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

WESLEY RAMOS CAVALCANTE GONÇALVES

ORGANIZAÇÃO SOCIAL E NORMATIVA DE UM SISTEMA MULTIAGENTE PARA ALOCAÇÃO DE VAGAS EM UM ESTACIONAMENTO INTELIGENTE

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA
2017

WESLEY RAMOS CAVALCANTE GONÇALVES

ORGANIZAÇÃO SOCIAL E NORMATIVA DE UM SISTEMA MULTIAGENTE PARA ALOCAÇÃO DE VAGAS EM UM ESTACIONAMENTO INTELIGENTE

Trabalho de Conclusão de Curso apresentado como requisito parcial para 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
2017

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Ponta Grossa
Diretoria de Graduação e Educação Profissional
Departamento Acadêmico de Informática
Bacharelado em Ciência da Computação

TERMO DE APROVAÇÃO

ORGANIZAÇÃO SOCIAL E NORMATIVA DE UM SISTEMA MULTIAGENTE PARA ALOCAÇÃO DE VAGAS EM UM ESTACIONAMENTO INTELIGENTE

por

WESLEY RAMOS CAVALCANTE GONÇALVES

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 25 de maio de 2017 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Prof. Dr. Gleifer Vaz Alves
Orientador

_____
Prof. Dr. André Pinz Borges
Membro titular

_____
Prof. Msc. Geraldo Ranthum
Membro titular

_____
Prof. Dr. Ionildo José Sanches
Responsável pelo Trabalho de Conclusão de Curso

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

RESUMO

GONçALVES, Wesley Ramos Cavalcante. **Organização social e normativa de um Sistema Multiagente para alocação de vagas em um Estacionamento Inteligente**. 2017. 117 f. Trabalho de Conclusão de Curso em Ciência da Computação, Universidade Tecnológica Federal Do Paraná. Ponta Grossa, 2017.

O desenvolvimento de sistemas multiagentes colaboram para a criação de sistemas inteligentes que tem como objetivo otimizar as atividades humanas. Este conceito é aplicado a estacionamento inteligentes, que utilizam a tecnologia para melhorar o uso do estacionamento pelos motoristas. Este trabalho está inserido no projeto MAPS desenvolvido na UTFPR que visa o desenvolvimento de um SMA estendido do MAPS com um sistema normativo, que tem como objetivo enviar vagas para os motoristas que mais colaboram com o sistema. A ferramenta utilizada para implementar o sistema normativo é o MOISE, o qual integra-se ao Jason e CArtAgO por meio da plataforma JaCaMo. O sistema desenvolvido chama-se MAPS-NORMS, que controla o funcionamento do estacionamento e determina algumas ações como obrigatórias para organizar as ações dos motoristas. Fora implementado artefatos para representar as vagas e alterações nos agentes, essas modificações contribuíram para o desenvolvimento do sistema normativo. Foram implementados e testados três níveis de restrições e quatro comportamentos que os agentes motoristas podem assumir. Apesar da complexidade ter aumentado no MAPS-NORMS se tornou mais robusto e flexível quando comparado ao MAPS.

Palavras-chaves: sistema multiagente. sistema normativo. moise. estacionamento.

ABSTRACT

GONçALVES, Wesley Ramos Cavalcante **Normative and Social Organization of a Multiagent System to manage parking spots in a Smart Parking**. 2017. 117 f. Work of Conclusion Course Graduation in Computer Science, Federal Technological University - Paraná. Ponta Grossa, 2017.

The development of multi-agent systems collaborate to create intelligent systems that aim to optimize human activities. This concept is applied to intelligent parking, which use the technology in parking lots to improve the use of parking by drivers. This work is part of the MAPS project developed at UTFPR. It aims to develop an extended SMA of MAPS with a normative system, which aims to send vacancies to the drivers who collaborate with the system. The tool used to implement the normative system is MOISE, which integrates with Jason and CArtAgO through the JaCaMo platform. The developed system is called MAPS-NORMS, which controls the operation of the parking lot and determines some actions as mandatory to organize the actions of the drivers. It was implemented artifacts to represent the vacancies and changes in the agents, these modifications contributed to the development of the normative system. Three levels of restriction and four behaviors that driver agents can assume have been implemented and tested. Although the complexity has increased the SMA has become more robust and flexible.

Key-words: multiagent system. normative system. moise. parking lot.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

ABGS	*Agent Based Guiding System*
BDI	*Belief -Desire-Intention*
CArtAgO	*Common "Artifacts for Agents" Open framework*
CSR	***C**hoice **S**pot **R**andomly*
EE	Especificação Estrutural
EF	Especificação Funcional
EN	Especificação Normativa
EO	Entidade Organizacional
ES	Esquema Social
FSF	*First Spot Free*
GC	Grau de Confiança
GCA	Grau de Confiança Antigo
JaCaMo	Jason-CArtAgO-MOISE
KQML	*Knowledge Query Management Language*
MAPS	*MultiAgent Parking System*
MadKit	*Multi-Agent Development Kit*
MOISE	*Model of Organization for multI-agent SystEms*
NOE	Nível de ocupação do estacionamento
NRA	*Not Reserved to Anyone*
NTD	*Not reserved To the Driver*
OE	Organização Estrutural
OI	Organização Individual
OO	Orientação à Objetos
OS	Organização Estrutural
RTA	*Reserved To Another*
RTD	*Reserved To the Driver*
SIGE	Sistema de Informação e Guia em Estacionamento
SMA	Sistema Multiagentes

SN	Sistema Normativo
SNM	Sistema Normativo Multiagente
XML	*EXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

As cidades são ambientes modernos e complexos que proporcionam benefícios para sua população, como o aumento da expectativa de vida da população, saneamento básico, dentre outros. Além disso, pode causar malefícios como estresse, doenças respiratórias e outras mazelas. Para eliminar ou minimizar os problemas urbanos que impactam na vida da população é necessário aplicar soluções inovadoras, inteligentes e eficientes. O conceito de Cidades Inteligentes foi criado com o objetivo de aumentar a segurança, eficiência e comodidade nas atividades diárias da população a partir da tecnologia. Este conceito visa o uso eficiente dos recursos naturais e urbanos por meio de sistemas tecnológicos inteligentes (MEIJER; BOLÍVAR, 2015). A eficiência é buscada no uso racional de recursos naturais e urbanos. Os recursos naturais são: água, ar, alimentos, dentre outros e recursos urbanos: matrizes energéticas, vias de trânsito, recursos financeiros e outros.

Os sistemas criados a partir do conceito de Cidade Inteligente são desenvolvidos, construídos e mantidos por meio de dispositivos modernos como sensores eletrônicos, alarmes, localizadores globais, entre outros. Estes itens são conectados em sistemas computacionais, como servidores. Estes servidores utilizam os dados recebidos pelo ambiente, processa-os via algoritmos e extraem informações úteis para os usuários. O usuário ao ser informado pelo sistema, poderá tomar a melhor decisão em função do contexto em que se encontra. Estes sistemas são aplicados em diversas componentes de uma cidade: educação, indústria, trânsito, etc. A partir da união de Cidades Inteligentes e as componentes de uma cidade, surgem: Economia Inteligente, Mobilidade Inteligente, Estacionamento Inteligente, dentre outras subdivisões.

Trânsito Inteligente é uma subdivisão do conceito Cidade Inteligente, o qual surge da união do conceito Cidade Inteligente ao componente Trânsito (PETROLO; LOSCRÌ; MITTON, 2015). O trânsito é um item essencial no meio urbano, responsável pela estrutura que permite o transporte de pessoas e objetos por vias públicas. Um item importante do trânsito é o estacionamento que permite aos motoristas estacionarem seus veículos próximos ao seu destino. A dificuldade em encontrar vagas livres em grandes centros tornou-se um problema pois pode causar congestionamentos. Quando um motorista procura uma vaga, seu veículo se locomove mais devagar que os veículos que estão trafegando normalmente, fazendo com que os veículos atrás dele se locomovam devagar gerando uma reação em cadeia e cause um congestionamento. Se muitos motoristas procuram vagas, as chances de haver congestionamento aumenta. Uma alternativa para minimizar este problema é o conceito de Estacionamento Inteligente (KOSTER; KOCH; BAZZAN, 2014).

O objetivo principal de um Estacionamento Inteligente é a otimização da alocação de vagas por meio de sistemas tecnológicos com a capacidade de organizar de forma sistemática e autônoma as vagas existentes de um estacionamento, e por meio de sensores, componentes eletrônicos, aplicativos para *smartphone*, dentre outros. Diante de diversas soluções e tecnolo-

gias que foram criadas para auxiliar o estacionamento de veículos. Estão as taxonomias para classificar as soluções em estacionamentos inteligentes. Uma delas desenvolvida por Lin (LIN, 2015) detalha as diversas soluções em todos os aspectos existentes para estacionar um veículo. Nesta taxonomia existem três ramificações principais: Coleta de Informações, Desenvolvimento de Sistemas e Serviço de Disseminação.

A Coleta de Informação utiliza sensores para detectar as ações nos estacionamentos e em vias urbanas. Existem sensores estáticos como em vagas para detectar se a mesma está ocupada e sensores dinâmicos para saber quantos motoristas passaram por determinada rua, por exemplo. Além disso, usa redes sem fio para saber a localização de todos os veículos numa área, a partir do uso de aplicativos em *smartphones* que fornecem os dados dos motoristas.

O Desenvolvimento de Sistemas trabalha com as informações em larga escala do ambiente. O objetivo é guiar os motoristas para seu destino a partir da análise da situação das rotas em tempo real e também realiza predições sobre o trânsito e o estacionamento, como modelos preditivos sobre a porcentagem de ocupação de um estacionamento em determinada época do ano.

O Serviço de Disseminação lida com a noção de recursos sociais, competição por vagas e o comportamento dos motoristas. Estes itens incentivam os motoristas a terem um determinado comportamento que possibilitem colaborar com o estacionamento. Uma forma de incentivar o bom comportamento de um motorista é via ações sociais como punições e bonificações que alteram o tempo que um motorista consegue uma vaga, por exemplo. O estímulo que o agente motorista recebe é apoiado pela Teoria de Agentes.

A Teoria de Agentes estabelece que um agente é um ser que tem proatividade, reatividade, autonomia e habilidades sociais (WEISS, 1999). Cada agente possui um conjunto de receptores e atuadores para perceber o ambiente em que está inserido e poder modificá-lo. Desta forma, um agente tem a capacidade de atingir os próprios objetivos via sua autonomia para escolher a próxima ação a tomar. As habilidades sociais fornecem ao agente a capacidade de interagir com outros agentes. Um conjunto de agentes interagindo entre si num mesmo ambiente é chamado de Sistema Multiagente (SMA).

O uso de SMA para representar e modelar um estacionamento inteligente está contido no Serviço de Disseminação e representa o estacionamento e suas respectivas vagas como o ambiente do SMA. As vagas além de ambiente também são um recurso que os agentes motoristas objetivam ter. Neste ambiente os agentes apresentam autonomia, reatividade, proatividade e habilidades sociais individuais. Estas características podem representar comportamentos dinâmicos e interativos, assim podem representar agentes como abstrações de motoristas e gerentes do estacionamento (CHOU; LIN; LI, 2008).

O trabalho aqui apresentado está inserido no projeto MAPS (*MultiAgent Parking System*) desenvolvido no grupo de pesquisa GPAS da UTFPR - Câmpus Ponta Grossa. O projeto MAPS utiliza o conceito de SMA para desenvolver estacionamentos inteligentes. Neste projeto foi criado um SMA de mesmo nome, chamado MAPS (CASTRO, 2015). O MAPS é a

primeira versão de um estacionamento inteligente implementado com o conceito ABGS e tem como objetivo implementar os agentes e o ambiente por meio da plataforma JaCaMo(**Ja**son-**CA**rtAgO-**MO**ISE).

Contudo, em sua primeira versão, o projeto MAPS não contemplou a implantação da organização normativa, a qual deve ser realizada por meio do MOISE (*Model of Organization for multI-agent SystEms* ou Modelo Organizacional para Sistemas Multiagente). Portanto, o objetivo principal deste trabalho é justamente preencher essa lacuna do projeto MAPS por meio do desenvolvimento de um SMA normativo.

Este SMA normativo é denominado aqui MAPS-NORMS, o qual possui modificações e novas funcionalidades implementadas no MAPS para a criação de uma organização normativa. A organização normativa é realizada pela ferramenta MOISE (HÜBNER, 2003). A ferramenta implementa as normas que os agentes devem seguir para manter os agentes organizados e coordenados para atingir seus objetivos individuais e globais do sistema. Um sistema normativo estabelece a organização normativa de um SMA.

A organização normativa é um conjunto de normas que serão utilizadas para manter a harmonia entre a autonomia de cada agente e os objetivos globais do sistema. As normas são importantes para construir relações sociais entre os agentes, uma vez que estabelece a forma como os agentes vão interagir entre si. Cada agente tem para si: direitos, responsabilidades, obrigações, deveres, privilégios, penalidades, autorizações e permissões, cabendo ao desenvolvedor de um SMA levar em consideração na especificação do sistema normativo quais aspectos são mais relevantes a seus agentes (SEARLE, 1995). Cada agente ao entrar num SMA possui um papel, o papel estabelece um conjunto de permissões e obrigações que restringem o comportamento de um agente, e o papel define o relacionamento entre diferentes agentes.

Para implementar a organização social do sistema é utilizada a ferramenta MOISE, responsável pela aplicação das normas desenvolvidas aos agentes e a forma como será estruturada a organização social do sistema (BOISSIER *et al.*, 2013). Desta forma, organizar as ações do agentes de acordo com a situação dos agentes e do ambiente. O $\mathcal{M}$OISE+ é o modelo utilizado para ser implementado na plataforma MOISE. Por meio da linguagem de programação orientada a organização, codificada em arquivo ***EX**tensible **M**arkup **L**anguage* (XML).

O maior desafio para a implementação de um sistema normativo é estabelecer o que é permitido e o que é obrigatório para um agente. Ao passo que as normas organizam um SMA também limitam a autonomia dos agentes, o que torna o agente menos flexível e robusto. Por isto, durante a utilização do estacionamento pelo agente motorista, a organização no MAPS-NORMS não determina que o agente motorista é obrigado a estacionar na vaga reservada à ele, isto permite poder estacionar em uma outra vaga do estacionamento.

A autonomia do agente motorista de estacionar em outra vaga que não seja a vaga reservada à ele, abre a possibilidade do estacionamento funcionar de forma menos eficiente, haja vista que um motorista pode demorar para ocupar uma vaga, ou ocupar uma vaga reservada à outro motorista, o que neste caso levará o motorista que teve sua vaga ocupada procurar outra

vaga para estacionar o que demandaria ainda mais tempo. Como forma de incentivar os motoristas a ocupar a vaga reservada à ele, fora desenvolvido um sistema de negociação em que cada vaga será alocada para o melhor agente motorista (GONÇALVES; ALVES, 2015).

O melhor agente motorista é aquele que possui o maior Grau de Confiança (GC) no conjunto de motoristas que requisitaram vaga ao estacionamento. O GC é um valor que é incrementado quando o motorista recebe uma bonificação e decrementado quando recebe uma punição (CASTELFRANCHI; FALCONE, 1998).

No MAPS-NORMS existem três níveis de restrição onde em cada uma, a ação do agente motorista tem um peso sobre o GC. Essa avaliação se dá pela ação de ocupar uma vaga, em que há três possibilidades: o motorista ocupar uma vaga reservada à ele, ocupar uma vaga que não estava reservada para ninguém e ocupar uma vaga que estava reservada à outro motorista. Cada nível de restrição possui para as três possibilidades um peso diferente que influência no GC.

Assim, visa-se desenvolver um sistema normativo e avaliá-lo diante de diferentes níveis de restrições. A fim de verificar se o sistema normativo e a negociação por vagas ocorrem de forma adequada e em quais cenários é benéfica ou não para os motoristas; e também, verificar se é alcançado o objetivo global do estacionamento que é o uso eficiente das vagas levando em conta os objetivos individuais de cada agente (HÜBNER; SICHMAN; BOISSIER, 2007).

Após a implementação foram realizadas as simulações em diversos cenários onde agentes com diferentes comportamentos, utilizaram o estacionamento e foram analisados as seguintes variáveis: tempo de espera que um agente motorista leva para conseguir uma vaga e o GC dos motoristas ao longo da utilização das vagas. Apesar do estacionamento não saber qual comportamento um agente motorista possui, o agente recebeu punição para que utilizasse menos o estacionamento, desta forma o objetivo principal fora atingido.

1.1 OBJETIVOS

A seguir serão descritos o objetivo geral e os objetivos específicos deste trabalho.

1.1.1 Objetivo Geral

Este trabalho tem por objetivo implementar a organização social e normativa de um Sistema Multiagente para alocação de vagas em um Estacionamento Inteligente.

1.1.2 Objetivos Específicos

Os objetivos específicos deste trabalho são:

(i) Definir as normas para a organização do SMA MAPS-NORMS para o funcionamento adequado de um estacionamento inteligente.

(ii) Alterar e implementar novas funcionalidades, tanto na implementação dos agentes (Jason), quanto na construção dos artefatos (CArtAgO), tendo como referência o SMA MAPS.

(iii) Definir a coordenação e a especificação dos papéis dos agentes.

(iv) Implementar a organização normativa utilizando o $\mathcal{M}$OISE+, considerando as normas, as regras e a coordenação definida.

(v) Validar a integração da implementação do sistema normativo através de testes e simulações no MAPS-NORMS.

(vi) Avaliar as correspondências entre os níveis de restrição do estacionamento e o impacto no tempo que os motoristas esperaram para conseguir uma vaga.

1.2 JUSTIFICATIVA

No ano de 2005, haviam 26,7 milhões de carros no Brasil, já em 2015 existiam 50 milhões de automóveis (FICHMAN; DUARTE; NETO, 2016). Em 10 anos a frota veicular dobrou, enquanto o crescimento da infraestrutura urbana não seguiu este ritmo de crescimento. Isto causou a dificuldade no funcionamento do trânsito nas cidades, uma vez que a infraestrutura não estava preparada para tal crescimento. Em 2014 no Brasil, acidentes de trânsito causaram 43.075 óbitos e 201.000 feridos hospitalizados (VIASSEGURAS, 2014), assim há a necessidade de alternativas para melhorar a segurança e eficiência do trânsito nas cidades brasileiras.

A alocação de vagas é um dos problemas que afeta o trânsito em várias cidades do mundo. Relatórios de Nova Iorque apontam que 40% do congestionamento existente é causado por motoristas à procura de vagas livres (KOSTER; KOCH; BAZZAN, 2014). Em Pisa na Itália, um estudo pela Telekon (TELEKOM, 2014) identificou que a procura por vagas em cidades grandes impactaram no aumento de 30% do espaço ocupado no trânsito.

Uma alternativa para minimizar este impacto negativo no trânsito é a criação de sistemas inteligentes agregados ao conceito de Cidade Inteligente. Desta união entre componentes modernos e sistemas inteligentes surgem novas formas de organizar o trânsito que podem melhorar seu funcionamento e o cotidiano da população, com os seguintes objetivos: aumentar a

segurança para pedestres e passageiros, diminuir o tempo dos motoristas procurando vagas em estacionamentos, aproveitar de forma eficiente o espaço urbano, como vias e ruas.

Para tentar diminuir o tempo que os motoristas levam à procura de uma vaga, o grupo MAPS desenvolve na plataforma JaCaMo uma solução baseada em SMA para organizar um estacionamento. O sistema auxiliará os motoristas a conseguirem uma vaga a partir de um sistema autônomo de distribuição de vagas. Desta forma, diminuindo o número de carros trafegando nos corredores do estacionamento, os quais estão à procura de uma vaga livre. A plataforma JaCaMo possui três dimensões: Jason para os agentes, CArtAgO para o ambiente e o MOISE para a organização. Este trabalho visa desenvolver a organização por meio da ferramenta MOISE, responsável pelo estabelecimento de normas e regras para coordenar os agentes no sistema.

Uma vantagem em desenvolver a solução computacional em SMA é que se trata de uma solução de alto nível. Desta forma, não está diretamente relacionado a uma plataforma física, uma vez que tem como característica trabalhar com agentes e ambientes heterogêneos. A tecnologia SMA permite ao estacionamento adotar seu próprio modo para os agentes perceberem o ambiente e atuarem no mesmo, assim pode ser aplicada de acordo com necessidades específicas, como sistemas simples ou complexos. Desta forma, sua implementação é flexível para diversos cenários existentes, por exemplo: um SMA pode identificar a entrada dos motoristas no estacionamento por diversas formas: visão computacional, sensores de estrutura metálicas, sistema de posicionamento global, entre outros.

A abordagem utilizada pela plataforma JaCaMo parte da premissa que o conceito de SMA é compatível com a representação de um estacionamento. Dentre as formas de implementação, este trabalho considera que os motoristas são os agentes, o estacionamento é o ambiente onde estes agentes vão interagir e as vagas são os recursos que os agentes objetivam possuir. Este trabalho visa estabelecer as regras e normas do SMA, para tornar o uso do estacionamento mais eficiente e seguro aos motoristas. Por meio de bonificação e penalização em relação as ações que os agentes tomarem, se são boas ou ruins para a organização do estacionamento.

Outra possível aplicação para o MAPS (CASTRO, 2015) e o MAPS-NORMS (SMA apresentado neste trabalho) é a utilização para veículos autônomos (KESSLER, 2015). Veículos autônomos são automóveis equipados com sensores de movimento, computador com localização via-satélite, radar e visão computacional, e tem como capacidade auto dirigir. O automóvel consegue partir de um ponto e ir a outro ponto predeterminado sem auxílio humano. Este tipo de tecnologia está relacionada ao escopo deste trabalho, uma vez que os agentes motoristas terão sua organização automatizada. Assim, o sistema pode ser utilizado por motoristas humanos e robôs.

1.3 ORGANIZAÇÃO DO TRABALHO

O restante deste documento está organizado da seguinte forma: o próximo capítulo apresenta os trabalhos relacionados que colaboraram para a revisão das publicações que estão relacionados a Sistemas Multiagente, Estacionamento Inteligente e Sistemas Normativos.

O Capítulo 3 descreve um sistema normativo multiagente e os elementos que o compõe, agentes, ambientes, sistemas normativos e o sistema normativo multiagente.

O Capítulo 4 descreve a ferramenta JaCaMo, apresenta a linguagem de programação orientada a agentes Jason e a plataforma de desenvolvimento de ambientes CArtAgO. Descreve o modelo de organização de SMA $\mathcal{M}$OISE+. O modelo é descrito em detalhes juntamente a um exemplo. Apresenta-se como realizar a especificação organizacional, e a implementação dos conceitos do modelo.

O Capítulo 5 estabelece o desenvolvimento do trabalho, como foi realizado a implementação do modelo e as mudanças que foram provocadas no funcionamento do MAPS com o $\mathcal{M}$OISE+ implementado.

O Capítulo 6 apresenta os resultados obtidos das simulações realizadas no sistema, e por fim, o Capítulo 7 apresenta as conclusões do trabalho e estabelece os trabalhos futuros.

2 TRABALHOS RELACIONADOS

Este capítulo apresenta os trabalhos relacionados que colaboraram para uma revisão crítica a respeito das publicações mais recentes e relevantes sobre os tópicos: Sistemas Multiagente, Sistemas Normativos e Estacionamentos Inteligentes. Os trabalhos relacionados estão localizados nas intersecções de cada par de campo de estudo, como representado na Figura 1. O MAPS-NORMS utiliza conceitos dessas três grandes áreas, por isto se localiza na intersecção central da Figura, pois desenvolve um sistema normativo para um SMA que tem como objetivo organizar um Estacionamento Inteligente.

Figura 1 – Intersecção de temas relacionados

Fonte: Autoria Própria

2.1 SISTEMAS MULTIAGENTE E ESTACIONAMENTO INTELIGENTE

Os estacionamentos são áreas com comportamento tipicamente dinâmicos, uma vez que o motorista pode escolher ocupar arbitrariamente o estacionamento, e assim o número de vagas livres não obedece um padrão. Por isso, é necessário uma solução que possua a flexibilidade que este problema exige. Sistemas Multiagentes podem ser aplicados como solução em Estacionamentos Inteligentes, pois permitem a representação de motoristas e gerentes de vagas. A teoria de agentes oferece abstrações úteis aos gerentes e motoristas autonomia, proatividade, habilidade social e cooperação. Além disso, permite a organização do funcionamento do estacionamento, o que pode trazer maior eficiência para a distribuição de vagas livres.

A autonomia permite ao agente liberdade para escolher uma ação útil a ele. No contexto de um motorista, a autonomia permite escolher a vaga que quiser estacionar. A proatividade indica que um agente pode realizar uma ação por conta própria, para um motorista essa abstração permite ceder voluntariamente uma vaga para outro motorista que esteja precisando urgentemente. Habilidade social permite a um agente estabelecer uma interação social com outros agentes. Um exemplo deste relacionamento é um motorista seguir uma ordem de um gerente

de estacionamento a fim de melhorar o tráfego. E a cooperação que permite a dois ou mais agentes agirem em harmonia a fim de alcançar um objetivo. Quando um motorista sai de uma vaga e avisa aos outros motoristas que a vaga está livre, existe a cooperação para o objetivo ocupar vaga.

Em (CHOU; LIN; LI, 2008) apresenta-se um sistema que realiza a negociação de vagas entre motoristas e gerentes de vagas em um estacionamento privado. Utiliza-se o conceito de agentes para guiar os motoristas até a vaga livre, apresentando o caminho que o motorista precisa seguir no estacionamento para chegar a vaga reservada a ele. Apresenta como resultado a possibilidade de um estacionamento funcionar completamente sem intervenção humana. Porém, não trata da forma como os agentes racionalizam suas ações e a arquitetura de raciocínio dos agentes. Assim, o modelo organiza um estacionamento mas não especifica a forma como o agente motorista toma sua decisão.

Um suporte de serviço automatizado para encontrar uma vaga livre é desenvolvido em (NAPOLI; NOCERA; ROSSI, 2014). Utiliza-se o conceito de Sistema Multiagente a fim de realizar a negociação do valor pago por uma vaga em um estacionamento. Neste caso existe um agente com o papel de um gerente do estacionamento e um agente que representa cada motorista. A interação é realizada entre o motorista e o gerente do estacionamento por meio dos agentes que os representam. Os agentes dos motoristas procuram vagas que possuam características interessantes a seu perfil e próximos ao destino desejado. Os agentes gerentes de vagas buscam alugar vagas com o maior valor possível. O agente do motorista terá a possibilidade de adaptar-se ao usuário, por exemplo: um motorista que utiliza o sistema esporadicamente possui necessidades diferentes de um motorista que precisa de vagas próximos ao ambiente de trabalho. Uma vez que o motorista precisa todos os dias de uma vaga, não importando sua localização. A partir dessa informação, o sistema pode agilizar o processo de distribuição de vagas. A maior dificuldade para implementar este sistema é unificar o compartilhamento de informações de vários estacionamentos em um sistema único, sendo assim, necessário utilizar a mesma tecnologia em vários estacionamento diferentes.

