteligente abriga diversos sub-termos, devido ao fato de uma cidade possuir diversos elementos que a compõem, não sendo diferente para uma cidade inteligente.

(CARAGLIU; BO; NIJKAMP, 2011) atribuem significados ao termo *smartcity* o qual pode ser decomposto em mais seis termos, sendo eles: *smart economy, smart mobility, a smart environment, smart people; smart living, e smart governance* ou economia inteligente, mobilidade inteligente, ambiente inteligente, pessoas inteligentes, convívio inteligente e governança inteligente respectivamente. Além disso, elenca algumas cidades européias que já possuem tais características e como essas podem afetar os profissionais atuantes nestas cidades. No contexto do projeto, o termo *smart parking* enquadra-se em três categorias: *smart mobility, smart living* e *smart environment*. Em específico, o termo *smart parking* é definido como um estacionamento que utilizam tecnologias que procuram viabilizar e automatizar as rotinas diárias de um estacionamento (REVATHI; DHULIPALA, 2012).

Além do fato de viabilizar e otimizar o estacionamento em si, é apresentada diversas maneiras de otimização do trânsito em um âmbito geral, como algoritmos de otimização para trânsito com base os sistemas multiagentes (ITO *et al.*, 2012). Além disso, autores elencam a importância da modelagem e simulação desses sistemas para verificar sua eficácia, e apontar as vantagens da utilização dos sistemas multiagentes para o tema trânsito (BAZZAN; KLüGL, 2013). Outro ponto importante é a análise dos fatores que norteiam um estacionamento inteligente como o fator de reciprocidade, dinheiro, altruísmo e reputação que os motoristas possuem perante o estacionamento (KOSTER; KOCH; BAZZAN, 2014). Além da questão de alocação de vagas, estudos também analisam o excesso de vagas concentradas em um único local, as quais devem ser remanejadas de acordo com a utilização, como é o caso da cidade de Dallas, no estado do Texas, onde está sendo desenvolvido um estudo a fim de verificar se há vagas de espaço no centro do cidade suficientes, ou se as existentes não estão má distribuídas (WILONSKY, 2015).

2. Desenvolvimento de sistemas multiagentes utilizando as ferramentas da plataforma JaCaMo :

   Devido ao desenvolvimento de um sistema multiagente envolver diversos fatores como os agentes em si, o ambiente, a comunicação e as normas sociais, há inúmeras ferramentas que destacam-se pela eficácia que apresentam. A utilização do JaCaMo objetiva o desenvolvimento completo do sistema multiagente. Para verificar tal eficácia, há trabalhos envolvendo diferentes contextos da área de agentes coma a utilização do JaCaMo ou de seus módulos integrantes.

   Em (ZHANG *et al.*, 2014) é apresentado o desenvolvimento de um sistema multiagente utilizando o JaCaMo para a modelagem de uma *smart home* para economizar e otimizar a distribuição de energia elétrica. Além desse trabalho, (ROLOFF *et al.*, 2014) é proposta um sistema multiagente para o controle de produção de placas de circuitos impressos baseada em uma abordagem de sistemas multiagentes.

   As ferramentas para o desenvolvimento de um SMA promovem um sistema flexível e ro-

busto, porém devida a tal robustez, o uso de recursos computacionais torna-se elevado. Em (JR., 2015) é analisado a perfomance da linguagem Jason e de que maneira é possível otimizá-la.

3. Soluções para alocações de vagas em estacionamentos:

   Várias cidades desenvolveram seus próprios sistemas de alocação ou previsão de vagas em estacionamentos, como no caso da empresa BestParking (PARKINGEDGE, 2013). Através de um sistema on-line a empresa aloca e reserva vagas em diversos estacionamentos de acordo com a preferência do cliente (carro, preço, horário), o sistema cobre mais de 100 cidades na América do Norte. Outro exemplo semelhante é em São Francisco no estado da Califórnia nos Estados Unidos, a empresa SFPark desenvolveu um sistema para checar as vagas dos estacionamentos da cidade e informar ao usuário o preço das mesmas para que o motorista escolha o local adequado para estacionar (SFPARK, 2015). Na Europa, em particular Pisa (Itália) (GRIFFITHS, 2014) os motoristas podem utilizar aplicativo móvel para requisitar vagas aos seus veículos. A comunicação entre o aplicativo e o sistema de alocação de vagas é realizada utilizando sensores instalados nos postes da rua e quando uma vaga é encontrada pelo sistema, o motorista recebe a informação da vaga através do aplicativo. De modo similar, (RICO *et al.*, 2013) desenvolve uma plataforma de alocação de vagas para estacionamentos privados onde subdivide-se em 4 módulos: interface com o usuário (aplicativo Android), módulo de verificação de estado da vaga, módulo de comunicação e servidor de administração para o gerenciamento das vagas.

   Na perspectiva de soluções utilizando agentes, (NAPOLI; NOCERA; ROSSI, 2014a) apresentam uma proposta para negociação de vagas entre agentes em uma cidade inteligente através do protocolo de negociação iterada, onde através da representação de um agente centralizador as vagas dos estacionamentos são negociadas. Essa proposta é simular ao sistema proposto nesse trabalho devido ao fato da problemática da alocação de vagas através de um agente centralizador. Na sequência, em (NAPOLI; NOCERA; ROSSI, 2014b) os mesmos autores descrevem com mais detalhes como a negociação entre o agente centralizador e os agentes motoristas ocorrem. Para a negociação, foi utilizado o protocolo de iteração FIPA, o qual é baseado em *rounds* de negociação. E finalmente, os autores apresentam uma solução desse sistema de alocação das vagas em Nápoli utilizando a plataforma JADE para os agentes e ferramentas WEB para a captura de informações da cidade como as informações e distâncias das vagas (NOCERA; NAPOLI; ROSSI, 2014).

   E por fim, (ZHAO; ZHAO; HAI, 2014) define um algoritmo para otimização dos espaços para vagas de estacionamento a fim de tornar alocação de vagas otimizada de acordo com o valor da distância da vaga até o motorista.

Com base nos trabalhos apresentados, o desenvolvimento de soluções visando a otimi-

zação de trânsito e estacionamentos têm se tornado frequente devido ao constante crescimento das cidades e o seu número de automóveis. Em particular no caso dos estacionamentos, há soluções para a questão de alocação de vagas utilizando sistemas multiagentes. Em destaque para o trabalho apresentado em (NOCERA; NAPOLI; ROSSI, 2014) onde de modo similar com a proposta desse atual trabalho, é utilizado um agente centralizador, o qual possui conhecimento e controle sobre as vagas, e aguarda por requisições dos agentes motoristas para vagas. Além das soluções utilizando agentes, há outras que visam também viabilizar a alocação de vagas através de sistemas WEB, como foi o exemplo da cidade de São Francisco, nos EUA, (PARKINGEDGE, 2013). De modo geral, há a preocupação das entidades governamentais a respeito dos estacionamentos em grandes cidades, devido ao seu crescimento exponencial, como é o exemplo citado anteriormente de Dallas, no estado do Texas (WILONSKY, 2015). Com base nesse contexto, o trabalho aqui apresentado alia-se a esses objetivos, pois está inserido no cenário de estacionamentos inteligentes e pretende aplicar sistemas multiagentes para organizar e alocar vagas de estacionamento.

3 AGENTES E SISTEMAS MULTIAGENTES

Um agente é um software no qual é definido um objetivo, entretanto, diferentemente de outros tipos de software, o agente possui certa autonomia para cumprir com o seu propósito e assim decidir por conta própria qual o melhor caminho para alcançá-lo. Além disso, um agente está inserido em um ambiente o qual o percebe e é capaz de comunicar-se com outros agentes nesse mesmo ambiente (WOOLDRIDGE, 2009).

A capacidade de um agente é limitada na verdade pelo conhecimento que possui, pelos recursos de hardware disponíveis e pela sua percepção do ambiente no qual ele se encontra inserido. Além disso, um agente deve ser capaz de trocar informações com outros agentes formando uma rede social onde há uma cooperação para a solução de um problema (SYCARA, 1998).

Um agente, além disso, é todo aquele capaz de perceber seu ambiente por meio de sensores e de agir sobre esse ambiente por meio de atuadores. Como por exemplo, em um ser humano são os olhos e ouvidos os sensores e mãos, boca, pernas e outras partes do corpo como atuadores ou até mesmo sensores (RUSSELL; NORVIG, 2004). A figura 1 ilustra um agente interagindo com o ambiente por meio dos atuadores.

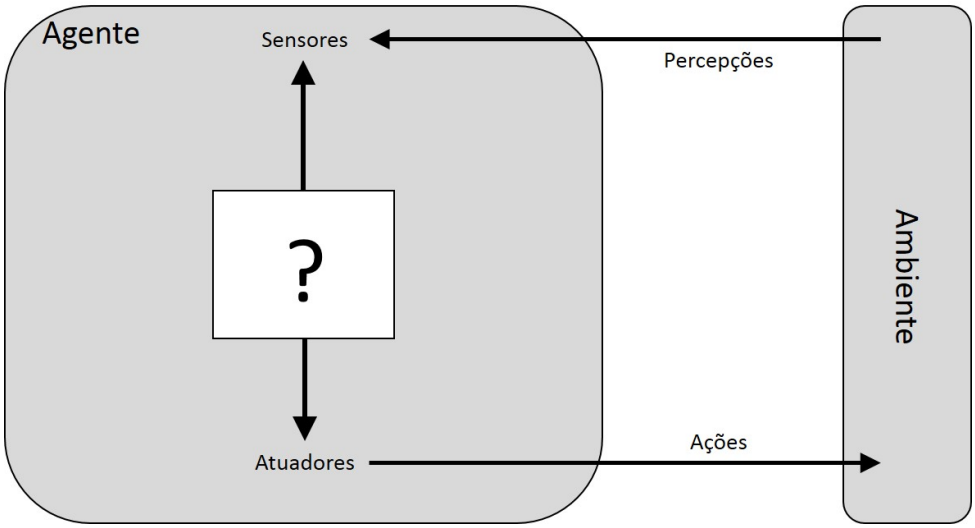

Figura 1 – Agente interagindo com o ambiente por meio de sensores e atuadores

Fonte: Adaptado de (RUSSELL; NORVIG, 2004)

Com base nessa estrutura, é possível estabelecer que vários agentes sejam empregados na cooperação para a resolução de problemas extremamente complexos, sendo que cada um utiliza o paradigma mais apropriado na solução de algum aspecto específico do problema. A cooperação entre os agentes, não necessariamente é relacionada com a cooperação propriamente dita, como por exemplo, o compartilhamento de recursos, mas sim, em relação a habilidade social que um agente possui com os demais agentes presentes no ambiente.

Finalmente, um sistema multiagente é um conjunto de agentes que compartilham o mesmo ambiente e são comunicáveis entre si, porém, cada agente pode ter um grau maior de

influência que um outro agente no mesmo ambiente, caracterizando-se assim a possibilidade de níveis de hierarquia dentro do sistema multiagente (WOOLDRIDGE, 2009).

O restante desse capítulo divide-se da seguinte forma: a seção 3.1 as principais características, a seção 3.2 o modelo BDI e por fim a seção 3.3 a respeito de modelagem de sistemas multiagentes.

3.1 PRINCIPAIS CARACTERÍSTICAS

Como citado no início desse capítulo, um agente possui importantes características que definem como ele é, bem como se comporta em um determinado ambiente. Um agente pode ser rígido ou flexível, reativo ou cognitivo, entre outras particularidades, porém, existem algumas características essenciais para uma solução cooperativa de problemas: (WOOLDRIDGE, 2009)

- Autonomia;
- Proatividade;
- Reatividade;
- Habilidade Social.

3.1.1 Autonomia

Uma das principais, se não a principal, característica que define um agente é a sua autonomia. Sistemas computacionais comuns, como por exemplo, um sistema desenvolvido utilizando uma linguagem orientada a objetos é destinado a seguir uma funcionalidade em específico, baseado em um objetivo inicial onde são definidas regras e uma maneira única de se atingir esse objetivo. Um agente autônomo é aquele que decide qual é a abordagem ideal para a solução de um problema, seja cooperar ou não com o problema, utilizar as percepções do ambiente ou não, e com base no seu conhecimento tomar uma decisão. Sendo assim, um agente autônomo forma as suas decisões independentemente de como irá atingir os objetivos delegados a ele e as decisões sob seu controle e não controlada por outros, tal como por um usuário (WOOLDRIDGE, 2009).

Isso pode implicar um problema em um sistema multiagente, onde a cooperação pode vir a falhar. Um agente com sua autonomia pode decidir ou não cooperar com os demais agentes por *n*-fatores que, na maioria das vezes, estão além do seu próprio controle e até mesmo de quem os programou. Por essa razão, é crucial definir bem a autonomia do agente e seu comportamento racional para que esse tipo de falha não ocorra (LESSER, 1995).

3.1.2 Proatividade

Característica que define a capacidade de um agente tomar a iniciativa para alcançar um objetivo que lhe foi delegado. Diferentemente do conceito de objetos, por exemplo, em que se espera passivamente até que alguém ou algo invoque algum método para que uma tarefa seja realizada (BORDINI; HüBNER; WOOLDRIDGE, 2007).

3.1.3 Reatividade

Para (BORDINI; HüBNER; WOOLDRIDGE, 2007) basicamente um agente reativo significa ser responsivo para mudanças no ambiente. Além do mais, os agentes podem se distinguir em dois tipos de reatividade: a primeira similar ao reflexo humano e a segunda a uma reatividade com um certo nível de cognitividade.

De modo similar, para (HONAVAR, 1999) um agente é todo aquele que reage ao ambiente. O conceito tenta imitar o comportamento humano, onde o agente está em todo o momento verificando e tomando decisões dependendo das ações vindas do ambiente. Essas reações podem ser rápidas (reflexo) ou podem vir a partir de uma decisão racional.

Atribuir esses conceitos ao agente é uma tarefa difícil, visto que o ambiente onde os agentes estão inseridos é altamente dinâmico. O desenvolvimento do software que irá implementar esse agente reativo torna-se complexo, pois, se para um ser humano analisar situações com diversas variáveis e aplicações é algo difícil, desenvolver um agente capaz de possuir a mesma capacidade analítica, torna-se uma tarefa completamente complexa.

3.1.4 Habilidade Social

O processo de comunicação entre os agentes é um ponto fundamental em um sistema multiagente, visto que, diferentes agentes precisam de informações a todo momento sobre o ambiente e de outros agentes. Essas informações, na maioria das vezes, é proveniente da troca de mensagens entre os agentes.

O processo de troca de mensagens não se restringe apenas ao fato de uma simples troca de informações, uma simples *string* por exemplo, mas, sim mensagens mais complexas que podem coordenar atividades a fim de tornar o ambiente e o sistema multiagente em si muito mais dinâmicos e flexíveis a mudanças. O fator comunicação entre os agentes pode ser dividido em três categorias: comunicação, cooperação e negociação (ALONSO; FUERTES; MARTINEZ, 2008).

- Comunicação: Simples troca de mensagem entre o agente destinatário e o agente remetente;
- Cooperação: Indica a capacidade do agente de responder aos serviços requisitados por outros agentes e oferecer serviços a outros agentes;
- Negociação: Capacidade do agente de realizar compromissos, resolver conflitos e chegar a acordos com outros agentes com o intuito de assegurar o compromisso com seus objetivos.

Como citado no início desse capítulo, diferentes agentes podem estar inseridos no mesmo ambiente com diferentes níveis de influência nesse ambiente, caracterizando um ambiente de cooperação-competição, gerando assim, áreas de conflito entres esses mesmos agentes. A figura 2 ilustra essa característica.

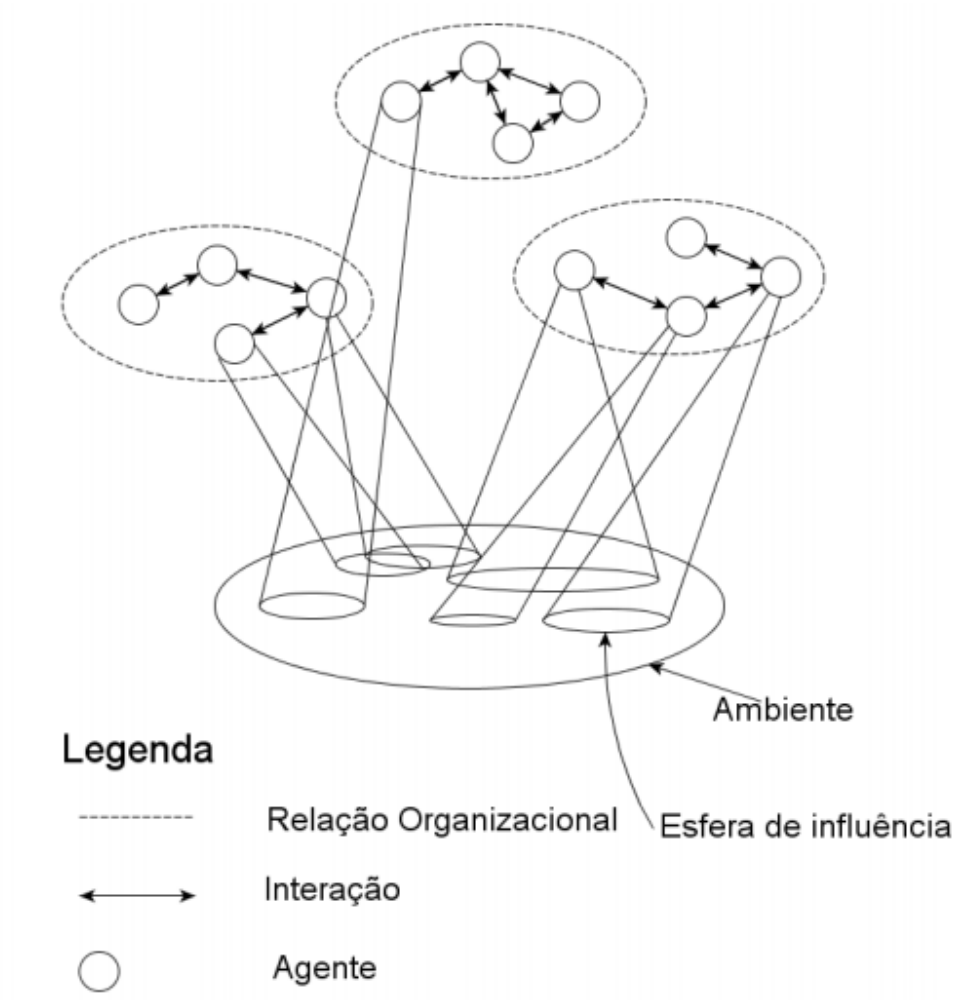

Figura 2 – Agentes com áreas de influência

Fonte: Adaptado de (BORDINI; HüBNER; WOOLDRIDGE, 2007)

3.2 MODELO BDI

BDI, do acrônimo *Belief, Desire e Intention* ou (Crença, Desejo e Intenção) é um modelo que vem sendo analisado pela perspectiva teórica quanto a prática. Entretanto, há uma barreira entre a teoria e a prática visto que há complexidade em se provar os teoremas definidos nas especificações lógicas do modelo (RAO, 1996).

Para (BORDINI; HüBNER; WOOLDRIDGE, 2007)um dos pontos importantes a levar-se em consideração sobre o modelo BDI é a base de que todo sistema computacional possui um estado mental, sendo assim, pode-se considerar o conjunto crença-desejo-intenção. Assim, pode-se definir:

- Crença: São as informações que o agente possui sobre o mundo ou ambiente em que está situado. Essa informação poderá estar desatualizada ou imprecisa.
- Desejos: Define-se por ser todos os possíveis estados que o agente pode desejar alcançar. Entretanto, ter um desejo não implica que o agente automaticamente irá satisfazê-lo ou agir conforme isto. Desejo é uma influência em potencial nas ações do agente. Outro ponto que é fundamental é o fato que um desejo pode ser incompatível com outro desejo. De modo geral, desejos são todas as opções que o agente possui a fim de atingir um objetivo.
- Intenções: São os estados que o agente decidiu seguir em frente, ou até mesmo os objetivos que são delegados a este agente. De modo geral, um agente analisa todas os seus desejos para atingir um objetivo. Ao escolher este desejo, este mesmo desejo torna-se uma intenção. Portanto, um agente com um objetivo a ser cumprido, analisa os seus desejos, escolhe uma ou mais intenções a fim de atingir e completar esse objetivo.

No código 3.1 é apresentado um exemplo básico de um agente baseado no modelo BDI. O agente inicialmente acredita que o agente é feliz e tem como objetivo dizer *"hello"*. Na linha 2 é utilizado o operador !, o qual indica um objetivo a ser atingido. Na linha 3 é exibido o agente executando a intenção de dizer *"hello"*. Na linha 3 é apresentado um plano, o qual é descrito com detalhes na seção 4.3.3.

```
1 feliz(agente).
  !diga(hello).
3 +!diga(X) : feliz(agente) <- .print(X).
```

Código 1 – Exemplo de um agente BDI

Para (BORDINI; HüBNER; WOOLDRIDGE, 2007) o conceito de desenvolver sistemas computacionais em termos de noções mentais, tais como: crença, desejo e intenção é o

componente chave para o modelo BDI. Esse conceito pode-se entender como um novo paradigma: programação orientada a agentes (AOP), o qual se baseia em alguns argumentos, tais como:

- Esse paradigma proporciona uma maneira não-técnica para tratar de sistemas complexos. Para nós, não é necessário nenhum treinamento formal para entender o sistema ou a maneira mental de raciocionar, pois isso é uma parte da nossa capacidade linguística do dia-dia.