2.2 SISTEMAS MULTIAGENTE E SISTEMAS NORMATIVOS

Os agentes que pertencem a um SMA possuem a autonomia como uma das características principais. A autonomia é a liberdade de escolha que um agente possui e deve ser regulada a fim de beneficiar o próprio sistema a atingir seus objetivos globais. Quando os agentes têm pouca autonomia o sistema tende a ser mais eficiente, pois as ações dos agentes estarão limitadas, então estes agentes se tornam limitados a resolver apenas um conjunto restrito de problema. Muita autonomia permite aos agentes terem muitas possibilidades, dificultando que o objetivo global seja alcançado. Uma vez que os agentes poderão estar realizando outras tarefas que em nada contribui para o sistema como um todo. Assim, sistemas normativos são ferramentas importantes

para o desenvolvimento de sistemas multiagentes que permitam aos agentes terem autonomia necessária para alcançar suas metas e também conseguir atingir os objetivos globais (WEISS, 1999).

O *framework* KARMA (PYNADATH; TAMBE, 2003) possui como objetivo principal proporcionar uma infraestrutura para agentes heterogêneos e humanos organizarem uma equipe para realizar um conjunto de objetivos. Proporciona um algoritmo para escolher os agentes que vão atuar num time. Após escolher os papéis, define-se a sequência de ações que os agentes irão realizar a partir do desenvolvimento de cenários para atingir cada objetivo. Porém, não existe a definição da dimensão normativa para os papéis dos agentes.

A infraestrutura para a institucionalização eletrônica implementada em AMELI (ESTEVA *et al.*, 2004), proporciona o reforço institucionais de regras para agentes. Apresenta como característica diferenciada ser utilizada para propósito geral. Assim, existe a compatibilidade para diversos protocolos existentes permitindo que infraestruturas heterogêneas consigam se comunicar e cooperar entre si.

A plataforma para sistema multiagente MadKit (Multi-agent Development Kit) (GUTKNECHT; FERBER, 2000), baseia sua implementação em um modelo organizacional. Utiliza conceitos de grupos e papéis para os agentes atuarem para administrar diferentes modelos de agentes e sistemas multiagentes ao mesmo tempo, desta forma manter a estrutural global coesa. Não trata do funcionamento do sistema e nem das normas aplicadas aos agentes.

A ferramenta MOISE (HANNOUN *et al.*, 2000) realiza a normatização de um sistema multiagente. Os agentes conhecem as normas que estão expostos e existe a institucionalização delas, isto é, estas normas são criadas inicialmente pelo desenvolvedor e podem ser modificadas por agentes por meio de inclusão e exclusão de normas em tempo de execução.

2.3 ESTACIONAMENTOS INTELIGENTES E SISTEMAS NORMATIVOS

Estacionamentos comuns e Estacionamentos Inteligentes podem ter sua infraestrutura estabelecida a partir do uso de sistemas normativos, uma vez que são úteis para estabelecer as normas que os agentes motoristas devem seguir. Um sistema normativo também organiza o funcionamento de equipamentos inteligentes que possibilitam entender o estado atual dos veículos e vagas no estacionamento. Desta forma, colaborando para a solução inteligente de alocação de vagas.

Em (FISMAN; MIGUEL, 2007) são estudadas normas culturais e formas legais de reforçá-las para o controle da corrupção no uso de um estacionamento. Foram analisados os comportamentos dos oficiais das Nações Unidas ao estacionar em Manhattan. Este estudo tem o viés relacionado a corrupção em relação as normas do estacionamento durante o uso por diplomatas. Foi constatado uma alto correlação entre diplomatas que são de países com alto índice

de corrupção e suas ações corruptivas durante o uso do estacionamento, acumulando mais violações em normas relacionadas ao não pagamento da vaga utilizada. As normas culturais dos motoristas e as ações que reforçam a atitude são determinantes para que ocorra a manutenção da corrupção.

2.4 SISTEMAS MULTIAGENTE, ESTACIONAMENTOS INTELIGENTES E SISTEMAS NORMATIVOS

Na intersecção das três grandes áreas citadas anteriormente está o trabalho de (BILAL *et al.*, 2012), é proposto um modelo que utiliza o sistema normativo Moise para a organização de um sistema multiagente. Que define um conjunto global de papéis para cada componente da infraestrutura utilizada em um Estacionamento Inteligente. Estes componentes são necessários para o gerenciamento flexível das vagas em um estacionamento. Neste trabalho é integrado a plataforma SensCity que é dedicada ao desenvolvimento e implementação de sistemas que realizam a interação entre dispositivos.

Dentre os trabalhos relacionados (BILAL *et al.*, 2012) e (NAPOLI; NOCERA; ROSSI, 2014) são os que possuem objetivos mais próximo deste trabalho. O primeiro realiza a organização normativa de um sistema multiagente em um estacionamento inteligente público criado para monitorar e selecionar vagas de estacionamentos. O segundo realiza a negociação entre agentes para a alocação de vagas. Estes trabalhos contribuíram para a forma como foi implementados a solução desenvolvida no MAPS-Norms, que relaciona sistemas normativos por meio do modelo organizacional $\mathcal{M}$OISE+ num sistema multiagente. Desenvolvido por meio da integração da ferramenta MOISE as linguagens Jason e CArtAgO, para alocar autonomamente vagas num Estacionamento Inteligente, visando ser uma solução para alocação de vagas e organização entre os agentes motoristas.

3 SISTEMA NORMATIVO MULTIAGENTE

Neste capítulo são apresentados os conceitos de Agentes, Sistemas Multiagentes, Sistemas Normativos e como estes conceitos reunidos formam um Sistema Normativo Multiagente.

3.1 AGENTES

Um agente é uma entidade lógica ou física que está situado em um ambiente e percebe este ambiente e outros agentes a partir de sensores que possui, a partir das informações coletadas pelos sensores, o agente realiza um raciocínio e então, realiza ações no ambiente por meio de seus atuadores. Cada agente possui objetivos próprios e que atua no sistema a partir da sua autonomia. A autonomia é o ponto central do conceito de agentes, pois permite ao agente ter sua individualidade preservada perante os demais agentes.

Podem ser utilizados por um sistema como parte de uma solução que surge quando as ações dos agentes coordenadamente atinjam determinados objetivos, por exemplo, agentes guarda de trânsito que informam os agentes motoristas que sigam por outras vias para evitar congestionamentos. O Paradigma Orientado a Agentes permite representar indivíduos de natureza dinâmica como motoristas de trânsito, os quais possuem rotas particulares e podem alterá-las conforme sua necessidade. No aspecto coletivo os motoristas procuram não causar acidentes, o que implicaria na dificuldade de atingir seu próprio objetivo, que é chegar em determinado ponto da cidade.

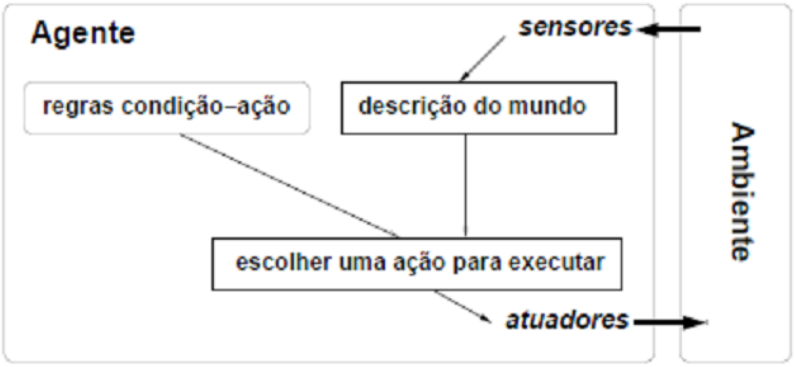
Existem duas arquiteturas clássicas para a construção de um agente: reativa e cognitiva. A primeira se caracteriza por ser uma arquitetura simples e ideal para simular centenas de milhares de agentes interagindo entre si, enquanto a arquitetura cognitiva se caracteriza por ser mais complexa e desta forma demandar mais recursos computacionais para seu funcionamento.

3.1.1 Agentes Reativos

Os agentes reativos possuem o comportamento determinado diretamente por sua percepção do ambiente e de outros agentes. Essas percepções servem de entrada para as regras do tipo condição-ação que determinarão qual tipo de ação o agente irá realizar. Sensores para capturar a percepção do ambiente e de outros agentes, e utiliza os atuadores para executar ações no ambiente e em outros agentes. As regras condição-ação estabelecidas pelo desenvolvedor e possivelmente modificadas em tempo de execução pelo próprio agente em prol de sua própria melhoria. SMA's que possuem agentes deste tipo de arquitetura geralmente utilizam um grande número de agentes para seu funcionamento, a fim de que a partir da interação entre diversos

agentes surja algum tipo de ação inteligente e organizada no sistema.

Figura 2 – Arquitetura resumida de um agente reativo.



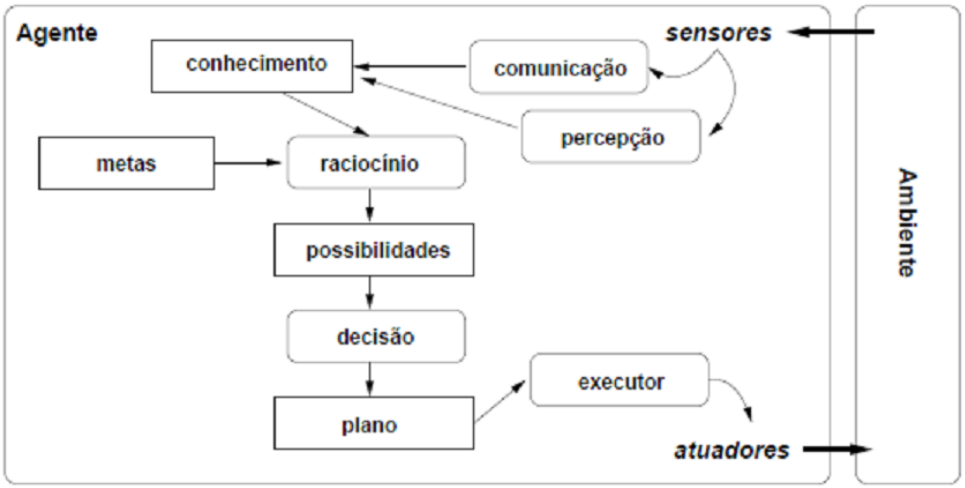
Fonte: (RUSSELL; NORVIG, 2003)

A Figura 2 representa como as regras condição-ação permitem ao agente fazer a conexão entre percepção (sensores) e ação (atuadores). Retângulos denotam o estado interno do processo de decisão do agente e elipses representam as informações suplementares usadas no processo. Este tipo de agente tem como característica ser simples, o que determina também uma limitação, pois toma decisões apenas do seu estado atual, o que pode ser inviável para resolver determinados problemas, já que há casos que exige do agente a habilidade de observar todo o estado do ambiente para conseguir agir adequadamente. Uma simples impossibilidade pode fazer com que o agente tome a decisão inadequada. Algumas arquiteturas para agentes reativos são: ABLE (BIGUS *et al.*, 2002) e Pengi (AGRE; CHAPMAN, 1987).

3.1.2 Agentes Cognitivos

Os agentes cognitivos possuem alguma forma de representar explicitamente o ambiente situado, bem como com os agentes que interagem. Possuem a capacidade de realizar planejamento para futuras ações, baseadas em experiências anteriores. Seu sistema de percepção permite examinar o ambiente em que está situado e o sistema de comunicação permite trocar informações com outros agentes. Utiliza uma linguagem de comunicação, como a *Knowledge Query Management Language* (KQML) (SINGH, 1998). Os agentes cognitivos possuem mecanismo deliberativo, onde raciocinam e decidem quais ações executar, quais planos seguir e quais objetivos alcançar. Usualmente um SMA com agentes cognitivos utilizam poucos agentes, na ordem de algumas dezenas no máximo. Os modelos de organização SMA cognitivo geralmente simulam situações sociológicas, como as interações humanas.

Figura 3 – Arquitetura resumida de um agente cognitivo.



Fonte: Adaptado de (DEMAZEAU; MULLER, 1991)

Na Figura 3 é representado um agente cognitivo baseado no modelo BDI (*belief-desire-intention*). Os sensores recebem informações do ambiente, seja ela por comunicação com outros agentes ou percepção do ambiente. Esta informação é registrada na base de conhecimentos do agente. Esta base permite ao agente raciocinar em como realizar suas ações. Estas ações são influenciadas pelas metas que o agente visa atingir. O agente calcula suas chances de atingir seus objetivos e estabelece sua decisão. E finalmente seu plano será executado por meio dos atuadores no ambiente situado.

Alguns exemplos de arquiteturas cognitivas: PRS (GEORGEFF; LANSKY, 1987) e TouringMachines (FERGUSON, 1992). Nota-se que a arquitetura PRS deu origem ao modelo BDI (WOOLDRIDGE; JENNINGS, 1994).

O modelo de agentes BDI (BRATMAN; ISRAEL; POLLACK, 1988) é inspirado num modelo comportamental humano desenvolvido pelo campo da Filosofia. A sigla BDI ou crença-desejo-intenção, se trata de um modelo baseado em estados mentais humanos. Assim, quando falamos em um sistema crença-desejo-intenção, estamos nos referindo a programas com análogos computacionais. Crenças são informações que o agente possui sobre o ambiente em que está inserido. Estas informações podem ser verdadeiras, desatualizadas ou falsas, cabe ao agente verificar a validade da crença. Desejos são todos os estados possíveis que o agente pode querer atingir por meio de suas ações. O agente escolhe qual dentre as opções melhor favorece sua situação no momento. Intenções são os desejos que o agente decide trabalhar para atingir. Intenções podem ser objetivos delegados por agentes terceiros ou pode ser uma decisão do próprio agente. A princípio depois que um desejo se torna intenção, o agente irá fazer tudo o que estiver ao seu alcance para atingir a intenção e não apenas uma mera tentativa, para isso utilizará todos os plano que possui.

3.2 SISTEMAS MULTIAGENTE

Sistemas Multiagente (SMA) é uma subárea da Inteligência Artificial que utiliza o conceito de agente como elemento central. O agente recebe uma determinada missão e é capaz de cumprir de maneira autônoma e em coordenação com outros agentes (BRIOT, 2001). Em um SMA, existe um conjunto homogêneo ou heterogêneo de agentes que interagem entre si, e cada agente busca alcançar seus objetivos. O agente se encontra em um ambiente virtual ou físico, onde os objetos existentes deste ambiente irão ser percebidos, controlados e até mesmo disputados entre os agentes. Estes recursos podem ser utilizados para a resolução de um objetivo que o agente possua.

Um ambiente em sistema multiagente é considerado um contexto de desenvolvimento, onde os agentes são localizados, bem como a infraestrutura de comunicação, topologia de rede e recursos físicos disponíveis (VIROLI *et al.*, 2007). É nele que os agentes se relacionam e interagem. A forma para alcançar os objetivos pode variar de acordo com as interações com outros agentes e as interações com o ambiente em que os agentes estão inseridos. O agente pode encontrar uma solução melhor para ser utilizada. Cada agente possui uma forma de raciocinar sobre os outros agentes e sobre o ambiente, para decidir qual atitude tomar, influenciada pela arquitetura do agente.

3.2.1 Metodologia Prometheus

Uma forma de construir um sistema multiagente é utilizando uma metodologia que contenha os conceitos utilizados por agentes. A metodologia Prometheus é detalhista e completa para essa tarefa (PADGHAM; WINIKOFF, 2005). A metodologia Prometheus utiliza uma descrição de um SMA para extrair as funcionalidades necessárias. Cada funcionalidade é um item no SMA que realiza uma ação completa (PADGHAM; WINIKOFF, 2002).

As funcionalidades são descrições de como determinada tarefa será realizada, a metodologia não oferece a forma de implementação, cabe ao desenvolvedor interpretar os conceitos que a funcionalidade usa e aplicar em sua linguagem. A partir das funcionalidades foram criados os planos dos agentes e as operações nos artefatos.

3.2.2 Organização em Sistemas Multiagente

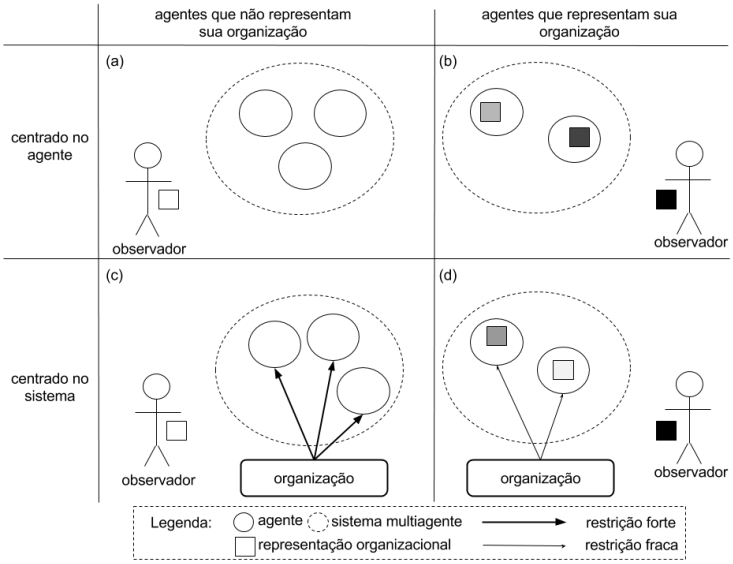
A organização é necessária em um SMA pois estabelece a forma que o sistema irá atingir seus objetivos globais, pois sem uma forma padronizada os agentes poderão realizar ações apenas para benefício próprio, sem aproveitar a coletividade que um conjunto de agentes traba-

lhando unidos possui. A organização de um SMA pode ser classificados segundo a localização da organização e a representação da organização.

A localização da organização pode ser no agente ou no sistema (institucionalizada), *i.e.* se estiver no agente, somente ele tem a organização explícita conhecida, se estiver no sistema um observador pode enxergar como a organização está estabelecida. A representação da organização é institucionalizada se o agente possui uma representação interna sobre a organização que está inserido, ou se ele segue essa organização sem saber que ela existe (LEMAÎTRE; EXCELENTE, 1998).

Segundo estas classificações tem-se na Figura 4 a representação de quatro tipos de organizações. Em cada uma tem-se os seguintes elementos: o observador, o SMA, o agente, a representação de organização e a organização explícita. O observador é um agente humano externo que observa o comportamento do SMA, e mantem para si a representação da organização que observa. A elipse pontilhada representa o sistema multiagente com os agentes situados nele. Os agentes são representados por elipses contínuas menores que podem possuir representação da organização ou não. A organização explícita pode existir ou não e a mesma pode ter restrições altas ou baixas sobre a autonomia dos agentes, ou seja, limitar ou não as ações que os agentes podem tomar.

Figura 4 – Representação de um agente e sua relação com as normas de um SMA



Fonte: Adaptado de (HÜBNER; SICHMAN; BOISSIER, 2007)

O modelo representado de (a) na Figura 4, consiste de agentes que não podem representar a organização explicitamente, os agentes não possuem representação interna da organização e não contam com nenhum suporte externo para manter a organização de suas ações. Um observador externo pode deduzir uma organização implícita do SMA a partir da observação dos agentes e suas decisões, que tendem a obedecer um padrão de execução de tarefas. Um exemplo deste modelo é o conceito de colônia de formigas (DORIGO; BLUM, 2005), onde existe uma estrutura padrão que os agentes seguem, porém não é explícito a sua existência para cada formiga.

O grande número de formigas interagindo podem alcançar o objetivo do SMA. A organização somente existe no momento que o sistema funciona e após uma interação pelo menos, haja visto a impossibilidade de representação da organização nos agentes.

Em (b) da Figura 4, os agentes possuem uma representação interna da organização e diferente entre si. Esta representação explícita no agente é construída de acordo com a interação entre agentes e também com o ambiente. As cores de cada agente são diferentes porque cada agente possui uma experiência particular com o mundo que o cerca. O observador externo tem uma interpretação da organização que pode ou não coincidir com a interpretação de algum agente. Por exemplo, o algoritmo de formação de coalizão (KAHAN; RAPOPORT, 2014). O foco deste algoritmo é ter o funcionamento baseado na existência de conflitos de interesses e a necessidade de solução. Através de barganha e outras formas de negociação os agentes ganham ou perdem. Assim, a organização não é representada de forma explícita, ela é formada por meio de alianças entre os agentes.

No tipo de organização (c) da Figura 4, a organização existe, mas os agentes não raciocinam sobre ela. Eles simplesmente obedecem, uma vez que a organização restringe de forma direta as ações do agente. Por isto, na Figura os agentes não possuem representação explícita para a organização, eles não vêem as regras que estavam seguindo, simplesmente porque não precisam entender, mas apenas seguir. O observador externo possui a representação exata da organização por ele determinada. Um exemplo deste tipo de organização, são os sistemas multiagentes resultantes da metodologia orientada a agente, onde os agentes são códigos gerados baseados nas especificações correspondentes aos conceitos organizacionais, como papéis e responsabilidades (MASSONET; DEVILLE; NÈVE, 2002). Os agentes podem se comunicar e tomar decisões, porém baseados na estrutura determinada pela organização.

No modelo (d) da Figura 4, existe a representação disponível da organização no agente em tempo de execução e os agentes estão habilitados a ler, representar e raciocinar sobre a organização. Cada agente representa a organização de forma diferente pois cada um irá agir de forma particular as normas, uma vez que existe hierarquia entre os agentes, as regras funcionam de forma diferente entre eles. O observador externo possui uma representação da organização a partir da especificação organizacional realizada por ele. Neste tipo de sistema os agentes podem exercer autonomia organizacional, isto é, decidir se seguirão a organização ou não. Um exemplo deste tipo de modelo organizacional é o MOISE (HÜBNER; SICHMAN; BOISSIER, 2007), onde é possível determinados agentes até mesmo alterarem as normas do sistema.

3.3 SISTEMAS NORMATIVOS MULTIAGENTE

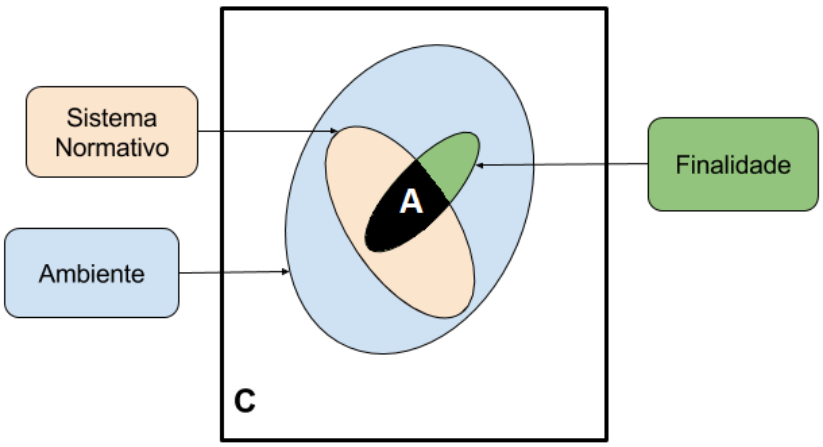
Sistemas Normativos (SN) são estudados principalmente por filósofos e sociólogos, porém o SMA empresta conceitos de Ciências Sociais para serem aplicadas aos agentes de um sistema, como: coordenação, organização, convenção, normas, grau de confiança, dentre outras.

O principal objetivo da criação de um sistema normativo é a implementação de uma organização centrada em regras institucionalizadas. A organização é vista como um conjunto de regras que restringem o comportamento do agente a partir de normas. Quando um agente entra em uma organização estabelecida, ele se torna parte de uma rede de agentes que possuem obrigações, proibições e permissões relacionadas a natureza do agente.

Um Sistema Normativo Multiagente surge da união entre um SMA e um SN. Tem como característica possuir normas opcionais e obrigatórias. As opcionais são aquelas que os agentes podem seguir se quiserem e as normas obrigatórias o agente deve seguir. O sistema normativo especifica como e qual o poder que os agentes têm para modificar as normas explicitadas pelo SN (BOELLA; TORRE; VERHAGEN, 2006).

Na Figura 5 pode-se ver como um SN interage com a autonomia de um SMA. O quadro C representa todos os comportamentos possíveis de um sistema. Comportamento do sistema são suas ações no ambiente dadas as percepções que teve do ambiente. A área Finalidade compreende todas as ações do agente que leva à finalidade do SMA, a finalidade está contida no comportamento. Sendo O o conjunto de percepções possíveis e A o conjunto de ações para o sistema, um comportamento b é um ponto no espaço de comportamentos, representa-se por um mapeamento entre uma sequência de observações do ambiente (denotada por ($o1$,$o2$,...) $\in O$*) e uma ação do sistema (denotada por $\alpha \in A$) onde b: $O^* \rightarrow A$. O conjunto ambiente representa todos os comportamentos possíveis em um dado ambiente. O SN restringe o comportamento dos agentes possibilitando um grupo de ações que estão contidas num conjunto de todas as ações que o agente consegue realizar. Desta forma, o comportamento do agente está mais próximo dos comportamentos desejados para atingir a finalidade do sistema.

Figura 5 – Representação das restrições que a organização causa no agente.



Fonte: Adaptado de (HÜBNER; SICHMAN, 2003).

Na intersecção entre finalidade, ambiente e sistema normativo está a autonomia, que são as ações dos agentes que levam a finalidade desejada pelo objetivo do sistema de modo eficiente, que na Figura está representado pela letra A. Normalmente não se deseja especificar

uma organização onde os comportamentos possíveis são exatamente aqueles que levam a finalidade. Como a finalidade depende do ambiente, uma vez que diferentes ambientes demandam diferentes comportamentos para atingir a finalidade, se o comportamento permitido for igual a finalidade e o ambiente mudar, ou seja, os comportamentos que levam à finalidade passarem a ser outro conjunto de ações possíveis, a organização pode deixar de ser eficaz. Ou seja, se o comportamento permitido é muito pequeno, o SMA pode ter problemas de adaptação a mudanças ambientais, já que a autonomia dos agentes pode ter sido eliminada. Por outro lado, se o conjunto de ações possíveis for muito grande, a organização não é eficiente já que não restringe a autonomia dos agentes. Surge aqui, de forma mais clara, o problema de conciliar a organização com a autonomia dos agentes, isto é, como especificar uma boa organização.

Uma das formas de estabelecer uma boa organização é conhecer o domínio do problema, o objetivo a ser atingido, o tipo de agente implementado e as características do ambiente. As simulações auxiliam no estabelecimento de normas, pois irão mostrar quais regras incentivam ou atrapalham a interação dos agentes a fim de atingir o objetivo. Desta forma, o desenvolvedor pode estabelecer o que é melhor para ser definido inicialmente para o SNM.

4 JACAMO

A plataforma JaCaMo é uma ferramenta para a execução de um SMA. Ela reune três plataformas: Jason, CArtAgO e MOISE, responsáveis pela implementação de um sistema multiagente (SORICI *et al.*, 2012). Para a implementação de agentes utiliza-se a linguagem Jason (BORDINI; HüBNER; WOOLDRIDGE, 2007), para a infraestrutura de interação entre os agentes, utiliza-se Jason (centralizada) ou Jade (distribuída) (BELLIFEMINE; CAIRE; GREENWOOD, 2007), o CArtAgO (RICCI; VIROLI; OMICINI, 2007) é utilizado para a configuração do ambiente e finalmente para a organização normativa do SMA é utilizado a ferramenta MOISE.