- AOP pode-se considerar uma programação pós-declarativa. Em programação procedural, ao desenvolver um algoritmo é necessário definir exatamente como cada ação irá funcionar. Na programação declarativa, por exemplo: Prolog, onde o objetivo é a redução na ênfase no controle de aspectos. Inicia-se com um objetivo que almeja que o sistema o complete, e implementa-se um mecanismo interno de controle, o qual irá procurar uma solução a fim de atingir esse objetivo. Todavia, ao invés de desenvolver sistemas eficientes e robustos em uma linguagem como Prolog, é altamente necessário que o programador possua um detalhado entendimento de como esse mecanismo interno funciona. Em contrapartida, o paradigma AOP é similar à programação declarativa onde inicia-se com os objetivos e deixa a cargo do mecanismo interno de controle agir para atingir esses objetivos, porém, esse mesmo mecanismo interno de controle implementa algum modelo de organismo racional. Esse modelo baseia-se com a racionalidade do entendimento intuitivo do ser humano, do mesmo modo com crenças e desejos.

Dessa forma, agentes BDI são programas inseridos em um ambiente totalmente dinâmico, onde continuamente recebem estímulos provindos desse ambiente, realizando ações (intenções) baseadas no seu estado mental e seu conhecimento sobre o mundo (crenças), bem como a todo momento analisando as opções disponíveis (desejos) a fim de atingir o objetivo a ele delegado. Na figura 3 é ilustrado a arquitetura genérica do modelo BDI.

Onde o módulo FRC da figura 3 (Função de Revisão de Crenças) recebe informações provindas do ambiente ao qual o agente está inserido, podendo assim ler e atualizar a base de crenças do agente. Com as alterações do ambiente, pode-se atribuir novos objetivos ao agente. Já a função "Gera Opções" é responsável pela elaboração de novos desejos (estados) que o agente irá possuir e verificar se esses estados serão atingidos, bem como as intenções com as quais o agente já está comprometido. A função "Filtro"é utilizada para atualizar o conjunto de intenções de acordo com as crenças que o agente possui.

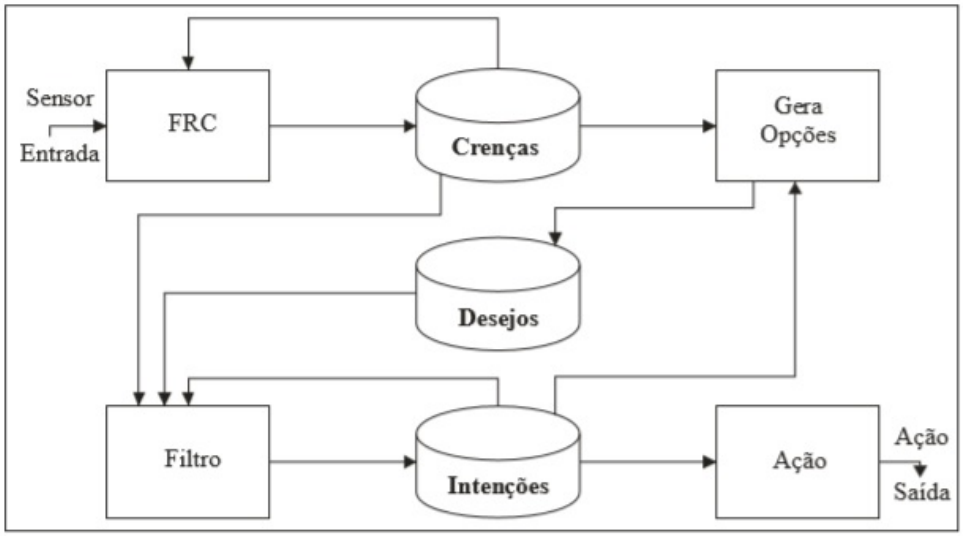

Figura 3 – Arquitetura BDI genérica

Fonte: Adaptado de (WOOLDRIDGE, 2009)

3.3 MODELAGEM DE SISTEMAS MULTIAGENTES

Devido ao fato do desenvolvimento de sistemas multiagentes envolverem diversos fatores como o ambiente em que os agentes estão envolvidos e os próprios agentes em si, o sistema torna se complexo de se desenvolver e modelar. Diante disso, assim como no desenvolvimento de software, há metodologias e modelagens destinadas a sistemas multiagentes, as quais visam modelar as interações dos agentes, o sistema em si, a troca de mensagens e também visa dividir o desenvolvimento do SMA em diferentes perspectivas.

Uma das modelagens existentes é *agent UML*, a qual é baseada na especificação UML (*Unified Modelling Language*), destinada a modelagem de sistemas computacionais, porém com enfoque destinada a agentes. Outra modelagem utilizada o desenvolvimento de SMAs é a metodologia Prometheus, a qual será explicada adiante na subseção 3.1.1.

3.3.1 Metodologia Prometheus

A metodologia Prometeus consiste em um processo detalhado de especificação, design e implementação de agentes inteligentes de maneira fácil e prática com enfoque no desenvolvimento de início ao fim do SMA. A metodologia é composta de três fases (PROMETHEUS, 2015)

1. **Especificação do sistema:** Composta por identificar objetivos dos agentes, desenvolver casos de uso, descrever ações, percepções e interações dos agentes.

2. **Design arquitetural:** Grupo de funcionalidades que determinam o tipo de dados que o agente utiliza, os tipos de agente, análise e elaboração de diagramas de visão geral do sistema multiagente.

3. **Design detalhado:** Consiste em desenvolver diagramas de processos, diagramas de análise dos agentes; definir detalhes dos eventos, planos e crenças.

Para o desenvolvimento do atual trabalho, será empregado todas as fases da metodologia Prometheus através da análise detalhada dos agentes (fase 3) e o diagrama de visão geral do sistema multiagente (fase 2). Para a fase 1, é apresentado diagramas ilustrando os objetivos e as interações dos agentes.q

4 JACAMO - FRAMEWORK DE DESENVOLVIMENTO DE SISTEMAS MULTIAGENTES

Em (WOOLDRIDGE, 2009) e (BORDINI; HüBNER; WOOLDRIDGE, 2007) destaca-se a utilização de ferramentas a fim de tornar a implementação de um sistema multiagente viável, flexível e ao mesmo tempo robusta e capaz de propor soluções que atendem os diversos problemas e modelagem de sistemas utilizando agentes.

Dentre as ferramentas, essas se subdividem em categorias, como: linguagens de programação para os agentes, linguagem de definição dos artefatos dos ambientes, protocolos de comunicação e interação dos agentes, normas da organização e simuladores de sistemas multiagentes e até mesmo plataformas completas para o desenvolvimento de sistemas multiagentes.

Para o desenvolvimento do atual trabalho, como descrito no capítulo 1 o *framework* JaCaMo é utilizado como ferramenta de desenvolvimento para o sistema multiagente proposto. O JaCaMo é composto por três principais módulos, sendo eles: Jason, Cartago e Moise (JACAMO, 2011).

Inicialmente, o projeto baseava-se no modelo de programação JaCa (**Ja**son + **Ca**rtago), utilizando-se apenas das ferramentas Jason para a linguagem de programação para a implementação e execução dos agentes. Já o *framework* Cartago é responsável pelo desenvolvimento e execução do ambiente onde os agentes estão inseridos. A figura 4 ilustra a arquitetura do *framework* JaCaMo . Observa-se que os agentes da *Workspace* A observam e utilizam apenas da sua própria *Workspace*. O mesmo ocorre com os agentes da *Workspace* B.

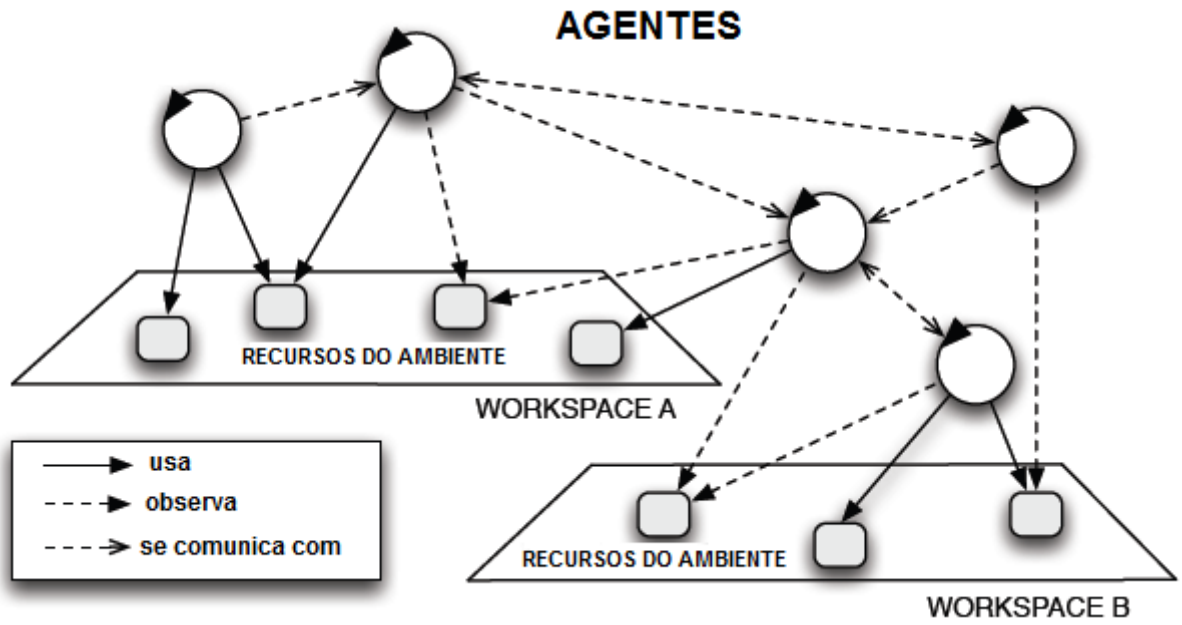

Figura 4 – Arquitetura JaCa

Fonte: Adaptado de (JACAMO, 2011)

O restante do capítulo subdivide-se da seguinte forma, a seção 4.1 apresenta a abordagem do *framework* , a seção 4.2 a descrição da linguagem AgentSpeak(L), a seção 4.3 sobre a linguagem Jason, a seção 4.4 sobre o desenvolvimento dos artefatos do ambiente com o Cartago, a seção 4.5 abordando a integração do Jason com o Cartago e por fim a seção 4.6 com a descrição das configurações do sistema multiagente no JaCaMo por meio da linguagem JCM.

4.1 ABORDAGEM DO FRAMEWORK

Um sistema multiagente em JaCaMo equivale-se a um sistema multiagente delimitado pela ferramenta Moise, que é um modelo organizacional baseado em regras, grupos e missões (MOISE, 2002). No nível do ambiente em que os agentes estão inseridos é utilizado um ambiente compartilhado-distribuído baseado em artefatos no Cartago. A figura 5 ilustra a abordagem geral do *framework* JaCaMo .

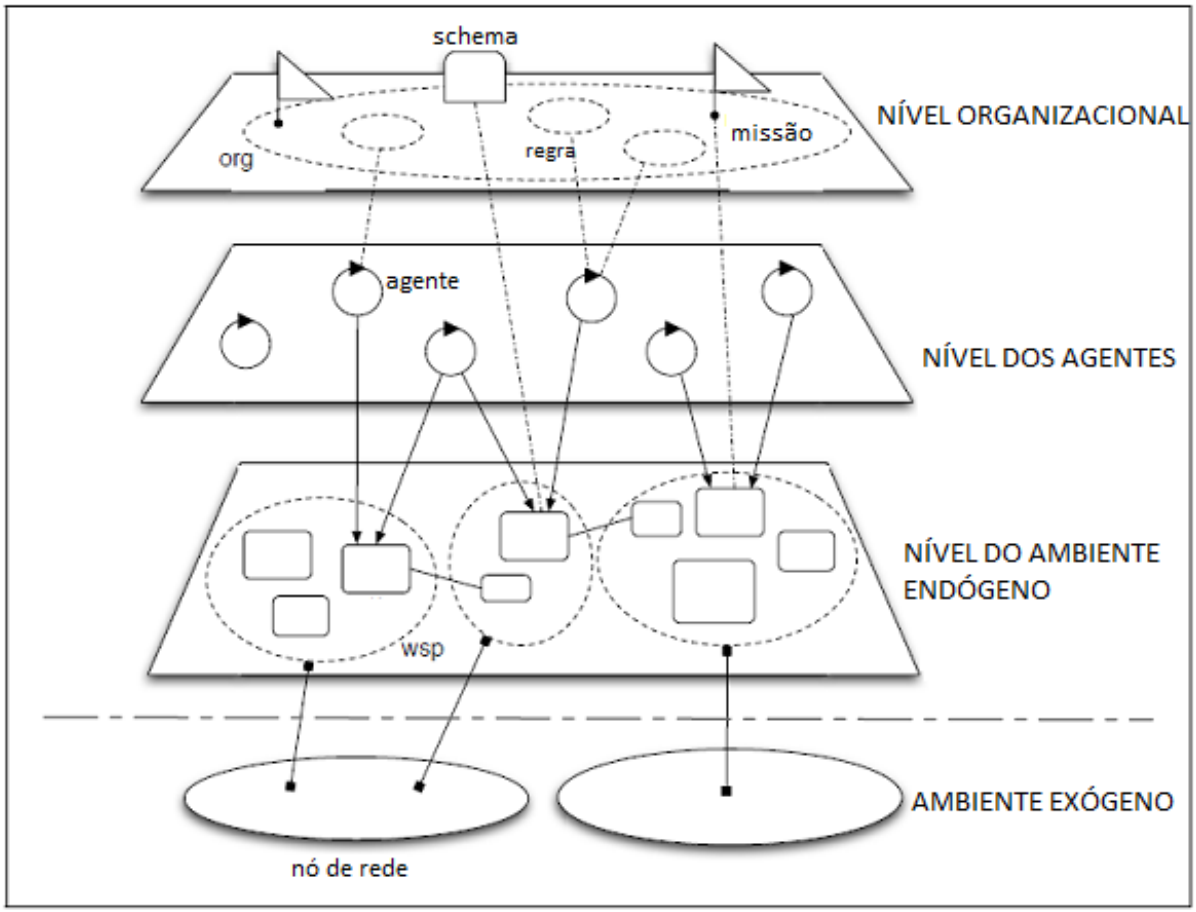

Figura 5 – JaCaMo - Abordagem Geral

Fonte: Adaptado de (JACAMO, 2011)

Cada uma das ferramentas que compõem o *framework* JaCaMo possui o seu próprio conjunto de abstrações para a programação, bem como, seu próprio modelo e meta-modelo de

programação. Portanto, para o *framework* considerou-se como peça fundamental a definição do modelo global de programação, tornando assim possível a integração de todas as abstrações disponíveis em cada plataforma, Jason, Cartago e Moise.

O meta-modelo presente no JaCaMo tem como objetivo definir as dependências, conexões, mapeamentos conceituais das sinergias entre as diferentes abstrações disponíveis nas três plataformas que compõem o *framework* (JACAMO, 2011).

Na dimensão dos agentes com relação ao meta-modelo do Jason, os quais são inspirados pela arquitetura BDI. Nessa dimensão, um agente é composto por um conjunto de crenças, desejos e intenções. Em particular no Jason, desejos entende-se como *goals* ou objetivos. Intenções em Jason entende-se como *plans* ou planos.

Por outro lado, na dimensão do ambiente, cada instância do ambiente Cartago, na figura 6 a entidade "*Work Environment*" é composto por um ou mais entidades de *workspace*. Onde cada *workspace* é formado por um ou mais artefatos. Um artefato provê um conjunto de operações e propriedades observáveis, definindo assim uma interface de uso de artefatos (RICCI; PIUNTI; VIROLI, 2011) (JACAMO, 2011). A entidade *Operation* é responsável pela atualização das propriedades observáveis e eventos especificamente observáveis. Finalmente, a entidade *Manual* é utlizada para representar a descrição de funcionalidades oferecidas por um artefato.

Pelo lado do meta-modelo organizacional do Moise, pode-se separar nos seguintes itens:

- Especificação estrutural é descrita pelo grupo e pelas regras das entidades. Onde ambas definem diferentes grupos de agentes e sub-grupos dentro da organização;
- Especificação funcional é definida pelo esquema social, missão e entidades objetivo. O esquema social define os objetivos da organização (missões).
- Especificação normativa é definida através da entidade *norm* a qual vincula regras para as missões, restringindo assim o comportamento do agente.

4.2 AGENTSPEAK(L)

Anand S. Rao, criador do AgentSpeak(L), traz o ideal de possibilitar uma abordagem prática do modelo BDI a fim de desenvolver sistemas multiagentes utilizando como base o sistema PRS (*Procedural Reasoning System*) e o dMARS (*Distributed Multi-Agent Reasoning System* (RAO, 1996).

O *framework* PRS foi desenvolvido para aplicações embarcadas em ambientes dinâmicos e de tempo de real, bem como para aplicações militares e industriais (INGRAND; GEORGEFF; RAO, 1992). A figura 7 ilustra a arquitetura de um agente baseado na arquitetura BDI e no *framework* PRS.

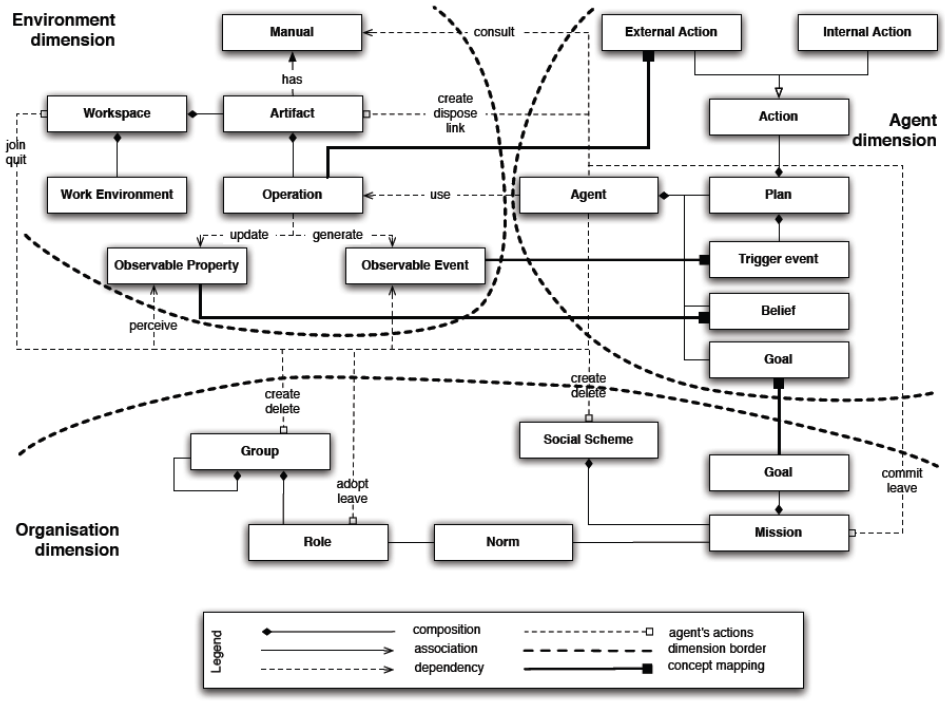

Figura 6 – JaCaMo - Meta-Modelo

Fonte: (JACAMO, 2011)

Esta arquitetura ilustrada na figura 7 consiste em uma base de dados que contém as informações sobre o mundo (ambiente), objetivos a serem atingidos, planos que descrevem como esses objetivos podem ser realmente alcançados e as sequências de ações que devem ser tomadas para tal através das intenções. O interpretador é fundamental, pois é através dele que a interação entre todos os módulos é possível. A comunicação do mundo com o interpretador utilizando a base de dados é responsável para que o sistema perceba e tenha conhecimento sobre o mundo em que está inserido e suas mudanças, visto que esse mundo é altamente dinâmico.

Finalmente, AgentSpeak(L) é uma linguagem de programação com base na arquitetura PRS. Alguns detalhes desnecessários foram retirados da implementação do sistema por ser um modelo inicial, sendo que a própria especificação da linguagem provê como ponto de início para futuras implementações deste modelo.

Características da linguagem

A sintaxe da linguagem AgentSpeak(L) é composta de variáveis, constantes, símbolos de funções, predicados e ações, conectivos, quantificadores e símbolos de pontuação (BORDINI; HüBNER; WOOLDRIDGE, 2007). Além desses, AgentSpeak proporciona outros operadores, tais como:

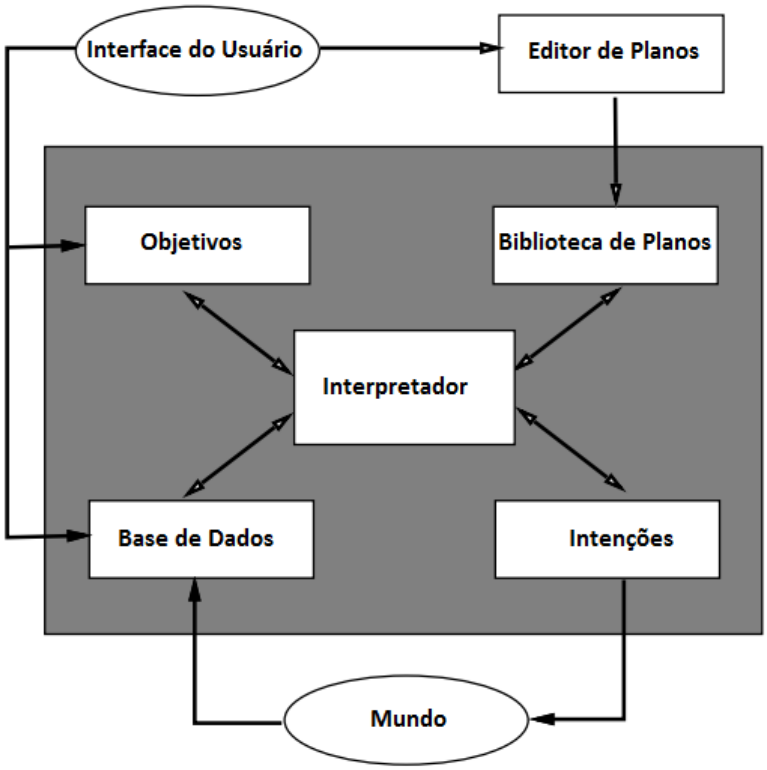

Figura 7 – Arquitetura de um agente baseado em BDI:PRS

Fonte: Adaptado de (MYERS, 1993)

- **!** - Objetivos a serem atingidos;
- **?** - Objetivos utilizados para testes;
- **;** - Utilizado para comandos em cadeia ou sequência;
- **<-** - Utilizado para implicação, por exemplo: em planos e eventos gatilho.