Na Figura 6 é representado a abordagem que a plataforma JaCaMo utiliza. Integra-se quatro dimensões: Agentes, Interações, Ambiente e Organização. Todas são independentes entre si e podem ser utilizadas juntas. Em cada quadro possui os principais conceitos de cada dimensão, os quais são implementados em sua respectiva ferramenta e serão explicados em detalhes nas seções: 4.2, 4.3, 4.4 e 4.5.

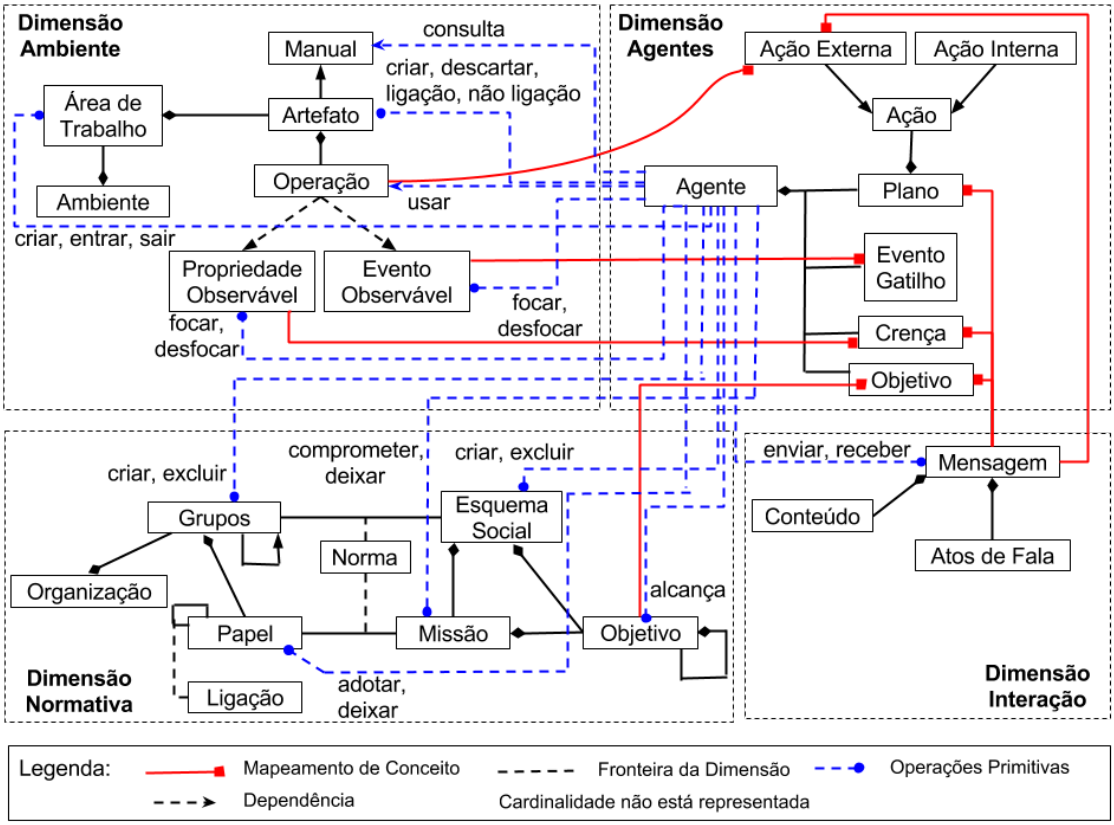
Figura 6 – Abordagem implementada pela plataforma JaCaMo

Fonte: Adaptado de (SICHMAN *et al.*, 2014)

4.1 INTEGRAÇÃO JASON, CARTAGO E MOISE

A plataforma JaCaMo integra três plataformas: Jason, CArtAgO e MOISE. Estas ferramentas possuem independência entre si, o que permite a flexibilidade e eficiência no desenvolvimento do sistema. O metamodelo do JaCaMo está representado na Figura 7, o qual interliga as quatro dimensões: agentes, ambiente, interação e normas.

Figura 7 – Metamodelo do JaCaMo.



Fonte: Adaptado de (SICHMAN *et al.*, 2014)

Os símbolos existentes na legenda da Figura 7 denominam os mecanismos que integram as dimensões. A fronteira da dimensão delimita quais componentes são pertencentes a uma determinada dimensão. Um elemento é um item de um modelo, como os elementos artefato e manual da dimensão do ambiente. O mapeamento dos conceitos são os conceitos que estão em um ambiente e possuem um correspondente em outra dimensão. A ligação de Dependência mostra que um conceito depende de outro, isto é, o elemento que aponta precisa do elemento apontado para realizar alguma atividade. A cardinalidade não está representada no metamodelo, ou seja, não especifica a quantidade de cada item na plataforma. Uma operação primitiva representa uma ação que é permitida a um elemento realizar em outro elemento.

A dimensão dos agentes se relaciona com a dimensão de interação por meio das operações primitivas que um agente pode realizar: enviar e receber uma mensagem. Uma mensagem enviada de um agente para outro é uma ação externa para ambos os agentes que trocam mensagem. Esta mensagem pode conter uma informação, um plano, uma crença ou um objetivo.

O ambiente interage com a dimensão do agentes por meio das ações que o agente realiza a seus componentes. Um agente pode consultar um manual e interagir com um artefato por meio da criação, descarte, se ligar ou desligar de um artefato. Usar as operações existentes nos artefatos, observar eventos e propriedades dos artefatos. Entrar e sair de áreas de trabalho. Uma operação na dimensão do ambiente é interpretada pelo agente como uma ação externa.

Um evento observável pode ser um gatilho para um plano. Uma propriedade observável de um artefato pode ser acrescentado as crenças de um agente.

Na dimensão normativa, um agente pode realizar a criação e exclusão de grupos, se comprometer ou deixar de cumprir uma missão, adotar ou deixar uma missão, criar ou excluir esquemas sociais, adotar ou deixar um papel e alcançar um objetivo. O objetivo da dimensão normativa pode ser acrescida aos objetivos que um agente deseja alcançar a partir da entrada deste agente a um esquema social.

Para implementar a integração das dimensões é preciso definir num arquivo de extensão .JCM os agentes existentes no sistema e a organização que estes agentes terão que seguir, desta forma a integração se torna simples por reunir as três dimensões.

4.2 DIMENSÃO DE AGENTES: JASON

A *Java-based interpreter for a extended version of Agent-Speak*, Jason (BORDINI; HüBNER; WOOLDRIDGE, 2007), é uma linguagem interpretada que surgiu da extensão do AgentSpeak(L), uma linguagem abstrata para agentes inteligentes (RAO, 1996). Jason é uma linguagem orientada a agente que utiliza a arquitetura denominada *Belief*, *Desire*, *and Intention* (BDI), (crença, desejo e intenção). A arquitetura BDI provê mentalidade ao agente por meio de um ciclo de raciocínio baseado nas percepções que o agente recebe, no que ele sabe sobre o ambiente e sobre os objetivos que ele quer atingir no ambiente onde estiver situado.

A linguagem Jason é construída tendo como base a programação lógica. Uma linguagem de programação lógica concilia o uso da lógica em uma linguagem declarativa que representa o conhecimento e a representação procedimental do conhecimento, por exemplo o Prolog. Os principais construtores da linguagem Jason são: crenças, objetivos e planos. As crenças, representam informações na forma de estado mental, informações sobre o ambiente, demais agentes e sobre si - o termo crença é usado para enfatizar que o agente pode ter uma informação correta ou incorreta e/ou incompleta sobre o ambiente e outros agentes. Os objetivos representam o estado das coisas que os agentes desejam alcançar e é necessário ao agente, e os planos indicam a forma como as ações de um agente podem alcançar determinado objetivo ou reagir as percepções captadas do ambiente e de outros agentes.

Para esclarecer os conceitos da linguagem Jason utilizados pelos agentes e apresentar como o código é implementado será apresentado um exemplo. Os códigos mostrados nesta seção é de autoria própria, o primeiro exemplo é o código 1, que apresenta a implementação de um agente. O agente tem as crenças semDinheiro (linha 1), *temTrabalho* (linha 2) e *bomHumor* (linha 3).

O agente tem o desejo trabalhar (linha 5), esta intenção aciona algum plano trabalhar que está implementado no agente. O agente tem dois planos trabalhar, o primeiro tem como

condição a crença *bomHumor* ter o valor *true* e existir a crença feliz (linha 7). O agente realiza o plano irAoTrabalho (linha 8).

Se a crença *bomHumor* ter valor falso, ou a crença feliz não existir, as condições não são atingidas e o agente verifica o próximo plano para tentar realizar. O próximo plano tem como condição a crença *bomHumor* ter o valor *false* guardado (10). Se isso for verdade, o agente adiciona a sua base de crenças a crença chamada *percebeMauHumor* (linha 11).

Quando uma crença é adicionada a base de dados, o agente pode ter um plano para perceber este evento e reagir à ele. Neste agente existe um plano para ser realizado caso uma crença seja adicionada (linha 13). O plano tem como condição a existência da crença dinheiro com o valor guardado na variável *Var_D* com valor igual a zero. Neste plano a crença *bomHumor* é atualizada com valor *true* (linha 14) e realiza o plano trabalhar (linha 15).

```
1  dinheiro(0).
2  temTrabalho.
3  bomHumor(false).
4
5  !trabalhar.
6
7  +!trabalhar : bomHumor(true) & feliz
8    <-  !irAoTrabalho.
9
10 -!trabalhar : bomHumor(false)
11   <-  +percebeMauHumor.
12
13 +percebeMauHumor : dinheiro(Var_D) & Var_D == 0
14   <-  -+bomHumor(true);
15       !trabalhar.
```

Código 1 – Codificação do agente que decide se irá trabalhar.

Um agente construído em Jason possui a capacidade de representar fatos, sobre o ambiente, outros agentes e si em forma de crença. Uma crença em Jason é representada por uma proposição lógica, essa proposição começa necessariamente com uma letra minúscula, se uma crença armazenar um valor, este valor é colocado entre parênteses no final do nome da crença. Como no exemplo do código 2, onde o tem as crenças dinheiro, *temTrabalho* e *bomHumor*. O código 2 representa as proposições lógicas existentes.

```
dinheiro(0).
temTrabalho.
bomHumor(false).
```

Código 2 – Crença e regras iniciais.

A Figura 3 que implementa o único objetivo deste agente simples. Um objetivo inicial possui antes do nome o caractere exclamação "!". Este objetivo determina que a primeira coisa que o agente tentará realizar é um plano com este nome.

```
!trabalhar.
```

Código 3 – Objetivos Iniciais.

O código 4 codifica os planos que o agente irá seguir cada vez que um plano for adicionado. Quando um plano é adicionado ao agente usa-se o caractere soma (+) antes de seu nome. Se um plano precisa ser removido pois o agente não quer manter este plano, utiliza-se o caractere subtração (-). O evento gatilho de um plano está depois do caractere (:) e antes dos caracteres maior e traço juntos (<-). São preposições lógicas que indicam ao as condições necessárias para um plano ocorrer. O caractere e comercial (&) é utilizado como conector lógico *E*, indicando que duas preposições precisam ser verdadeiras para o contexto ser verdade. As ações de um plano são separadas pelo caractere ponto e vírgula (;) e o final do plano possui ponto final (.).

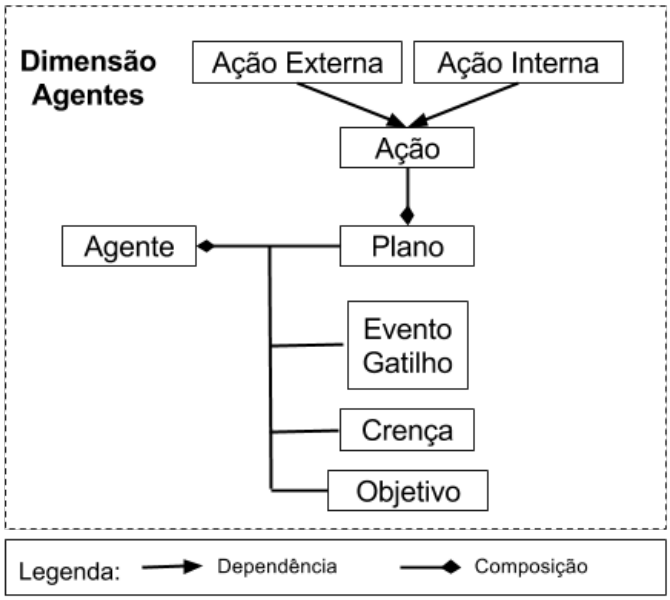
```
1 +!trabalhar : bomHumor(true) & feliz
2   <-  !irAoTrabalho.
3
4 -!trabalhar : bomHumor(false)
5   <-  +percebeMauHumor.
6
7 +percebeMauHumor : dinheiro(Var_D) & Var_D == 0
8   <-  -+bomHumor(true);
9       !trabalhar.
```

Código 4 – Codificação dos planos do agente.

A Figura 8 representa o metamodelo da linguagem Jason. Um agente é composto basicamente por crenças, objetivos, planos e eventos gatilhos. Uma crença é o que o agente sabe sobre o estado atual do mundo, pelo menos para ele é verdade. Um objetivo é o estado das coisas que o agente quer atingir. Um evento gatilho, também chamado de contexto, é o que determina quais as condições para que seu plano seja acionado. Um plano é a forma como o agente conseguirá cumprir seu objetivo.

Um plano é composto por uma ou mais ações. Uma ação pode ser de dois tipos: interna ou externa. Uma ação interna é aquela que o agente causa para si, que modifica algum estado mental próprio. Uma ação externa é o que o agente realiza para outro agente ou para o ambiente.

Figura 8 – Metamodelo da plataforma Jason.



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

4.3 DIMENSÃO DE INTERAÇÃO: PERFORMATIVAS

A comunicação entre agentes ocorre baseada na linguagem Jason que por sua vez é uma extensão de *AgentSpeak*. A linguagem Jason se baseia na teoria de atos de fala, em particular nos trabalhos de Austin (AUSTIN, 1962) e Searle (SEARLE, 1969). A teoria da fala é baseada no princípio que a linguagem é uma ação que tem como fim modificar os estados das coisas. Um agente racional realiza uma elocução na tentativa de mudar o estado do mundo, por exemplo: um general grita: "Atacar!". Ele espera que com esta fala seus soldados irão se mover contra o exército inimigo.

A interação entre agentes na linguagem Jason é baseada na linguagem KQML (FININ *et al.*, 1994), a qual tem como premissa o uso de atos de fala. Os atos de fala são mensagens enviadas de um agente para outro que realizará algo previsto por um padrão determinado. Cada ato de fala possui uma performativa e um conteúdo. Diferentemente da linguagem humana, onde existem situações que podem ocorrer a ambiguidade, para a comunicação entre agentes é necessário explicitar a força ilocucionária com o objetivo de simplificar a comunicação e evitar a ambiguidade. A força ilocucionária específica o contexto que o conteúdo da mensagem possui. Os tipos de forças ilocucionária são chamadas no contexto de agente de performativas.

O mesmo conteúdo com performativas diferentes formam mensagem diferentes, estas mensagens têm significados específicos, apesar de ter o mesmo conteúdo. A seguir será exemplificado o uso de performativas em geral, por meio de três tipos de performativas com o conteúdo passado pela mensagem e o resultado que é gerado no ato de fala. O primeiro ato de fala é um pedido relacionado ao café estar feito. O segundo ato de fala é a informação passada de um

agente para outro. O último ato de fala estabelece que o agente emissor da mensagem requisita uma resposta relacionado ao conteúdo passado.

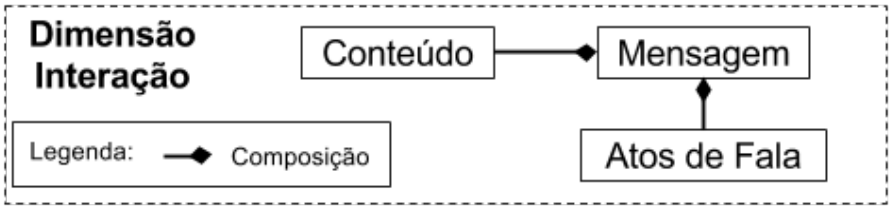
Figura 9 – Exemplo de performativas.

Performativa	Conteúdo	Ato de Fala
pedir	O café está feito.	Por favor, faça café.
informar	O café está feito.	O café está feito!
investigar	O café está feito.	O café está feito?

Fonte: Adaptado de (SICHMAN *et al.*, 2014)

O metamodelo utilizado pela plataforma JaCaMo para representar a interação entre agentes está representado na Figura 10, onde a mensagem de um agente é enviada para outro é composta por um conteúdo e um ato de fala, o conteúdo é o que o agente envia para outro agente e o ato de fala é o tipo de mensagem que está sendo enviado.

Figura 10 – Metamodelo de Interação.



Fonte: Adaptado de (SICHMAN *et al.*, 2014)

A seguir será detalhado as performativas existentes na linguagem Jason. O termo *AgO* representa o agente de origem da mensagem enviada e o termo *AgD* representa o agente destino, o qual recebe a mensagem do agente *AgO*.

- *broadcast*: *AgO* envia uma mensagem para todos os agentes informando sobre algo que acredita;
- *tell*: *AgO* pretende que *AgD* acredite no que *Ago* acredita, passa no conteúdo da mensagem;
- *untell*: *AgO* pretende que *AgD* não acredite no que *Ago* também não acredita, passado como conteúdo da mensagem;
- *achieve*: *AgO* pede a *AgD* para tentar alcançar um estado das coisas em que o literal conteúdo da mensagem seja verdadeiro, isto é, *AgO* está delegando um objetivo a *AgD*;
- *unachieve*: *AgO* pede a *AgD* que abandone o objetivo de alcançar um estado das coisas onde o conteúdo da mensagem é verdadeira, desta forma o agente *AgD* abandona um objetivo;
- *Askone*: *AgO* quer saber se o conteúdo da mensagem é verdadeiro para *AgD*, isto é, se há uma resposta que faz com que o conteúdo gere uma consequência lógica na base de crença de *AgD*;

- *askAll*: *AgO* quer a resposta de todos os agentes existentes para a questão que envia na mensagem;
- *tellHow*: *AgO* informa a *AgD* um plano, desta forma o agente *AgD* saberá realizar um plano;
- *untellHow*: *AgO* requisita que *AgD* ignore um determinado plano, ou seja, remove o plano da biblioteca de planos de *AgD*;
- *askHow*: *AgO* quer que todos os planos de *AgD* que são relevantes para o evento gatilho no conteúdo da mensagem seja satisfeito.

No código 5 é apresentado um agente que avisa a outro agente que choveu e conta quantas vezes avisou. O agente inicializa com a crença *numAvisos* com o valor zero (linha 1). O agente inicializa com o desejo de avisar (linha 2). Existem dois planos para realizar o objetivo de avisar, um para o caso se vai chover (linha 3) e outro para o caso de não chover (linha 10).

O primeiro plano tem como condição o agente ter a crença chuva em sua base de crenças (linha 3). Neste plano o agente recupera a crença *numAvisos* na variável *Var_NA* (linha 4). Cria-se uma variável nova *Var_NNA* que possuí o valor da variável *Var_NA* incrementado em um (linha 5). Atualiza a crença *numAvisos* com o novo valor do número de avisos efetuados (linha 6). Envia ao agenteCarlos a mensagem do tipo *tell* com a crença chuva (linha 7) e realiza recursivamente o plano avisa (linha 8).

Se o agente não tiver a crença chuva, o segundo plano é realizado, o segundo o plano tem a condição *true*. Quando este plano é testado com esta condição o mesmo sempre será feito, pois a condição guarda é sempre verdadeira, se um plao anterior for acionado este plano não será acionado. Assim, se o primeiro plano não ter sido realizado, o segundo plano será.

No segundo plano o agente realiza a ação interna *.wait* (linha 11). Nesta ação, o agente espera o tempo determinado em milissegundo. No código o agente espera cinco segundos e realiza o plano avisa novamente. Assim, não importa qual seja acionado, o agente sempre verificará se está chovendo para avisar o outro agente chamado agenteCarlos.

```
1 numAvisos(0).
2 !avisa.
3 +!avisa : chuva
```

```
4     <-  ?numAvisos(Var_NA);
5         Var_NNA = Var_NA + 1;
6         -+numAvisos(Var_NNA);
7         .send(agenteCarlos, tell, chuva);
8         !avisa.
9
10  -!avisa : true
11    <-  .wait(5000);
12        !avisa.
```

Código 5 – Codificação de um desejo e os planos existentes para sua execução.

4.4 DIMENSÃO DO AMBIENTE: CARTAGO

O CArtAgO é uma linguagem orientada a ambiente que codifica os itens chamados de artefatos e ambientes em sistemas multiagentes. O CArtAgO não introduz nenhum tipo de modelo ou arquitetura para agentes ou sociedades de agentes. A linguagem constrói uma plataforma para que os agentes existentes possam ser integrados e consigam interagir adequadamente, inclusive agentes de arquitetura diferentes. A plataforma utiliza artefatos como elemento central de seu funcionamento, um artefato é uma abstração para modelar e criar: objetos, recursos e ferramentas.

Permite a abstração dos artefatos, úteis para promover a autonomia necessária para um grupo de agentes poderem construir, compartilhar e cooperar entre si objetos que farão parte de suas soluções. Os artefatos podem ser utilizados em tempo de execução, e também como suporte para trabalhos cooperativos, facilitando a interação entre agentes (RICCI; VIROLI; OMICINI, 2007). Os artefatos podem ser recursos alvos para uso. Neste caso existe a situação de concorrência para a utilização de recursos em determinado ambiente, como itens escassos em um cenário.

Existe o conceito de corpo de agente, o qual permite a conexão entre a mente do agente e o ambiente de desenvolvimento do CArtAgO. Para cada agente dentro do ambiente é criado um corpo de agente. Este item contém um conjunto dinâmico de sensores que coletam estímulos provenientes do ambiente e os atuadores para realizar ações no ambiente. O agente utiliza este corpo para interagir com artefatos do ambiente, para realizar a construção, seleção e uso. E também, perceber eventos gerados por tais artefatos.

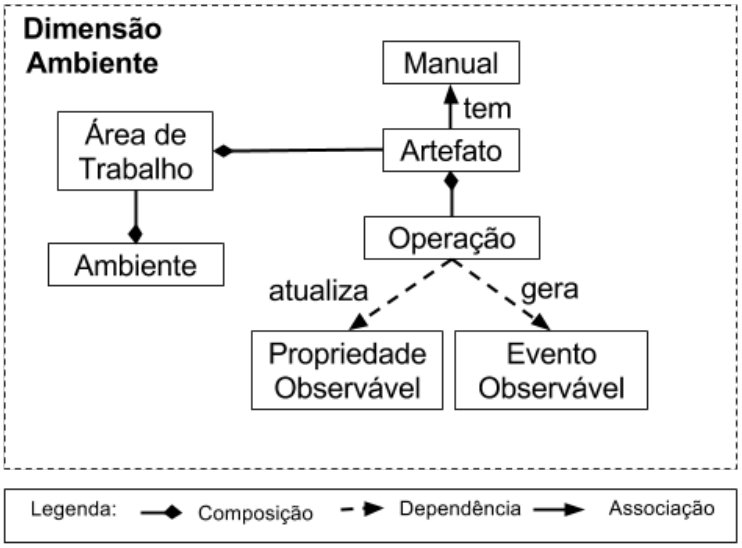
Uma Área de Trabalho (*workspace*) é uma área física ou lógica em que artefatos podem ser manipulados por agentes e os agentes podem criar, alterar, entrar e sair delas. As operações de um artefato são realizadas uma de cada vez, a mesma operação nunca é realizadas simulta-

neamente por dois ou mais agentes.

Analogamente aos artefatos em nossa sociedade, o modelo básico que caracteriza a interação entre agentes e artefatos são as noções de usabilidade e observação. Os agentes podem usar um artefato pela execução de operações listadas na interface de uso. Uma operação é caracterizada por um nome e um conjunto de parâmetros tipificados. A execução de uma operação pode causar uma atualização no estado interno de um artefato, e possivelmente a geração de múltiplos eventos observáveis, incluindo erros, que podem ser coletados por sensores de um agente.

O metamodelo do CArtAgO está representado pela Figura 11. Nesta dimensão é configurado o ambiente onde os agentes podem entrar e sair, e como os objetos de ambiente estão configurados. Um artefato tem um manual para que possa ser consultado e entendido por outros agentes. Também é composto por operações, as quais geram eventos observáveis e podem atualizar propriedades observáveis que já existiam. Uma Área de Trabalho é composta por vários artefatos e um ambiente é composto por várias Área de Trabalho.

Figura 11 – Metamodelo do CArtAgO.



Fonte: Adaptado de (SICHMAN *et al.*, 2014)

A integração do Jason e Cartago é realizada para que os agentes consigam interagir com os artefatos na área de trabalho. A seguir são listadas algumas operações que um agente pode realizar num artefato.

- *makeArtifact* - criar um artefato.
- *lookupArtifact* - descobrir o identificador de um artefato.
- *destroy* - destruir um artefato.
- *focus* - focalizar num artefato.

- *stopFocus* - interromper sua atenção num artefato.
- *createWorkspace* - criar uma área de trabalho.
- *joinWorkspace* - entrar numa área de trabalho.
- *quitWorkspace* - sair de uma área de trabalho.

Para exemplificar a integração é utilizado um exemplo em que um agente chamada mãe, irá criar uma área de trabalho chamado quarto e criar um artefato chamado despertador. Vai mandar o filho entrar no quarto e acordar quando o despertador tocar na hora que ela configurou.

O código 6 apresenta o código do agente mãe. A mãe tem o desejo criarQuarto (linha 1) que aciona o plano criarQuarto (linha 3). O agente cria a área de trabalho chamado quarto (linha 4) e entra nele (linha 5). Cria o artefato despertador com o tempo configurado em seis horas (linha 6). Manda o agente filho realizar o plano entrarQuarto, informando os parâmetros com o nome da área de trabalho e do artefato (linha 7). Em seguida, a mãe sai da área de trabalho que ela criou (linha 8).

```
1 !criarQuarto.
2
3 +!criarQuarto : true
4   <-  createWorkspace("quarto");
5       joinWorkspace("quarto", _);
6       makeArtifact("despertador", "quarto.Despertador",
           ["06:00:00"], IdClock);
7       .send(filho, achieve, entrarQuarto("quarto",
           "despertador")).
8       quitWorkspace.
```

Código 6 – Codificação da agente mãe.

O código 7 apresentado o código do agente filho. O filho realiza o plano que mãe mandou realizar (linha 1). Este plano tem as variáveis *Var_Quarto* e *Var_Des* com o nome da área de trabalho quarto e do artefato despertador. O filho entra na área de trabalho quarto (linha 2), recupera o identificador do artefato despertador (linha 3) e focaliza neste artefato (linha 4). O agente filho focaliza neste artefato para perceber quando o artefato enviar algum sinal para ele.

Quando o artefato despertador tocar, enviará um sinal chamado trimTrimTrim, este sinal é percebido pelo agente filho que está focalizado no despertador. Se o agente tiver a crença dormirCedo, ele acorda e avisa a sua mãe. Senão, apesar de estar focalizado no despertador, por não ter outro plano sem a crença dormirCedo, o agente filho não faz nada.

```
1 +!entrarQuarto(Var_Quarto, Var_Des)
2   <-  joinWorkspace(Var_Quarto, _);
```

```
3       lookupArtifact(Var_Des, IdRel);
4       focus(IdRel).
5
6  +trimTrimTrim : dormirCedo
7    <-   .print("Acordei");
8         .send(mae, tell, acordei).
```

Código 7 – Codificação do agente filho.