4.3 JASON

A Java-based interpreter for an extended version of AgentSpeak, Jason, é uma extensão da linguagem AgentSpeak(L). Foi desenvolvido utilizando a linguagem Java e possui código aberto sob a licença GNU LGPL e desenvolvido por Jomi F. Hübner e Rafael H. Bordini e outros colaboradores (JASON, 2005).

A interpretação do programa agente efetivamente determina o ciclo de raciocínio do agente. Um agente constantemente está recebendo informações sobre o ambiente através das suas percepções e reagindo de forma a responder a esses estímulos. Inicialmente, o agente possui planos pré-programados em sua base de dados, de como reagir a esses estímulos, porém a escolha

de que maneira o agente irá reagir é realizada de maneira autônoma (BORDINI; HüBNER; WOOLDRIDGE, 2007).

A seguir são descritos os componentes que pertencem a linguagem Jason: crenças, objetivos e planos e suas características.

4.3.1 Crenças

Assim como no AgentSpeak(L), a linguagem Jason possui uma base de crenças inicial, a qual é uma simples coleção de literais, de mesmo modo a uma programação lógica tradicional, sendo assim, a informação é representada através de predicados, como por exemplo:

1. disponivel(spot1)[1];
2. ocupado(spot2);
3. preferencia(driver0, spot0).

Neste exemplo acima, o item 1 expressa que a *spot1* está disponível, uma afirmação. No item 2 é informado ao agente que a *spot2* está ocupada e por fim no item 3 que o agente *driver0* tem preferência para a *spot0*. De modo geral, estes itens representam o que o agente sabe sobre o mundo, ou ambiente que está inserido. Porém, as crenças não podem ser tomadas como verdades absolutas, pois o agente muitas vezes recebe informações imprecisas ou inválidas provindas do ambiente.

Na figura 8 são ilustrados os tipos de termos(objetos) que o Jason utiliza para representar os predicados.

Devido as informações que o agente recebe vindas de um agente no ambiente ou do próprio ambiente, a linguagem Jason possui um recurso de anotações, onde é possível identificar a origem de cada informação recebida. As anotações são identificadas e delimitadas utilizando os colchetes.

1. **disponivel(spot0)[source(driver0)]:** *Spot0* anunciada como disponível informada pelo agente *driver0*;
2. **disponivel(spot1):** Agente acredita que a *spot1* está disponível.
3. **disponivel(spot3)[expires(tomorrow)]:** Agente acredita que a *spot1* está disponível.

[1] Com o objetivo de tornar os termos padronizados com os utilizados no SMA, denota-se:

- spot: valor equivalente ao "vaga";
- driver: valor equivalente a "motorista";
- manager: valor equivalente a "gerente";

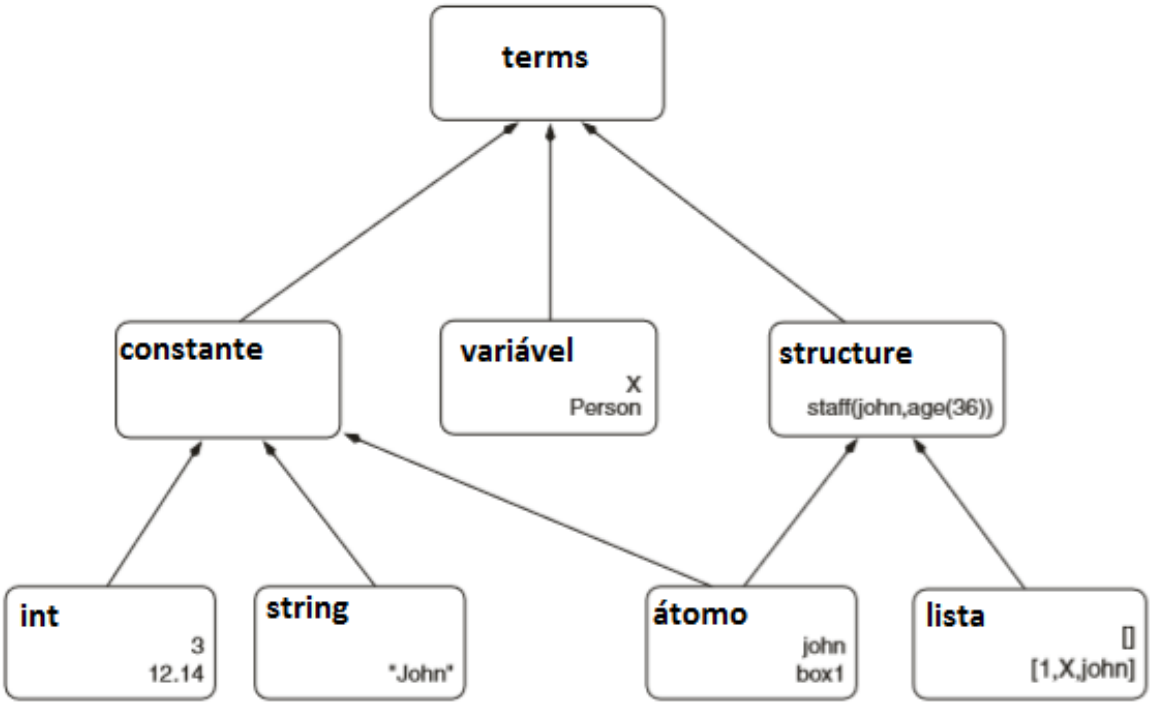

Figura 8 – Tipos de termos do AgentSpeak no Jason

Fonte: Adaptado de (BORDINI; HüBNER; WOOLDRIDGE, 2007)

No item 1, a anotação é apenas uma nota mental ao agente, pois o interpretador do Jason não o leva em conta para acreditar em tal informação *(expires(tomorrow))*. Diferente dos itens 1 e 2, onde há a utilização da anotação "*source*", nesse caso o interpretador do Jason leva em conta essa informação. A anotação em respeito a origem da anotação é válida e importante, pois em muitos casos dependendo da fonte de informação o agente pode desacreditar na crença ou tomar medidas de precaução ao acreditar na informação.

Em um sistema multiagente há três diferentes tipos de fontes de informação para os agentes, sendo elas:

- **Informação perceptiva:** Um agente adquire certas crenças de acordo com o ambiente em que está inserido através das percepções que esse agente tem;
- **Comunicação:** As percepções do ambiente podem ser informadas por um agente a outro agente. É importante o agente destinatário saber a origem dessa informação, visto que esta informação pode estar incompleta ou inválida;
- **Notas mentais:** Possui o objetivo de lembrar o agente informações sobre o passado e até mesmo um lembrete do que deve ser utilizado, como por exemplo na seleção de um plano. O conceito de notas mentais não deve ser confundido com o de anotações, pois notas mentais são apenas informações a serem usadas pelo agente como lembrete, não interferindo diretamente nas ações do agente.

Além disso, o Jason pode automaticamente inserir informações sobre as anotações, por exemplo:

- **source(percept):** Informa ao agente uma informação provinda do ambiente;
- **source(self):** Criação de uma nota mental criada pelo próprio agente a fim de relembrar a informação no futuro.
- **source(agent0):** Informar ao agente que a informação proveu de um agente no ambiente, nesse caso o agent0.

Negação forte

Negação é a fonte de muitas dificuldades em linguagens de programação lógica. Uma abordagem popular é trabalhar a negação como um "mundo fechado" ou a negação como falha. A suposição do "mundo fechado" é definida como: "Qualquer coisa que não é nem conhecida para ser verdade, nem derivada a partir dos fatos conhecidos, utilizando as regras em programa, é assumida como sendo falsa." (BORDINI; HüBNER; WOOLDRIDGE, 2007).

Em Jason, há o suporte para a negação de um predicado, sendo nomeado como negação forte. Para negar uma informação é utilizado o operador ~. A seguir um exemplo de lista de crenças de um agente, chamado agenteManager.

- **disponivel(spot0A)[source(percept):]**: agente *manager* acredita que *spot0A* está disponível;
- **~disponivel(spot1B)[source(driver1)]:** agente *manager* acredita que a *spot1B* não está disponível;
- **~disponivel(spot0B)[source(percept)]:** agente *manager* acredita que a *spot0B* está ocupada;
- **disponivel(spot3C)[source(driver2)]:** agente *manager* acredita que *spot3C* não está ocupada.

4.3.2 Objetivos

Os objetivos na linguagem Jason determinam um estado que o agente deve alcançar para cumprir seu objetivo. De mesmo modo ao AgentSpeak(L), a linguagem Jason possui duas categorias de objetivos: os de teste e os objetivos a serem alcançados. Para os objetivos a serem alcançados, utiliza-se o operador **"!"** e para os de teste o operador **"?"**.

Assim como existe a base de crenças iniciais que o programador define, há também a criação de objetivos iniciais, ou objetivos inicialmente delegados a um agente.

- **!alocarVaga(spot1)**: Objetivo desse agente para alocar a *spot1*;
- **?verificarExistenciaDriver(spot0)**: Objetivo de teste onde o agente verifica se o agente motorista ainda está na *spot0*.

Assim como as crenças, os objetivos são fundamentais para modelar um agente BDI, assim determina-se o que agente sabe sobre o ambiente, bem como, o que deve fazer nesse ambiente. Porém, há a importância dos planos, pois esses definem como o agente irá atingir esses objetivos.

4.3.3 Planos

Um plano utilizando o AgentSpeak(L) é composto em três partes: evento gatilho, o contexto e o corpo. O evento gatilho e o contexto são os que compõem a cabeça do plano. Sendo assim, as três partes são sintaticamente separadas por ":" e "← ". Assim define-se um plano como:

EVENTOGATILHO : CONTEXTO ← CORPO

O evento gatilho informa para o agente as condições para que a escolha do plano seja realizada. O contexto define as regras necessárias para que o evento seja reconhecido e o plano executado seja escolhido. O corpo é composto das ações que o agente irá realizar caso o plano seja escolhido.

Para elaborar um plano é imprescindível que o programador saiba definir os três elementos: evento gatilho, contexto e corpo. Em relação aos eventos, a tabela 1 ilustra como eles podem ser definidos.

Tabela 1 – Definição dos eventos em planos

Notação	Função
$+l$	Adição de crença
$-l$	Remoção de crença
$+!l$	Adição de objetivo a ser alcançado
$-!l$	Remoção de objetivo a ser alcançado
$+?l$	Adição de objetivo de teste
$-?l$	Remoção de objetivo de teste

Fonte: Adaptado de (BORDINI; HüBNER; WOOLDRIDGE, 2007)

Onde *l* é a representação de um literal. Os eventos de adição ou remoção de crença podem ocorrer a qualquer momento, dependendo das mudanças no ambiente. A tabela 2 mostra os tipos de literais que podem ser utilizadas na segunda parte que compõe um plano, o contexto.

Tabela 2 – Uso dos literais no contexto

Notação	Significado
l	Agente acredita que *l* é verdadeira
$\sim l$	Agente acredita que *l* é falsa
not l	Agente não acredita que *l* é verdadeira
not $\sim l$	Agente não acredita que *l* é falsa

Fonte: Adaptado de (BORDINI; HüBNER; WOOLDRIDGE, 2007)

A terceira parte que compõe o plano é o corpo. Esta parte é definida como o curso de ação que o agente irá tomar quando o evento gatilho acontecer e as condições do contexto sejam verdadeiras e então o plano será executado. O conjunto das ações no corpo é delimitado e separado utilizando um ponto-e-vírgula ";".

Ações Internas

Outra característica importante da linguagem Jason é a utilização de ações internas. Essas ações são funções pré-definidas na linguagem e prontas para utilização. Para utilizá-las, usa-se o operador **"."** para invocar as ações.

Uma das ações mais importantes, é a ação *send* a qual é responsável pela comunicação dos agentes. Esta função utiliza o protocolo KQML para o envio e recebimento de mensagens entre os agentes (BORDINI; HüBNER; WOOLDRIDGE, 2007).

A seguir, um exemplo de código em Jason, o qual estabelece um plano e faz uso da ação interna *send*.

```
1 +!alocarVaga(SPOT): isFull(CONDITION) & CONDITION = false <-
          +parking(SPOT,1);
3     .send(DRIVER, tell ,SPOT).
```

Código 2 – Agente em Jason com ação interna e definição de um plano

O código 4.1 inicia com um agente *manager* recebendo uma requisição de um agente *driver*. O agente *manager* verifica se há vaga livre. Sendo assim, o agente *manager* aloca a vaga para o *driver* e passa a crer que a vaga está ocupada (linha 2). Na linha 3 é executada a ação interna *send* com o *manager* enviando ao agente *driver* a informação sobre a vaga requisitada.

4.3.4 Interpretador

O interpretador da linguagem Jason executa um agente com base em um ciclo de raciocínio que pode ser descrito em dez etapas. A figura 9 ilustra de modo geral esse ciclo.

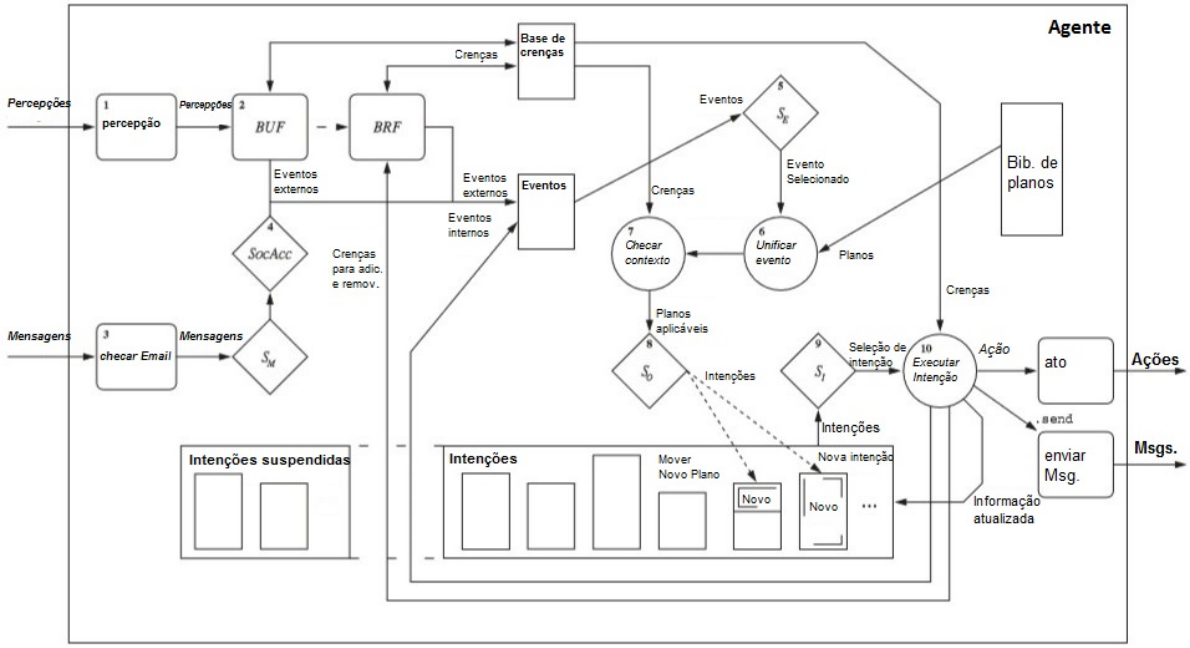

Figura 9 – Ciclo de raciocínio - Linguagem Jason

Fonte: (BORDINI; HüBNER; WOOLDRIDGE, 2007)

Percepção do ambiente

Os agentes necessitam constantemente avaliar o ambiente para assim alterar e validar sua base de crenças.

Atualização da base de crenças

De acordo com as percepções que o agente teve sobre o ambiente, é necessário atualizar a base de crenças. De modo padrão, o agente insere tudo em sua base de crenças sobre o ambiente, porém isso é personalizável, visto que em um ambiente complexo podem haver informações desnecessárias para um agente em particular.

Comunicação entre os agentes

Envio e recebimento de mensagens entre os agentes. Cada agente possui uma caixa de entrada de mensagens. Sendo assim, o agente necessita verificar se há novas mensagens disponíveis.

Escolha das mensagens

No ambiente em que o agente está inserido pode haver inúmeros agentes, e o número de mensagens pode crescer exponencialmente. Assim, nem todas as mensagens recebidas por um agente podem ser interessantes a ele. (BORDINI; HüBNER; WOOLDRIDGE, 2007) definem tais mensagens como "socialmente aceitáveis", onde o agente irá fazer uma seleção das mensagens.

Escolha de eventos

Os eventos necessitam ser processados para que os agentes executem as suas ações e assim alcançar os objetivos. Porém, a cada ciclo de raciocínio o Jason seleciona apenas um evento para ser tratado.

Lista de planos possíveis

Outro fator importante no ciclo é a escolha do plano que o agente irá executar. O agente necessita verificar todos os planos possíveis para determinado evento e decidir se o plano é relevante ou não.

Determinação dos planos utilizáveis

Sendo todos os planos relevantes apontados, o Jason verifica os planos que realmente podem ser utilizados de acordo com o contexto.

Escolha do plano a ser executado

Uma vez que todos os planos possíveis foram analisados e depois verificados se são utilizáveis, escolhe-se um plano a ser executado. De modo padrão, o Jason escolhe o primeiro plano da lista dos possíveis, porém há uma enorme linha de pesquisa na área que determina qual é o melhor método para a escolha de planos (HORTY; POLLACK, 2001).

Escolha da intenção a ser executada

O agente possui uma lista completa de intenções a serem executadas e nessa etapa é selecionada qual intenção será executada.

Execução da intenção

Esta etapa depende de como a fórmula da intenção está sendo executada, podendo ser categorizada em até seis diferentes tipos, sendo eles:

- Ação do ambiente;
- Objetivos a serem atingidos;
- Objetivos de teste;
- Notas mentais;
- Ações internas;
- Expressões.

Há, ainda, um último passo antes de cada novo ciclo iniciar novamente. Esse passo destina-se a verificação das intenções que ficaram aguardando alguma ação do ambiente ou até uma mensagem de outro agente. Caso essa intenção não tenha sido executada, ela é inserida como uma intenção a ser executada no próximo ciclo.

4.4 CARTAGO

Common ARTifact infrastructure for AGents Open environments, Cartago, é um *framework* que possibilita desenvolver e executar ambientes virtuais para sistemas multiagentes.

Cartago é baseado no meta-modelo de agentes e artefatos (A&A) para modelar, o qual é baseado e implementar sistemas multiagentes (CArtAgO, 2006).

Além disso, o *framework* é capaz de isolar o desenvolvimento do sistema multiagente em duas camadas, a programação dos agentes e a programação do ambiente, apenas estabelecendo uma interface comum entre os agentes e o ambiente para que possa haver um nível sociabilidade entre eles. Com isso, o Cartago não é dependente de nenhuma linguagem para agentes em específico, pois seu enfoque é no ambiente. No caso da linguagem Jason, há um suporte específico de integração das ferramentas, como no caso do projeto JaCa (Jason + Cartago) e também no caso do *framework* utilizado no desenvolvimento desse trabalho, o JaCaMo .

4.4.1 Workspaces

Um ambiente em Cartago é dado por um ou vários *workspaces*, sendo esses possivelmente distribuídos em uma rede. Um agente ao pertencer a um ambiente, esse mesmo agente deve obrigatoriamente pertencer a no mínimo um *workspace* para usufruir de um ambiente. Ou seja, um *workspace* está sempre inserido em um ambiente.

4.4.2 Repetório de Ações do Agente e Artefatos

Um artefato é definido como sendo uma entidade não-autônoma dentro de um ambiente. Essa entidade é apenas invocada pelos agentes. No exemplo do estacionamento, pode-se assumir que a cancela é um artefato, pois a mesma só é atividade quando o agente *manager* invoca-a para abrir ou fechar. As ações de um agente em relação ao ambiente é regida de acordo com os artefatos, pois um agente interage com o ambiente através do artefatos.

4.4.3 Artefatos Padrões

Por padrão, cada *workspace* possui um conjunto pré-definido de artefatos que fornecem algumas funcionalidades aos agentes:

- **Artefato *Workspace*:** Fornece as funcionalidades de criação, busca, link, foco em artefatos do *workspace*;
- **Artefato *Node*:** Conectar, criar em local ou *workspaces* remotos;
- **Artefato *Blackboard*:** Funcionalidade de comunicação e coordenação entre os agentes;
- **Artefato *Console*:** Imprimir mensagens na saída padrão.