O artefato é um despertador inicializado por um agente que determina um tempo que avisará aos agentes que estão com foco nele que o tempo foi atingido, o código deste artefato é apresentado em 8.

O artefato é inicializado recebendo como parâmetro o tempo para despertar (linha 1). Este tempo é atribuído a propriedade tempoDespertar (linha 2) e realiza a operação interna verificarTempo (linha 3). A operação interna verificarTempo espera um segundo (linha 8), converte o tempo configurado para o tipo inteiro (linhas 9 - 12), coloca o tempo atual na variável agora (linhas 13 - 14). Verifica se o tempo foi atingido (linha 15), se foi atingido realiza o sinal trimTrimTrim (linha 16). Senão realiza a operação verificarTempo novamente.

```
1  public class Despertador extends Artifact {
2    void init(String tempoDespertar) {
3      defineObsProperty("tempoDespertar", tempoDespertar);
4      execInternalOp("verificarTempo");
5    }
6
7    @INTERNAL_OPERATION void verificarTempo() throws
        OperationException{
8      await_time(1000);
9      String tempo =
          getObsProperty("tempoDespertar").stringValue();
10     int hora = Integer.parseInt(tempo.substring(0,2));
11     int min = Integer.parseInt(tempo.substring(3,5));
12     int seg = Integer.parseInt(tempo.substring(6,8));
13     Date tempoAtual = new Date();
14     int agora = (tempoAtual.getHours() * 3600) +
15                 (tempoAtual.getMinutes() * 60) +
                      tempoAtual.getSeconds();
16     if(agora == ((hora * 3600) + (min * 60) + seg)){
17       signal("trimTrimTrim");
18     }
```

```
19      else{
20        execInternalOp("verificarTempo");
21      }
22    }
23  }
```

Código 8 – Codificação do artefato despertador.

4.5 DIMENSÃO NORMATIVA: MOISE+

O $\mathcal{M}$OISE+ é o modelo estendido de uma versão anterior chamado MOISE (*Model of Organization for multI-agent SystEms*) (HANNOUN *et al.*, 2000). O modelo $\mathcal{M}$OISE+ é implementado na plataforma chamado MOISE. A visão da organização está centrada no sistema, ou seja, com normas institucionalizadas que são externas aos agentes e podem ser consultadas por eles. Os agentes têm a própria representação das normas. Este modelo tem a capacidade de reorganização, onde as ações do SMA modificam o sistema e o mesmo pode se reorganizar a fim de manter seu objetivo. Apresenta dimensões explicitamente distintas: estrutural, funcional e normativa. Onde as dimensões estruturais e funcionais podem ser especificadas independentemente e são integradas pela dimensão normativa (FERBER; GUTKNECHT; MICHEL, 2003).

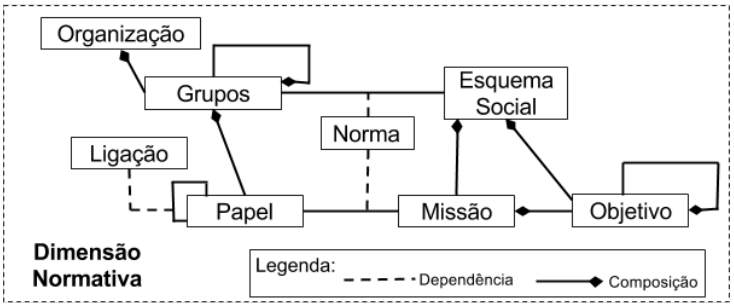
A dimensão estrutural é responsável pela forma como o papel de um agente está situado na hierarquia da organização e quais suas restrições comportamentais existentes aos agentes. A dimensão funcional especifica como as ações dos agentes são realizadas no SMA, qual a ordem de execução e quem é responsável por determinada tarefa. A dimensão normativa estabelece quais são as tarefas permitidas e obrigatórias para cada agente de acordo com o papel que o agente possui perante o SMA. Para apresentar os conceitos envolvidos no $\mathcal{M}$OISE+, será utilizado um exemplo na organização de uma equipe de pesquisadores que vão escrever um artigo, este exemplo é adaptado de (HÜBNER *et al.*, 2010). Cada dimensão possui uma especificação correspondente e que forma a Especificação Organizacional (EO). Assim que criada, a EO inicia e funciona por eventos de agentes, como sua entrada e saída da EO, criação de grupos, adoção de papéis, comprometimento para uma missão, dentre outros.

Para codificar o modelo realizado em $\mathcal{M}$OISE+, é necessário especificar as três dimensões em um arquivo XML. O arquivo XML será interpretado pela plataforma MOISE e a partir desta leitura especificará como a organização será aplicada ao SMA. Para apresentar os conceitos envolvidos no $\mathcal{M}$OISE+ e sua codificação na plataforma MOISE, será utilizado um exemplo de especificação de organização.

A Figura 12 representa o metamodelo do $\mathcal{M}$OISE+. A plataforma MOISE utiliza os conceitos da linguagem $\mathcal{M}$OISE+ para normatizar um sistema multiagente. A organização é o

elemento que engloba todos os itens do MOISE para determinado objetivo. Cada elemento será explicado na continuação desta seção.

Figura 12 – Metamodelo do MOISE.



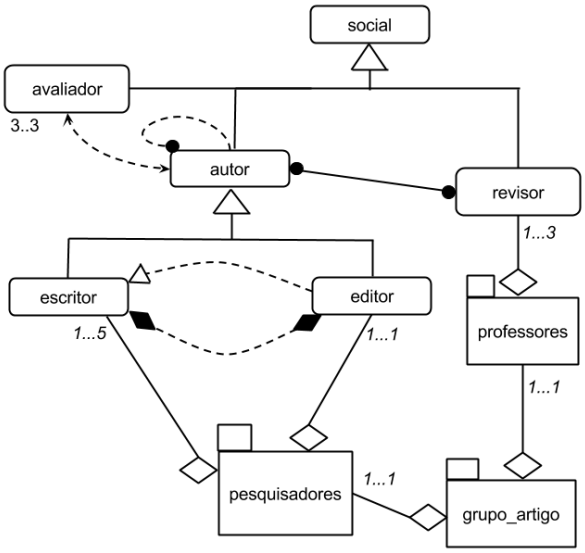
Fonte: Adaptado de (HÜBNER *et al.*, 2010)

4.5.1 Dimensão Estrutural

A dimensão estrutural é responsável por definir a estrutura da organização: os papéis, a relação entre eles e os grupos de papéis. A Especificação Estrutural (EE) é construída em três níveis: individual, social e coletiva. A primeira específica o comportamento que um agente é responsável ao assumir e as restrições pertencentes a ele. O nível social apresenta o comportamento do agente em relação aos outros agentes do sistema. O último implementa quais agentes possuem características semelhantes e assim são agrupados para organizar a estrutura.

Nesta seção serão apresentados os conceitos do modelo formal e exemplificados a partir do exemplo de pesquisadores que intencionam publicar um artigo. A Figura 13 representa a especificação da dimensão estrutural.

Figura 13 – Especificação Estrutural.



Fonte: Autoria Própria.

A implementação da EE fica entre as *tags* das linhas 1 e 3 do código 9, o restante do código referente a estrutura é codificado entre estas duas marcações:

```
1 <structural-specification>
2     ...
3 </structural-specification>
```

Código 9 – Início e fim da especificação estrutural.

A seguir são definidos formalmente papéis, grupos e ligações que constituem os componentes da organização estrutural e individual do modelo $\mathcal{M}$OISE+.

4.5.1.1 Nível individual

O nível individual é formado por papéis que os agentes representam numa organização. Um papel é um conjunto de restrições que um agente deve seguir ao aceitar participar de uma organização. As restrições são definidas de duas formas: ações possíveis ao agente realizar e qual a relação que este papel possui para os demais papéis. A relação que um agente tem com outros agentes está especificada no nível estrutural coletivo. Uma das propriedades que existem no nível individual é a relação hierárquica entre papéis, semelhante ao conceito de herança da Orientação Objeto (OO).

Se um papel p' herda p com $p \neq p'$. O papel p' recebe as mesmas propriedades de p, e p' é uma especialização de p. Por exemplo, num grupo de pesquisa todo escritor tem as todas as propriedades do papel autor (pEscritor $\sqsubset$ pAutor), porém um autor pode não ter propriedades de um escritor.

Um papel pode ser especializado em mais de um papel, *i.e.* receber propriedades de vários papéis. O conjunto de todos os papéis é denotado por $\mathcal{R}ss$. Outra inspiração em OO, é a definição de um *papel abstrato* que é um papel que não pode ser atuado por nenhum agente. É utilizado apenas com o objetivo de especificar a estrutura dos papéis, este tipo não tem fim prático.

(i) Papel Abstrato: existe apenas um papel abstrato, também chamado de social. Especificado no modelo estrutural da Figura 13. Nenhum agente pode ter este papel em particular, é um papel geral que todos os papéis existentes são especializações dele. A codificação no MOISE deste tipo de papel não é necessária, este papel está implícito na ferramenta.

(ii) Papel: existem um ou mais papéis na especificação. No modelo é representado por uma elipse. Temos o exemplo do papel autor que representa um autor do artigo. Corresponde ao agente que irá escrever o artigo. A codificação do papel autor está representado no código 10:

```
<role-definitions>
  <role id="autor"/>
</role-definitions>
```

Código 10 – Codificação do papel autor.

(iii) Grupo: é um conjunto de papéis que possuem um objetivo em comum. Cada papel de um grupo possui um conjunto de tarefas particulares que, quando realizadas por todos os agentes do grupo, atingem o objetivo principal. Por exemplo um departamento de uma faculdade, *i.e.* professores com habilidades e conhecimentos semelhantes mas com sua individualidade resguardada por sua área de pesquisa.

Um grupo é representado no modelo por um retângulo com um outro retângulo menor na parte superior esquerda. Existem na Figura 13 três grupos que os agentes podem participar: Pesquisador, Professores e GrupoArtigo. No código 11 temos a codificação do grupo GrupoArtigo, correspondente ao maior grupo da estrutura. Este grupo contém todos os papéis existentes do SMA. Um grupo possui agentes e especifica sua cardinalidade, as ligações entre os papéis e compatibilidade entre papéis, os quais serão detalhados a diante. A codificação de um grupo no MOISE está demonstrada no código 11, as reticências indicam a implementação de cada item.

```
<group-specification id="GrupoArtigo">
  <roles>
    ...
  </roles>
  <links>
    ...
  </links>
  <formation-constraints>
    ...
  </formation-constraints>
</group-specification>
```

Código 11 – Codificação do grupo GrupoArtigo.

4.5.1.2 Nível social

Enquanto a relação de herança não tem nenhum efeito direto sobre o comportamento do agente, existem outros relacionamentos entre papéis que restringem diretamente o comportamento entre os agentes. Estas relações são chamadas ligações e são representados pelo predicado ligação (po, pd, t), onde po é a origem da ligação, pd é o destino da ligação e $t \in con, com, aut$ é o tipo de ligação. O papel origem po estabelece que a ligação parte deste agente, ou seja, a ação é executada por ele. O papel destino pd aponta qual agente sofre a ligação do agente origem. O tipo de ligação estabelece o que é permitido ao agente origem realizar ao interagir com o agente destino.

Os tipos de ligação são: conhecimento, comunicação e autoridade. O tipo de ligação conhecimento (con) permite ao papel de origem po ter a representação mental do agente de papel pd, desta forma sabe que o agente pd existe no SMA. O segundo tipo, comunicação (t = com), o po é permitido se comunicar com pd, desta forma pode transmitir algum tipo de informação para ele. Na ligação de autoridade (t = aut), o agente po é permitido ter autoridade sobre o agente pd, para poder controlá-lo por meio de uma ordem.

Uma ligação de autoridade implica na existência da ligação de comunicação conforme a equação 4.1. Consequentemente implica na ligação de conhecimento, no código 4.2.

$$ligacao(po, pd, aut) \rightarrow ligacao(po, pd, com) \tag{4.1}$$

$$ligacao(po, pd, com) \rightarrow ligacao(po, pd, con) \tag{4.2}$$

Em relação a herança, a ligação de um papel para outro, é herdada por suas especializações. Conforme as seguintes regras:

$$((ligacao(po, pd, t) \wedge po) \sqsubset po') \rightarrow ligacao(po', pd, t) \tag{4.3}$$

$$((ligacao(po, pd, t) \wedge pd) \sqsubset pd') \rightarrow ligacao(po, pd', t) \tag{4.4}$$

Por exemplo, se o editor tem autoridade sobre o papel de escritor, a partir da ligação (pEditor, pEscritor, aut) e o escritor tem uma especialização chamada revisor linguístico (pEscritor $\sqsubset$ $p$revisorLinguistico), pela equação 4.4 temos que o editor tem autoridade sobre o escritor e também ao revisor linguístico. Além disso, um editor tem a permissão para se comunicar com o escritor, devido a equação 4.1 e é permitido conhecer o escritor.

A seguir serão apresentados a codificação dos oito tipo de ligações existentes entre papéis de agentes:

(i) Herança: representa o relacionamento de herança de um papel para outro. No modelo da Figura 13 a origem da seta está o papel especialização, o papel destino está o papel que foi especializado. Temos por exemplo, os papéis editor, escritor que são especializações do papel autor, ou seja, possuem as mesmas características que o papel *autor* com características próprias adicionadas em cada especialização. A implementação de herança na ferramenta MOISE do papel escritor herdando as características do autor é:

```
1 <role-definitions>
2   <role id="escritor"> <extends role="autor"/>
3 </role-definitions>
```

Código 12 – Codificação do papel escritor.

(ii) Compatibilidade: representa a compatibilidade entre dois papéis. Significa que um agente pode assumir outro papel sem precisar perder características anteriores e adquirir novas para isso. Na especificação do modelo sua representação é um losango pigntado de preto, sinalizando a que papel é compatível com uma ligação com o papel que pode assumir outro papel. No modelo anterior, todos os escritores têm a possibilidade de serem do tipo editor, desta forma qualquer escritor tem a permissão para se tornar editor. O código 13 é implementado a compatibilidade entre editor e escritor na ferramenta MOISE.

```
1 <formation-constraints>
2   <compatibility from="editor" to="escritor" />
3 </formation-constraints>
```

Código 13 – Codificação da ligação entre os papéis editor e escritor.

(iii) Composição: mostra quais papéis pertencem aos grupos do modelo e a cardinalidade do papel no grupo. A cardinalidade indica quantos indivíduos podem existir de um mesmo papel. No código 14 temos os papéis escritor e editor que compõem o grupo pesquisadores, podem existir de um até cinco escritores e apenas um editor.

```
1 <roles>
2   <role id="editor" min="1" max="1"/>
3   <role id="escritor" min="1" max="5"/>
4 </roles>
```

Código 14 – Codificação da composição do grupo pesquisadores.

(iv) Conhecimento: este tipo de ligação representa as ligações de conhecimento que um agente tem de outro. No modelo é indicado como uma seta com a ponta na forma de um losango

para o papel destino e a origem da seta está no papel origem da ligação. Representa que um agente pode saber se determinado agente existe ou não, sem ter havido contato entre ambos previamente. Por exemplo, o escritor sabe que existe um avaliador, porém o escritor não pode se comunicar com ele e muito menos ter autoridade sobre ele, isto é, somente saber que o mesmo existe. O código 15 implementa a ligação de conhecimento entre escritor e avaliador.

```
1 <links>
2   <link from="escritor" to="avaliador"
      type="acquaintance"/>
3 </links>
```

Código 15 – Codificação da ligação entre escritores e avaliadores.

(v) Comunicação: é a ligação que indica dois papéis com a capacidade de trocar mensagens, efetuando a comunicação entre eles. No modelo é identificado como uma ligação com um círculo pintado de preto no papel que recebe a mensagem enviada e a origem da seta está no papel que envia a mensagem. A comunicação pode ocorrer entre os agentes do papel *autor* e papéis que são especializações do mesmo. No modelo 13 apresenta a ligação de comunicação tendo como origem e destino o mesmo papel. A ligação de origem e destino o papel autor é representado no código 16.

```
1 <links>
2   <link from="autor" to="autor" type="communication"/>
3 </links>
```

Código 16 – Codificação da ligação entre os autores.

(vi) Autoridade: determina que um papel interfere nas atitudes de outro. Assim, um papel pode delegar a outro agente uma meta ou objetivo que precisa cumprir e não tem disponibilidade ou meios para realizá-la no momento. No modelo é indicado como uma seta preenchida o papel que recebe as ordens, e a origem da seta o papel que envia a ordem. No exemplo o editor que tem permissão de interferir nos escritores, o editor indica ao escritor a melhor forma de escrita ou ordem das ideias. No código 17 é apresentado a ligação de autoridade de editor para escritor.

```
1 <links>
2   <link from="editor" to="escritor" type="authority"/>
3 </links>
```

Código 17 – Codificação da ligação entre editores e escritores.

(vii) Intragrupo: indica que a ligação ocorre apenas entre papéis do mesmo grupo. No modelo é especificado com a linha de ligação pontilhada. O código 18 implementa uma ligação entre papéis que estão no mesmo grupo, na linha 2 especifica o escopo da autoridade existente entre editor e escritor.

```
1 <link from="editor" to="escritor" type="authority"
2       scope="intra-group"/>
```

Código 18 – Codificação da composição do grupo pesquisadores.

(viii) Intergrupo: mostra que a ligação ocorre apenas entre papéis de grupos diferentes. No modelo é especificado com linhas contínuas. No exemplo do código 19 apresenta-se a ligação de comunicação intergrupo entre autor e revisor na linha 2.

```
1 <link from="autor" to="revisor" type="communication"
    scope="inter-group" bi-dir="true"/>
```

Código 19 – Codificação da ligação de comunicação entre autor e revisor intergrupo.

Esta ligação é classificada como bidirecional (linha 3), significa que ambos os papéis possuem a mesma ligação de um para outro. Se for falso, somente o papel de origem pode ter este tipo de ligação com o papel destino.

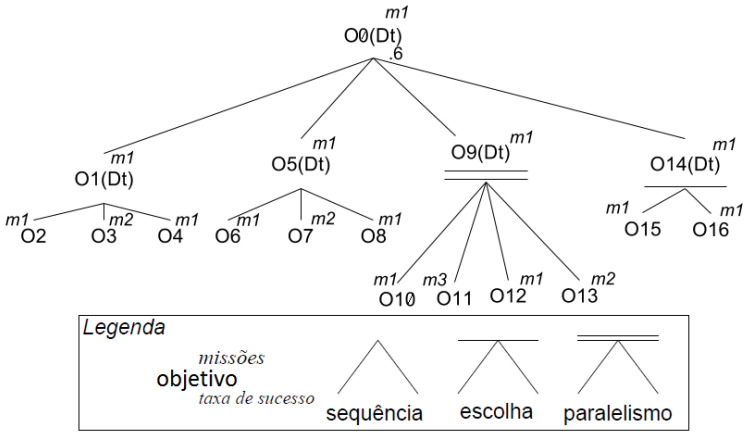
4.5.2 Dimensão Funcional

A dimensão funcional especifica a coordenação das ações de todos os agentes. Determina a organização das ações para atingir os objetivos globais do sistema. Um objetivo global é um conjunto de ações que deve ser realizada pelo sistema como um todo e por isso, é um processo complexo envolvendo muitos agentes. Cada agente é responsável por uma parte do objetivo, uma vez que os agentes têm uma limitação de conhecimento sobre o objetivo global e alguma habilidade para planejar. A dimensão funcional tem o intuito de melhorar a eficiência do SMA.

A dimensão estrutural define como os papéis estão distribuídos em prol da finalidade, mas não determina como alcançar a finalidade. Uma forma de melhorar a eficiência de uma sociedade é estabelecer procedimentos para realizar certas atividades. São atividades que estão bem estabelecidas, e que são aceitas por sua eficiência, por exemplo: comprar um produto, contratar um empregado, e assim por diante. Estes tipos de procedimentos serão denominados Esquemas Sociais. Um Esquema Social é um conjunto de objetivos (LUGO; HÜBNER; SICHMAN, 2001)

semelhantes a uma árvore binária, mas que diferencia que um nó pai pode ter mais de dois nós filhos. Cada nó representa um objetivo. No exemplo apresentado todo objetivo inicia seu nome com a letra maiúscula (*O*) seguido de um número, mas essa denominação é definida pelo desenvolvedor, sendo livre para escolher o nome do objetivo. Esta dimensão funcional lida com a coordenação dos agentes. A seguir será apresentado cada aspecto desta especificação, com o exemplo de um modelo funcional, onde pesquisadores querem publicar um artigo, o mesmo exemplo da seção 4.5.1. O modelo que especifica a dimensão funcional do exemplo da publicação de um artigo, está representado na Figura 14. Nas seções 4.5.2.1 e 4.5.2.2 são apresentados com detalhes as notações existentes do modelo funcional.

Figura 14 – Especificação Funcional.



Fonte: Autoria Própria.

O quadro 1 especifica os nós da árvore de objetivos. Os objetivo possuem uma descrição para facilitar o entendimento da árvore.

No código 20 representa o início e o fim da implementação da dimensão funcional num arquivo XML, entre as *tags* os objetivos são especificados na ordem que devem ser executados.

```
1 <functional-specification>
2   ...
3 </functional-specification>
```

Código 20 – Elementos de início e fim da especificação funcional.

4.5.2.1 Objetivos globais

A principal noção para os Esquemas Sociais, e consequentemente para a Especificação Funcional (EF), é a noção de um objetivo global. Um objetivo global representa um estado do mundo que é desejado pelo SMA. Objetivos globais diferenciam-se de objetivos locais devido

Quadro 1 – Descrição dos objetivos do plano global

Objetivo	*Descrição*
O0	publicar artigo
O1	fazer título
O2	escrever título
O3	revisar título
O4	reescrever título
O5	fazer introdução
O6	escrever introdução
O7	revisar título
O8	reescrever introdução
O9	fazer demais seções
O10	escrever seção
O11	revisar conceitos do desenvolvimento
O12	reescrever escrita seções
O13	revisar escrita resumo
O14	submeter artigo
O15	submissão eletrônica
O16	submissão por correio

Fonte: Autoria Própria.

a esta última ser um objetivo de um único agente enquanto a primeira é objetivo de todo o SMA. Na terminologia proposta por (CASTELFRANCHI, 1995), trata-se de um objetivo social cooperativo. A cada objetivo global O são atribuídos três valores, os quais indicam:

1. Nível de satisfação: aponta se a meta já foi alcançada (valor *satisfeito*) ou não (valor *não satisfeito*), ou mesmo se ela é impossível de ser alcançada (valor *impossível*).

2. Nível de alocação: indica se existe ou não algum agente comprometido a satisfazer a meta (valores *comprometido* e *não comprometido*, respectivamente).

3. Nível de ativação: indica se as pré-condições necessárias para que a meta seja satisfeita estão presentes (valores *permitido* e *proibido*). Por exemplo, a meta para submeter um artigo (*O0*) é proibido até que a meta escrever título do artigo (*O2*) seja realizada.

Seja o objetivo $O\theta$ um objetivo qualquer, o valor inicial correspondente a ela é: (*não satisfeito*, *não comprometido*, *proibido*) e, no decorrer do funcionamento do sistema, seu valor vai sendo alterado. A fim de simplificar a notação das fórmulas das seções seguintes, os seguintes predicados são definidos:

- *estáSatisfeito(*$O\theta$*)* é verdade se o objetivo $O\theta$ estiver satisfeito;

- *éPossível(*$O\theta$*)* é verdade se o objetivo $O\theta$ for possível de ser satisfeito.

4.5.2.2 Nível individual: missões

Um Esquema Social (ES) é constituído, no nível individual, por missões. Uma missão é um conjunto coerente de objetivos globais que podem ser atribuídos a um agente através de seu papel. O agente que se compromete com uma missão é responsável por satisfazer todas os objetivos desta missão, porém pode requisitar a outro agente realizar determinado objetivo em seu lugar. Estes ES podem ser configurados pelo desenvolvedor do SMA, o qual possui conhecimento sobre a solução ou por um agente que armazenou suas melhores soluções do passado. Por exemplo, como uma iniciativa nova criada em meio ao procedimento manual. Também especifica a política para alocação de tarefas para agentes, a coordenação da execução de planos, e a qualidade do plano (tempo de consumo, recursos utilizados, dentre outros).

Um ES é essencialmente uma árvore onde o objetivo principal é decomposto. A raiz desta árvore é o objetivo do ES e os nós posteriores são objetivos menores que são parte da resolução do objetivo maior, semelhante ao método de Divisão e Conquista (KARATSUBA; OFMAN, 1963). Pois resolve um problema dividindo-o num conjunto de problemas menores e mais fáceis de serem realizados. A responsabilidade dos objetivos menores são distribuídos ao longo das missões, como mostrado na Figura 14. Uma missão é um conjunto de objetivos que o agente responsável deve cumprir. No exemplo da publicação do artigo, a missão m2 possui como objetivos $O3$, $O7$ e $O13$. Estes objetivos tratam de revisão de determinadas partes do artigo, sendo assim responsabilidade do papel revisor realizar esta missão.

Os objetivos são partes que compõem missões. Estão organizados em ordem de execução: da esquerda para a direita. O objetivo global somente é alcançado quando todos os objetivos forem cumpridos. Um objetivo ao ser executado deve respeitar os operadores existentes: sequência, escolha ou paralelismo. Um operador define num conjunto de dois ou mais objetivos, como eles poderão ser executados. Desta forma, os operadores são utilizados para estabelecer a coordenação de execução. Os objetivos podem ter argumentos que detalham melhor o estado de satisfação de uma meta. Por exemplo, no objetivo *O1* uma data (argumento *Dt*) deve ser instanciada detalhando o prazo final para a escrita do título.

Para implementar um ES na ferramenta MOISE é necessário especificar a dimensão funcional conforme o código 21. Este código mostra a implementação do esquema *escrita_sch* utilizado para a implementação do código dos escritores da Figura 14.

```
1 <functional-specification>
2   <scheme id="escrita_sch">
3       ...
4   </scheme>
5 </functional-specification>
```

Código 21 – Codificação do esquema escrita.

Os três operadores da dimensão funcional sequência, escolha e paralelismo são implementados na ferramenta MOISE. O operador sequência impõe ao agente a obrigação de seguir a ordem estabelecida. Por exemplo, na Figura 14 o objetivo *O2* precisa ocorrer antes do objetivo *O3*. Executar o *O3* antes de *O2*, proporciona a não satisfação do objetivo *O1* e assim do objetivo global *O0*. No modelo, o operador de sequência é o mais simples, sem nenhum símbolo atrelado ao mesmo, conforme a legenda da Figura 14. A implementação deste operador está representada no código 22, onde o objetivo *escreverTitulo* precisa ser realizado anterior aos objetivos *revisarTitulo* e *reescreverTitulo*.

```
1 <goal id="fazerTitulo">
2    <plan operator="sequence">
3       <goal id="escreverTitulo"/>
4       <goal id="revisarTitulo"/>
5       <goal id="reescreverTitulo"/>
6    </plan>
7 </goal>
```

Código 22 – Código da implementação do objetivo fazerTitulo.

A operação de escolha oferece ao agente a possibilidade de escolher apenas um entre dois ou mais objetivos a serem realizados, permitindo ao agente escolher a partir de sua autonomia qual objetivo é mais viável e melhor para a situação em que se encontra. Em *O14* é possível notar que há o operador escolha, pode ser realizado o objetivo *O15* ou *O16*. No modelo é representado por um traço vertical nos objetivos que devem ter apenas uma escolha. O código 23 mostra a codificação do objetivo *submeterArtigo* ou *gO14*.

```
1 <goal id="submeterArtigo">
2    <plan operator="choice">
3       <goal id="submissaoPorCorreio"/>
4       <goal id="submissaoEletronica"/>
5    </plan>
6 </goal>
```

Código 23 – Implementação do objetivo submeterArtigo.

A operação de paralelismo dá a liberdade ao agente escolher a ordem de execução das tarefas. Permite que o agente possa realizar o conjunto de objetivos com flexibilidade. Esta opção existe no objetivo *O9*, onde os objetivos *O10*, *O11*, *O12* e *O13* podem ser realizados na sequência que os agentes desejarem. No modelo é representado pelo símbolo de sequência com dois traços horizontais sobre ele, como explicitado na legenda. O código 24 mostra sua codificação, o qual implementa o objetivo *fazerDemaisSecoes*.

```
1 <goal id="fazerDemaisSecoes">
2   <plan operator="parallel">
3     <goal id="escreverSecoes"/>
4     <goal id="revisarConceitos"/>
5     <goal id="reescreverSecoes"/>
6     <goal id="EscreverResumo"/>
7   </plan>
8 </goal>
```

Código 24 – Codificação do objetivo fazerDemaisSecoes.

Um conjunto de objetivos forma uma missão. Neste exemplo o agente que se comprometer com a missão *m2* terá que realizar os objetivos *O3*, *O7* e *O13*. Mas serão executas no momento adequado, já que esses objetivos precisam de determinadas condições para acontecerem seguindo as condições dos operadores. Na especificação da missão é determinado que apenas um agente pode realizá-la, devido a cardinalidade na linha 1.

```
1 <mission id="m2" min="1" max="1">
2   <goal id="O3"/>
3   <goal id="O7"/>
4   <goal id="O13"/>
5 </mission>
```

Código 25 – Codificação da missão m2.

4.5.3 Dimensão Normativa

A relação entre a especificação estrutural e funcional é feita pela Especificação Normativa (EN). No nível individual especifica-se quais as missões que um papel tem permissão ou obrigação de se comprometer a realizar. A norma de permissão *per(p, m, tc)*, estabelece ao agente representando o papel *p* cumprir a missão *m*, e o campo *tc* é uma restrição de tempo na permissão (*time constraint*). O tempo permitido especifica o intervalo de tempo onde determinada missão é válida, por exemplo: todo dia, nos Sábados das 13 horas às 15 horas, no primeiro dia do mês. O valor *Qualquer* no campo *tc* significa ser possível realizar em qualquer dia e qualquer hora.

A norma de obrigação *obri(p, m, tc)* estabelece que um agente representando o papel *p* deve se comprometer a realizar a missão *m* nos períodos listados em *tc*. Para os dois predicados referentes a permissão e obrigação as seguintes fórmulas são válidas:

$$obri(p, m, tc) \rightarrow per(p, m, tc) \tag{4.5}$$

$$per(p, m, tc) \wedge p \sqsubset p' \rightarrow per(p', m, tc) \tag{4.6}$$

$$obri(p, m, tc) \wedge p \sqsubset p' \rightarrow obri(p', m, tc) \tag{4.7}$$

A fórmula 4.5 indica que quando um papel é obrigado a uma missão, então ele também tem permissão para tal missão. As fórmulas 4.6 e 4.7 determinam que as normas referentes a permissão e obrigação aplicadas a um papel, geram a mesma norma para os papéis especializações.

Toda EN tem, pelo menos, a seguinte relação normativa: *obri*($psoc$, $msoc$), *Qualquer*. Como todos os papéis são especializados de $psoc$, os agentes são obrigados a se comprometer com a missão $msoc$ (a missão raiz do ES $schsoc$). Uma vez que $schsoc$ representa o plano global para alcançar a finalidade do SMA, esta obrigação implica que todos os agentes estejam comprometidos com a finalidade do sistema.

Uma vez que o ES esteja criado, outros agentes (representando escritor, revisor, ...) são obrigados pelas relações normativas do papel a participarem no SCH. Estes agentes devem perseguir os objetivos de suas missões somente no momento que o SCH permitir. As especificações normativas da equipe de pesquisadores para escrever um artigo podem incluir as seguintes especificações do quadro 2:

Quadro 2 – Especificação Normativa

Papel	*Relação Normativa*	*Missão*	*Restrições Temporais*
escritor	obri	m1	12/09/2016
editor	obri	m2	12/09/2016
revisor	obri	m3	12/09/2016

Fonte: Autoria Própria.

Neste exemplo da escrita do artigo, o editor tem a obrigação de realizar a missão identificada como *m2*. Toda missão é um conjunto de objetivos e neste exemplo o editor tem três objetivos a cumprir: O3, O7 e O13.

```
1 <normative-specification>
2   <mission id="m2" min="1" max="1">
3     <goal id="O3"/>
4     <goal id="O7"/>
5     <goal id="O13"/>
6   </mission>
7 </normative-specification>
```

Código 26 – Codificação da especificação normativa.

4.5.4 Integrar Jason e Moise

Para o MOISE ser utilizado por agentes, o MOISE é implementado via artefatos. Porém, existem operações específicas para criar os itens do MOISE. A seguir as principais operações que os agentes podem realizar:

1. *makeArtifact* - criar uma organização.
2. *createGroup* - criar um grupo.
3. *adoptRole* - adotar um papel num grupo.
4. *destroy* - destruir um grupo ou esquema.
5. *focus* - focalizar num grupo ou esquema.
6. *stopFocus* - interromper sua atenção num grupo ou esquema.
7. *addScheme* - adicionar um esquema para um grupo se responsabilizar a fazer.
8. *commitMission* - se comprometer a realizar uma missão.
9. *goalAchieved* - marcar um objetivo como cumprido.

Para exemplificar a integração é utilizado um exemplo em que um agente chamado técnico cria uma organização para um time de jogadores participarem e se organizarem em campo. O código do técnico é implementado no código 27. O técnico tem o desejo montarTime (linha 1). Este desejo aciona o plano montarTime (linha 3), em que é criado uma organização chamada OrganizacaoFlamengo (linha 4) e focaliza na organização (linha 5). Cria o grupo grupoFlamengo (linha 6) baseado no grupoTime especificado no arquivo organization.xml. O agente adota o papel tecnico (linha 7) e focaliza no grupo (linha 8). Cria os esquemas ataque e esquema (linhas 9 e 10). Focaliza nestes esquemas (linhas 11 e 12). Envia a mensagem receberInstrucoes com a informação do nome do grupo que devem participar (linha 13). Realiza os planos adicionarEsquema (linha 14 e 15).

O plano adicionarEsquema só pode ser realizado se o grupo estiver pronto, para isso, o número mínimo de agentes de cada papel deve ter adotado seu papel (linha 17). Se estiver pronto é adicionado o esquema ao grupo.

```
1  !montarTime.
2
3  +!montarTime : true
4    <-  makeArtifact("OrganizacaoFlamengo",
       "ora4mas.nopl.OrgBoard",
       ["src/org/organization.xml"], Id_Org);
5        focus(Id_Org);
6        createGroup("grupoFlamengo", grupoTime, IdGrupo);
7        adoptRole(tecnico)[artifact_id(IdGrupo)];
8        focus(IdGrupo);
9        createScheme("ataque", schemeAtacar,   IdEsqAta);
10       createScheme("defesa", schemeDefender, IdEsqDef);
11       focus(IdEsqAta);
12       focus(IdEsqDef);
13       .broadcast(tell, recInstr("grupoFlamengo"));
14       !adicionarEsquema("ataque");
15       !adicionarEsquema("defesa").
16
17 +!adicionarEsquema(IdGrupo, NomeEsquema) :
     formationStatus(ok)[artifact_id(IdGrupo)]
18   <-  addScheme(NomeEsquema)[artifact_id(IdGrupo)].
```

Código 27 – Codificação do agente técnico.

O código do agente jogador está implementado em 28. Quando recebe a crença *recInst* o jogador tem um plano que ocorre se o agente que enviou essa crença é o agente técnico. O técnico envia o nome do grupo que o agente deve participar (linha 1). O agente jogador descobre o identificador deste grupo (linha 2), focaliza neste grupo (linha 3), recupera o papel que deve adotar que está guardado na crença papel (linha 4), realiza a operação de adotar um papel (linha 5) com o nome guardado na variável P.

```
1  +recInstr(Grupo)[source(tecnico)]
2    <-  lookupArtifact(Grupo, IdGr);
3        focus(IdGr);
4        ?papel(P);
5        adoptRole(P)[artifact_id(IdGr)].
```

Código 28 – Codificação do agente jogador.

Após adotar o papel o sistema normativo MOISE irá enviar obrigações para os agentes por meio de sinais que serão percebidos pelos agentes que estão num grupo. O código 29 representa os jogadores lidando com as obrigações da organização para eles. O primeiro plano lida com missões que são obrigações do grupo realizar (linha 1). A condição para o plano ser realizado é o nome do agente no parâmetro Ag recebido ser o mesmo que o agente que percebeu essa obrigação (linha 2), o agente realiza a missão que está destinada para ele (linha 3). O segundo plano lida com a obrigação de realizar um objetivo (linha 5). O plano tem como condição o agente ter o mesmo nome que o parâmetro recebido em Ag e se o objetivo pode ser feito (What=*satisfied*(Esquema,Objetivo) ou já foi concluído (What = *done*(Esquema,Objetivo,Ag)) (linha 6). O agente realiza o objetivo (linha 7) e depois marca o objetivo como atingido (linha 8).

```
1 +obligation(Ag,Norma,committed(Ag,Mission,Esquema),TimeConst)
2   :   .my_name(Ag)
3   <-  commitMission(Mission)[artifact_name(Esquema),
      wid(W)].
4
5 +obligation(Ag,Norma,What,TimeConst)[artifact_id(ArtId)]
6   :   .my_name(Ag) & (What=satisfied(Esquema,Objetivo) |
      What = done(Scheme,Objetivo,Ag))
7   <-  !Objetivo[scheme(Scheme)];
8       goalAchieved(Objetivo)[artifact_id(ArtId)].
```

Código 29 – Codificação das obrigações que os agentes devem realizar.

5 DESENVOLVIMENTO MAPS-NORMS

Este capítulo apresenta a descrição do SMA que implementa um estacionamento inteligente. A seção 5.1 apresenta a descrição geral do estacionamento inteligente implementado, o uso da metodologia Prometheus para extrair as funcionalidades do sistema e o funcionamento do MAPS-NORMS. Nas seções 5.2, 5.3, 5.4 e 5.5 são explicadas e detalhadas as quatro fases principais do funcionamento do estacionamento: abrir estacionamento, requisitar vaga, ocupar vaga e desocupar vaga. Em cada fase é apresentado como o MAPS implementa, o que precisou ser alterado e como fora implementado no MAPS-NORMS.

5.1 DESCRIÇÃO GERAL

O MAPS é um sistema inteligente para alocação de vagas que utiliza um SMA para reger um estacionamento particular que tenha funcionamento autônomo na tarefa de distribuir vagas para os motoristas. O MAPS-NORMS visa estender a implementação do MAPS para aumentar a flexibilidade do SMA, aumentar a autonomia dos motoristas e a criação de uma organização normativa. Além disso, obter o histórico de grau de confiança dos motoristas para avaliar como o comportamento do motorista afeta o funcionamento do SMA.

A partir da metodologia Prometheus fora estabelecido um descritivo sobre o SMA: o gerente inicializa os itens que existem no estacionamento: cancela, controlador do estacionamento e todas as vagas. O gerente envia uma mensagem para todos os motoristas avisando que o estacionamento está pronto com as informações necessárias para requisitar vaga. Os motoristas podem requisitar vaga para o controlador do estacionamento. O controlador do estacionamento aguarda um determinado tempo para receber requisições por vagas. Depois de aguardar verifica se existe vaga disponível. Quando existir uma vaga livre é verificado se existe algum motorista na fila. Se houver é reservado a vaga para o motorista com maior grau de confiança e reserva a vaga disponível.

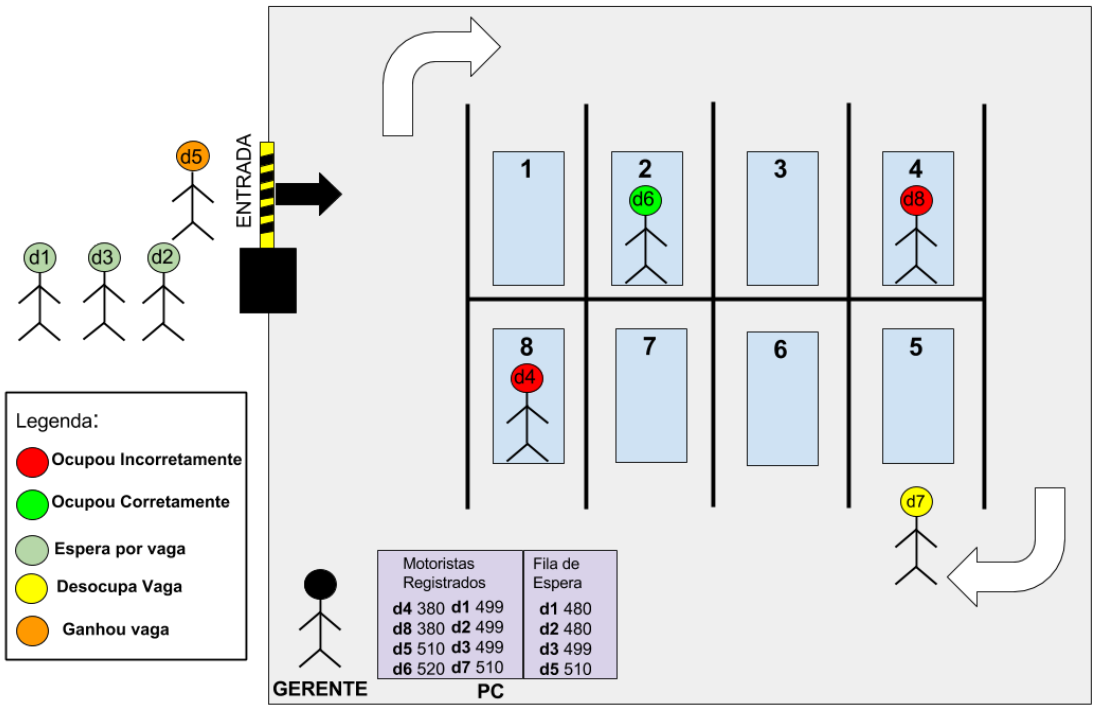
O gerente é notificado pelo controlador do estacionamento que existe uma vaga reservada para determinado motorista. O gerente envia uma mensagem para o motorista com as informações necessárias para o motorista ocupar a vaga. O motorista passa pela cancela e escolhe qual vaga quer estacionar. Quando um motorista ocupa uma vaga, a vaga detecta a ocupação e envia essa informação para o controlador do estacionamento. O controlador do estacionamento verifica se o motorista ocupou a vaga reservada. Se o motorista ocupar a vaga que recebeu do gerente, ganha um incremento no grau de confiança, caso contrário o controlador do estacionamento irá decrementar o grau de confiança deste motorista.

A Figura 15 representa um exemplo de estacionamento regido pelo MAPS-NORMS. Neste estacionamento existe um agente gerente, oito agentes motoristas, um artefato controlador

de estacionamento (*Parking Controller* - PC), um artefato cancela e oito artefatos vagas. O agente gerente é quem cria e configura o estacionamento, notifica os motoristas que o estacionamento está aberto e se ganhou uma vaga para ocupar e gerencia a organização normativa. O PC é um objeto que auxilia o agente gerente para tarefas automáticas, como: escolher o próximo motorista a receber uma vaga, detectar se uma vaga foi ocupada corretamente, dentre outros.

A cancela controla a entrada para que apenas motoristas com vagas reservadas entrem no estacionamento. A quantidade de vagas é determinada pelo gerente na inicialização do estacionamento, neste exemplo foram apenas oito. Exitem três motoristas esperando por uma vaga, um motorista que ganhou uma vaga, um motorista que ocupou a vaga corretamente, dois motoristas ocupando incorretamente duas vagas e um motorista desocupando, conforme a descrição da legenda. O motorista d5 recebeu uma vaga por ter o maior grau de confiança que os outros motoristas que estavam esperando por vaga. Os motoristas d4 e d8 recebem um decremento no grau de confiança registrado, o motorista d7 deixa a vaga porém o PC mantém registrado o grau de confiança de sua ação de ocupar vaga. O motorista d6 recebe um incremento por ter ocupado a vaga reservada pelo PC.

Figura 15 – Estrutura geral de um estacionamento inteligente.



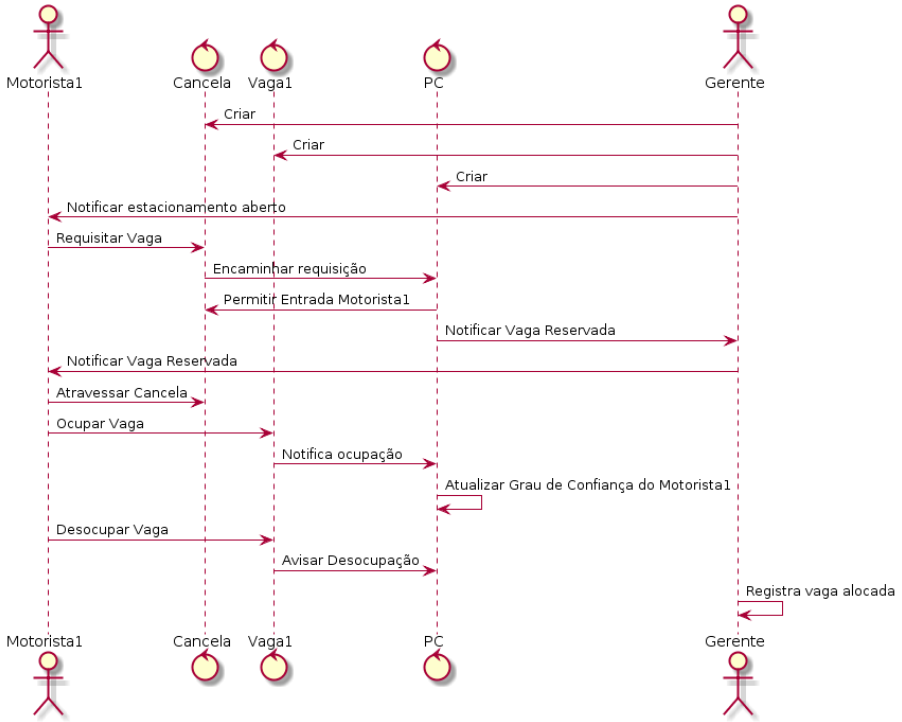
Fonte: Autoria Própria.

O funcionamento resumido do MAPS-NORMS é representado na Figura 16, o diagrama representa um exemplo em que existe apenas um motorista e uma vaga. O estacionamento é criado pelo gerente que, em seguida, notifica o motorista que o estacionamento está pronto. O motorista requisita uma vaga para a cancela que encaminha o pedido para o PC. Quando o Motorista1 ganhar uma vaga o PC avisa a cancela que o Motorista1 está permitido a entrar no

estacionamento e notifica o gerente que existe uma vaga reservada.

O gerente ao receber a notificação do controlador do estacionamento, envia uma mensagem para o motorista com as informações do estacionamento e qual vaga fora reservada para ele. O motorista passa pela cancela que o Motorista1 estava registrado. O motorista ocupa a Vaga1. A Vaga1 percebe que fora ocupada e informa ao PC. O PC atualiza o grau de confiança (GC) do Motorista1. O motorista desocupa a vaga que informa ao PC.

Figura 16 – Diagrama de sequência resumido do funcionamento do MAPS-NORMS.



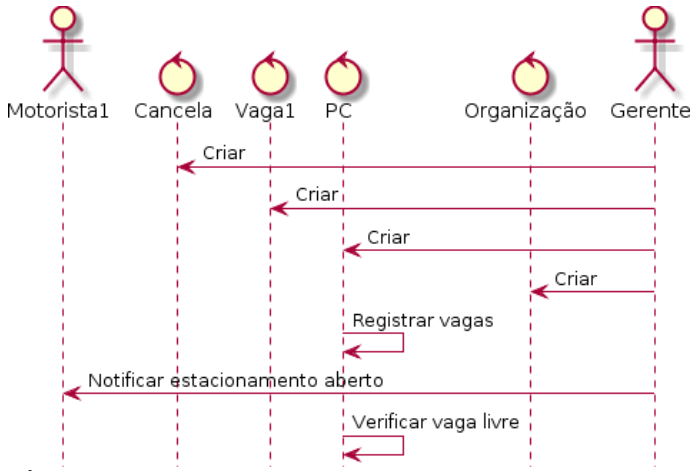
Fonte: Autoria Própria.

A seguir é apresentado como o MAPS e o MAPS-NORMS implementam as funcionalidades que compõem o estacionamento. A seguir serão apresentadas as quatro fases principais do estacionamento: abrir o estacionamento, requisitar vaga, ocupar vaga e desocupar vaga.

5.2 ABRIR ESTACIONAMENTO

Nesta fase acontece a criação do estacionamento e todos os agentes que estão no SMA são notificados que o estacionamento está aberto. A mensagem é enviada apenas uma vez para todos os agentes do SMA. Os agentes motoristas estão configurados para receber a mensagem e requisitar uma vaga. A Figura 17 apresenta detalhadamente como é realizada criação do estacionamento e a notificação aos agentes motoristas que o estacionamento está aberto.

Figura 17 – Diagrama de sequência da fase de abertura do estacionamento.



Fonte: Autoria Própria.

5.2.1 Criar Estacionamento

Nesta fase o agente gerente é responsável por criar os itens que compõem o estacionamento. O gerente no MAPS cria os artefatos cancela e o controlador de fila. As vagas são crenças armazenadas no gerente. No MAPS-NORMS o gerente cria os artefatos: organização, controlador do estacionamento, cancela, vagas e notifica os motoristas que o estacionamento está pronto.

Quando o MAPS-NORMS é inicializado o agente gerente tem o desejo de abrir o estacionamento e este desejo aciona o plano para realizá-lo. O código 30 apresenta o gerente com o desejo inicial chamado *des_openParking* (linha 1). Este desejo tem um plano para realizá-lo: *des_openParking* (linha 3). Este plano tem como condição a existência da crenças *bel_ws* que carrega na variável *Var_WS* o nome da área de trabalho que o estacionamento estará.

O gerente cria uma área de trabalho (linha 4) e entra nela (linha 5). Após entrar na área de trabalho, todo artefato criado pelo gerente apartir deste momento estará nesta área de trabalho. Executa o plano para criar a organização (linha 6), o plano para criar o PC (linha 7), o plano para criar a cancela (linha 8), o plano para criar todas as vagas (linha 9) e o plano para notificar os motoristas (linha 10).

```
1  !des_openParking.
2
3  +!des_openParking : bel_ws(Var_WS)
4    <-  createWorkspace(Var_WS);
5        joinWorkspace(Var_WS, _);
6        !plan_createOrg;
7        !plan_createPC;
8        !plan_createGate;
9        !plan_createSpots;
10       !plan_notifyDrivers.
```

Código 30 – Codificação do plano abrir estacionamento.

O plano *plan_createOrg* criará o artefato que realizará a organização normativa do SMA. Este artefato é responsável por especificar como os agentes devem agir no SMA. O código 31 implementa o plano do gerente de criar a organização. Este plano tem como condição a existência da crença *bel_org* que carrega na variável *Var_Org* o nome que a organização do estacionamento terá (linha 1).

Este plano cria o artefato com os campos: nome do artefato, caminho para o código fonte do artefato, o parâmetro de inicialização e o identificador do artefato (linha 2). Após criar o artefato, o gerente focaliza nele (linha 3).

```
1  +!plan_createOrg : bel_org(Var_Org)
2    <-  makeArtifact(Var_Org, "ora4mas.nopl.OrgBoard",
       ["src/org/organization.xml"], Var_IdOrg);
3        focus(Var_IdOrg);
```

Código 31 – Codificação do plano criar organização.

O plano *plan_createPC* é responsável por criar o PC. O PC substitui o controlador de fila do MAPS. Além de controlar a fila de espera, é responsável por manter o percentual de vagas ocupas do estacionamento, reservar vagas, avisar ao gerente que uma vaga está reservada, registrar e atualizar o GC dos motoristas, receber requisições por vaga e registrar se uma vaga está livre ou ocupada.

O gerente cria o PC e o configura com um nível de restrição. Existem três tipos de níveis de restrição desenvolvidos no MAPS-NORMS para testar o impacto das normas: *weakly*, *normal* e *heavily*. O nível de restrição determina o tamanho da punição e bonificação que um agente motorista irá receber em seu Grau de Confiança por seu comportamento no estacionamento. A restrição *weakly* determina uma punição e bonificação branda, *normal* o agente recebe um

incremento ou decremento maior que em *weakly* e o nível *heavily* puni e bonifica com maior intensidade que *normal*.

O código 32 mostra a implementação do plano *plan_createPC*. O plano tem como condição a existência das crenças *bel_numSpots*, *bel_namePC* e *bel_restrictLevel* (linha 1). A crença *bel_numSpots* carrega o número total de vagas no estacionamento na variável Var_NS, a crença *bel_namePC* carrega o nome do PC na variável *Var_NPC* e a crença *bel_restrictLevel* tem o nível de restrição do estacionamento carregado na variável Var_R.

Este plano cria o artefato com os campos: nome do artefato, caminho para o código fonte do artefato, o parâmetro de inicialização e o identificador do artefato (linha 2). O parâmetro de inicialização é o número total de vagas e o nível de restrição do SMA. Então o gerente focaliza no artefato PC a partir do identificador gerado em Var_IdPC (linha 3).

```
1 +!plan_createPC : bel_numSpots(Var_NS) &
     bel_namePC(Var_NPC) & bel_restrictLevel(Var_R)
2   <-  makeArtifact(Var_NPC,
       "park_art.ParkingController", [Var_NS, Var_R],
       Var_IdPC);
3       focus(Var_IdPC).
```

Código 32 – Codificação do plano criar PC no MAPS-NORMS.

A cancela criada no MAPS-NORMS apresenta algumas mudanças em relação a cancela do MAPS. No MAPS o gerente abre a cancela para o motorista entrar e depois a fecha, isto requer a dependência do gerente toda vez que algum motorista entra no estacionamento. No MAPS-NORMS a cancela é aberta pelo motorista que está permitido a entrar no estacionamento e fechada depois que o motorista passa pela cancela. Como o gerente não realiza nenhuma operação no artefato cancela, permite maior autonomia da cancela e flexibilidade do funcionamento do SMA.