4.5 INTEGRAÇÃO JASON-CARTAGO

Um dos principais objetivos da plataforma JaCaMo é a integração das diferentes ferramentas Jason, Cartago e Moise. Contudo, é possível a utilização das três ferramentas de forma independente, ou até mesmo realizar integrações entre elas de maneira manual. Com a utilização do JaCaMo essa integração torna-se menos complexa e mais direta. Com base na figura 4, pode-se observar a integração de diferentes agentes inseridos em um mesmo ambiente com diversos artefatos. Através da integração Jason-Cartago os agentes podem invocar ações diretamente ao Cartago, sem a necessidade de importação e exportação de parâmetros. Outro fator, a implementação dos artefatos em Cartago é baseada na linguagem Java, tornando assim o sistema multiagente mais versátil e robusto, visto a aplicabilidade da linguagem Java.

Como citado, através do *framework* JaCaMo é possível que o agente invoque diretamente uma ação provinda de um artefato, ou seja, o agente crê que ele executa a ação, porém, é através do *framework* que ação invocada pelo agente é enviada ao Cartago para que o artefato que implementa a ação em específico a invoque. Entretanto, é necessário que na inicialização do agente, ele execute duas ações para que isso se torne possível.

1. Criação do artefato:

```
1 makeArtifact("a_Gate","maS3.Gate",["Starting"],ArtId)
```

Código 3 – Agente Jason instanciando um artefato

O comando "*makeArtifact*" é uma ação provida pelo *framework* JaCaMo em que o primeiro agente que utilizará um artefato deverá executar a fim de instanciar o artefato. No exemplo acima, o artefato em específico é a Cancela (*Gate*) do estacionamento. O comando é composto por 4 parâmetros, sendo eles:

- **Nome do artefato:** "a_Gate";
- **Localização do artefato:** "maS3.Gate- "maS3- Nome do pacote onde a implementação do artefato encontra-se;
- **Parâmetros de inicialização:** "Starting"
- **Identificador do artefato:** Identificador utilizado para futuras referências ao artefato.

2. *Lookup* no artefato:

```
1 focus(ArtId);
```

Código 4 – Lookup no artefato em Jason

O comando "*focus*" assim como o "*makeArtifact*"é uma ação provida pelo JaCaMo. Após a instanciação do artefato, com esse comando o agente toma conhecimento do artefato e toma como sua ações as ações do artefato, podendo assim invocá-las diretamente sem a necessidade de utilizar o artefato de forma direta. O comando apenas utiliza um parâmetro, o qual é o identificador do artefato.

4.6 CONFIGURAÇÕES DO SISTEMA MULTIAGENTE NO JACAMO POR MEIO DA LINGUAGEM JCM

A fim de tornar o desenvolvimento do SMA centralizado, é utilizado um arquivo de controle geral do SMA, o arquivo baseia-se na linguagem JCM, a qual define as principais funcionalidades do sistema multiagente, tais como: instanciação dos agentes, definição de crenças iniciais e objetivos, instanciação de artefatos, definição do ambiente e outras funcionalidades. Para o atual trabalho, foram definidas as seguintes características:

- Instanciação dos agentes;
- Crenças iniciais para os agentes *Driver*;
- *Lookup* dos artefatos *Control* e *Gate* pelo agente *Manager*;

Tais funcionalidades descritas acima, poderiam ter sido implementadas sem a utilização do arquivo JCM, porém, com o objetivo de tornar o SMA mais flexível, robusto e otimizado foi utilizado o arquivo de controle. No código 5 é apresentado o arquivo JCM.

```
1   mas mAPS{
  agent manager
3     focus: maS3.Control
      focus: maS3.Gate
5         agent m1: driver.asl {
                beliefs: myTrust(100)
7                        timeToSpend(10000)
                         timeToArrive(17514)}
9
          agent m2: driver.asl {
11              beliefs: myTrust(450)
                         timeToSpend(3000)
13                       timeToArrive(25307)}
```

Código 5 – Arquivo JCM

No código 5 é apresentado o SMA com três agentes, sendo um deles o agente *Manager* e os outros dois agentes *Driver*. As crenças do agente *Manager* são dispostas no arquivo referente do agente (MANAGER.ASL). Para os agentes *Driver*, são inseridas as crenças iniciais que o agente possui, sendo elas: *myTrust*, *timeToSpend* e *timeToArrive*. As crenças apresentadas são descritas na seção 5.2.1.

5 MAPS: DESENVOLVIMENTO DE UM SISTEMA MULTIAGENTE PARA ALOCAÇÃO DE VAGAS DE UM ESTACIONAMENTO

Como citado no capítulo de Introdução, o objetivo principal do atual trabalho é desenvolver um sistema multiagente baseado no *framework* JaCaMo para o projeto `MAPS`. Sendo assim, o atual capítulo destina-se a descrever como o estacionamento irá operar e como o sistema multiagente foi desenvolvido a fim de abstrair as funcionalidades exigidas pelo estacionamento.

O termo *Smart* adota-se por ser "inteligente" ou "esperto", que no contexto do projeto, trata-se de um estacionamento inteligente, capaz de administrar as vagas (recursos) para os seus clientes (agentes) de forma automática e otimizada. Os estacionamentos abrangidos pelo trabalho são os privados, ou seja, que estão são utilizados por clientes de shoppings centers, teatros, aeroportos, estádios e entre outros.

Os motoristas (*drivers*) que utilizam o estacionamento são os agentes que interagem e utilizam o sistema multiagente. Já as vagas (*spots*) de estacionamento são consideras os recursos que o SMA utiliza e aloca para os agentes. Com o objetivo de administrar as vagas é utilizado um agente centralizador, nomeado o agente gerente (*manager*) do estacionamento, sendo ele o responsável por informar, alocar e gerenciar todas as vagas do estacionamento. O diagrama de caso de uso exibido na figura 10 ilustra o que os agentes podem realizar.

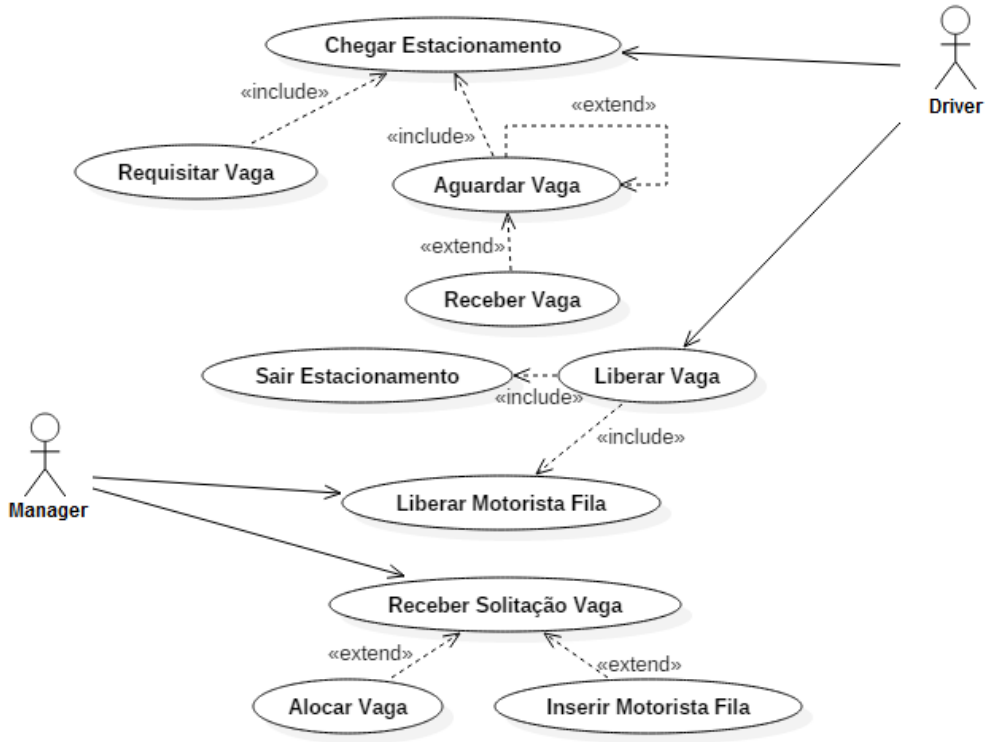

Figura 10 – Diagrama de Caso de Uso - SMA

Fonte: Autoria Própria

5.1 PRINCIPAIS FUNCIONALIDADES

No início e durante o desenvolvimento do sistema multiagente foram elencados diversos requisitos que o SMA deveria ser capaz de proporcionar aos agentes a fim de tornar o sistema robusto, porém flexível e expansível na continuidade do projeto `MAPS`.

- Agente *Manager*:
    1. **Receber solicitação de requisição de vaga:** O *manager* pode a qualquer momento receber requisições providas dos agentes *driver* para vagas no estacionamento. Nem sempre uma requisição de vaga é atendida, pois o estacionamento pode estar lotado;
    2. **Decidir para qual agente alocar uma vaga:** Caso o estacionamento esteja lotado, ou próximo da sua lotação máxima, o *manager* se baseia na valor da confiança (*trust*) dos agentes *driver* e no tempo que ele está aguardando a vaga. Sendo assim, não necessariamente o primeiro *driver* que requisitou uma vaga irá receber a vaga por primeiro;
    3. **Possuir conhecimento e controle sobre o estacionamento:** O agente *manager* possui todo controle do estacionamento, pois é através dele que um agente requisita, ganha, estaciona e deixa uma vaga. Além disso, o *manager* tem conhecimento de todas as vagas, o seu estado, sua localização e quem está ocupando-a;
    4. **Receber o aviso de um motorista que está saindo do estacionamento:** Caso um agente *driver* sair do estacionamento, antes disso ele deve avisar ao *manager* que está deixando a vaga livre. Após isso ocorrer, o *manager* decide qual agente *driver* irá receber a vaga, caso haja uma fila de espera.
- Agente *Driver*:
    1. **Requisitar uma vaga ao agente *manager*:** O *driver* ao chegar no estacionamento requisita uma vaga ao agente *manager*, podendo ser respondido com uma vaga ou tendo que aguardar por uma;
    2. **Receber uma vaga e estacionar:** Ao receber uma vaga, o agente deverá dirigir-se a esta vaga e estacionar o seu veículo;
    3. **Aguardar uma vaga:** Caso o estacionamento esteja lotado, ao requisitar uma vaga, o agente *driver* deverá aguardar até que o agente *manager* o notifique com uma liberação de vaga;
    4. **Possuir características e crenças de um usuário de estacionamento:** A fim de tornar a abstração do agente *driver* próxima a um motorista real, os agentes *driver* presentes no sistema possuem as seguintes crenças: tempo para chegada no estacionamento, tempo aproximado de estadia no estacionamento, vaga que está estacionado e valor de confiança.

5. **Possuir um valor de confiança perante o agente *manager*:** Assim como descrito no item anterior, um agente *driver* é capaz de possuir um valor de confiança (*trust*) perante o *manager*. Este valor compreende as atitudes desse motorista no estacionamento, como por exemplo: não infringir as regras do estacionamento e utilizar o estacionamento de forma regular. Este valor poderá diminuir caso o motorista não cumpra as regras do estacionamento.

A seguir, é apresentado na figura 11 a estrutura básica do estacionamento e seu funcionamento de forma simplificada.

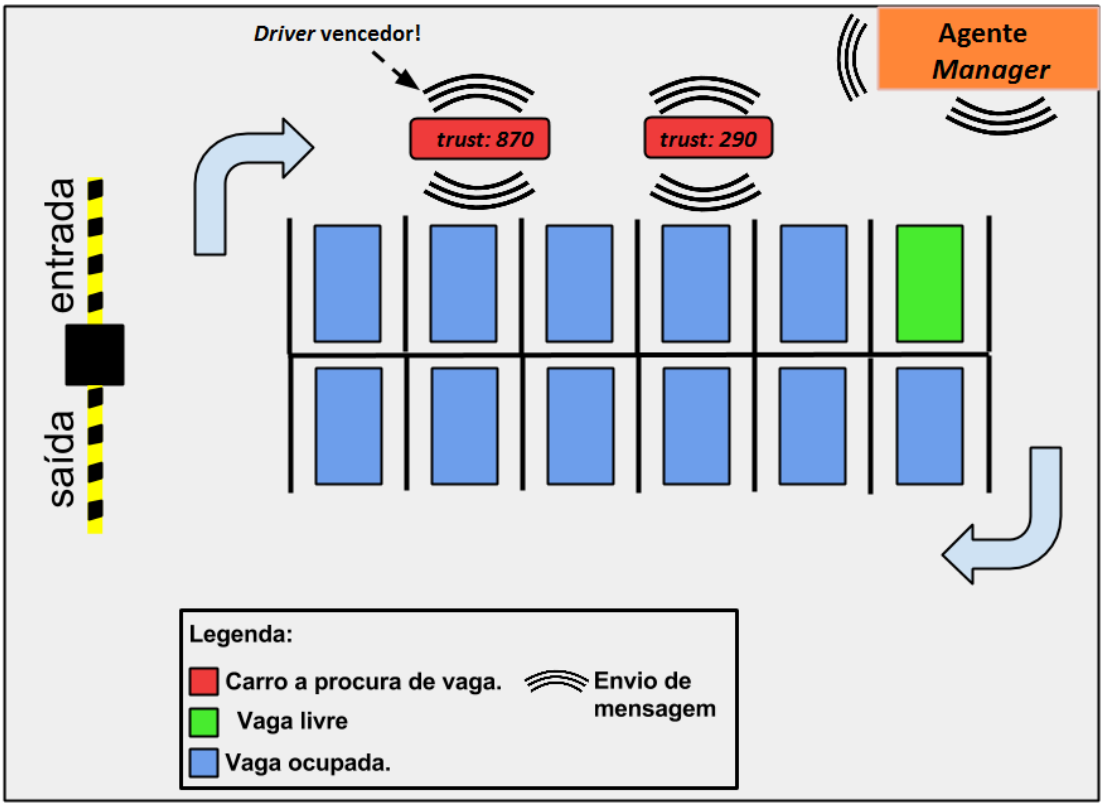

Figura 11 – Estrutura básica do estacionamento

Fonte: Modificado de (GONçALVES; ALVES, 2015)

5.2 IMPLEMENTAÇÃO DO SMA UTILIZANDO O JACAMO

O desenvolvimento do SMA foi realizado em três etapas e utilizando repositórios e controles de versão, como o GitHub[1] e Bitbucket. O grupo de pesquisa MAPS atuou no levantamento de requisitos, exibidos na seção 5.1, e como o SMA deveria ser capaz de tratá-los. As três etapas foram:

1. Implementação dos agentes e suas interações em Jason;
2. Implementação dos artefatos em Cartago;
3. Otimização da utilização dos agentes com os artefatos.

A seguir é apresentado na figura 12 um diagrama de visão geral utilizando a metodologia Prometheus, vista anteriormente na seção 3.3.1, para ilustrar a interação entre os agentes bem como suas ações e crenças.

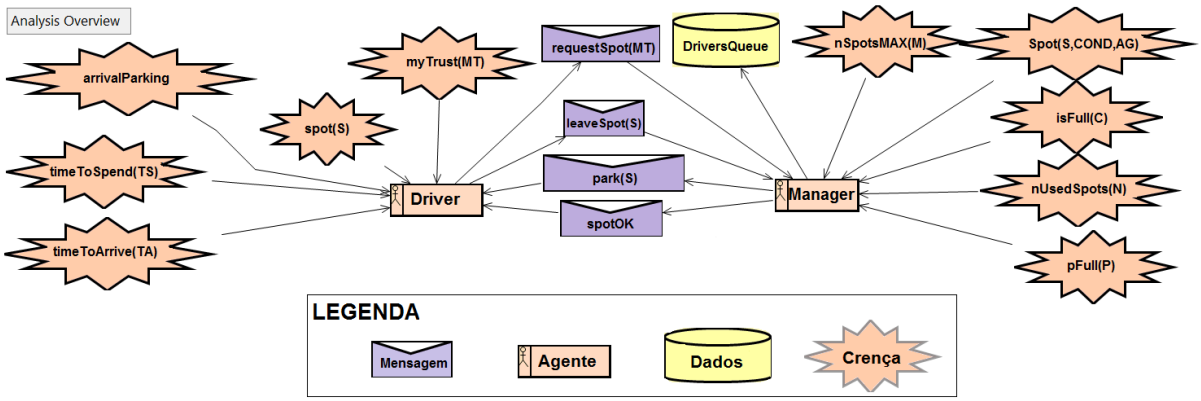

Figura 12 – Diagrama de visão geral - SMA

Fonte: Autoria Própria

5.2.1 Implementação dos Agentes em Jason

O Sistema multiagente desenvolvido possui dois tipos principais de agentes: *driver* e *manager*. A linguagem Jason utilizada pelo JaCaMo baseia-se no modelo BDI (*Belief, Desire and Intention*), sendo assim respectivamente no contexto do estacionamento: o que o agente sabe e crê do estacionamento, o que o agente deseja atingir e finalmente como atingirá esses objetivos.

[1] Repositório GitHub: github.com/MAPS-UTFPR/MAPS

Agente *Driver*

A figura 13 mostra um diagrama através da metodologia Prometheus. O diagrama mostra o funcionamento do agente *driver*, crenças, desejos (objetivos) e intenções (planos).

O código 5.1 demonstra as crenças e os objetivos iniciais do agente *driver*. Neste exemplo, o *driver* é nomeado *"m1"*, possuindo as seguintes crenças e objetivo:

- Tempo aproximado de chegada no estacionamento: 2 minutos;
- Tempo aproximado de estadia no estacionamento: 10 minutos;
- Valor de confiança: 300;
- Objetivo de chegar no estacionamento;

```
1 myTrust(300).
  timeToArrive(120).
3 timeToSpend(6000).
  !arriveParking.
```

Código 6 – Crenças e objetivos iniciais - Agente *Driver*

Além das crenças e objetivos, o agente *driver* possui planos (intenções). A estrutura do plano é composta por três elementos: evento gatilho, contexto e corpo de ações (Ver seção 4.3.3). O Agente *Driver* possui quatro planos (Ver figura 13), sendo eles:

1. ***arriveParking*:** Plano responsável pela chegada de um agente *driver* no estacionamento. Ao chegar no estacionamento o agente possui um novo objetivo: requisitar uma vaga. Ao fim do plano, o agente possui o objetivo, o qual é o evento gatilho para o próximo plano: requisitar vaga (*requestSpot*). Na linha 2 do código 7 é utilizada para fazer o agente "aguardar" até chegar ao estacionamento.

```
1 +!arriveParking : timeToArrive(TA) <-
2                   .wait(math.random(TA));
                    !requestSpot;
4                   +arrivalParking.
```

Código 7 – Plano de chegada estacionamento - Agente *Driver*

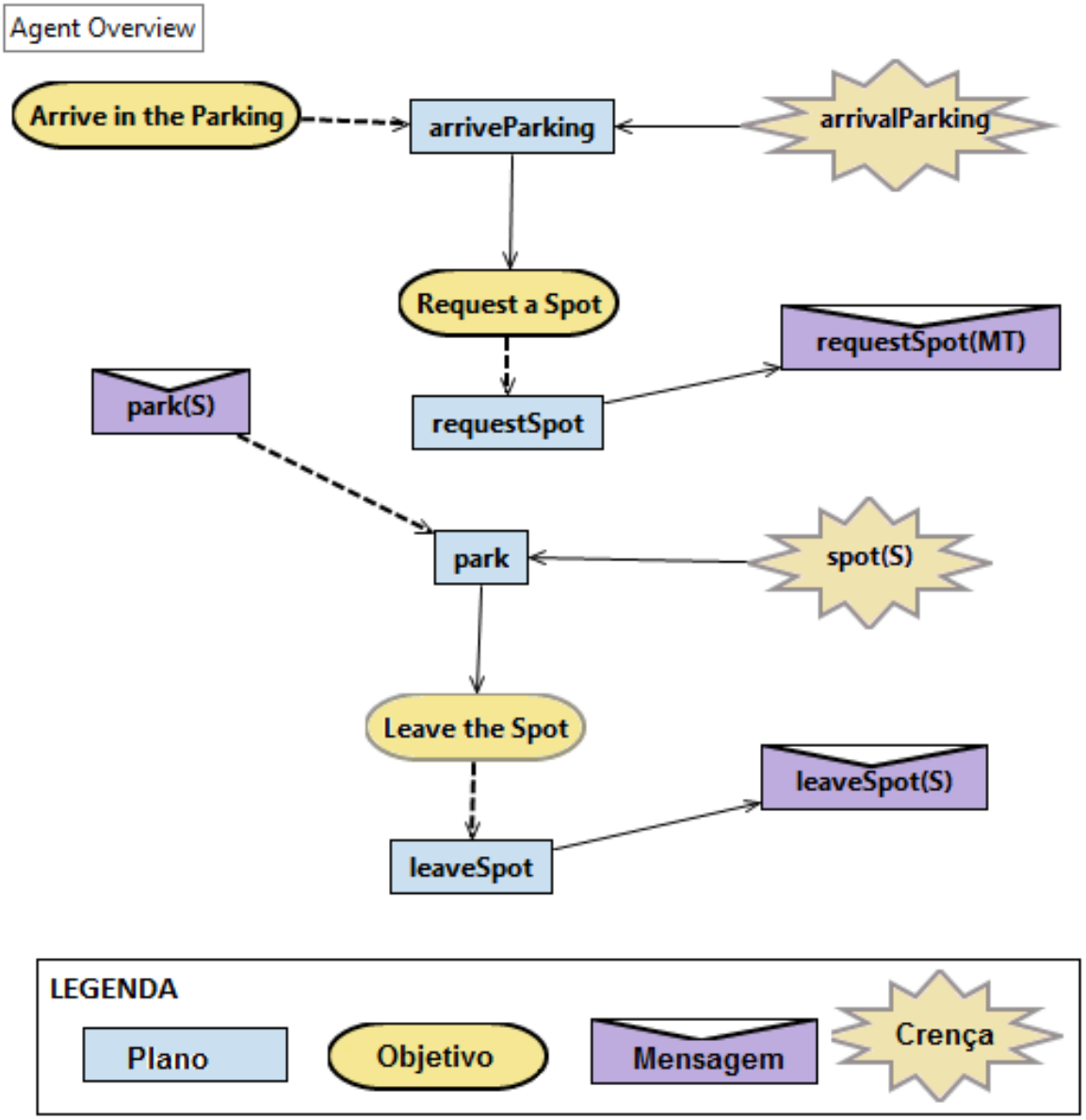

Figura 13 – Visão Geral - Agente *Driver*

Fonte: Autoria Própria

2. ***requestSpot*:** Plano responsável após a chegada do agente *driver* no estacionamento pela requisição de vaga para o agente *manager*. Na linha 4 do código 8 é utiliza uma ação interna da linguagem Jason (*send*) a qual é responsável pelo envio de mensagens entre os agentes. No atual plano é enviado ao agente *manager* o valor de confiança para uma futura utilização. Na linha 2, é utilizado também uma ação interna do Jason (*print*), sendo essa a responsável por imprimir no console as mensagens emitidas pelos agentes.

```
1 +!requestSpot : myTrust(MT) <-
2         .print("Arrived in the parking!");
              .send(manager, achieve, requestSpot(MT)).
```