O artefato cancela apresenta duas propriedades: *prop_isOpen* e *prop_PC*. A *prop_isOpen* representa se a cancela está aberta ou fechada. A propriedade *prop_PC* carrega o nome do artefato que a cancela irá se comunicar para receber o nome dos agentes motoristas permitidos a entrar.

O artefato possui as operações: *requestSpot*, *permitted*, *openGate*, *crossGate* e *closeGate*. A operação *requestSpot* é realizada por um motorista para requisitar uma vaga. A operação *permitted* é realizada somente pelo PC e determina que um motorista está permitido a entrar no estacionamento. A operação *openGate* é realizada por um motorista para entrar no estacionamento. A operação *crossGate* é realizada por um motorista para passar pela cancela. A operação interna *closeGate* é realizada pela própria cancela para fechar o acesso a entrada.

O código 33 mostra a implementação do plano *createGate*. A condição para o plano ser realizado é existir as crenças *bel_nameGate* e *bel_namePC* (linha 1). A crença *bel_nameGate*

carrega na variável *Var_NG* o nome que a cancela terá. A crença *bel_namePC* carrega na variável *Var_NPC* o nome do PC que a cancela irá se comunicar.

Este plano cria o artefato com os campos: nome do artefato, caminho para o código fonte do artefato, o parâmetro de inicialização e o identificador do artefato (linha 2). O parâmetro de inicialização é o nome do PC. O gerente não focaliza neste artefato porque não interage com ele.

```
1 +!plan_createGate : bel_nameGate(NG) & bel_namePC(NPC)
2   <-  makeArtifact(NG, "park_art.Gate", [NPC]).
```

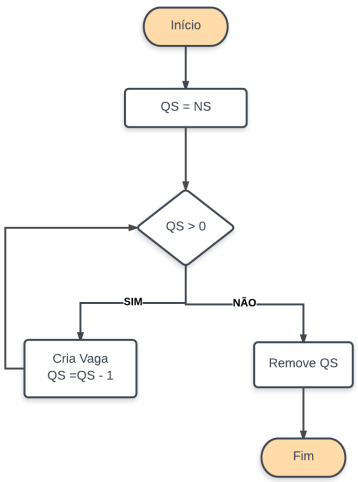
Código 33 – Codificação do plano criar cancela no MAPS-NORMS.

No MAPS-NORMS a vaga é implementada como artefato. A vaga possui as propriedades: *isFree*, *driverOcc* e *PC*. A propriedade *isFree* representa se a vaga está livre ou não, a propriedade *driverOcc* indica o nome do motorista que está ocupando a vaga e a propriedade *PC* guarda o nome do PC que a vaga irá se comunicar quando for ocupada ou desocupada.

Existem duas operações: *occupy* e *vacate*. A operação *occupy* é realizada por um motorista para ocupar a vaga e a operação *vacate* é realizada pelo motorista que a ocupava, desocupá-la.

Para criar os artefatos vagas existe um plano recursivo. A Figura 18 apresenta como funciona este plano: inicializa a variável QS com o número total de vagas NS. Se a variável QS tiver valor maior que zero, uma vaga é criada e a variável decrementada. Quando o valor de QS for igual a zero, a variável QS é removida e a recursão para.

Figura 18 – Fluxograma para criar vagas.



Fonte: Autoria Própria.

O plano *plan_createSpots* no código 34 implementa as vagas do estacionamento. A criação de vagas é realizada recursivamente, o primeiro plano realiza a parada da recursão para

o caso de não existir mais vagas a serem criadas e o segundo plano realiza a criação de uma vaga e aciona recursivamente o próprio plano.

O primeiro plano tem como condição a existência da crença *bel_quantSpots* e a variável Var_QS com valor igual a zero (linha 1). Este plano apenas remove a crença *bel_quantSpots* (linha 2).

O segundo plano (linha 4) tem como condição a existência da crença *bel_quantSpots* e a variável Var_QS com valor maior que zero e a existência da crença *bel_namePC*. A crença *bel_namePC* carrega na variável *Var_NPC* o nome do PC que as vagas irão se comunicar. O plano converte o número de vagas que são do tipo inteiro para o tipo *String* na variável *Var_SpotName* (linha 5), Cria o artefato com os campos: nome do artefato, caminho para o código fonte do artefato, o parâmetro de inicialização e o identificador do artefato (linha 6). Atualiza o número de vagas a serem criadas na base de crenças do gerente (linha 7) e aciona recursivamente o mesmo plano (linha 8). O parâmetro de inicialização é o nome do PC que a vaga irá se comunicar. O gerente não focaliza neste artefato porque não interage com ele.

```
1 +!plan_createSpots : bel_quantSpots(Var_QS) & Var_QS == 0
2   <-  -bel_quantSpots(Var_QS).
3
4 +!plan_createSpots : bel_quantSpots(Var_QS) & Var_QS > 0
     & bel_namePC(Var_NPC)
5   <-  .term2string(Var_QS, Var_SpotName);
6       makeArtifact(Var_SpotName, "park_art.Spot",
           [Var_NPC]);
7       Var_NQS = Var_QS - 1; -+bel_quantSpots(Var_NQS);
8       !plan_createSpots.
```

Código 34 – Codificação do plano criar vagas no MAPS-NORMS.

5.2.2 Notificar Motoristas

No MAPS não existe a notificação para os motoristas informando que o estacionamento está pronto, pois os agentes motoristas estão programados a requisitar vaga em sua inicialização. A notificação foi acrescentada no MAPS-NORMS para tornar as ações dos agentes motoristas e gerente coordenadas.

O código 35 apresenta a implementação do plano *plan_notifyDrivers*. O plano tem

como condição a existência das crenças *bel_ws* e *bel_numSpots*. A crença *bel_ws* carrega na variável nome da área de trabalho para ter acesso a cancela *wS* e o número total de vagas. A partir dessas crenças recuperadas nas variáveis: WS e NS (linha 1) a mensagem *parkReady* do tipo *broadcast* é enviada (linha 2).

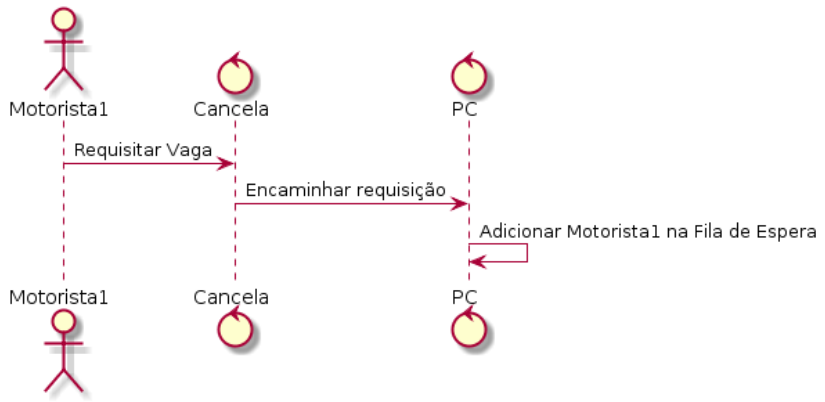
```
1 +!plan_notifyDrivers : bel_ws(Var_WS) &
     bel_numSpots(Var_NS)
2   <-  .broadcast(tell, parkReady(WS, NS)).
```

Código 35 – Codificação do plano notificar motoristas no MAPS-NORMS.

5.3 REQUISITAR VAGA

Na segunda fase do funcionamento do estacionamento o motorista realiza a requisição por vaga, como ilustrado na Figura 19.

Figura 19 – Diagrama de sequência da fase requisição por vaga.

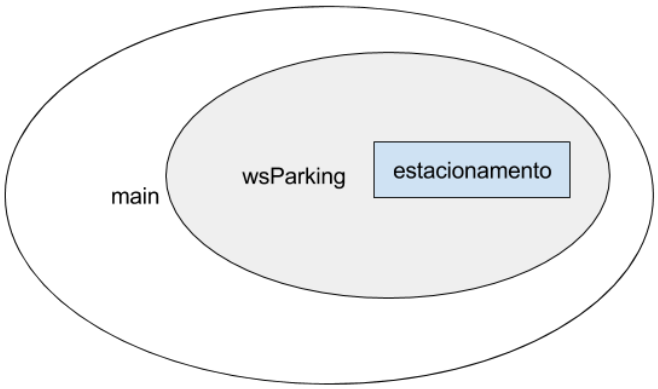


Fonte: Autoria Própria.

Ao receber notificação que o estacionamento está aberto, o motorista entra na área de trabalho do estacionamento e realiza o plano requisitar vaga a partir das informações recebidas pelo gerente. No MAPS não existia uma área de trabalho específica para o estacionamento, os artefatos ficavam na área de trabalho principal.

Para tornar organizado o acesso ao estacionamento, criou-se uma área de trabalho específica para o estacionamento criado, assim permite que os artefatos fiquem reunidos num local comum, em que agentes que estão fora desta área de trabalho não tenham acesso aos artefatos do estacionamento. A área de trabalho que contém o estacionamento se chama *wsParking*, como representado na Figura 20. A área de trabalho contém o estacionamento e isola do contato dos agentes que estão na área de trabalho principal e que não querem utilizar o estacionamento.

Figura 20 – Área de Trabalho *wsParking* contém o estacionamento.



Fonte: Autoria Própria.

Ao receber a mensagem *parkReady*, o agente motorista recebe o nome da área de trabalho e o número total de vagas. O código 36 implementa o plano para o recebimento da notificação. Este plano não tem condição para ser realizado. O motorista guarda esses valores que recebeu do gerente por meio da criação das crenças: *bel_ws* e *bel_numTotalSpots* (linha 2). O motorista entra na área de trabalho que recebe do agente gerente (linha 3) e realiza o plano requisitar vaga *plan_requestSpot* (linha 4).

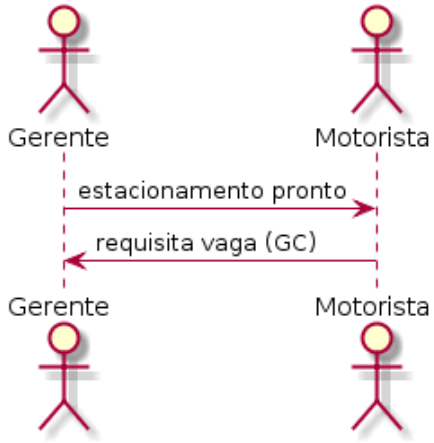
```
1 +parkReady(Var_WS, Var_NS)
2   <-  +bel_ws(Var_WS); +bel_numTotalSpots(Var_NS);
3       joinWorkspace(Var_WS, _);
4       !plan_requestSpot.
```

Código 36 – Codificação do plano estacionamento está pronto no MapsNorms.

5.3.1 Agente Motorista Requisita Vaga

No MAPS o motorista requisita uma vaga a partir de uma mensagem enviada ao gerente, esta mensagem contém o valor do GC do motorista como representado na Figura 21.

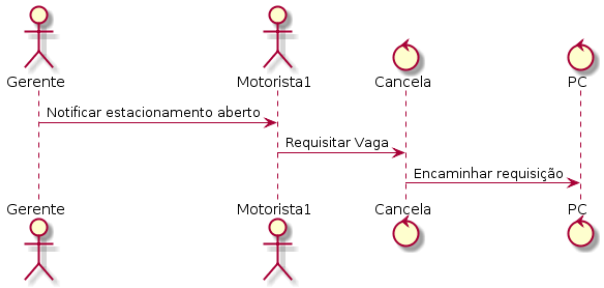
Figura 21 – Diagrama de Sequência para requisitar vaga no MAPS.



Fonte: Autoria Própria.

No MAPS-NORMS a forma de requisitar uma vaga foi alterada, como representado na Figura 22. O motorista requisita uma vaga para a cancela sem enviar sua pontuação. Isto evita que o motorista envie uma nota mais alta que o valor verdadeiro para benefício próprio, e assim burlar o sistema. Para evitar este problema e continuar utilizando o GC como critério para enviar vaga aos motoristas, o PC cadastra o motorista em sua base de dados e atribui um valor inicial para o GC do agente motorista, e conforme o motorista utiliza o estacionamento o PC atualiza o valor do GC de cada agente motorista que utiliza o estacionamento.

Figura 22 – Diagrama de Sequência para requisitar vaga no MAPS-NORMS.



Fonte: Autoria Própria.

Para requisitar vaga, o agente motorista realiza o plano *plan_requestSpot* do código 37. Se o motorista não quiser mais utilizar o estacionamento o primeiro plano é ativado (linha 1), pois tem como condição o valor da variável Var_Nus ser zero, neste plano o agente motorista apenas remove a crença com o número de vagas ainda a serem ocupadas (linha 2).

Senão o motorista realiza o segundo plano (linha 4), cria a variável Var_NNUS para receber o número de vagas ainda a serem ocupadas decrementado em um e atualiza a crença *bel_numUseSpot* (linha 5). Realiza a operação *op_requestSpot* na cancela (linha 6).

```
1 +!plan_requestSpot : bel_numUseSpot(Var_NUS) & Var_NUS
    == 0
2   <-  -bel_numUseSpot(Var_NUS).
3
4 +!plan_requestSpot : bel_numUseSpot(Var_NUS) & Var_NUS >
    0
5   <-  Var_NNUS = Var_NUS - 1; -+bel_numUseSpot(Var_NNUS);
6       op_requestSpot.
```

Código 37 – Codificação do plano chegar no estacionamento.

5.3.2 Registrar Requisição

O MAPS registra o pedido do motorista num plano do gerente. O plano verifica se o estacionamento está cheio. Se estiver cheio, o gerente registra o GC recebido do motorista no artefato controlador de fila. Senão o gerente procura em sua base de crenças uma vaga livre e a envia para o motorista.

No MAPS-NORMS o motorista requisita uma vaga para o artefato cancela e não para o gerente. Essa mudança foi realizada porque o agente gerente possui uma arquitetura que não comporta mais que uma requisição ao mesmo tempo, ou seja, se dois ou mais motoristas requisitarem uma vaga para o gerente ao mesmo tempo, o mesmo dará a mesma vaga livre para todos, pois quando o gerente consulta sua crença sobre a vaga, o gerente a percebe livre. Desta forma, o gerente envia a mesma vaga livre para os motoristas que requisitaram uma vaga. A mudança para a requisição ser no artefato cancela elimina a possibilidade de uma vaga livre ser enviada para mais de um motorista ao mesmo tempo.

O motorista requisita uma vaga para a cancela por meio da operação *op_requestSpot* apresentada no código 38. A cancela recupera o nome do artefato PC (linha 3) e realiza novamente a operação de requisitar vaga (linha 4). O motorista não requisita diretamente a vaga para o PC por segurança. O único agente que pode realizar operações no PC é o gerente.

```
1 @OPERATION void op_requestSpot() {
2     String namePC = getObsProperty("PC").getValue().toString();
3     execLinkedOp(lookupArtifact(namePC), "requestSpot");
4 }
```

Código 38 – Implementação da operação *requestSpot* do PC.

A operação realizada pela cancela no PC está implementado no código 39. A operação cria um objeto da classe *Driver* a partir do método *GetDriver* que retorna o objeto que representa

o motorista que requisitou para o estacionamento. O PC adiciona o motorista na fila de espera (linha 3). A fila de espera chama-se *dbDriReqSpot*, é uma lista ligada que armazena os objetos que representam os motoristas.

```
1  @LINK void requestSpot() {
2    Driver driver =
        GetDriver(this.getCurrentOpAgentId().getAgentName());
3    dbDriReqSpot.add(driver);
4  }
```

Código 39 – Implementação da operação *requestSpot* do PC.

O método *GetDriver* é implementado no código 40. Este método recebe o nome do motorista como parâmetro (linha 1), utiliza um laço para percorrer toda a lista de motoristas registrados no PC chamado *dbDrivers*. O *dbDrivers* se trata de uma lista ligada de objetos da classe *Driver*. A classe *Driver* possui os atributos: nome e GC. Se for encontrado um motorista com o mesmo nome que aquele que requisitou uma vaga, é retornado o objeto com o nome e o GC registrado (linha 6). Se não for encontrado o motorista que requisitou vaga, significa que o mesmo não está registrado, por isso é criado um novo objeto motorista com o GC inicial padrão (linha 9), adiciona-se a base de dados dos motoristas registrados (linha 10) e retorna o objeto do motorista (linha 11).

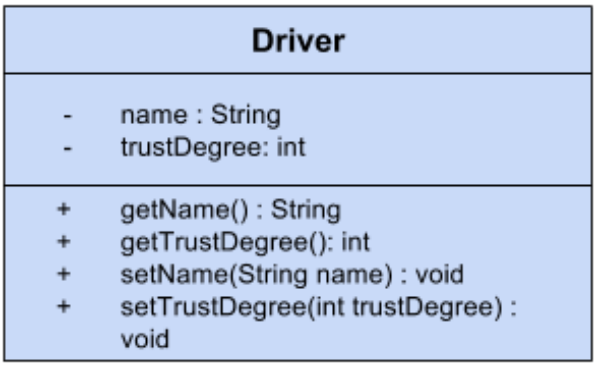
O GC pode variar entre zero e 999, é inicializado com 499 pois é um valor que fica entre os extremos. Se o GC inicial dado pelo gerente for muito baixo, pode ser que um bom motorista que utilizou o estacionamento poucas vezes não tenha prioridade para ganhar a vaga do que um mau motorista que utiliza o estacionamento há muito tempo. Se o GC inicial for muito alto, pode ser que um mau motorista com o GC muito alto que utiliza o estacionamento pela primeira vez, atrapalhe o estacionamento por ter prioridade em relação ao restante dos motoristas.

```
1  Driver GetDriver(String nameDriver){
2    Iterator<Driver> ItDrivers = dbDrivers.iterator();
3    while (ItDrivers.hasNext()){
4      Driver driver = ItDrivers.next();
5      if (driver.getName().equals(nameDriver)){
6        return driver;
7      }
8    }
9    Driver driver = new Driver(nameDriver, 499);
10   dbDrivers.add(driver);
11   return driver;
12 }
```

Código 40 – Implementação do método adicionar motorista do PC.

A classe *Driver* que representa o motorista para o estacionamento possui sua representação na Figura 23. A classe tem as propriedades: *name* do tipo *String* que representa o nome do motorista e *trustDegree* do tipo inteiro que representa o GC do motorista. A classe possui os métodos para atribuir e obter as propriedade do motorista.

Figura 23 – Representação da classe Motorista.



Fonte: Autoria Própria.

Toda requisição do motorista é colocada numa fila de espera. Continuamente o PC verifica se existe alguma vaga livre para realizar o envio de uma vaga para um dos motoristas que estão na fila de espera. Na inicialização do artefato PC, realiza-se a operação interna *checkSendSpot* implementado no código 41.

No início da operação é realizada a espera de um segundo (linha 2), para esperar as requisições por vaga dos motoristas. Se houver pelo menos uma vaga livre , é verificado se algum motorista está na fila de espera (linha 3). Se houver algum motorista esperando por vaga, decrementa em 1 o número de vagas livres (linha 4), atualiza a porcentagem de vagas livres (linha 5), atualiza o percentual de vagas ocupadas (linha 6), obtém o objeto do motorista que ganhou a vaga (linha 7). Realiza-se uma interação na lista de vagas registradas no PC para descobrir a vaga que está livre (linha 8 e 9). A primeira vaga livre que for encontrada é reservada para o motorista (linha 11), atualiza o objeto da vaga com o nome do motorista que está reservado (linha 12), envia para a cancela o nome do motorista que ganhou uma vaga e está permitido entrar no estacionamento (linha 14), envia um sinal *sendSpot* com os parâmetros nome da vaga, do motorista, do grupo e do esquema (linha 18). Finaliza a procura por vaga livre (linha 19). Após verificar se havia vaga livre, é realizada novamente a operação interna de checar se existe vaga livre (linha 24).

```
1 @INTERNAL_OPERATION void checkSendSpot(){
2   await_time(1000);
3   if(numFreeSpots > 0){
4     if(!dbDriReqSpot.isEmpty()){
5       numFreeSpots--;
6       updateCoefOc();
7       String driver = getNextDriver();
```

```
8         Iterator<SpotClass> iterSpots = dbSpots.iterator();
9         while(iterSpots.hasNext()){
10          SpotClass spot = iterSpots.next();
11          if(spot.getIsFree()){
12            updateSpotRes(spot, driver);
13            execLinkedOp(lookupArtifact("gate1"), "permitted",
                driver);
14
15            signal("sendSpot", spot.getId(), driver, getGroup(),
                getScheme());
16            break;
17          }
18        }
19      }
20    }
21    execInternalOp("checkSendSpot");
22  }
```

Código 41 – Implementação da operação interna enviar vaga do PC.

O método para escolher o motorista que irá conseguir uma vaga é o *getNextDriver* implementado no código 42. Este método seleciona o motorista para receber uma vaga por GC e ordem de chegada. Se o motorista for escolhido apenas por GC pode ser que um motorista nunca receba uma vaga por sempre chegar motoristas com GC maior que ele. Por isso, a cada quantidade de vagas reservada para o motorista por GC uma vaga é reservada por ordem de chegada e evitar que um motorista nunca receba uma vaga.

Neste método é inicializado um objeto chamado *winner* (linha 2), este objeto representa o agente que terá uma vaga reservada para ele. *winner* é inicializado com o nome vazio e com GC menos um. Se o critério para escolher um motorista for por maior GC, escolhe-se o agente com maior pontuação (linhas 3 até 11), em que realiza-se uma interação na lista de motoristas registrados na fila de espera (linha 5). Se o objeto que representa o motorista na na fila de espera tiver valor de GC maior que o objeto *winner* (linha 7), copia-se o objeto do motorista para o *winner* (linha 8) e decrementa em um o número de vagas escolhidas pelo GC (linha 10). Se houver empate no GC entre os motoristas que esperam vaga, o motorista que irá ganhar a vaga é aquele que requisitou primeiro, pois o objeto *winner* só recebe o objeto do motorista se o GC for maior.

Se a escolha for por ordem de chegada (linha 12), é escolhido o primeiro motorista na fila de espera (linha 13) e reinicia o contador *choiceByPointing* para o valor inicial (linha 14). Após obter o objeto que representa o motorista escolhido, remove-se o objeto do agente motorista que ganhou a vaga da fila de espera (linha 16) e retorna o nome do agente que ganhou

a vaga (linha 17).

```
1  public String getNextDriver(){
2    Driver winner = new Driver("", -1);
3    Iterator<Driver> iterDrivers = dbDriReqSpot.iterator();
4    if(choiceByPointing > 0){
5      while (iterDrivers.hasNext()) {
6        Driver driver = iterDrivers.next();
7        if(driver.getTrustDegree() > winner.getTrustDegree())
8          winner = driver;
9      }
10     choiceByPointing--;
11   }
12   else{
13     winner = iterDrivers.next();
14     choiceByPointing = this.PriByDegree;
15   }
16   dbDriReqSpot.remove(winner);
17   return winner.getName();
18 }
```

Código 42 – Implementação do método para escolher o motorista a receber uma vaga.

A operação *permitted* que o PC realiza na cancela tem como parâmetro o nome do motorista que ganhou uma vaga e pode entrar no estacionamento (linha 1). Ela adiciona o nome a lista de motoristas permitidos a entrar no estacionamento (linha 2). A lista *dbPermittedDrivers* registra o nome de todos os motorista que podem entrar no estacionamento. Se um motorista tentar entrar e não estiver na lista a cancela não irá abrir.

```
1 @LINK void permitted(String nameDriver){
2   dbPermittedDrivers.add(nameDriver);
3 }
```

Código 43 – Implementação da operação que registra o motorista permitido a entrar.

5.4 OCUPAR VAGA

Esta é a terceira fase do funcionamento do estacionamento, em que o PC, após reservar a vaga para o motorista, notifica o gerente que o motorista ganhou uma vaga. O gerente envia uma mensagem ao motorista passando as informações necessárias para poder participar da organização e ocupar uma vaga.

5.4.1 Receber Vaga

No MAPS o agente gerente verifica em sua base de crença qual vaga está livre. A primeira vaga vazia que encontrar é atualizada para ocupada e registra o motorista nesta vaga. O gerente realiza a operação de abrir a cancela, envia uma mensagem com o nome da vaga que recebeu e fecha a cancela.

No MAPS-NORMS o agente gerente percebe o sinal *sendSpot* do artefato PC. O sinal *sendSpot* indica que uma vaga foi reservada para um motorista e é necessário avisá-lo que há uma vaga reservada para ele. O código 44 implementa o plano para enviar a vaga para o motorista. O sinal *sendSpot* envia quatro parâmetros: vaga, motorista, grupo e esquema.

A vaga é o nome da vaga que o motorista tem reservada para ele, o motorista é o nome do agente motorista que recebeu uma vaga, o grupo é o nome do grupo que o gerente e o motorista devem fazer para para alocar uma vaga e o esquema é o nome do esquema que irá indicar quais passos devem ser feitos para alocar corretamente uma vaga.

O gerente cria o artefato grupo (linha 3). Em seguida, realiza a operação de adotar o papel *manager* no grupo criado (linha 3) e focaliza no grupo criado (linha 4). Cria o esquema utilizado pelo grupo para ocupar uma vaga (linha 5) e focaliza neste esquema (linha 6). Envia a mensagem *spotInf* para o motorista que ganhou a vaga com as informações: nome da vaga, o nome do papel que deve adotar e o nome do grupo que deve fazer parte (linha 7) e realiza um plano para adicionar o esquema ao grupo (linha 8).

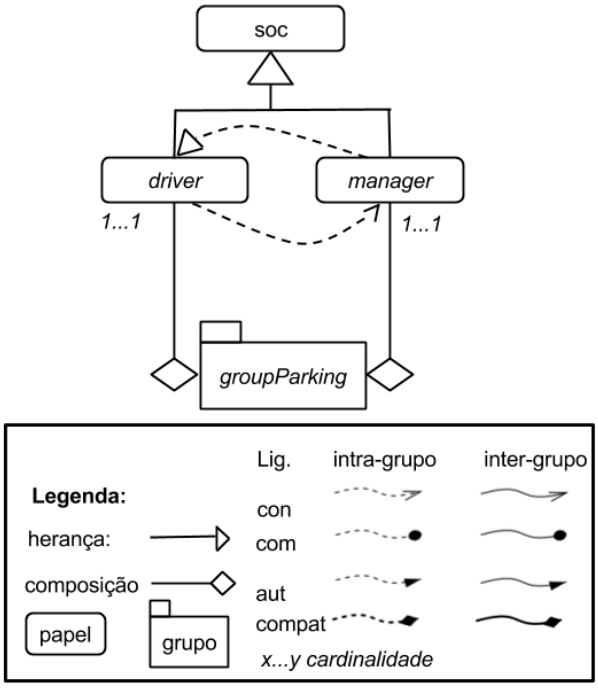
```
1 +sendSpot(Spot, Driver, Group, Scheme)  : true
2   <-  createGroup(Group, groupParking, GrArtId);
3       adoptRole(manager)[artifact_id(GrArtId)];
4       focus(GrArtId);
5       createScheme(Scheme, schemeOccupySpot, SchArtId);
6       focus(SchArtId);
7       .send(Driver, achieve, spotInf(Spot, "driver",
           Group));
8       !addSchemeToGroup(GrArtId, Scheme).
```

Código 44 – Codificação do plano enviar vaga reservada.

A Figura 24 representa o grupo que o gerente instancia para ocupar uma vaga. Existem dois papéis: *driver* e *manager*. Estes papéis fazem parte do grupo *groupParking*. O papel *driver* representa o motorista e o papel *manager* representa o gerente na organização. Num grupo pode existir apenas um agente no papel de *driver*, pois este grupo é responsável por alocar apenas uma vaga e existe apenas um agente no papel de *manager* porque existe apenas um gerente no estacionamento. O *manager* tem a ligação de autoridade sobre o *driver* e o *driver* tem a ligação de

conhecimento para o *manager*. Assim, o gerente tem autoridade sobre o motorista e o motorista apenas sabe que o gerente existe.

Figura 24 – Representação do grupo para ocupar uma vaga.



Fonte: Autoria Própria.

O motorista recebe do gerente as informações para estacionar: nome da vaga, nome do papel e o nome do grupo. O nome da vaga que recebe é a vaga que o PC reservou no estacionamento para o motorista, o papel é aquele que o motorista deve adotar para participar do grupo criado. O nome do grupo representa o grupo que o gerente e o motorista irão atuar para cumprir o objetivo de utilizar a vaga.

O código 45 implementa o plano *spotInf*. O motorista recebe uma mensagem do gerente com três variáveis: o nome da vaga (*Spot*), nome do papel (*Role*) e o grupo (*Group*) (linha 1). O motorista descobre o identificador do artefato grupo (linha 2), focaliza neste artefato (linha 3), realiza a operação no artefato grupo de adotar o papel que recebeu do gerente (linha 4) e adiciona a crença *occupy* com o nome da vaga que foi reservada para ocupar (linha 5).

```
1 +!spotInf(Spot, Role, Group) : true
2   <-  lookupArtifact(Group, IdGr);
3       focus(IdGr);
4       adoptRole(Role);
5       +occupy(Spot).
```

Código 45 – Codificação do plano receber informações sobre a vaga.

5.4.2 Escolher Vaga

No MAPS o motorista não tem autonomia para escolher em qual vaga estacionar, isto restringe a ação do agente motorista. Quando um motorista ocupa uma vaga o SMA não leva em consideração o tempo que um motorista levaria para estacionar. O gerente apenas registra que o motorista ganhou determinada vaga.

No MAPS-NORMS após o motorista ganhar uma vaga o gerente adiciona o esquema para ocupar vaga como responsabilidade do grupo do gerente e do motorista que recebeu a vaga. O plano *addSchemeToGroup* no código 46, implementa a ação do gerente de adicionar o esquema criado ao grupo responsável. Se a formação do grupo está pronta, o gerente realiza o primeiro plano (linha 1), senão realiza o segundo plano (linha 4). A formação do grupo só acontece quando o número mínimo de agentes adotam cada papel do grupo.

```
1 +!addSchemeToGroup(GrArtId, SchemeName) :
     formationStatus(ok)[artifact_id(GrArtId)]
2   <-  addScheme(SchemeName)[artifact_id(GrArtId)].
3
4 -!addSchemeToGroup(GrArtId, SchemeName) : true
5   <-  !addSchemeToGroup(GrArtId, SchemeName).
```

Código 46 – Codificação do plano adicionar esquema ao grupo.

A partir do momento que o esquema é adicionado ao grupo, o gerente e o motorista que estão no grupo são informados pela organização normativa quais as missões que devem ser realizada para finalizar o esquema. O código 47 implementa os planos dos agentes motoristas e gerente para realizarem uma obrigação que está configurada no esquema adicionado ao grupo que eles pertencem.

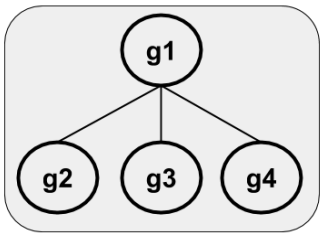
O primeiro plano lida com missões que são obrigações do grupo realizar (linha 1). A condição para o plano ser realizado é o nome do agente no parâmetro Ag recebido ser o mesmo que o agente que percebeu essa obrigação (linha 2), o agente realiza a missão que está destinada para ele (linha 3). O segundo plano lida com a obrigação de realizar um objetivo (linha 5). O plano tem como condição o agente ter o mesmo nome que o parâmetro recebido em Ag e se o objetivo pode ser feito (What=satisfied(Sch,Goal) ou já foi concluído (What = done(Sch,Goal,Ag)) (linha 6). O agente realiza o objetivo (linha 7) e depois marca o objetivo como atingido (linha 8).

```
1 +obligation(Ag,Norm,committed(Ag,Mission,Sch),TimeConst)
2   :    .my_name(Ag)
3   <-   commitMission(Mission)[artifact_name(Sch), wid(W)].
4
5 +obligation(Ag,Norm,What,TimeConst)[artifact_id(ArtId)]
6   :    .my_name(Ag) & (What = satisfied(Sch,Goal) | What
        = done(Sch,Goal,Ag))
7   <-   !Goal[scheme(Sch)];
8        goalAchieved(Goal)[artifact_id(ArtId)].
```

Código 47 – Codificação dos agentes realizando uma obrigação imposta pela organização.

A Figura 25 representa a ordem estabelecida dos objetivos para o esquema ser realizado corretamente. O primeiro objetivo é atravessar a cancela para o motorista entrar no estacionamento (g2), o segundo objetivo é o motorista ocupar uma vaga (g3), o terceiro é desocupar a vaga que ocupou (g4) e o quarto objetivo é o gerente saber que uma vaga fora desocupada registrando a alocação (g1).

Figura 25 – Representação do esquema para alocação de uma vaga.



Obj	Nome
g1	Vaga Utilizada
g2	Atravessar Cancela
g3	Ocupar Vaga
g4	Desocupar Vaga

Fonte: Autoria Própria.

Cada objetivo no esquema ocupar vaga é realizado pelo agente responsável a partir de um plano. O objetivo g2 é realizado pelo papel *driver*, o plano para realizar este objetivo é implementado no código 48. O plano *crossGate* não tem condição para ocorrer, basta ser acionado (linha 1). O motorista realiza as operações *openGate* (linha 2) e *crossGate* (linha 3). A operação *openGate* é realizada pelo motorista para abrir a cancela e a operação *crossGate* é realizada para o agente ter acesso ao estacionamento. Caso alguma operação falhe em sua execução, é realizado um plano de contenção que contornar o problema (linha 5). O segundo plano espera um segundo (linha 6) e tenta realizar o plano *crossGate* novamente (linha 7).

```
1 +!crossGate
2   <-   openGate;
3        crossGate.
4
```

```
5 -!crossGate
6   <-  .wait(1000);
7       !crossGate.
```

Código 48 – Codificação do plano atravessar vaga.

A operação *openGate* é responsável por abrir a cancela. A operação tem uma condição guarda que não permite que a operação seja realizada caso a condição não seja verdadeira. A condição implementada em 49 chamada *guardClose* retorna se a cancela está fechada ou não. A operação retorna o valor oposto a propriedade *isOpen* (linha 2), assim retorna um valor verdadeiro caso esteja fechada e falso caso esteja aberta.

```
1 @GUARD boolean guardClose() {
2   return !getObsProperty("isOpen").booleanValue();
3 }
```

Código 49 – Implementação da operação guarda que *guardClose*.

A operação para abrir cancela é apresentada no código 50. A operação *openGate* identifica qual agente realizou a operação (linha 2), verifica se este motorista está registrado na lista de motoristas permitidos a entrar no estacionamento (linhas 3 e 4). Se o motorista estiver registrado (linha 6), a cancela é aberta (linha 7). O motorista é removido da lista de motoristas permitidos a entrar no estacionamento (linha 8), para evitar que o motorista consiga acessar novamente o estacionamento sem ter uma vaga reservada. Após encontrar o motorista na lista de motoristas permitidos o laço é finalizado (linha 9). Caso a vaga não encontre o motorista na lista de motoristas permitidos a entrar no estacionamento a vaga se manterá fechada e o motorista não conseguirá entrar no estacionamento.

```
1  @OPERATION(guard = "guardClose") void openGate(){
2    String driver = getCurrentOpAgentId().toString();
3    Iterator<String> iterdbPermittedDrivers = drivers.iterator();
4    while(iterdbPermittedDrivers.hasNext()){
5      String regDriver = iterdbPermittedDrivers.next();
6      if(driver.equals(regDriver)){
7        getObsProperty("openGate").updateValue(true);
8        drivers.remove(driver);
9        break;
10     }
11   }
12 }
```

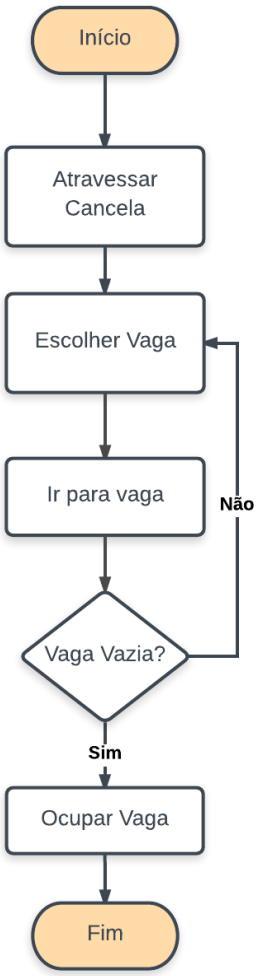
Código 50 – Implementação da operação para abrir a cancela.

Após atravessar a cancela e cumprir o objetivo *crossGate*, o agente realizará o objetivo *occupySpot*. Para cumprir o objetivo de ocupar uma vaga, o agente motorista tem autonomia para escolher qual vaga ele irá ocupar, desta forma o motorista pode escolher outra vaga para estacionar que não seja a que foi reservada para ele. Devido ao motorista poder ocupar uma vaga que não esteja reservada à ele, o mesmo precisa ter uma estratégia para ocupar uma vaga no estacionamento levando em consideração que outro motorista pode estacionar em sua vaga. Por isso, é necessário um mecanismo para o agente conseguir encontrar uma vaga livre mesmo se a sua estiver ocupada.

Para ocupar uma vaga, é realizado a sequência de passos apresentada no fluxograma da Figura 26. O motorista atravessa a cancela, escolhe a vaga para estacionar, vai até a vaga e verifica se a vaga está vazia. Se estiver vazia, o motorista ocupa a vaga. Senão volta para a ação de escolher vaga.

Apesar do motorista poder chegar na vaga e está ocupada, existe a garantia que o motorista irá encontrar uma vaga. O PC somente libera a entrada de um motorista no estacionamento, se houver reserva de vaga para o motorista. Assim, o número limite de motoristas no estacionamento é o número de vagas, desta forma se o motorista entrar no estacionamento, existirá uma vaga que ele pode estacionar.

Figura 26 – Sequência de ações para estacionar.



Fonte: Autoria Própria.

A escolha da vaga pelo motorista é realizada de acordo com seu comportamento. Foram implementados quatro comportamentos que um motorista pode assumir para tentar ocupar uma vaga. Foram implementados para verificar como o seu comportamento afeta o funcionamento do SMA e como isso impacta no GC atribuído a eles.

1. RTD (***R**eserved **T**o the **D**river*) - o motorista escolhe ocupar a vaga que recebeu do gerente.
2. NTD (***N**ot reserved **T**o **D**river*) - o motorista dá prioridade para escolher vagas que não tenha recebido do gerente.
3. CSR (***C**hoice **S**pot **R**andomly*) - o motorista escolhe uma vaga aleatoriamente para ocupar.
4. FSF (***F**irst **S**pot **F**ree*) - o motorista escolhe ocupar a primeira vaga livre que encontrar.

Cada tipo de comportamento possui uma estratégia para escolher qual vaga ocupar, a estratégia de escolha de cada motorista é implementada no plano *occupySpot*.

O motorista do tipo de comportamento RTD inicialmente tenta ocupar a mesma vaga que recebeu do gerente. Seja *x* a vaga que o motorista recebeu do gerente e *n* a quantidade de vagas do estacionamento. O motorista RTD tenta ocupar uma vaga na ordem: *x*, *x* + 1, *x* + 2, ..., *n*. Quando chega no final do estacionamento sem conseguir estacionar verifica as vagas 1, 2, ..., *x* - 1, para estacionar, assim a procura é feita de forma circular.

O quadro 3 mostra um conjunto de cinco vagas e a ordem que o agente motorista escolhe qual vaga tentará estacionar. O motorista recebe a vaga três do gerente e tenta estacionar inicialmente nela. Caso a vaga esteja ocupada, este motorista tenta ocupar a vaga seguinte, e assim sucessivamente.

Quadro 3 – Estratégia de escolha do comportamento RTD

Vaga	1	2	**3**	4	5
Ordem de escolha	4	5	1	2	3

Fonte: Autoria Própria

O código 51 mostra o plano utilizado por este agente para escolher qual vaga ocupar. O plano tem como condição a crença *behaviour* tiver valor RTD e existir a crença *occupy* (linha 1). O plano inicia com a execução do plano *arriveParking* (linha 2), em seguida converte o nome da vaga que fora recebida como inteiro para o tipo String (linha 3), recupera o identificador da vaga que recebeu (linha 4), focaliza a vaga (linha 5) e realiza o plano *tryOccupy*.

```
1 +!occupySpot : occupy(SpotInt) & behaviour("RTD")
2   <-  !arriveParking(SpotInt);
3       .term2string(SpotInt, SpotName);
4       lookupArtifact(SpotName, SpotId);
5       focus(SpotId);
6       !tryOccupy.
```

Código 51 – Plano chegar na vaga com comportamento RTD.

O motorista do tipo de comportamento NTD irá priorizar escolher todas as vagas existentes antes de tentar ocupar a vaga que recebeu do gerente. Seja *x* a vaga que o motorista recebeu do gerente e *n* a quantidade de vagas do estacionamento, o motorista NTD tenta ocupar uma vaga na ordem: *x* + 1, *x* + 2, ..., *n*. Caso chega no final do estacionamento sem encontrar uma vaga, procura as vagas que estão no início do estacionamento 1, 2, ..., até *x*.

O quadro 4 mostra um conjunto de cinco vagas e a ordem que o agente motorista escolhe qual vaga tentará estacionar. O motorista recebe a vaga três do gerente e tenta estacionar inicialmente na vaga quatro. Caso a vaga esteja ocupada, este motorista tenta ocupar a vaga cinco, e assim sucessivamente.

Quadro 4 – Estratégia de escolha do comportamento NTD

Vaga	1	2	**3**	4	5
Ordem de escolha	3	4	5	1	2

Fonte: Autoria Própria

O código 52 implementa o plano *occupySpot* do motorista de comportamento NTD. O plano tem como condição a existência da crença *occupy*, a crença *behaviour* com valor NTD, a existência da crença *numTotalSpots* e a vaga a ser escolhida não ser a última vaga (SpotInt < NTS).

O motorista escolhe a vaga seguinte que recebeu originalmente (linha 2). Executa o plano para chegar nessa vaga (linha 3). Atualiza a crença da vaga desejada (linha 4), converte o nome da vaga de inteiro para o tipo *String* (linha 5). Recupera o identificador da vaga (linha 6). Focaliza na vaga (linha 7) e realiza o plano *tryOccupy* (linha 8). Então, se a quantidade total de vagas, ou seja, o valor de NTS é igual a 20 e a vaga atual é 20 ele volta para a vaga 1, como se a busca fosse uma lista circular. Realiza o segundo plano (linha 10), para reiniciar a escolha da vaga para 1. Assim, o motorista não tentara estacionar numa vaga que não existe.

```
1  +!occupySpot : occupy(SpotInt) & behaviour("NTD") &
       numTotalSpots(NTS) & SpotInt < NTS
2    <-  NewSpotInt = SpotInt + 1;
3        !arriveParking(NewSpotInt);
4        -+occupy(NewSpotInt);
5        .term2string(NewSpotInt, SpotName);
6        lookupArtifact(SpotName, SpotId);
7        focus(SpotId);
8        !tryOccupy.
9
10 +!occupySpot : occupy(SpotInt) & behaviour("NTD") &
       numTotalSpots(NTS) & SpotInt == NTS
11   <-  NewSpotInt = 1;
12       !arriveParking(NewSpotInt);
13       -+occupy(NewSpotInt);
14       .term2string(NewSpotInt, SpotName);
15       lookupArtifact(SpotName, SpotId);
16       focus(SpotId);
17       !tryOccupy.
```

Código 52 – Plano para chegar na vaga com comportamento NTD.

O motorista do tipo de comportamento CSR irá escolher inicialmente uma vaga aleatória para estacionar. Seja *x* a vaga que o motorista recebeu do gerente, ele escolhe a vaga *x'*,

x' = *random*. O valor de *x'* é gerado entre as vagas 1, ..., *qtde*. Assim, o motorista pode acabar escolhendo a mesma vaga que recebeu do gerente, uma vaga ocupada ou uma vaga livre.

O quadro 5 mostra um conjunto de cinco vagas e a ordem que o agente motorista escolhe qual vaga tentará estacionar. Por exemplo, apesar do motorista receber a vaga três, o motorista aleatoriamente pode escolher a vaga dois. Caso a vaga esteja ocupada, este motorista tenta ocupar a vaga seguinte previamente escolhida. Neste exemplo a próxima vaga é a três, e assim sucessivamente. Se chegar na última vaga e estiver ocupada, reinicia a procura ao tentar ocupar a primeira vaga do estacionamento.

Quadro 5 – Estratégia de escolha do comportamento CSR

Vaga	1	2	**3**	4	5
Ordem de escolha	5	1	2	3	4

Fonte: Autoria Própria

O código 53 implementa o plano *occupySpot* do motorista de comportamento CSR. O plano tem como condição a existência das crenças: *occupy*, *numTotalSpots* e a crença *behaviour* com valor CSR.

A vaga a ser escolhida é obtida apartir de um valor inteiro aleatório entre o número um e o número total de vagas (linha 2). Realiza-se o plano para chegar na vaga que fora escolhida (linha 3). A crença sobre qual vaga estacionar é atualizada (linha 4). O nome da vaga é convertido do tipo inteiro para o tipo *String* (linha 5), o identificador da vaga é obtido (linha 6), o agente focaliza na vaga (linha 7) e realiza o plano *tryOccupy* (linha 8).

```
1 +!occupySpot : occupy(SpotInt) & numTotalSpots(NTS) &
     behaviour("CSR")
2   <-  NewSpotInt = math.ceil(math.random(NTS));
3       !arriveParking(NewSpotInt);
4       -+occupy(NewSpotInt);
5       .term2string(NewSpotInt, SpotName);
6       lookupArtifact(SpotName, SpotId);
7       focus(SpotId);
8       !tryOccupy.
```

Código 53 – Plano para chegar na vaga com comportamento CSR.

O motorista do tipo de comportamento FSF irá tentar ocupar a primeira vaga que estiver livre. Seja *x* a vaga que o motorista recebeu do gerente e *n* a quantidade de vagas do estacionamento, o motorista FSF tenta ocupar uma vaga na ordem: 1, 2, 3, ..., n.

O quadro 6 mostra um conjunto de cinco vagas e a ordem que o agente motorista escolhe estacionar. O motorista recebe a vaga três e escolhe tentar ocupar a primeira vaga. Caso a vaga esteja ocupada, este motorista tenta ocupar a vaga seguinte, e assim sucessivamente.

Quadro 6 – Estratégia de escolha do comportamento FSF

Vaga	1	2	**3**	4	5
Ordem de escolha	1	2	3	4	5

Fonte: Autoria Própria

O código 54 implementa o plano *occupySpot* do motorista com comportamento FSF. Este plano tem como condição a existência da crença *occupy* e *behaviour* com valor valor FSF. O plano atribui zero para a crença *occupy* e aciona o plano *occupySpotLinear* (linha 2).

O plano *occupySpotLinear* (linha 5) acontece com a condição: existência da crença *occupy* (linha 5). Uma variável é criada com o nome da vaga que ele tentará ocupar chamada NewSpotInt e lhe é atribuído o valor SpotInt incrementado em um (linha 6), o plano para o motorista chegar na vaga é realizado (linha 7), atualiza-se a crença de qual vaga tentará ocupar (linha 8). Converte o nome da vaga de inteiro para o tipo *String* (linha 9). Recupera o identificador da vaga (linha 10), focaliza no artefato vaga (linha 11) e realiza o plano para tentar ocupar a vaga (linha 12).

```
1  +!occupySpot : occupy(SpotInt) & behaviour("FSF")
2    <-  -+occupy(0);
3        !occupySpotLinear.
4
5  +!occupySpotLinear : occupy(SpotInt)
6    <-  NewSpotInt = SpotInt + 1;
7        !arriveParking(NewSpotInt);
8        -+occupy(NewSpotInt);
9        .term2string(NewSpotInt, SpotName);
10       lookupArtifact(SpotName, SpotId);
11       focus(SpotId);
12       !tryOccupy.
```

Código 54 – Plano para chegar na vaga com comportamento FSF.

Após escolher uma vaga, os motoristas irão chegar na vaga para verificar se a mesma está vazia. Cada vaga tem uma localização única no estacionamento, a localização da vaga é definida pelo nome. Por exemplo, a vaga 1 tem a localização 1, a vaga 2 a localização 2, e assim por diante até a quantidade total de vagas. Quando um motorista vai até uma vaga, ele demora para chegar na vaga de acordo com a localização dela. Para chegar na vaga 1, o motorista leva 1 segundo, na vaga 2, são 2 segundos, na vaga 3, são 3 segundos e assim por diante até a quantidade total de vagas. O plano para simular a chegada do motorista numa vaga escolhida é o *arriveParking*.

O código 55 mostra o cálculo para qualquer motorista chegar na primeira vaga que ten-

tará estacionar. Esse plano tem como condição a crença *lastSpot* ter valor nulo, pois o motorista não tentou estacionar em nenhuma vaga antes. Neste plano atribui-se o nome da vaga para a variável que tem a distância até ela (linha 2) e o motorista espera este valor em segundos (linha 3), simulando que o motorista está indo até a vaga e por fim, atualiza a crença *lastSpot* registrando qual foi a última vaga que o motorista chegou (linha 4).

```
1 +!arriveParking(SpotInt) : lastSpot(null)
2   <-  TimeToArrive = SpotInt;
3       .wait(TimeToArrive * 1000);
4       -+lastSpot(SpotInt).
```

Código 55 – Plano para o motorista chegar na vaga selecionada.

Caso o motorista tente estacionar numa vaga que está ocupada, o motorista escolhe a próxima vaga para ir e vai até a outra vaga para verificar se está vazia. Para calcular o tempo de chegada nessa nova vaga é levado em consideração a posição atual do motorista. Desta forma a distância para a próxima vaga não é diretamente a localização da vaga, mas sim o módulo da diferença entre a distância da vaga que o motorista tentou estacionar para a próxima que foi escolhida.

Se o motorista tentar estacionar numa vaga que está mais próxima da entrada do que a vaga anterior é realizado o primeiro plano do código 56, senão é realizado o segundo plano.

No plano para chegar na vaga mais próxima da entrada (linha 1), o tempo para chegar na vaga é calculado pela distância da vaga atual menos a distância vaga anterior (linha 2), espera o tempo calculado (linha 3) e atualiza a última vaga que se tentou estacionar (linha 4).

O segundo plano é para chegar na vaga que está mais longe da entrada (linha 6), o tempo para chegar na vaga é calculado pela distância da vaga anterior menos a distância da vaga atual (linha 7), espera o tempo calculado (linha 8) e atualiza a última vaga que se tentou estacionar (linha 9).

```
1 +!arriveParking(SpotInt) : lastSpot(LS) & (LS < SpotInt)
2   <-  TimeToArrive = SpotInt - LS;
3       .wait(TimeToArrive * 1000);
4       -+lastSpot(LS).
5
6 +!arriveParking(SpotInt) : lastSpot(LS) & (LS > SpotInt)
7   <-  TimeToArrive = LS - SpotInt;
8       .wait(TimeToArrive * 1000);
9       -+lastSpot(LS).
```

Código 56 – Plano chegar na vaga.

Para ocupar uma vaga no MAPS o motorista apenas adiciona a crença que ocupa uma vaga, isso não permite ao motorista poder escolher qual vaga quer estacionar. No MAPS apartir da negociação da vaga, o motorista deve necessariamente estacionar na vaga atribuída pelo gerente. Não há possibilidade do agente motorista tentar estacionar em outra vaga.

Por outro lado, no MAPS-NORMS o agente motorista tem uma certa autonomia, pois ocupa uma vaga sem a dependência do gerente. As vagas são artefatos que os motoristas interagem diretamente. O motorista realiza a ocupação de uma vaga por meio de uma operação no artefato vaga, a partir do plano 57. Para realizar a ocupação de uma vaga, a vaga deve estar vazia, ou seja, naquele momento não pode haver outro motorista a ocupando, porém pode estar reservada a outro motorista. Assim, para realizar a operação *occupy* é necessário que a propriedade *isFree* tenha valor verdadeiro e o nome da vaga a ser ocupada na crença *occupy* (linha 1). O nome da vaga é convertido do tipo inteiro para o tipo *String* na variável *SpotName* (linha 2), recupera o identificador da vaga (linha 3) e realiza a operação *occupy* na vaga (linha 4).

```
1 +!tryOccupy : isFree(true) & occupy(Spot)
2   <-  .term2string(Spot, SpotName);
3       lookupArtifact(SpotName, IdSpot);
4       occupy[artifact_id(IdSpot)].
```

Código 57 – Codificação do plano *tryOccupy*.

Se a vaga que os motoristas de comportamento RTD ou CSR escolheram estiverem ocupadas, os motoristas utilizam o comportamento de escolha de vaga NTD. Por exemplo, se o motorista tenta ocupar a vaga dez e ela está ocupada o mesmo tenta ocupar a vaga onze e assim sucessivamente. O código 58 apresenta o plano utilizado pelo motorista RTD e CSR para encontrar outra vaga para estacionar.

A condição para que o plano aconteça é a vaga estar ocupada e a crença *behaviour* com valor RTD ou CSR (linha 1). O motorista atualiza a crença *behaviour* para NTD e assim adota o comportamento NTD (linha 2). Adiciona a crença *backTo* com o valor do comportamento anterior, para quando o motorista ocupar uma vaga voltar a ter o comportamento original (linha 3). Por fim, realiza o plano *occupySpot* novamente (linha 4).

```
1 +!tryOccupy : isFree(false) & behaviour(B) & ((B ==
    "RTD") | (B == "CSR"))
2   <-  -+behaviour("NTD");
3       +backTo(B);
4       !occupySpot.
```

Código 58 – Codificação do escolher outra vaga do motorista RTD e CSR.

Quando encontrar uma vaga livre com o comportamento NTD, os motoristas que tinham comportamento RTD ou CSR, retornam ao comportamento original. O plano *tryOccupy* do código 59 implementa o plano que tem como condição a vaga estar livre, ter a crença *occupy* e a crença *backTo* carregando o comportamento anterior (linha 1).

O comportamento do agente é atualizado com o comportamento anterior (linha 2). Remove-se a crença *backTo* que não será mais utilizada (linha 3). Converte o nome da vaga do tipo inteiro para o tipo *String* (linha 4). Recupera o identificador da vaga (linha 5) e realiza a operação para ocupar a vaga (linha 6).

```
1 +!tryOccupy : isFree(true) & occupy(Spot) & backTo(BT) &
     ((BT == "RTD") | (BT == "CSR"))
2   <-  -+behaviour(BT);
3       -backTo(_);
4       .term2string(Spot, SpotName);
5       lookupArtifact(SpotName, IDSPOT);
6       occupy[artifact_id(IDSPOT)].
```

Código 59 – Codificação do escolher outra vaga do motorista RTD e CSR.