Código 8 – Plano de requisição de vaga - Agente *Driver*

3. ***park*:** Ao agente requisitar vaga ao *manager*, ele deve aguardar pela resposta da vaga. Após o *manager* alocar a vaga, é enviado ao agente *driver* a informação da vaga (Ver figura 12) pelo *manager*. O plano *"park"* exibido no código 9 é o responsável pelo movimento do agente *driver* dentro do estacionamento e pelo estacionamento do veículo na vaga recebida. Na linha 3 é adicionado a base de crenças do agente *driver* a vaga em questão. Na linha 4 é utilizada uma ação interna (*wait*) a qual fica como responsável pela "estadia"do agente no estacionamento. Na linha 5 é adicionado um desejo ao agente: sair do estacionamento (*leaveSpot*).

```
1 +!park(S)[source(AGENT)] : spotOk & arrivalParking &
    timeToSpend(TS)
        <- .print("Parking at the spot: ",S);
3               +spot(S);
                .wait(TS);
5               !leaveSpot.
```

Código 9 – Plano para estacionar - Agente *Driver*

4. ***leaveSpot*:** Após o agente permanecer no estacionamento pelo tempo desejado (*timeToSpend(TS)*), o *driver* realiza a saída do estacionamento. Para isso, deve comunicar ao agente *manager* através da ação interna *send* qual vaga está sendo liberada para que seja novamente ocupada por um outro *driver*. Na linha 4 do código 10 a crença relacionada a vaga é removida da base de crenças do agente.

```
1 +!leaveSpot : spot(S) <-
        .print("Leaving the parking...");
3       .send(manager, achieve, leaveSpot(S));
        -spot(S).
```

Código 10 – Plano para sair do estacionamento- Agente *Driver*

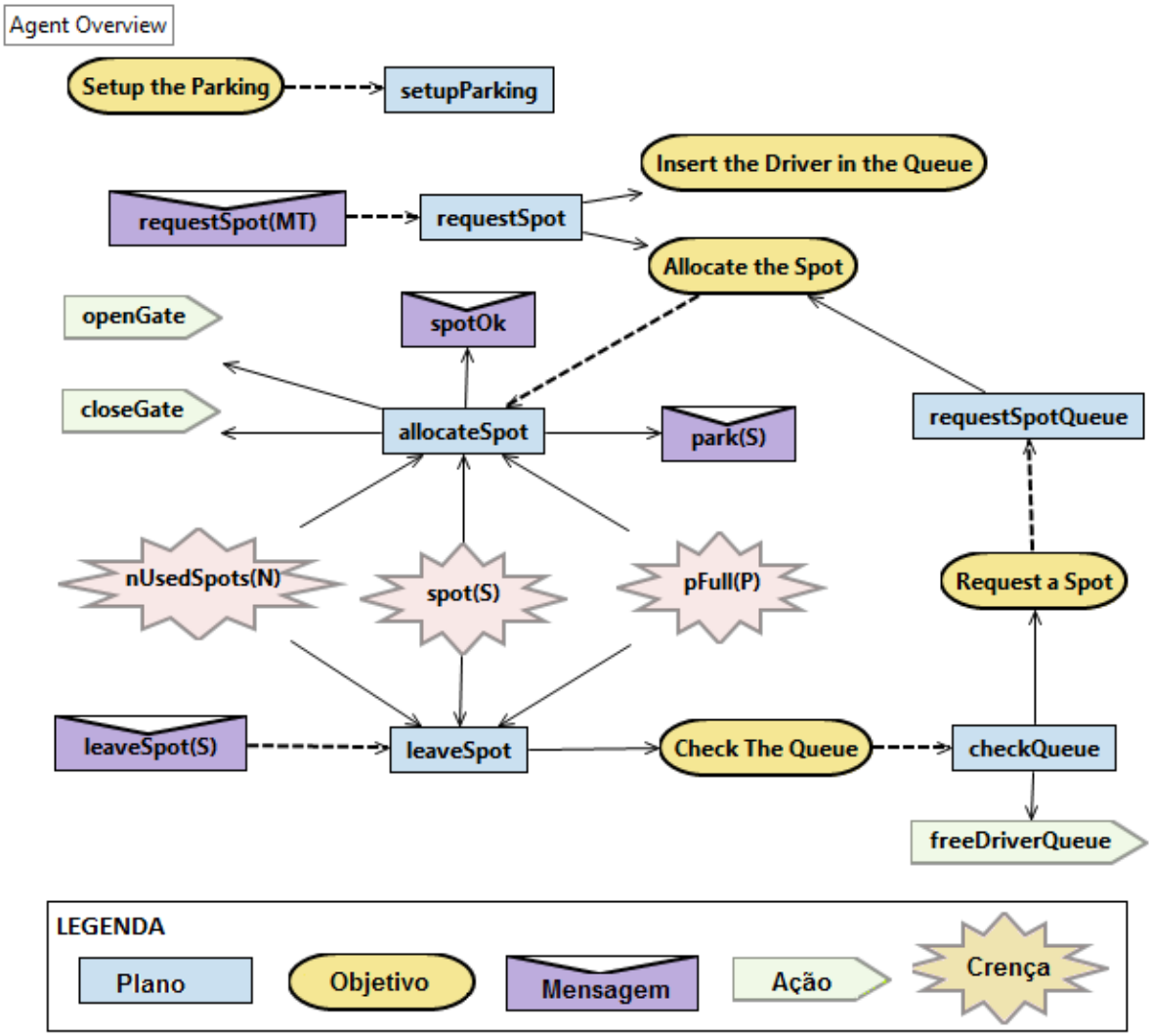

Figura 14 – Visão Geral - Agente *Manager*

Fonte: Autoria Própria

Agente *Manager*

Assim como no agente *driver*, a figura 14 ilustra os planos que o agente *manager* possui e utiliza, bem como os objetivos e crenças presentes nos planos.

A seguir é apresentado as crenças e objetivos iniciais do agente *manager* e atribuídos valores simbólicos a título de explicação.

- Número de vagas utilizadas: 0 vagas;
- Número de vagas máximas que o estacionamento comporta: 4 vagas;
- Se o estacionamento está cheio ou não;
- Percentual de lotação;

- Vagas: A crença do agente para o vaga é baseada em tupla, sendo composta por três valores: ID da vaga, estado da vaga e nome do agente que a ocupa;

    1. ID Vaga: Valor inteiro correspondente a um identificador único da vaga;
    2. Estado da vaga: Valor numérico (0 ou 1): 0 - Vaga livre / 1 - Vaga Ocupada;
    3. Nome agente: Valor String que armazena o nome do agente que ocupa a vaga em questão. Caso a vaga esteja vazia é atribuído o valor *"EMPTY"* a este campo.

```
1 nSpotsMAX(4).
2 nUsedSpots(0).
  isFull(false).
4 pFull(0).
  spot(0,0, "EMPTY").
6 spot(1,0, "EMPTY").
  spot(2,0, "EMPTY").
8 spot(3,0, "EMPTY").
  !setupParking.
```

Código 11 – Crenças e objetivos iniciais - Agente *Manager*

No código 5.6 é apresentado as crenças e objetivos iniciais do agente *manager*. A partir da linha 1 até a 8 é apresentado as crenças, e na linha 9 o objetivo inicial: *setupEstacionamento*. Além dos objetivos e crenças, o agente *manager* implementa os planos responsáveis pela alocação de vagas e gerenciamento do estacionamento, sendo eles:

1. ***setupParking*:** Plano responsável de inicializar o estacionamento e gerar os artefatos do Cartago. Nas linhas 2 e 3 do código 12 são utilizados os comandos de integração do Jason com Cartago. Essa integração foi apresentada e explicada previamente na seção 4.5.

```
1 +!setupParking <-
      makeArtifact("a_Gate","maS3.Gate", ["Starting"],ArtId);
3     focus(ArtId);
      .print("Parking␣has␣been␣opened!");
5     makeArtifact("a_Control","maS3.Control",["20"] ArtId2);
      focus(ArtId2).
```

Código 12 – Plano de setup estacionamento - Agente *Manager*

2. ***requestSpot*:** Ao agente *driver* enviar uma mensagem requisitando a vaga, o plano de requisição de vaga é ativado. O agente *manager* é notificado e é adicionado um novo objetivo a ser alcançado: alocar a vaga (*allocate the spot*).

```
1 +!requestSpot(TRUST)[source(AG)] <-
2         .term2string(AG,AGENT);
          .print("Agent:",AGENT,"has␣requested␣a␣spot!-Trust:
             ",TRUST,")");
4         !allocateSpot(AGENT,TRUST).
```

Código 13 – Plano de requisição de vaga - Agente *Manager*

3. ***requestSpotQueue*:** Ao agente *driver* deixar o estacionamento, ele deve avisar ao agente *manager* que está liberando a vaga. Ao *manager* receber tal aviso é realizado uma busca na fila para encontrar o *driver* com as melhores condições (valor de confiança ou tempo de espera) de utilizar a vaga. Ao executar o plano o agente *manager* possui um novo objetivo: alocar a vaga para o *driver* da fila. O plano *requestSpotQueue* exibido no código 14 é responsável pela alocação da vaga provinda de um *driver* da fila.

```
1 +!requestSpotQueue(AGENT,TRUST) <-
2         .print("Agent:␣",AGENT,"␣has␣requested␣a␣spot!");
          !allocateSpot(AGENT,TRUST).
```

Código 14 – Plano de requisição de vagas de motoristas na fila - Agente *Manager*

4. ***allocateSpot*:** Estacionamento com vagas.

```
1 +!allocateSpot(AGENT,TRUST) : nUsedSpots(N) & nMAXSpots(M)
     & isFull(COND) & pFull(P) & COND = false <-

3         +~find;
          for(spot(S,C,A)){
5                 if( Z = 0 & ~find & (COND = false)){
                        -spot(S,C,A); +spot(S,1,AGENT);
7                       .print("Spot␣(",S,")␣has␣allocated␣
                           for␣the␣agent:␣",AGENT);
                        openGate;
9                       .send(AGENT,tell,spotOk); .send(
                           AGENT,achieve,park(S));
                        closeGate;
11                      -nUsedSpots(N); +nUsedSpots(N+1);
                        if((N+1) = MAX){
13                       -isFull(COND); +isFull(true);
                         .print("Parking␣FULL!");
15                      };
```

```
                                -pFull(P);
17                              +pFull(((N+1) * 100) / MAX);
                                .print("Parking usage: ",((N+1) *
                                    100) / MAX,"%");
19                              -~find;
                        }
21              };
                +~find.
```

Código 15 – Plano de alocação de vagas - estacionamento não cheio - *Manager*

Plano responsável pela alocação de vagas para os *drivers*. A partir da linha 4 do código 15, é realizado um laço para a procura de uma vaga disponível para alocação a um agente *driver*. Nas linhas 5 e 6 a vaga caso encontrada, é alocada. Na linha 8 é utilizada uma ação provinda do artefato "*Gate*", a qual o agente *manager* invoca para abrir a cancela do estacionamento. Nas linhas 12 até 14 é verificado se o estacionamento está cheio. Caso esteja cheio o agente *manager* passa ter uma nova crença em sua base de crenças: *isFull(true)*. Esse plano é utilizado apenas quando o estacionamento possui vagas, caso contrário ele não é utilizado.

5. ***allocateSpot*:** Estacionamento cheio. Plano utilizado quando estacionamento está lotado. Ao agente *driver* requisitar uma vaga e o estacionamento estiver cheio, ele é inserido em uma fila de espera. Na linha 2 do código 16 é utilizado uma ação provinda pelo artefato *"Control"* a qual insere o agente *driver* na fila de espera.

```
1 +!allocateSpot(AGENT,TRRUST):isFull(COND) & COND = true <-
2                       insertDriverQueue(AGENT,TRUST).
```

Código 16 – Plano de alocação de vagas - estacionamento cheio - Agente *Manager*

6. ***leaveSpot*:** Ao agente *driver* avisar o *manager* que está liberando a vaga, o plano do código 17 é ativado. O agente *manager* remove a crença sobre a vaga ocupada (linha 8) e passa a ter a nova crença que a vaga está livre (linha 7). Na linha 10 o agente *manager* exclui do SMA o agente *driver*. Após isso o agente *manager* deverá verificar se há agentes esperando por uma vaga, assim é adicionado um novo objetivo ao *manager*: verificar a fila (linha 11) *(check the Queue)*.

```
1  +!leaveSpot(S)[source(AG)] : nUsedSpots(N) & nSpotsMAX(M) &
       isFull(COND) & pFull(P) <-
2          .term2string(AG,AGENT);
           .print(AGENT,"␣leaving␣the␣spot:␣",S);
4          -nUsedSpots(N); +nUsedSpots(N-1);
           -isFull(COND);
6          +isFull(false);
           +spot(S,0,"EMPTY");
8          -spot(S,1,AGENT);

10         .kill_agent(AGENT);
           !checkQueue.
```

Código 17 – Plano de liberação de vagas - Agente *Manager*

7. ***checkQueue*:** Plano responsável por verificar se há *drivers* na fila esperando por uma vaga. Caso sim, é verificado qual *driver* irá a receber. Na linha 2 do código 18 é verificado através do artefato "Control" se há alguém na fila. Caso sim, é realizada uma requisição de vaga para o *driver* selecionado (linha 5).

```
1  +!checkQueue : nUsedSpots(N) & isFull(COND)<-
           isAnyone(C);
3          if(C = true){
                   freeDriver(AG,BG);
5                  !requestSpotQueue(AG,BG);
           }else{
7                  .print("Nobody␣at␣queue");
           }.
```

Código 18 – Plano de verificação de motoristas na fila - Agente *Manager*

5.2.1.1 Interação entre os agentes

Durante a execução do SMA é realizado troca de mensagens entre o agente *Manager* e o agente *Driver*, sendo estas mensagens destinadas a requisição de vagas e avisos.

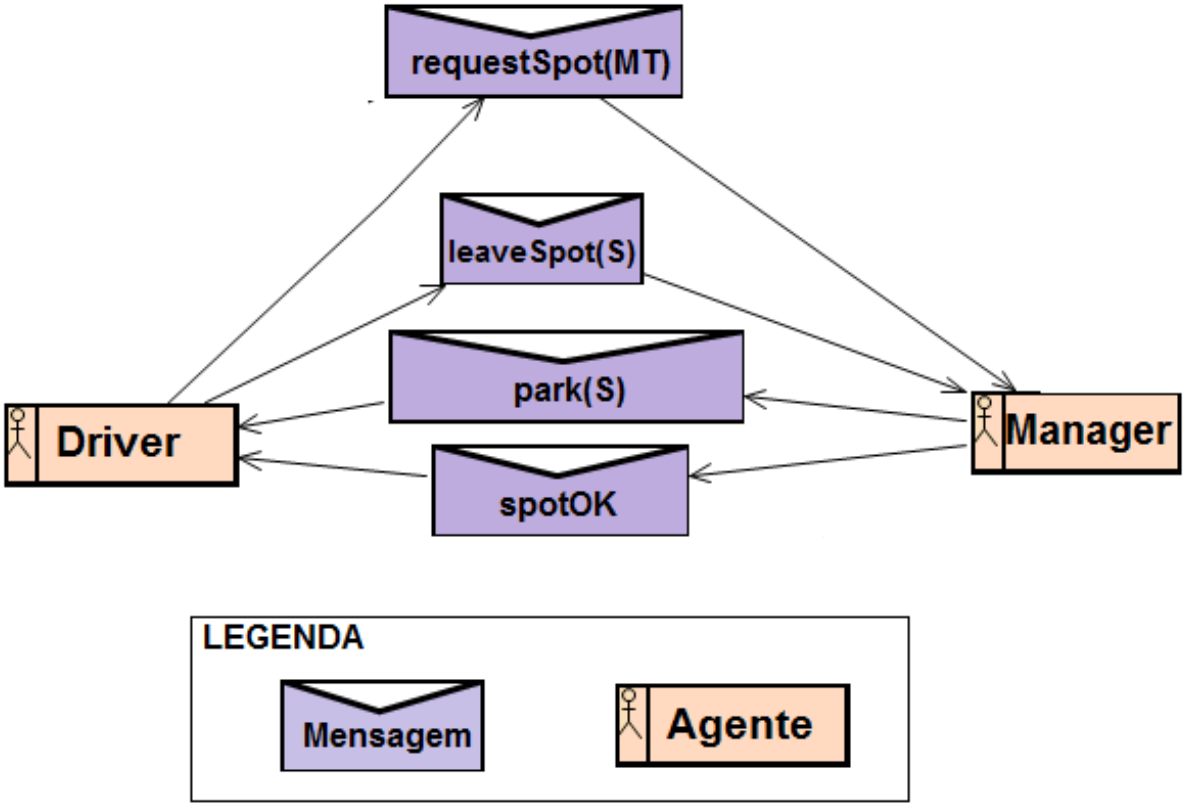

Figura 15 – Troca de mensagens entre os agentes

Fonte: Autoria Própria

- ***requestSpot(MT):*** Mensagem enviada ao agente *Manager* informando-o sobre uma requisição de vaga. O valor de parâmetro (MT) representa a crença *myTrust* utilizada pelo agente *Driver*;
- ***spotOk:*** Ao agente *Manager* receber a requisição de vaga e alocar a vaga ao *Driver*, é enviado uma mensagem a este *Driver* com a liberação da vaga;
- ***park(S):*** Após a mensagem *spotOk* ser enviada, é enviado ao *Driver* a qual vaga ele deve estacionar. O parâmetro (S) possui o identificador da vaga.
- ***leaveSpot(S):*** Ao agente Driver sair do estacionamento, é notificado o agente Manager qual vaga está sendo liberada através do parâmetro (S).

5.2.2 Implementação dos Artefatos em Cartago

Os artefatos em Cartago são responsáveis por prover ações aos agentes que estão no seu ambiente. No contexto do estacionamento, o ambiente em si é o próprio estacionamento e os agentes *drivers* e o agente *manager*. Foram implementados dois artefatos, sendo eles:

1. **Artefato *Control*:** Responsável pelo gerenciamento dos *drivers* na fila, sendo inserindo-os ou selecionando o *driver* com as melhores condições de receber uma vaga;

2. **Artefato *Gate*:** Situando na entrada do estacionamento, responsável pela abertura e fechamento da cancela.

Os artefatos são implementados utilizando a linguagem Java, proporcionando assim ao SMA uma flexibilidade e robustez providas por essa linguagem.

Artefato *Control*

Artefato utilizado pelo agente *manager*. A figura 16 ilustra o diagrama de classe do artefato e sua interação com classe "*Driver*", sendo essa classe a responsável pela representação dos agentes *drivers* na fila de espera.

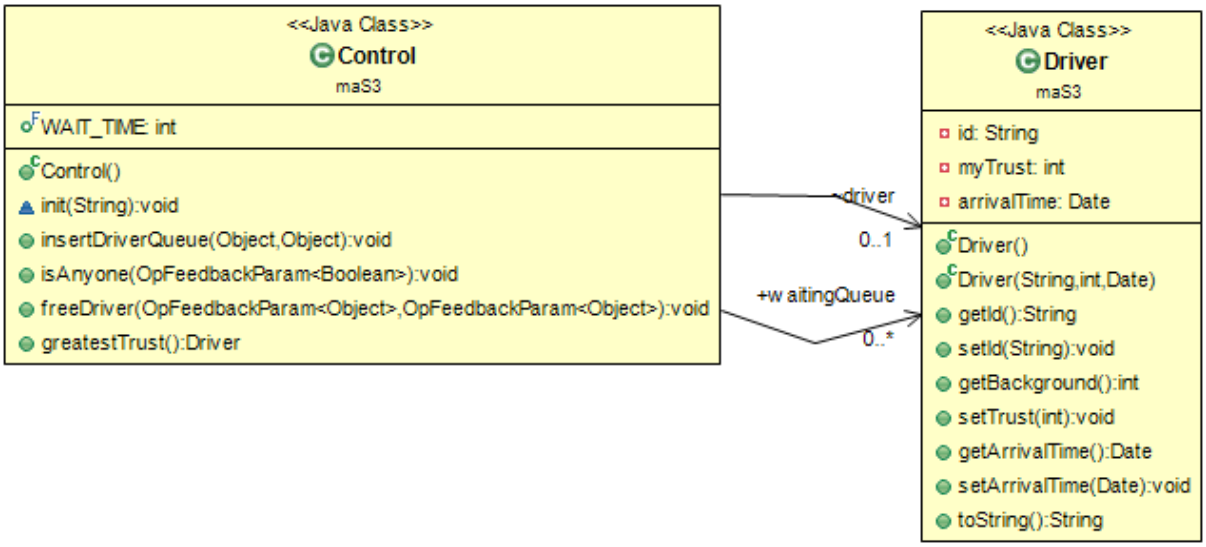

Figura 16 – Diagrama de Classe - Artefato *Control*

Fonte: Autoria Própria

1. **Fila de motoristas:** Para o armazenamento dos *drivers* na fila, é utilizado uma lista encadeada provida pela classe *LinkedList* utilizando o estereótipo da classe "*Driver*"(Ver figura 16). Na linha 5 do código 19 é determinado uma constante com o valor máximo teórico que um agente *driver* poderá esperar na fila para receber uma vaga.

```
1 public class Control extends Artifact {
2
        public LinkedList <Driver> waitingQueue;
4       Driver driver;
        public final int WAIT_TIME = 60000;
6
        void init(String msg){
8               waitingQueue = new LinkedList<Driver>();
        }
```

Código 19 – Fila de motoristas - Artefato *Control*

2. **Inserir *driver* na fila:** Ao agente requisitar uma vaga e o estacionamento estiver lotado, ele é inserido na fila de espera. O código 20 ilustra como o artefato implementa a função de inserção do agente *driver* na fila.

```
1 @OPERATION
        public void insertDriverQueue(Object idDriver,
           Object tDriver){
3
        driver = new Driver(idDriver.toString(), Integer.
           parseInt(tDriver.toString()), new Date());
5       waitingQueue.add(driver);
        }
```

Código 20 – Inserir motorista na fila - Artefato *Control*

Na linha 1 é utilizado a anotação *"@OPERATION"*, a qual é utilizada para demarcar quais métodos estarão disponíveis como ações para os agentes no Jason. Na linha 4 é instanciado um objeto da classe "*Driver*"e inserido na fila. No método construtor da classe "*Driver*"é requerido três parâmetros, sendo eles: *idDriver*, valor de confiança (*trust*) e hora de chegada no estacionamento. O valor da hora de chegada será utilizado quando o *driver* está na fila de espera aguardando uma vaga, sendo dado preferência aos *drivers* que estão com tempo de espera maior ou superior a 60 segundos.

3. **Selecionar agente *driver* para receber vaga:** Ao liberar uma vaga, o agente *manager* verifica se há agentes *drivers* aguardando vaga. Caso sim, é selecionado um agente para ocupar a vaga. A seleção busca o agente *driver* com a maior valor de confiança (*trust*) na

fila. Porém, caso haja algum agente *driver* com o tempo de espera na fila superior que 60 segundos, esse agente será o selecionado.

```
1  @OPERATION
2    public void isAnyone(OpFeedbackParam<Boolean> cond){
                cond.set(waitingQueue.isEmpty());
4        }
   @OPERATION
6    public void freeDriver(OpFeedbackParam<Object> idDriver,
       OpFeedbackParam<Object> tDriver){

8               Driver d = greatestTrust();
                idDriver.set(d.getId());
10              tDriver.set(d.getTrust());
                waitingQueue.remove(d);
12       }

14       public Driver greatestTrust(){

16              Driver d1 = new Driver();
                for(Driver d : waitingQueue){
18                      if(new Date().getTime() - m.
                           getWaitTime().getTime() >
                           WAIT_TIME)
                                return d;
20              }

22              for(Driver m : waitingQueue){
                        if(m1.getTrust() < m.getTrust())
24                              d1 = d;
                }
26              return d1;
         }
```

Código 21 – Selecionar *driver* da fila - Artefato *Control*

Na código 21 é ilustrado três operações responsáveis pela busca de *drivers* na fila. A primeira *isAnyOne* busca verificar se há alguém na fila de espera. A segunda, "*freeDriver* trabalha em conjunto com a terceira *"greatestTrust"*. A partir da linha 17 é buscado na lista de espera se há algum agente *driver* com tempo de espera maior que 60 segundos.

Caso contrário é realizado a busca novamente para eleger o agente *driver* com o maior valor de confiança, e finalmente este agente é o selecionado para a vaga.

Artefato Gate

Artefato utilizado pelo agente *manager* utilizado na entrada e sáida dos agentes *drivers* do estacionamento. A figura 17 ilustra o diagrama de classe do artefato. O artefato *Gate* implementa as seguintes ações: *openGate* e *closeGate*.

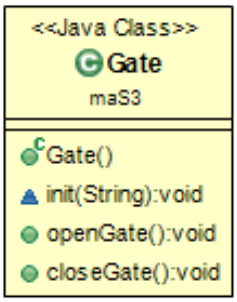

Figura 17 – Diagrama de Classe - Artefato Gate

Fonte: Autoria Própria

A seguir no código 5.17 é apresentado a implementação do artefato.

```
1  public class Gate extends Artifact {
2
3          void init(String msg) {
4                  System.out.println("System message: " + msg );
5          }
6          @OPERATION
7          public void openGate(){
8                  System.out.println("Opening gate!");
9          }
10         @OPERATION
11         public void closeGate(){
12                 System.out.println("Closing gate!");
13         }
14 }
```

Código 22 – Implementação artefato *Gate*

6 RESULTADOS

Neste capítulo são demonstrados diferentes cenários de aplicação do SMA desenvolvido. Especificamente, onze cenários com configurações e características particulares foram estabelecidos.

6.1 CONFIGURAÇÃO DOS EXPERIMENTOS

A configuração dos onze cenários utilizados é apresentada na tabela 3.

Tabela 3 – Configuração dos cenários

Cenário	Número de Drivers	Número de vagas
0	2	1
1	50	1
2	50	2
3	50	10
4	50	25
5	25	1
6	25	2
7	25	10
8	10	1
9	10	3
10	10	5

Fonte: Autoria Própria

Tabela 4 – Configuração dos agentes

Agente	*timeToArrive(TA)*	*timeToSpend(TS)*	*myTrust(MT)*
m1	4s	10s	100
m2	2s	3s	450
m3	7s	5s	999
m4	3,5s	7,5s	4
m5	4,5s	10s	120
m6	6,7s	9s	770
m7	9s	12s	180
m8	10,29s	8,5s	10
m9	9,9s	6,6s	803
m10	4,6s	6,6s	5

Fonte: Autoria Própria

O número de agentes utilizados nos cenários variam de 2 até 50 agentes (m1,m2,m3...m50), vide tabela 3. No cenário 1 até o cenário 10 tiveram ao mínimo 10 agentes em sua execução. A

seguir será detalhado cada cenário e como o SMA desenvolvido comportou-se durante a execução. O cenário 0 foi designado a detalhar o funcionamento do SMA e todas as suas interações, devido ao fato de possui uma quantidade pequena de agentes. Os cenários de 1 a 10 foram designados a testar o SMA em diferentes configurações, tendo em vista ao mínimo dez agentes nesses cenários. Durante a descrição dos cenários de 1 a 10 [1] foram selecionados dez agentes, devido ao fato desses agentes estarem em todos esses cenários, sendo eles apresentados na tabela 4.

Com o objetivo de demonstrar a eficácia do SMA em diferentes cenários e o como o valor de *trust* dos agentes *Drivers* impacta no processo de alocação de vagas será demonstrado para alguns cenários o tempo de alocação dos motoristas, comparando com seus valores de *trust* e também como esse valor impacta de modo geral durante a execução do SMA nos cenários.

6.2 CENÁRIO 0

Nesse cenário será utilizado apenas 2 agentes e 1 vaga no estacionamento, com o intuito de apresentar o funcionamento do SMA. A fim de clarificar o entendimento e comunicação entre os agentes, a figura 19 ilustra um diagrama de sequência com as interações e utilização do SMA no Cenário 0.

Com a finalidade de tornar mais claro as mensagens exibidas no *console* do SMA, a seguir é apresentado na figura 18 o SMA em execução, porém com a adição de algumas letras para melhor explicar cada funcionalidade.

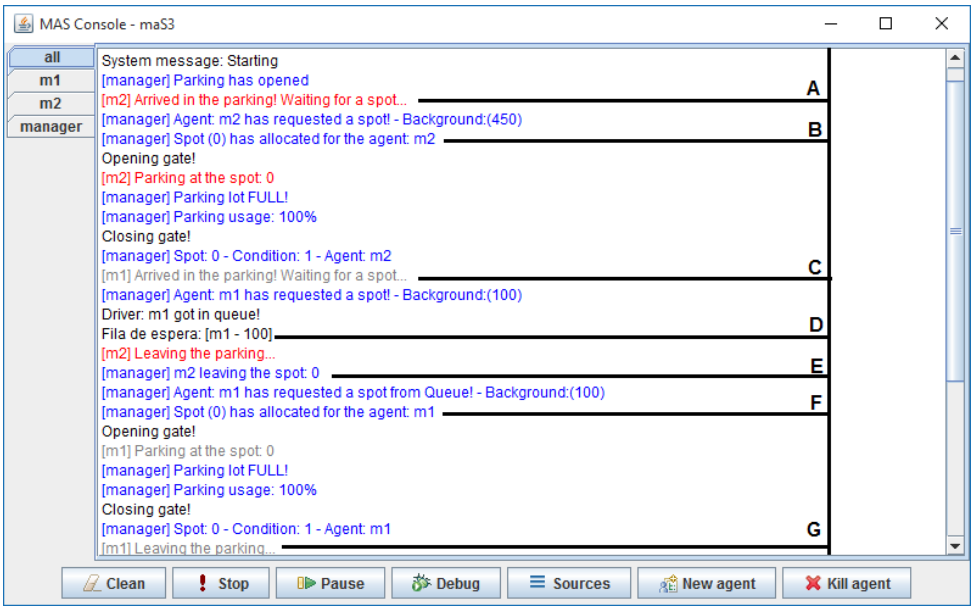

Figura 18 – Execução SMA - Cenário 1 - Início

Fonte: Autoria Própria

[1] Os cenários 4,7,9 e 10 não serão descritos devido ao fato de que por existir uma quantidade significativa de vagas, os agentes *Drivers* não ficaram aguardando na fila por uma vaga

A tabela 5 apresenta a funcionalidade cada letra exibida na figura 18.

Tabela 5 – Descrição das funcionalidades das letras do SMA

Letra	Funcionalidade
A	Driver m2 chegou no estacionamento e aguarda por uma vaga
B	Vaga 0 alocada ao driver m2
C	Driver m1 chegou no estacionamento e aguarda por uma vaga
D	Estacionamento cheio. Driver m1 entra na fila para aguardar uma vaga
E	Driver m2 deixa o estacionamento e libera a vaga
F	Vaga 0 alocada ao driver m1
G	Driver m1 deixa o estacionamento e libera a vaga

Fonte: Autoria Própria

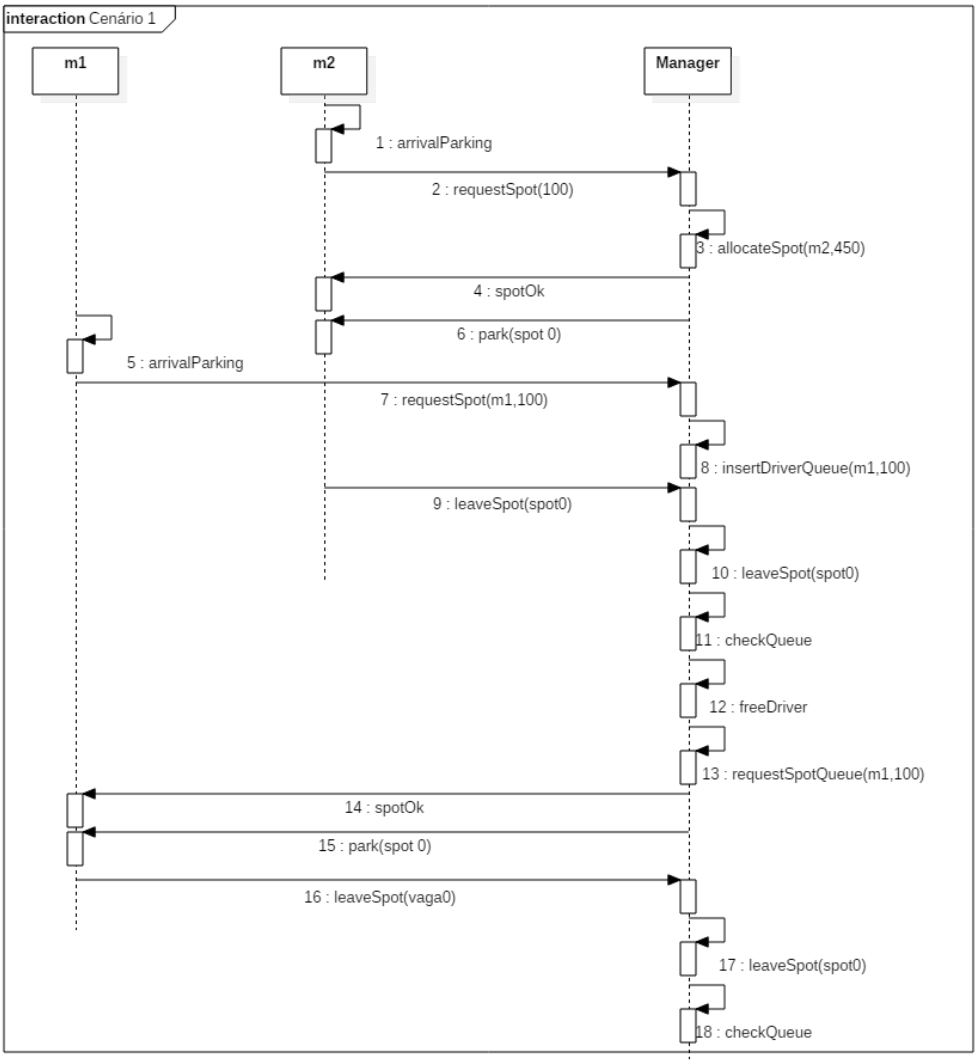

Figura 19 – Diagrama de Sequência - Cenário 0

Fonte: Autoria Própria

6.3 CENÁRIO 1

Nesse cenário será utilizado 50 agentes *Driver* e uma vaga de estacionamento. A figura 20 apresenta um gráfico de como os *Drivers* apresentados na tabela 4 são alocados.

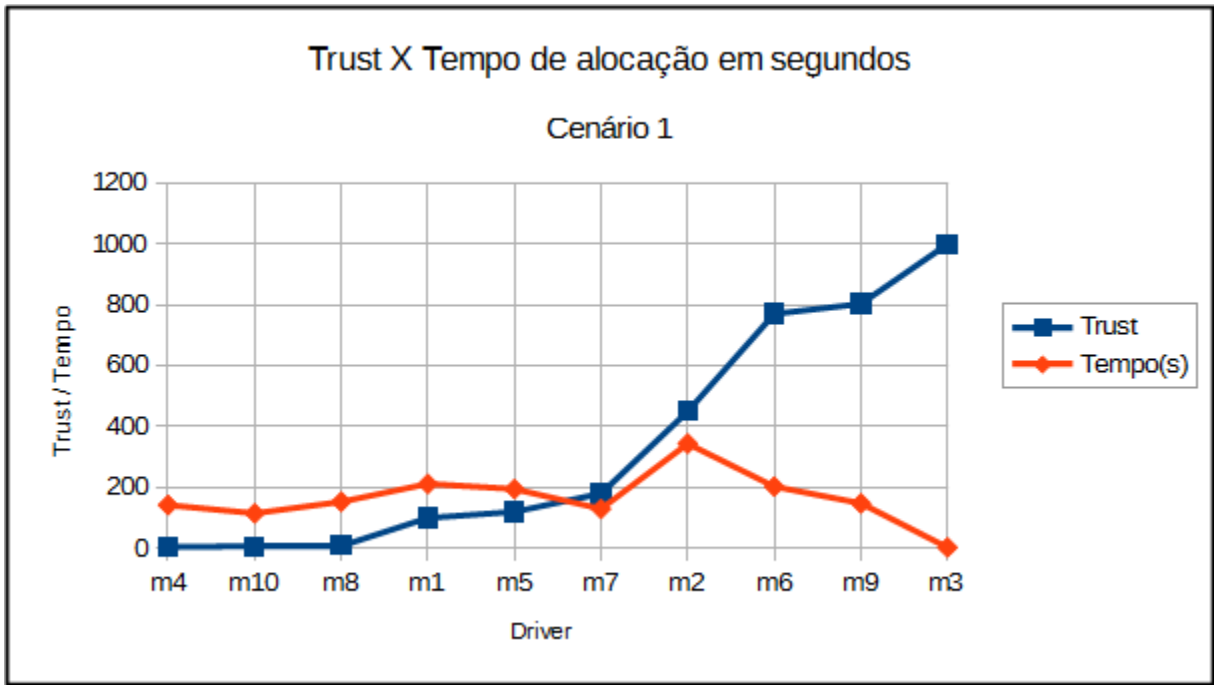

Figura 20 – Alocação *Drivers* - Cenário 1

Fonte: Autoria Própria

O gráfico ilustrado pela figura 20 apresenta duas curvas, sendo a curva com triângulos apresenta o tempo em segundos de alocação para uma vaga. Já a curva com quadrados representa o valor de *trust* para cada *Driver*.

Nota-se no gráfico de que a medida que o valor de *trust* é maior, o tempo de alocação é menor, pois é dado preferência aos Drivers com maior valor de *trust* na alocação de vagas, vide Código 5.6. Ocorre que em alguns momentos mesmo um Driver com um valor de *trust* alto pode levar um tempo alto para ser alocado. Isso ocorre devido ao fato de que no momento que esse *Driver* chegou no estacionamento, todas as vagas estavam ocupadas e os *Drivers* que possuíam estas vagas levaram um tempo maior para deixar o estacionamento de acordo com seu valor de `timeToSpend(TS)`.

Vale destacar o fato do *Driver* `m3` possuir um valor muito alto de *trust*, sendo assim seu tempo de alocação foi extremamente baixo. A figura 21 apresenta um gráfico do histórico do tempo de alocação em segundos baseado no valor de *Trust* dos *Drivers* neste cenário.

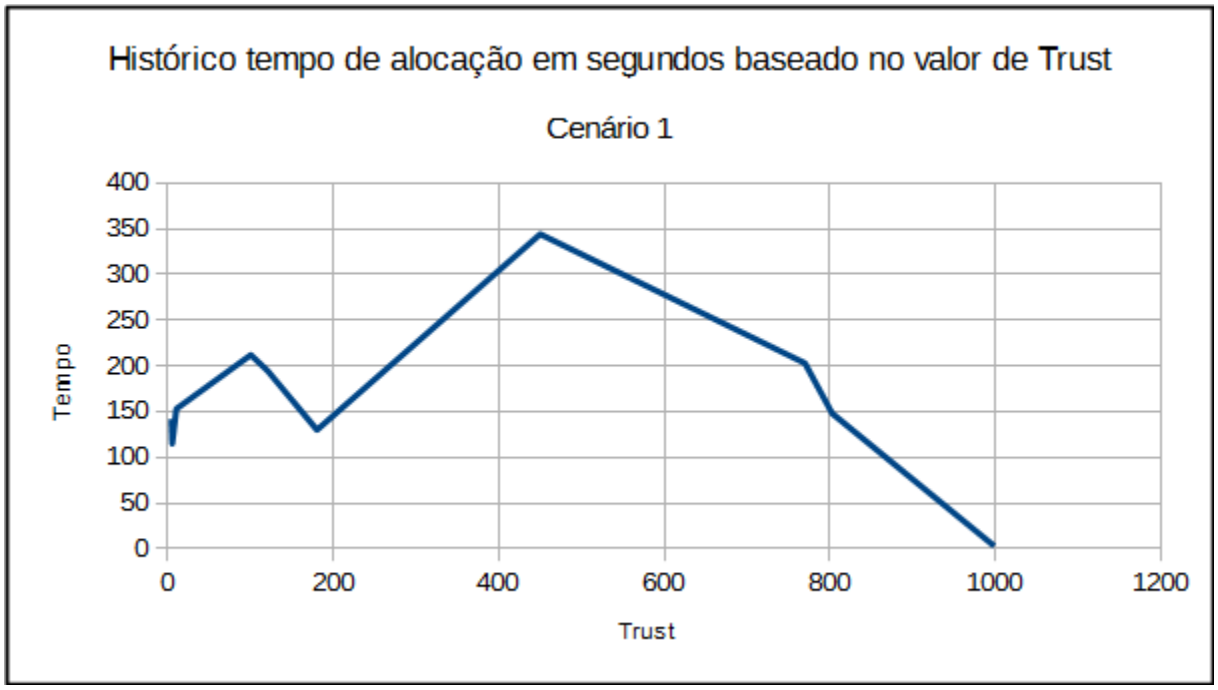

Figura 21 – Histórico tempo de alocação de acordo com valor de *trust* - Cenário 1

Fonte: Autoria Própria

6.4 CENÁRIO 2

Nesse cenário será utilizado 50 agentes *Driver* e duas vagas de estacionamento. A figura 22 apresenta um gráfico de como os *Drivers* apresentados na tabela 4 são alocados.

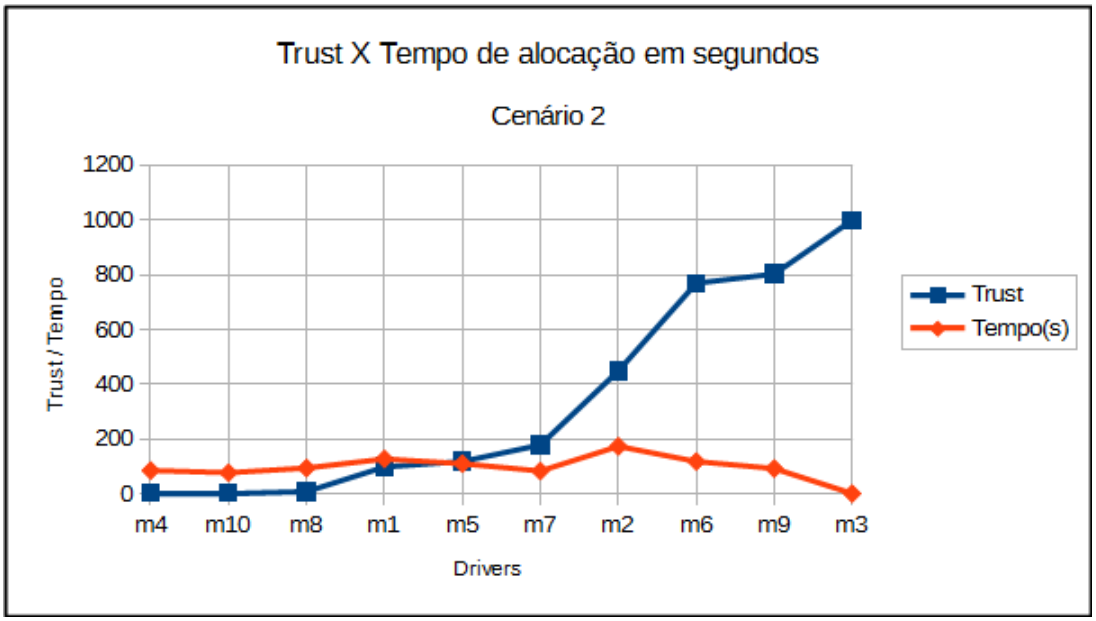

Figura 22 – Alocação *Drivers* - Cenário 2

Fonte: Autoria Própria

Neste cenário em contraste com cenário 1 possui uma vaga de estacionamento a mais disponível, diminuindo assim a média geral de tempo de alocação em segundos. Nota-se também o fato do valor de *trust* possuir um grande impacto no processo de alocação das vagas. O gráfico exibido na figura 23 reforça esta afirmação, porém, em específico nesse cenário os *Drivers* que tiveram um valor médio de *trust* foram os que mais demoraram para ser alocados.

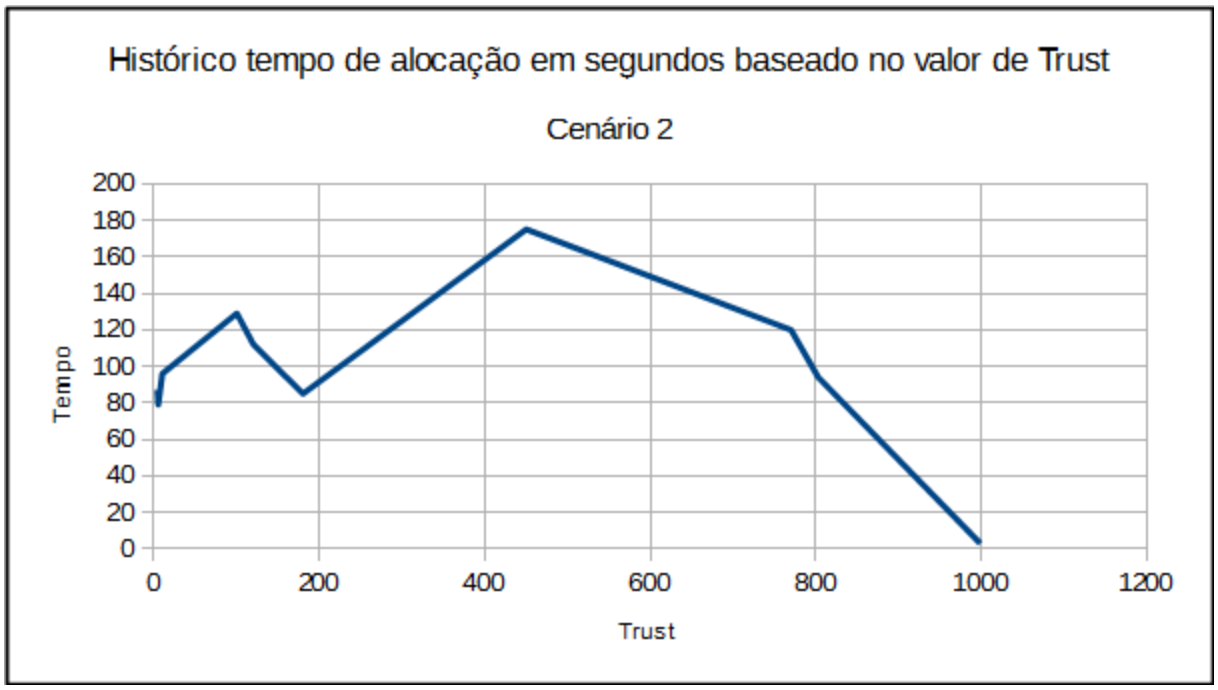

Figura 23 – Histórico tempo de alocação de acordo com valor de *trust* - Cenário 2

Fonte: Autoria Própria

6.5 CENÁRIO 3

Nesse cenário será utilizado 50 agentes *Driver* e dez vagas de estacionamento. A figura 24 apresenta um gráfico de como os *Drivers* apresentados na tabela 4 são alocados. Este cenário apresenta um maior contraste com os anteriores, devido ao fato de haver mais vagas disponíveis, sendo assim os *Drivers* com um valor de *trust* alto e baixo não tiveram diferenças grandes de tempo de alocação para uma vaga. O gráfico exibido na figura 25 demonstra que alguns *Drivers* com valor *trust* médio tiveram um tempo maior de alocação em comparação com os valores mais baixos. Porém, esse valor em relativamente baixo (14 segundos).

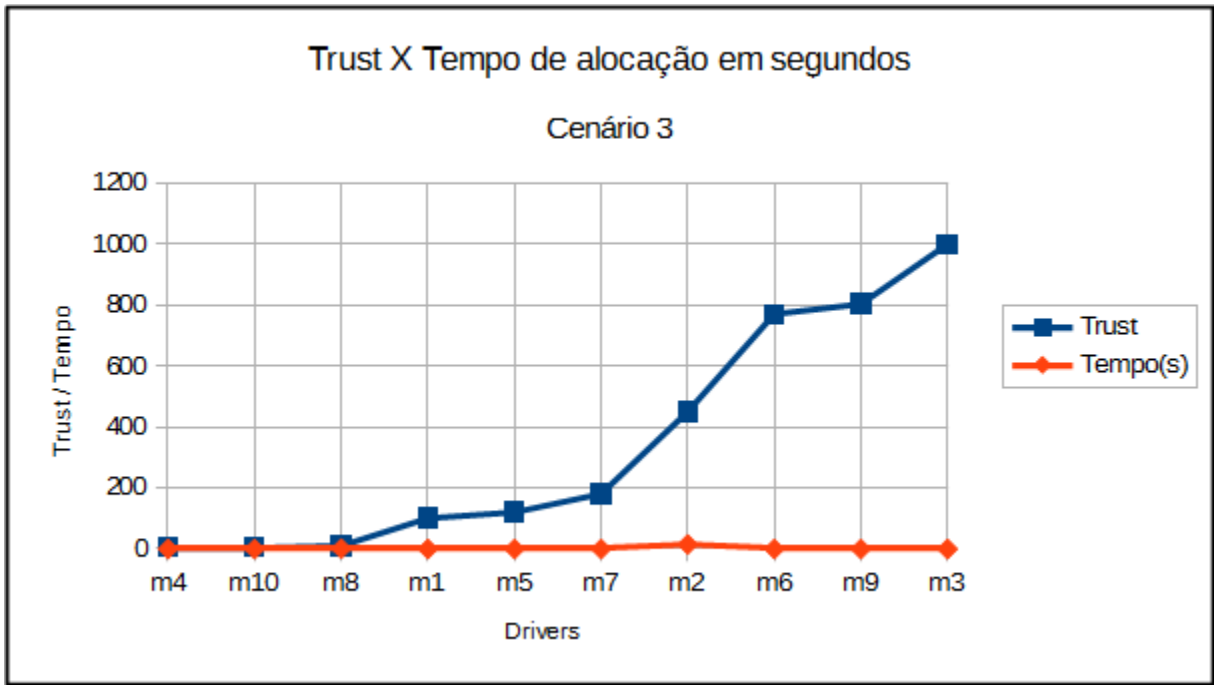

Figura 24 – Alocação *Drivers* - Cenário 3

Fonte: Autoria Própria

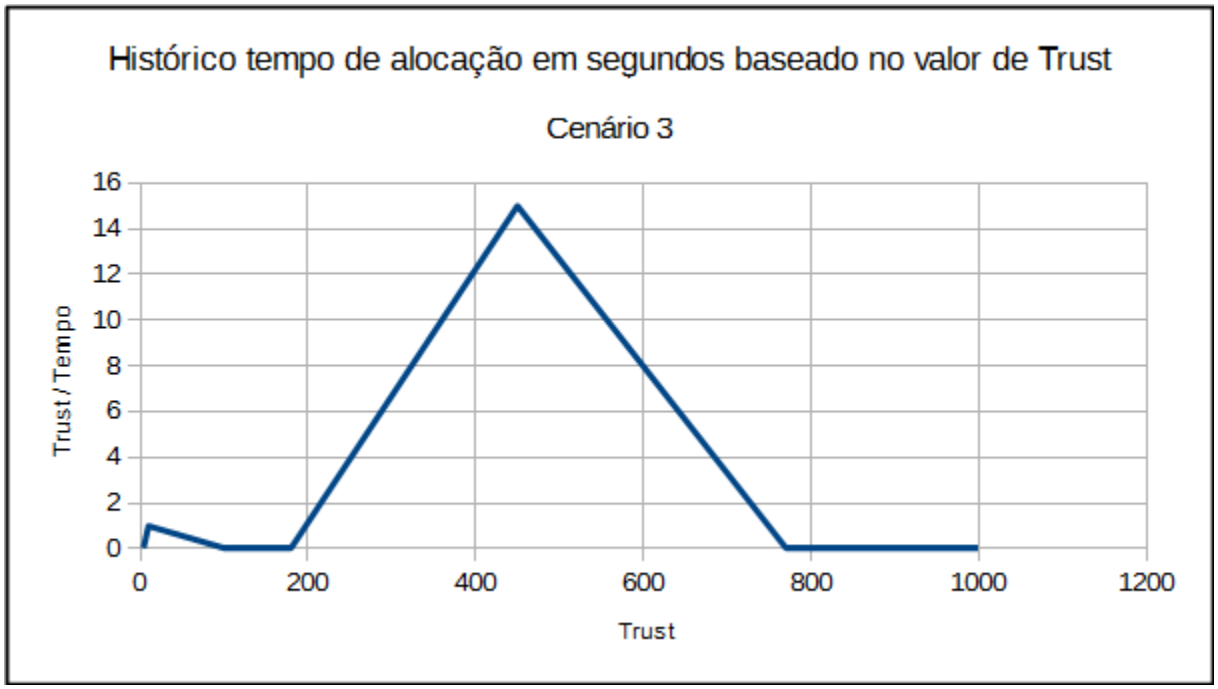

Figura 25 – Histórico tempo de alocação de acordo com valor de *trust* - Cenário 3

Fonte: Autoria Própria

6.6 CENÁRIO 5

Nesse cenário será utilizado 25 agentes *Driver* e 1 vaga de estacionamento. A figura 26 apresenta um gráfico de como os *Drivers* apresentados na tabela 4 são alocados. Neste cenário de maneira similar aos cenários anteriores, conforme o valor de *trust* dos agentes *Drivers* aumenta, o tempo de alocação diminui. No gráfico exibido na figura 27 mostra que alguns *drivers* de valor *trust* médio tiveram um tempo de alocação maior que os *drivers* de *trust* menor. Entretanto, os *drivers* com valor de *trust* alto, obtiveram um tempo de alocação menor que os demais.

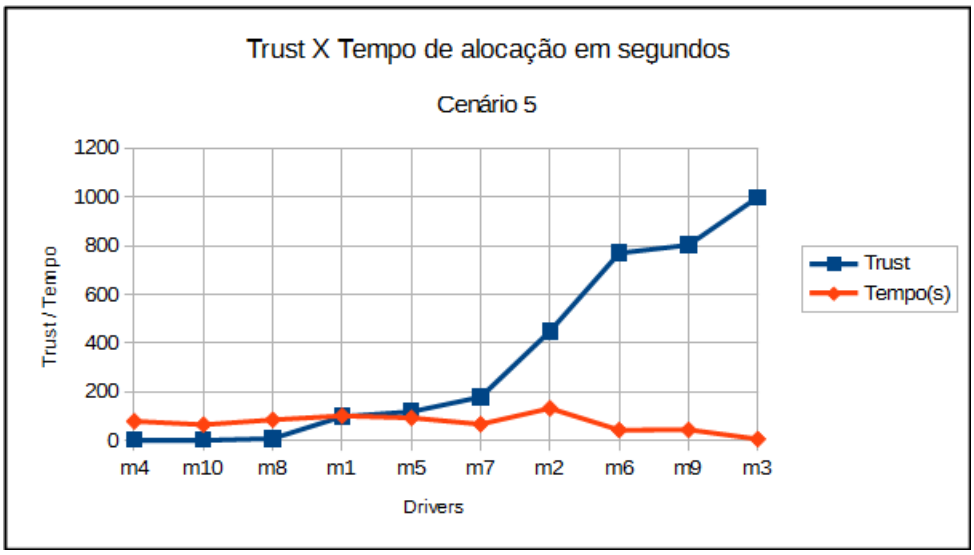

Figura 26 – Alocação *Drivers* - Cenário 5

Fonte: Autoria Própria

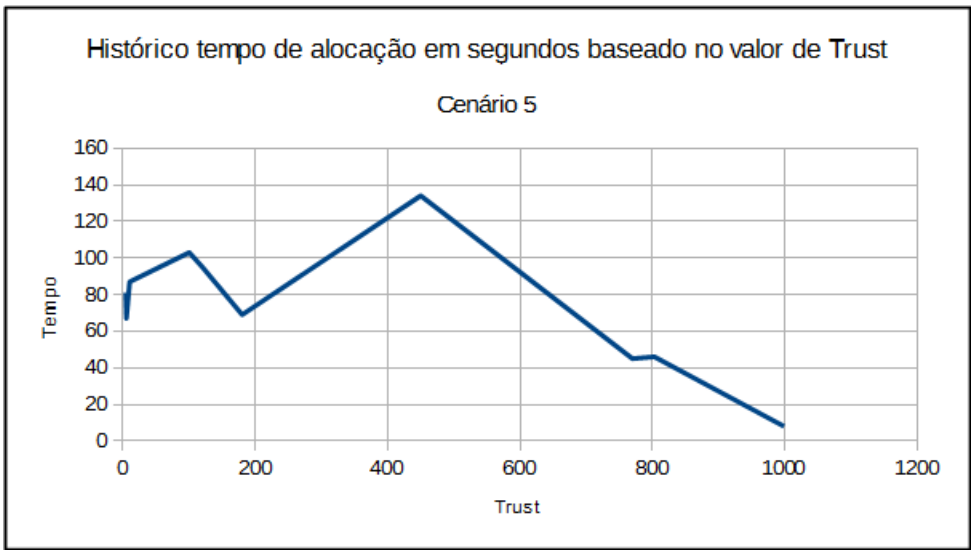

Figura 27 – Histórico tempo de alocação de acordo com valor de *trust* - Cenário 5

Fonte: Autoria Própria

6.7 CENÁRIO 6

Nesse cenário será utilizado 25 agentes *Driver* e 2 vagas de estacionamento. A figura 28 apresenta um gráfico de como os *Drivers* apresentados na tabela 4 são alocados. Nota-se que neste cenário no gráfico da figura 29 a curva do tempo de alocação teve uma queda inicial em um *Driver* valor de *trust* muito baixo. Para os demais *drivers* conforme o valor de *trust* aumentava, diminuía o tempo de alocação.

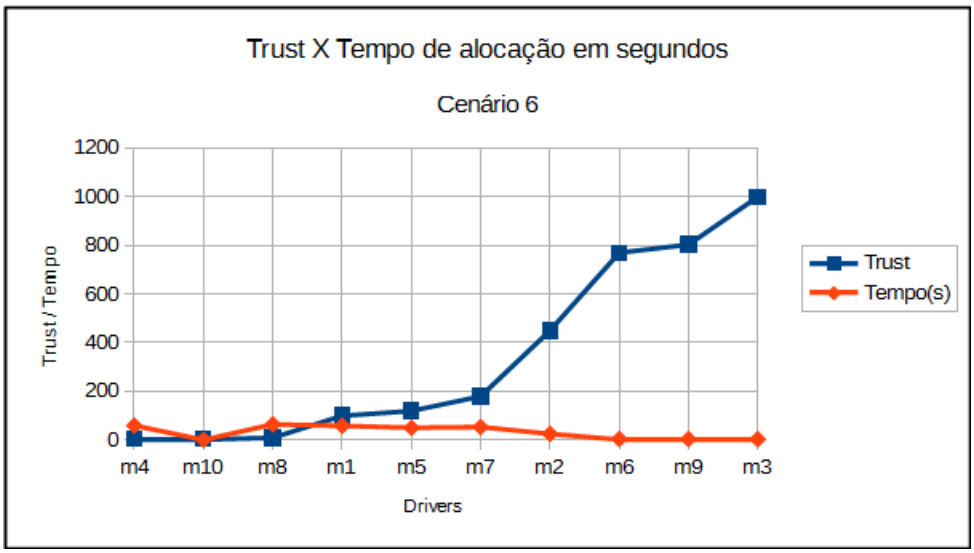

Figura 28 – Alocação *Drivers* - Cenário 6

Fonte: Autoria Própria

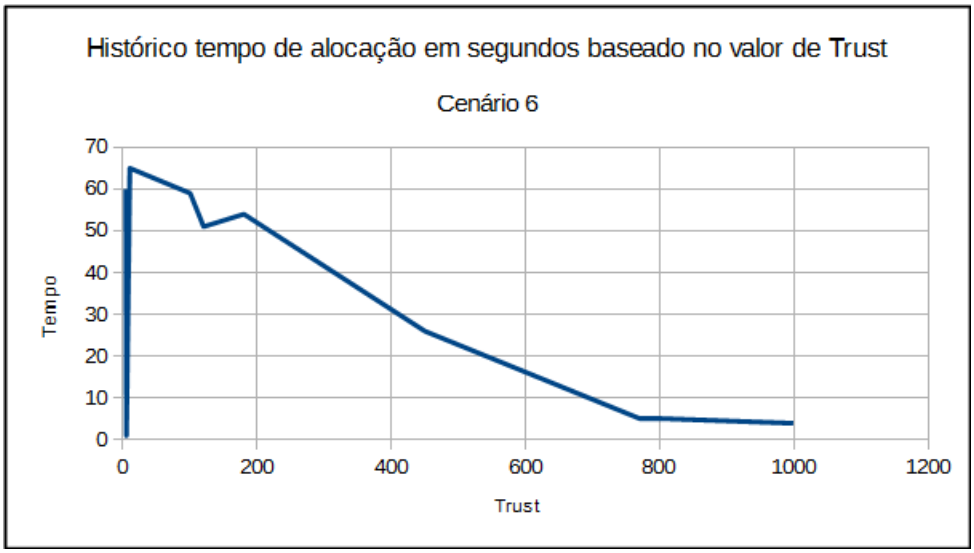

Figura 29 – Histórico tempo de alocação de acordo com valor de *trust* - Cenário 6

Fonte: Autoria Própria

6.8 CENÁRIO 8

Nesse cenário será utilizado 10 agentes *Driver* e 1 vaga de estacionamento. A figura 30 apresenta um gráfico de como os *Drivers* apresentados na tabela 4 são alocados. Vale ressaltar que no gráfico da figura 31 apresenta uma queda no tempo de alocação inicial, e novamente uma queda após o valor de *trust* aumentar.

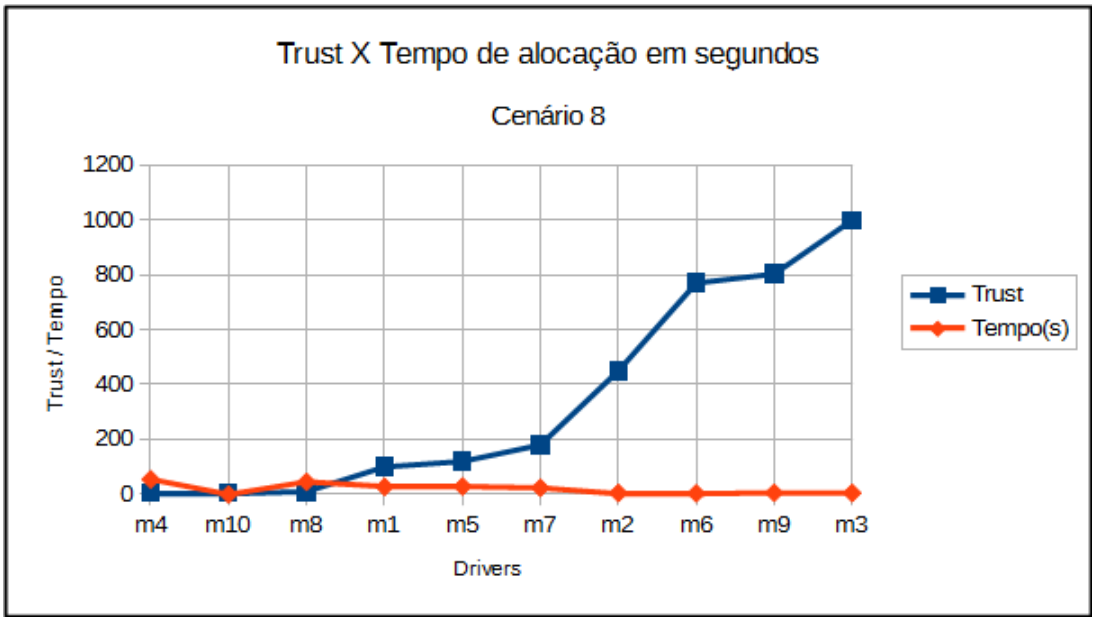

Figura 30 – Alocação *Drivers* - Cenário 8

Fonte: Autoria Própria

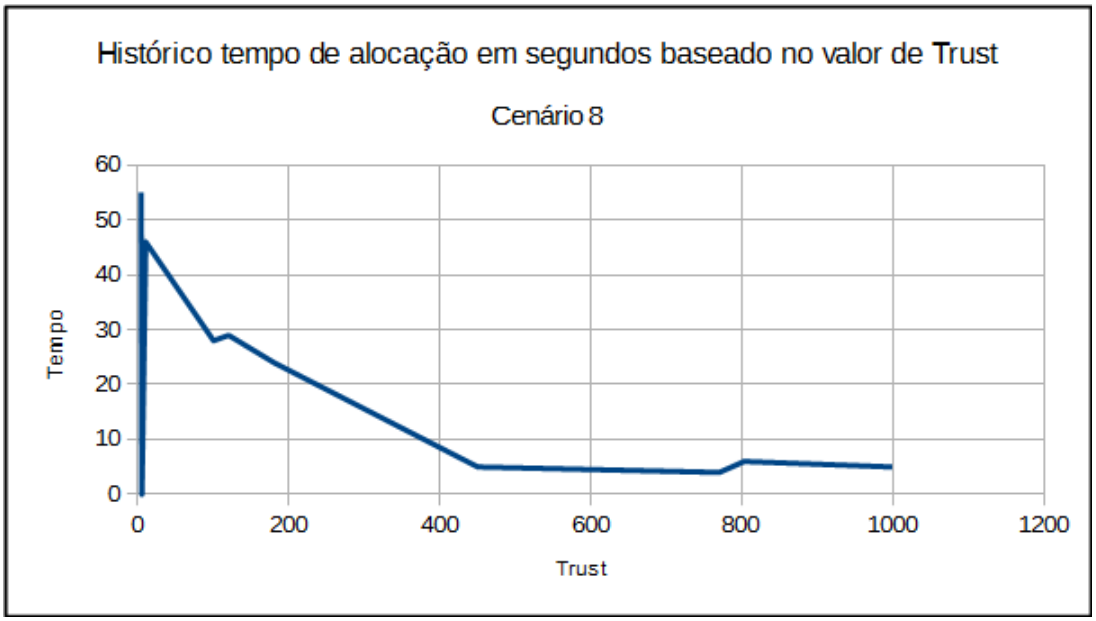

Figura 31 – Histórico tempo de alocação de acordo com valor de *trust* - Cenário 8

Fonte: Autoria Própria

6.9 DISCUSSÃO DOS CENÁRIOS

O principal objetivo da descrição dos cenários nas seções anteriores foi a análise do impacto que o valor de *trust* gera na alocação de vagas dos agentes *Drivers*. Vale notar que durante a alocação de vagas também é analisado o tempo de espera do *Driver*. A fim de sumarizar os cenários apresentados, com base na tabela 4 foram selecionados agentes para cada nível de valor de *trust*: dois agentes para o valor alto (700->999), um agente para o valor médio (350->699) e três agentes para o valor baixo (0->349). A tabela 6 ilustra os agentes selecionados e seu tempo de alocação para receber uma vaga nos cenários 1,2,3,5,6 e 8. Além disso é apresentado a média aritmética de alocação do agente nos cenários baseada no tempo de alocação para os cenários 1 à 10.

Tabela 6 – Agentes selecionados para análise

Agt.	***myTrust (MT)***	***timeToArrive (TA)***	***timeToSpend (TS)***	**C1**	**C2**	**C3**	**C5**	**C6**	**C8**	**Média**
m4	4	3,5s	7,5s	142	87	0	81	60	55	43.1
m10	5	4,6s	6,6s	115	79	0	67	1	0	26.2
m7	180	9s	12s	130	85	0	69	54	24	36.2
m2	450	2s	3s	344	175	15	134	26	5	69.9
m9	803	9,9s	6,6s	148	94	0	46	5	6	30
m3	999	7s	5s	3	3	0	8	4	5	2.3

Fonte: Autoria Própria

Com base na tabela 6 foi elaborado um gráfico (vide 32) com o objetivo de ilustrar o tempo de alocação dos agentes nos cenários desenvolvidos.

Segundo a figura 32 é possível notar o impacto que o valor de *trust* gerou nos motoristas. Conforme o valor de *trust* aumenta, consequentemente o tempo de alocação diminui. Em alguns casos, como por exemplo o Driver `m2` que possui um valor de *trust* médio (450) teve um tempo alto de alocação de vagas. Sendo assim, um *Driver* com um valor médio de *trust* não garante um tempo médio de alocação de vagas, porém, um tempo em muitos casos alto. Isso ocorre em algumas situações devido a função de verificação se há algum agente com mais de 60 segundos de espera (Ver seção 5.2.2). No caso dos Drivers com valor de *trust* alto, `m9`,`m3`, é possível notar que os tempos de alocação são baixos, dando destaque para o `m3` que na média geral de tempo de alocação teve em média 2.3 segundos de espera, ao contrário dos agentes `m4`,`m10`,`m2` que com um valor de *trust* baixo e médio tiveram um tempo de espera maior que 30 segundos. Na figura 32 foi retirado o cenário 3 devido ao fato que apenas um *Driver* da tabela 6 (`m2`) teve um tempo de alocação de vaga superior a zero.

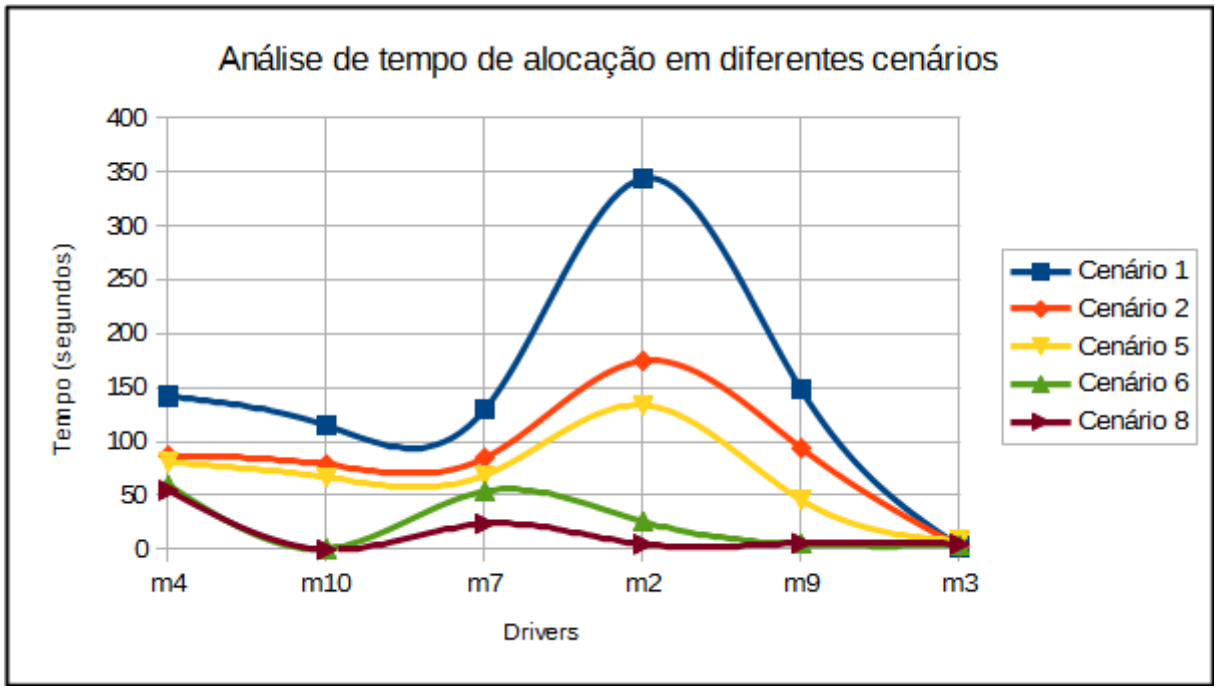

Figura 32 – Análise dos agentes *Drivers* nos cenários 1,2,5,6 e 8

Fonte: Autoria Própria

7 CONCLUSÃO

O projeto MAPS baseia-se no contexto de estacionamentos inteligentes inseridos em cidades inteligentes, as quais se denominam cidades que possuem recursos que automatizam e facilitam as tarefas diárias. Dentro desse contexto, um estacionamento inteligente é o responsável por proporcionar uma automatização com o uso de tecnologias que facilitam a realização das tarefas. Sendo assim, o presente trabalho apresenta o desenvolvimento de um sistema multiagente capaz de alocar vagas em um estacionamento através de um agente centralizador, o qual é responsável pelo controle do local.

O sistema multiagente foi desenvolvido utilizando o *framework* JaCaMo devido ao fato por ser uma plataforma robusta e flexível, proporcionando desde o desenvolvimento dos agentes, do ambiente e da organização social dos agentes. A utilização desta plataforma aponta uma das contribuições do atual trabalho, visto por ser uma ferramenta completa e recente na área de sistemas multiagentes. Outro fator de destaque se dá ao fato de como os módulos que compõem o JaCaMo, em específico Jason e Cartago interligam-se. Durante o desenvolvimento do SMA foi utilizado e apresentado como os agentes interagem com os artefatos do ambiente. Além disso, a utilização do JaCaMo no contexto de estacionamentos, especificamente no problema de alocação de vagas torna-se uma contribuição deste trabalho em vista que há soluções de *smartparking* utlizando agentes para precificação de vagas, entretanto até então inexistente no contexto de alocação de vagas.

A fim de verificar o funcionamento do SMA, foram realizados testes com diferentes configurações exibidas através dos cenários 0->11 (Seção 6.1), onde vários agentes disputam pela mesma vaga, sendo essa vaga prioritária aos usuários de maior valor de confiança (*trust*) ou com maior tempo de espera (> 60 segundos), verificando-se assim a capacidade do sistema de atender diferentes agentes em um mesmo intervalo de tempo. Na seção 6.9 ainda é apresentado uma discussão onde os cenários são analisados e como os agentes *Drivers* foram alocados, com o objetivo de analisar a relação *trust versus* tempo de alocação. Onde foi possível verificar através de tabelas e gráficos que um valor de *trust* alto implica em um tempo de alocação baixo, e em contrapartida um valor de *trust* baixo ou médio implica e um tempo de alocação para uma vaga alto. Foi apresentado também um cenário específico para ilustrar as interações dos agentes (Cenário 0) onde através de um diagrama de sequência é possível verificar etapa a etapa a execução do SMA desenvolvido.

O desenvolvimento do MAPS implicou não somente em um sistema capaz de alocar vagas em um estacionamento para motoristas, mas sim em um contexto onde os motoristas possuem um valor que corresponde a confiança *(trust)* que o gerente (*manager*) possui sobre ele, podendo assim influenciar como esse motorista irá receber a sua vaga, podendo ter um tempo de espera por uma vaga baixo ou alto. O sistema MAPS além de objetivar a alocação de vagas, tem como meta abstrair as características de um motorista real, sendo essas características

implementadas através de crenças, como o tempo de chegada ao estacionamento e tempo de utilização. O MAPS no atual trabalho através da sua execução nos cenários de teste apresentados cumpriu as suas metas iniciais, porém com o cumprimentos destas, novas metas foram elencadas a fim de tornar o sistema ainda mais completo.

7.1 TRABALHOS FUTUROS

O Sistema Multiagente desenvolvido para o MAPS possui como objetivo primário a alocação de vagas para motoristas. Porém já está em desenvolvimento possíveis extensões desse SMA, podendo destacar:

- A descentralização do gerenciamento do estacionamento, podendo os agentes *drivers* negociar vagas entre si e dessa forma implantar mecanismos de coordenação e cooperação entre os agentes;
- Implementação das normas sociais que regem o estacionamento desenvolvidas pela ferramenta Moise;
- Disposição geográfica das vagas no estacionamento, não apenas sendo identificadas por identificadores, mas também por coordenadas espaciais;
- Desenvolvimento do módulo Android onde os agentes *drivers* poderão requisitar as vagas por meio de um aplicativo móvel;
- Implementação de diferentes mecanismos de negociação entre os agentes, para assim lidar com as diferentes lotações de um estacionamento (quase vazio, intermediário e quase cheio);
- Utilização da ferramenta de simulação de trânsito SUMO *(Simulation of Urban MObility)*, para melhor simulação do SMA e visualização dos resultados (BEHRISCH *et al.*, 2011).
- Implementação de novos mecanismos de confiança (*trust*) para analisar e comparar com os resultados aqui apresentados, com o objetivo de selecionar o método mais adequado para calcular e atualizar o valor de *trust* para cada *Driver* do sistema.
- Aplicação de algoritmos de organização e gerenciamento de vagas no estacionamento. Para assim, procurar melhorar a organização e distribuição de vagas.

REFERÊNCIAS

ALONSO, F.; FUERTES, J.L.; MARTINEZ, L. Measuring the social ability of software agents. In: **Software Engineering Research, Management and Applications, 2008. SERA '08. Sixth International Conference on**. [S.l.: s.n.], 2008. p. 3–10.

BAZZAN, Ana L. C.; KLüGL, Franziska. A review on agent-based technology for traffic and transportation. **The Knowledge Engineering Review**, FirstView, p. 1–29, 5 2013. ISSN 1469-8005. Disponível em: <http://journals.cambridge.org/article_S0269888913000118>.

BEHRISCH, Michael *et al.* Sumo–simulation of urban mobility. In: **The Third International Conference on Advances in System Simulation (SIMUL 2011), Barcelona, Spain**. [S.l.: s.n.], 2011.

BORDINI, Rafael H.; HüBNER, Jomi Fred; WOOLDRIDGE, Michael. **Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)**. [S.l.]: John Wiley & Sons, 2007. ISBN 0470029005.

CARAGLIU, Andrea; BO, Chiara Del; NIJKAMP, Peter. Smart cities in europe. **Journal of Urban Technology**, v. 18, n. 2, p. 65–82, 2011.

CArtAgO. 2006. Disponível em: <http://cartago.sourceforge.net/>.

GONçALVES, Wesley R. C.; ALVES, Gleifer Vaz. Smart parking: mecanismo de leilão de vagas de estacionamento usando reputação entre agentes. In: **Anais do IX Workshop-Escola de Sistemas de Agentes, seus Ambientes e Aplicações – IX WESAAC**. [S.l.: s.n.], 2015.

GRIFFITHS, SARAH. **Parking in a city centre just got easier: Sensors find spaces and smart LAMP POSTS guide you to the nearest spot**. 2014. Disponível em: <http://www.dailymail.co.uk/sciencetech/article-2667536/Smart-LAMP-POSTS-end-parking-woes-App-locates-space-lights-guide-spot.html>.

HONAVAR, Vasant. Intelligent agents and multi-agent systems. **IEEE Conference on Evolutionary Computation**, 1999.

HORTY, John F.; POLLACK, Martha E. Evaluating new options in the context of existing plans. **Artificial Intelligence**, v. 127, n. 2, p. 199 – 220, 2001. ISSN 0004-3702. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0004370201000601>.

INGRAND, F.F.; GEORGEFF, M.P.; RAO, A.S. An architecture for real-time reasoning and system control. **IEEE Expert**, v. 7, n. 6, p. 34–44, Dec 1992. ISSN 0885-9000.

ITO, T. *et al.* Innovating multiagent algorithms for smart city: An overview. In: **Service-Oriented Computing and Applications (SOCA), 2012 5th IEEE International Conference on**. [S.l.: s.n.], 2012. p. 1–8.

JACAMO. **The JaCaMo approach**. 2011. Internet. Disponível em: <http://jacamo.sourceforge.net/?page_id=40>.

JASON. **Jason | a Java-based interpreter for an extended version of AgentSpeak**. 2005. Disponível em: <http://jason.sourceforge.net/wp/>.

JR., Jaime S. Sichman Marcio F. Stabile. Incorporando filtros de percepção para aumentar o desempenho de agentes jason. In: **Anais do IX Workshop-Escola de Sistemas de Agentes, seus Ambientes e Aplicações – IX WESAAC**. [S.l.: s.n.], 2015.

KOSTER, Andrew; KOCH, Fernando; BAZZAN, Ana L.C. Incentivising crowdsourced parking solutions. In: NIN, Jordi; VILLATORO, Daniel (Ed.). **Citizen in Sensor Networks**. Springer International Publishing, 2014, (Lecture Notes in Computer Science, v. 8313). p. 36–43. ISBN 978-3-319-04177-3. Disponível em: <http://dx.doi.org/10.1007/978-3-319-04178-0_4>.

LESSER, Victor R. Multiagent systems: An emerging subdiscipline of ai. **ACM Comput. Surv.**, New York, NY, USA, v. 27, n. 3, p. 340–342, set. 1995. ISSN 0360-0300.

MOISE. **The Moise Organisation Oriented Programming Framework**. 2002. Disponível em: <http://moise.sourceforge.net/>.

MYERS, Karen L. **User's Guide for the Procedural Reasoning System**. Menlo Park, CA, 1993.

NAPOLI, Claudia Di; NOCERA, Dario Di; ROSSI, Silvia. Agent negotiation for different needs in smart parking allocation. In: **Advances in Practical Applications of Heterogeneous Multi-Agent Systems. The PAAMS Collection**. Springer, 2014. p. 98–109. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-319-07551-8_9>.

____. Negotiating parking spaces in smart cities. In: **Proceeding of the 8th International Workshop on Agents in Traffic and Transportation, in conjunction with AAMAS**. [S.l.: s.n.], 2014.

NOCERA, Dario Di; NAPOLI, Claudia Di; ROSSI, Silvia. A Social-Aware Smart Parking Application. In: **Proceedings of the 15th Workshop "From Objects to Agents"**. [S.l.: s.n.], 2014. v. 1269.

PARKINGEDGE. **San Francisco - Best Parking**. 2013. Disponível em: <http://sanfrancisco.bestparking.com>.

PROMETHEUS. **The Prometheus Methodoloy**. 2015. Disponível em: <http://www.cs.rmit.edu.au/agentsSAC2/methodology.html>.

RAO, Anand S. **AgentSpeak(L): BDI Agents speak out in a logical computable language**. 1996.

REVATHI, G.; DHULIPALA, V.R.S. Smart parking systems and sensors: A survey. In: **Computing, Communication and Applications (ICCCA), 2012 International Conference on**. [S.l.: s.n.], 2012. p. 1–5.

RICCI, Alessandro; PIUNTI, Michele; VIROLI, Mirko. Environment programming in multi-agent systems: An artifact-based perspective. **Autonomous Agents and Multi-Agent Systems**, Kluwer Academic Publishers, Hingham, MA, USA, v. 23, n. 2, p. 158–192, set. 2011. ISSN 1387-2532.

RICO, J. *et al.* Parking Easier by Using Context Information of a Smart City: Enabling Fast Search and Management of Parking Resources. In: . IEEE, 2013. p. 1380–1385. ISBN 978-1-4673-6239-9 978-0-7695-4952-1. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6550588>.

ROLOFF, M.L. *et al.* A multi-agent system for the production control of printed circuit boards using jacamo and prometheus aeolus. In: **Industrial Informatics (INDIN), 2014 12th IEEE International Conference on**. [S.l.: s.n.], 2014. p. 236–241.

RUSSELL, Stuart J; NORVIG, Peter. **Inteligência artificial**. Rio de Janeiro: Elsevier ; Campus, 2004. ISBN 8535211772 9788535211771.

SFPARK. **SFPark**. 2015. Disponível em: <http://www.sfpark.org/>.

SYCARA, Katia P. Multiagent systems. **AI magazine**, v. 19, n. 2, p. 79, 1998.

WILONSKY, Robert. 2015. Disponível em: <http://cityhallblog.dallasnews.com/2015/11/dallas-is-ready-to-pay-someone-to-find-out-if-the-city-has-enough-downtown-parking-spaces.html/>.

WOOLDRIDGE, Michael. **An Introduction to MultiAgent Systems**. 2nd. ed. [S.l.]: Wiley Publishing, 2009. ISBN 0470519460, 9780470519462.

ZHANG, Guangzhi *et al.* Agent-based simulation and optimization of urban transit system. **Intelligent Transportation Systems, IEEE Transactions on**, v. 15, n. 2, p. 589–596, April 2014. ISSN 1524-9050.

ZHAO, Xuejian; ZHAO, Kui; HAI, Feng. An algorithm of parking planning for smart parking system. In: . IEEE, 2014. p. 4965–4969. ISBN 978-1-4799-5825-2. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7053556>.