Se a vaga que o motorista FSF tentar estacionar estiver ocupada, o motorista realiza o plano *tryOccupy* implementado no código 60. O plano tem como condição a vaga estar ocupada e existir o comportamento FSF (linha 1). Neste plano o agente irá continuar procurando com sua estratégia inicial (linha 2).

```
1 +!tryOccupy : isFree(false) & behaviour("FSF")
2   <-  !occupySpotLinear.
```

Código 60 – Codificação do escolher outra vaga do motorista FSF.

Se a vaga que o motorista NTD escolheu para ocupar estiver ocupada, o motorista realiza o plano *occupySpot* do código 61, em que realiza o plano *occupySpot* novamente (linha 2).

```
1 +!tryOccupy : isFree(false) & behaviour("NTD")
2   <-  !occupySpot.
```

Código 61 – Codificação para escolher outra vaga do comportamento NTD.

5.4.3 Detectar Ocupação da Vaga

A vaga tem um operador guarda que garante que a operação somente seja realizada se estiver livre. Para isso o operador guarda retorna o valor da propriedade *isFree* (linha 2).

Quando a vaga está livre e a operação é realizada o artefato atribui para a variável *namePC* o nome do PC que a vaga se comunica (linha 6). Atribui a variável *nameDriver* o nome do motorista que realizou a operação de ocupar (linha 7). Atualiza a propriedade da vaga para falso (linha 8). Atualiza o nome do motorista que ocupa a vaga (linha 9) e realiza a operação *spotOccupied* no artefato PC (linha 11) passando como parâmetro o nome da vaga que fora ocupada.

```
1  @GUARD boolean spotFree() {
2    return getObsProperty("isFree").booleanValue();
3  }
4
5  @OPERATION(guard = "spotFree") void occupy() {
6    String namePC = getObsProperty("PC").getValue().toString();
7    String nameDriver =
       this.getCurrentOpAgentId().getAgentName();
8    getObsProperty("isFree").updateValue(false);
9    getObsProperty("driverOcc").updateValue(nameDriver);
10     execLinkedOp(lookupArtifact(namePC), "spotOccupied",
11         Integer.parseInt(this.getId().getName()));
12 }
```

Código 62 – Implementação da operação ocupar vaga.

O PC é avisado pela vaga que a mesma fora ocupada via a operação *spotOccupied*. A operação é implementada no código 63. Nesta operação o PC recebe um valor inteiro com o nome da vaga (linha 1). O PC obtém o nome do motorista a partir do nome do agente que realizou a operação de ocupar a vaga (linha 2), realiza uma interação nas vagas existentes (linhas 3 e 4), verifica qual objeto de vaga registrado representa a vaga que fora ocupada (linha 6), verifica se a vaga estava reservada para o motorista (linha 7). Se estiver reservada é realizado o método *updateTrustDriver* passando como parâmetro o nome do motorista e o parâmetro POC que representa que o motorista ocupou a vaga reservada a ele (linha 8).

Se o motorista não ocupou uma vaga reservada para ele (linha 10) a vaga que ele ocupou tem a propriedade *isFree* atualizada para falsa (linha 11), atribui o nome do motorista que a ocupa (linha 12), atualiza a vaga que estava reservada para o motorista (linha 13). Se o motorista ocupou uma vaga sem reserva (linha 14) é realizado o método *updateTrustDriver* passando como parâmetro o nome do motorista e POE que representa que o motorista ocupou indevidamente

uma vaga que não estava reservada (linha 15). Senão o motorista ocupou uma vaga reservada para outro motorista (linha 16), o método *updateTrustDriver* é realizado passando como parâmetro o nome do motorista e POR que representa que o motorista ocupou indevidamente uma vaga que estava reservada a outro motorista (linha 17). Após achar a vaga, atualizar o GC de acordo com sua ação, realiza-se a parada da interação (linha 20).

```
1  @LINK void spotOccupied(int idSpot){
2    String nameDriver =
        this.getCurrentOpAgentId().getAgentName();
3    Iterator<SpotClass> iterSpots = dbSpots.iterator();
4    while(iterSpots.hasNext()){
5      SpotClass spot = iterSpots.next();
6      if(spot.getId() == idSpot){
7        if(spot.getReservedDriver().equals(nameDriver)){
8          execInternalOp("updateTrustDriver", nameDriver,
              POC);
9        }
10       else{
11         spot.setIsFree(false)
12         spot.setReservedDriver(nameDriver);
13         updateOtherSpot(nameDriver);
14         if(spot.getReservedDriver().equals("")){
15           execInternalOp("updateTrustDriver",
                nameDriver, POE);
16         } else{
17           execInternalOp("updateTrustDriver",
                nameDriver, POR);
18         }
19       }
20       break;
21     }
22   }
23 }
```

Código 63 – Codificação da operação *spotOccupied*.

A operação que atualiza a vaga que o motorista não ocupou e estava reservada, é implementada no código 64. O método realiza uma interação nas vagas registradas (linhas 2 e 3). Encontra a vaga que tem o motorista reservado (linha 5). Atualiza as propriedades atribuindo verdadeiro para a propriedade que indica que a vaga está livre e vazio para o nome do motorista

que ocupa a vaga.

```
1  public void updateOtherSpot(String driver){
2    Iterator<SpotClass> iterSpots = dbSpots.iterator();
3    while(iterSpots.hasNext()){
4      SpotClass spotToUpdate = iterSpots.next();
5      if(spotToUpdate.getReservedDriver().equals(driver)){
6        spotToUpdate.setReservedDriver("");
7        spotToUpdate.setIsFree(true);
8      }
9    }
10 }
```

Código 64 – Método *updateOtherSpot*.

A operação interna *updateTrust* irá atualizar o GC que o PC tem do motorista. O código 65 mostra que é recebido o nome do motorista *driver* e o valor que será acrescentado em seu GC antigo *degree*. O objeto que representa o motorista é obtido via o método *getDriver*. Neste objeto o novo GC é calculado (linha 3), se ultrapassar o limite estabelecido de 999 (linha 4) o grau se torna 999 (linha 5), senão se o GC alcança um valor abaixo de 0 (linha 7), o GC recebe zero (linha 8) e por fim, o objeto do motorista recebe o novo GC (linha 10).

```
1  @INTERNAL_OPERATION void updateTrustDriver(String
      nameDriver, int updateTrusDegree){
2    Driver driver = GetDriver(nameDriver);
3    int newTrustDegree = driver.getTrustDegree() +
       (updateTrusDegree * CoefficientOccupancy);
4    if(newTrustDegree > 999){
5      newTrustDegree = 999;
6    }
7    else if(newTrustDegree < 0) {
8      newTrustDegree = 0;
9    }
10   driver.setTrustDegree(newTrustDegree);
11 }
```

Código 65 – Codificação da operação interna atualizar GC dos motoristas.

O quadro 7 apresenta as equações utilizadas para calcular o novo grau de confiança de um motorista. A equação realiza a soma do grau de confiança antigo (GCA) com a multiplicação entre o valor que recebe de sua ação vezes o Nível de Ocupação do Estacionamento (NOE).

Existem três níveis de restrição: alto, normal e baixo. O nível de restrição alto atribui maior punição e bonificação ao motorista do que em relação aos níveis normal e baixo. No nível normal é atribuído um valor menor que o do alto para a bonificação e a punição, e o nível baixo tem o menor valor para bonificação e punição.

A ação de ocupar uma vaga reservada ao motorista (*Reserved To the Driver* - RTD) gera uma bonificação, no nível de restrição alto tem valor 20, no normal 10 e no baixo 2. Estes valores foram escolhidos por representarem uma porcentagem da pontuação máxima que um motorista pode conseguir: 20 é 2% de mil, 10 é 1% e 2 é 0.2%.

A ação de ocupar uma vaga vazia não reservada a algum motorista (*Not Reserved to Anyone* - NRA) gera uma punição, no nível de restrição alto tem valor -4, no normal -2 e no baixo não sofre nenhuma alteração. Estes valores foram escolhidos por representarem uma porcentagem da pontuação máxima que um motorista pode conseguir: 4 é 0.4% de mil, 2 é 0.2%.

A ação de ocupar uma vaga vazia reservada a outro motorista (*Reserved To Another* - RTA) gera uma punição, no nível de restrição alto tem valor -20, no normal -10 e no baixo -2. Estes valores foram escolhidos por representarem uma porcentagem da pontuação máxima que um motorista pode conseguir: 20 é 2% de mil, 10 é 1% e 2 é 0.2%.

Quadro 7 – Tipos de Restrição

Nível de Restrição	RTD	NRA	RTA
Alto	NGC = GCA + (20 * NOE)	NGC = GCA + (-4 * NOE)	NGC = GCA + (-20 * NOE)
Normal	NGC = GCA + (10 * NOE)	NGC = GCA + (-2 * NOE)	NGC = GCA + (-10 * NOE)
Baixo	NGC = GCA + (2 * NOE)	NGC = GCA	NGC = GCA + (-2 * NOE)

Fonte: Autoria Própria

O NOE é determinado seguindo o quadro 8. Este nível de ocupação é utilizado para potencializar a bonificação ou a punição que um motorista recebe. Assim, a nova pontuação será de acordo com o estado do estacionamento, ou seja, quando o estacionamento está vazio e o motorista não utiliza a vaga que recebe do gerente, tem impacto diferente do que se estivesse lotado, o que permite flexibilidade em contextos menos urgentes.

Além de potencializar a pontuação dos motoristas, o nível de ocupação do estacionamento define quantas vagas serão enviadas utilizando o critério de maior GC. Quanto mais vagas estiverem ocupadas, mais vagas serão enviadas para motoristas com a pontuação melhor.

Quadro 8 – Nível de Ocupação

Nível de Ocupação	% de vagas ocupadas
1	>= 0% e < 30%
2	>= 30% e < 70%
3	>= 70% e <= 100%

Fonte: Autoria Própria

O método do PC chamado *updateLevelOc* atualiza o nível de ocupação e é implemen-

tado no código 66. Nesta operação é obtido o percentual de vagas ocupadas a partir do número total de vagas existentes e o percentual de vagas ocupadas (linha 4). Se o percentual de vagas ocupadas for maior ou igual a setenta (linha 5), o valor do nível de ocupação é um (linha 6) e o número de vagas distribuídas a partir do GC é cinco porcento do total das vagas do estacionamento (linha 7). Senão se o valor for menor que setenta e ser maior ou igual a trinta (linha 8), o valor do nível de ocupação é dois (linha 9) e o número de vagas distribuídas a partir do GC é dez porcento do total de vagas do estacionamento (linha 10). Senão se o valor for menor que trinta (linha 11), o valor do nível de ocupação é três (linha 12) e o número de vagas distribuídas a partir do GC é quinze porcento do total de vagas do estacionamento (linha 13).

Os valores utilizados para enviar vaga a partir do GC foram escolhidas por serem representativas em número de vagas e proporcionais ao estado do estacionamento, porém não são valores muito altos que pudessem deixar um motorista com pontuação muito baixa esperando um tempo muito alto por vaga.

```
1  public void updateLevelOc(){
2    ObsProperty PNTS = getObsProperty("numTotalSpots");
3    ObsProperty PPOS = getObsProperty("percOccuSpot");
4    PPOS.updateValue(100 - ((numFreeSpots * 100) /
       PNTS.intValue()));
5    if(PPOS.doubleValue() >= 70){
6      this.LevelOccupancy = 1;
7      this.PriorityByDegree = (PNTS.intValue() * 5) / 100;
8    } else if(PPOS.doubleValue() < 70 &&
       PPOS.doubleValue() >= 30){
9      this.LevelOccupancy = 2;
10     this.PriorityByDegree = (PNTS.intValue() * 10) / 100;
11   } else if(PPOS.doubleValue() < 30){
12     this.LevelOccupancy = 3;
13     this.PriorityByDegree = (PNTS.intValue() * 15) / 100;
14   }
15 }
```

Código 66 – Codificação da operação altera o nível de ocupação.

5.5 DEIXAR ESTACIONAMENTO

Esta seção contém a fase final do uso do estacionamento, em que o motorista desocupa a vaga e a própria vaga avisa o PC que está desocupada.

5.5.1 Desocupar Vaga

No MAPS, o motorista desocupa a vaga apenas removendo a crença que ocupa uma vaga e envia uma mensagem ao gerente avisando que saiu da vaga. O gerente decrementa o número de vagas ocupadas, atualiza a crença sobre o estacionamento estar cheio para falso e atualiza a crença da vaga para vazia e retira o agente do SMA, e em seguida, verifica se existe algum motorista na fila esperando.

No MAPS-NORMS, o motorista não depende do gerente para desocupar a vaga, o motorista realiza a operação de desocupar a vaga diretamente no artefato vaga. O motorista realiza o plano *vacateSpot* do código 67. O plano ocorre com as condições: existir a crença *occupy* e *timeOS*.

A crença *timeOS* representa o tempo o motorista ocupará uma vaga (linha 2). Converte o nome da vaga do tipo inteiro para o tipo *String* (linha 3). Remove a crença, se existir, *lastSpot* (linha 4). Recupera o identificador do artefato vaga (linha 5), realiza a operação desocupar vaga neste artefato (linha 6), interrompe o foco que tinha sobre o artefato vaga (linha 7), remove a crença da vaga que ocupava (linha 8) e realiza o plano de ocupar vaga novamente (linha 9).

```
1 +!vacateSpot : occupy(SpotInt) & timeOS(TOS)
2   <-  .wait(TOS);
3       .term2string(SpotInt, SpotName);
4       -lastSpot(_);
5       lookupArtifact(SpotName, SpotId);
6       vacate[artifact_id(SpotId)];
7       stopFocus(SpotId);
8       -occupy(SpotInt);
9       !requestSpot.
```

Código 67 – Codificação do plano *vacateSpot*.

No plano *requestSpot*, se o motorista ainda quiser utilizar o estacionamento, realiza a operação de requisitar vaga para a cancela, caso contrário, não requisita e não utiliza mais o estacionamento.

5.5.2 Detectar Desocupação da Vaga

No MAPS não existe a desocupação da vaga, é uma crença para os agentes gerente e o motorista. O gerente só atualiza a vaga como desocupada se o motorista avisar que saiu da vaga. Assim, se o motorista sair da vaga e não avisar o gerente, a vaga estará livre, porém o gerente

não pode enviar para outro agente por não ter essa informação.

No MAPS-NORMS não é necessário o motorista avisar ao gerente que está saindo da vaga. O motorista realiza a operação de desocupar a vaga e a própria vaga avisa para o PC que não existe mais motorista a ocupando. O código 68 implementa a operação deixar vaga que o motorista realiza no artefato vaga. O nome da vaga é atribuído a variável *Var_NameSpot* (linha 2) e o nome do PC é atribuído na variável *Var_NamePC* (linha 3). Atualiza as propriedades *isFree* e *driverOcc* para verdadeiro e para vazio, respectivamente (linhas 4 e 5). Realiza a operação *spotFree* no PC passando como parâmetro o nome da vaga (linha 7).

```
1 @OPERATION void vacate(){
2   int Var_NameSpot = Integer.parseInt(this.getId().getName());
3   String Var_NamePC =
       getObsProperty("PC").getValue().toString();
4   getObsProperty("isFree").updateValue(true);
5   getObsProperty("driverOcc").updateValue("");
6     execLinkedOp(lookupArtifact(Var_NamePC), "spotFree",
        Var_NameSpot);
7 }
```

Código 68 – Implementação da operação desocupar vaga.

A operação *spotFree* que a vaga realiza no PC está no código 69. O PC recebe como parâmetro o identificador da vaga (linha 1), atualiza o número de vagas livres (linha 2), atualiza a porcentagem de vagas livres (linha 3), percorre as vagas registradas (linha 4 e 5) para encontrar a vaga que fora ocupada. Ao encontrar a vaga atualiza as informações de que a vaga está livre e finaliza a procura (linhas 7 a 10).

```
1 @LINK void spotFree(int idSpot){
2   numFreeSpots++;
3   updateCoefOc();
4   Iterator<SpotClass> iterSpots = dbSpots.iterator();
5   while(iterSpots.hasNext()){
6     SpotClass spot = iterSpots.next();
7     if(spot.getId() == idSpot){
8       spot.setIsFree(true);
```

```
9           spot.setReservedDriver("");
10          break;
11        }
12      }
13  }
```

Código 69 – Implementação da operação *spotFree* que a vaga realiza no PC.

Após o motorista desocupar a vaga, o agente deve realizar o objetivo de registrar a alocação. O plano para realizar este objetivo se chama *registerUse* implementado no código 70. Este plano tem como condição a existência da crença *numUsed* e carrega na variável NU o número de vezes que as vagas do estacionamento foram utilizadas por completo no estacionamento.

O plano atualiza o número de vagas alocadas incrementando em um e atualiza a crença *numUsed* com o valor atualizado (linha 2).

```
1 +!registerUse : numUsed(NU)
2   <-  NNU = NU + 1; -+numUsed(NNU);
```

Código 70 – Codificação do plano *registerUse* do gerente.

Assim que o gerente realiza o objetivo *registerUse* o esquema está completo. O grupo e o esquema não serão mais utilizados, então o gerente exclui o grupo e o esquema criado. Todo grupo tem um esquema com mesmo número, por exemplo, quando o estacionamento abre, a primeira vaga alocada é enviada para o grupo g1 e utiliza o esquema s1. Quando o gerente percebe que o objetivo *registerUse* foi realizado ativa um plano codificado em 71. Neste plano é deletado o nome do esquema e colocado na variável *Number* (linha 2). Concatena a letra g com o número do esquema na variável *Group* (linha 3). Recupera o identificador do grupo (linha 4), interrompe a atenção para o grupo (linha 5) e destrói o artefato grupo (linha 6). Realiza as mesmas ações para o esquema (linhas 7, 8 e 9).

```
1 +goalState(Scheme,registerUse, _, _, satisfied)
2   <-  .delete("s", Scheme, Number);
3       .concat("g", Number, Group);
4       lookupArtifact(Group,GroupId);
5       stopFocus(GroupId);
6       destroy[artifact_id(GroupId)];
7       lookupArtifact(Scheme,SchemeId);
8       stopFocus(SchemeId);
9       destroy[artifact_id(SchemeId)].
```

Código 71 – Codificação do plano *goalState*.

5.6 ORGANIZAÇÃO NORMATIVA

Nesta seção é apresentada a implementação da organização normativa no MAPS-NORMS. O MAPS-NORMS apresenta o uso do estacionamento por meio de uma organização institucionalizada. Essa organização é implementada pela ferramenta MOISE, em que há o estabelecimento da sequência de objetivos que devem ser cumpridos para que uma vaga seja ocupada adequadamente.

No MAPS a organização estava implícita nos planos dos agentes, a partir da organização implementada, realiza-se um padrão que todos os agentes devem seguir para ocupar uma vaga. A seguir serão apresentados as especificações implementadas no MOISE: estrutural, funcional e normativa.

5.6.1 Especificação Estrutural

Para se criar uma organização é necessário a especificação estrutural que estabelece quais papéis serão necessários para compôr um grupo que realizará um objetivo. O código 72 implementa os papéis da organização e os grupos que estes papéis fazem parte. A especificação estrutural é representado na Figura 24 da seção 5.4.1.

A implementação da especificação estrutural é realizada via os códigos: 72 e 73. O código 72 implementa os papéis: *manager* e *driver*. O papel *manager* (linha 2) representa o agente gerente e o papel *driver* representa os agentes motoristas (linha 3).

```
1 <role-definitions>
2   <role id="manager"/>
3   <role id="driver" />
4 </role-definitions>
```

Código 72 – Codificação dos papéis existentes na organização.

Existe apenas um tipo de grupo no MAPS-NORMS, chamado *groupParking* e toda instância de grupo criada pelo gerente é baseada nele. Cada vaga alocada possui uma instância de *groupParking* para ocupar uma vaga.

Existe um e somente um agente com papel *manager* (linha 3), uma vez que existe apenas um gerente no estacionamento. Existe apenas um papel *driver*, devido uma vaga ser ocupada por um motorista por vez (linha 4). O *manager* tem autoridade sobre o *driver* (linha 7), assim o motorista obedece o agente que cuida do funcionamento do estacionamento e um *driver* sabe que o *manager* existe (linha 8), uma vez que o motorista receberá mensagens do gerente. Porém o motorista não se comunica com o gerente, o motorista requisita a vaga para a cancela.

O escopo de ambas as ligação é intra-grupo, ou seja, o agente destino da ligação e o agentes origem pertencem ao mesmo grupo (linhas 7 e 8). As ligação também não são bidirecionais, então a ligação que o agente origem tem para o agente destino não funciona para o caminho inverso da ligação.

```
1  <group-specification id="groupParking">
2    <roles>
3      <role id="manager" min="1" max="1" />
4      <role id="driver"  min="1" max="1"/>
5    </roles>
6    <links>
7      <link from="manager" to="driver"  type="authority"
              scope="inter-group" bi-dir="false"/>
8      <link from="driver"  to="manager"
         type="acquaintance"  scope="inter-group"
         bi-dir="false"/>
9    </links>
10 </group-specification>
```

Código 73 – Codificação do grupo estacionamento no MOISE.

5.6.2 Especificação Funcional

A especificação funcional determina a ordem que os objetivos devem ser cumpridos. A sequência estabelecida realiza a organização das ações dos agentes.

Esta organização dos objetivos é estabelecida em um esquema. Um esquema é um conjunto de objetivos que são organizados para cumprir um objetivo. O esquema utilizado para ocupar uma vaga é o *schemeOccupySpot* no código 74. Para o estacionamento foi estabelecido que o primeiro objetivo a ser cumprido é o *crossGate*, o qual tem um prazo para ser realizado de dez segundos (linha 4). Este objetivo deve ocorrer antes que os objetivos *occupySpot* (linha 5), *vacateSpot* (linha 6) e *registerUse* (linha 2) sejam realizados. Assim, existe uma relação de dependência entre os objetivos. O objetivo *registerUse* só pode ser realizado quando os objetivos *crossGate*, *occupySpot* e *vacateSpot* forem satisfeitos. Todos os objetivos são do tipo *achievement*, ou seja, precisam ser realizados apenas uma vez.

As missões *mManagementSpot* (linha 10) e *mUseSpot* (linha 14) especificam um conjunto de objetivos que devem ser realizados. Neste esquema só pode existir uma missão *mManagementSpot* (linha 10) e *mUseSpot* (linha 14). A missão *mManagementSpot* é composta pelo objetivo *registerUse* e a missão *mUseSpot* é composta pelos objetivos *occupySpot*, *crossGate* e

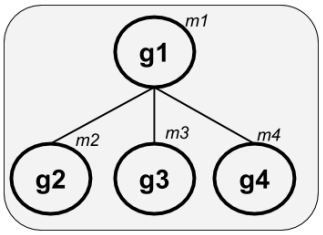
vacateSpot.

```
<scheme id="schemeOccupySpot">
  <goal id="registerUse" type="achievement">
    <plan operator="sequence" >
      <goal id="crossGate"  type="achievement" ttf="10
         seconds"/>
      <goal id="occupySpot" type="achievement"/>
      <goal id="vacateSpot" type="achievement"/>
    </plan>
  </goal>

  <mission id="mManagementSpot" min="1" max="1">
    <goal id="registerUse"/>
  </mission>

  <mission id="mUseSpot" min="1" max="1">
    <goal id="occupySpot"/>
    <goal id="crossGate"/>
    <goal id="vacateSpot"/>
  </mission>
</scheme>
```

Código 74 – Codificação da especificação ocupar vaga.

Este código gera o esquema representado na Figura 27, em que cada objetivo faz parte de uma missão. Cada missão é de responsabilidade de um papel. O agente que adotar este papel deve realizar os objetivos que cada missão possui.

Figura 27 – Representação do esquema para alocação de uma vaga detalhado.



Obj	Nome
g1	Vaga Utilizada
g2	Atravessar Cancela
g3	Ocupar Vaga
g4	Desocupar Vaga

Sigla	Missão	Responsável
m1	mManagementSpot	Manager
m2	mUseSpot	Driver

Fonte: Autoria Própria.

5.6.3 Especificação Normativa

Todos os objetivos dos agentes no MAPS estão implementados nos agentes, ou seja, não existe uma institucionalização dos objetivos que os agentes devem seguir. No MAPS-NORMS existe a garantia que o agente irá realizar determinadas ações relacionadas ao papel que o agente assume.

A garantia é implementada por normas que possuem obrigatoriedade para serem cumpridas. No MAPS-NORMS existem duas normas: *n1* e *n2*. A implementação dessas normas é realizada no código 75. A norma *n1* e *n2* são do tipo obrigação. A norma *n1* é realizada por agentes no papel de *manager* e *n2* é realizada por agentes no papel de *driver*. A norma *n1* tem a missão *mManagementSpot*, assim determina que um agente no papel de *manager* é obrigado a realizar a missão *mManagementSpot*. A norma *n2* tem a missão *mUseSpot*, assim determina que um agente no papel de *driver* é obrigado a realizar a missão *mUseSpot*.

```
1 <normative-specification>
2   <norm id="n1" type="obligation" role="manager"
       mission="mManagementSpot" />
3   <norm id="n2" type="obligation" role="driver"
       mission="mUseSpot"     />
4 </normative-specification>
```

Código 75 – Codificação do esquema *schemeOccupySpot*.

5.7 COMPARAÇÃO ENTRE MAPS E MAPS-NORMS

O quadro 9 apresenta as diferentes formas que um determinado item está implementado no MAPS e no MAPS-NORMS. A vaga é implementada no MAPS como uma crença, enquanto no MAPS-NORMS é um artefato que o motorista pode interagir diretamente. Desta forma torna o funcionamento mais flexível e sem a dependência do gerente para o motorista usar o estacionamento, o motorista tem a autonomia para escolher qual vaga quer estacionar.

A organização antes implícita nos planos que os agentes realizavam, agora está explícita no SMA. A organização fora obtida apartir do uso da ferramenta MOISE que delimitou o que poderia ser feito no estacionamento.

O motorista requisitava a vaga para o gerente, porém para o caso de haver requisições simultâneas, fora alterado a requisição de vaga para a cancela receber.

O GC que no MAPS era recebido do motorista, no MAPS-NORMS é inicializado e mantido no PC que tem a capacidade de atualizar o GC dos agentes.

Quadro 9 – Itens diferentes para cada estacionamento

Item	MAPS	MAPS-NORMS
Vaga	Crença	Artefato
Organização	Implícita	Explícita
Motorista Requisita Vaga Para	Gerente	Cancela
GC do Motorista	Recebe do Motorista	Registrado no PC

Fonte: Autoria Própria

6 SIMULAÇÕES E RESULTADOS

Neste capítulo são apresentadas as simulações realizadas e os resultados obtidos em relação ao MAPS-NORMS apresentado no capítulo anterior.

As simulações foram realizadas num PC com processador Intel(R) Core(TM) i3-4005U CPU @ 1.70GHz de 64-bits, sistema operacional Linux Mint 18 Sarah, no Eclipse IDE versão: *Neon Release (4.6.1)*.

As simulações foram realizadas com 48 agentes motoristas e cada motorista utiliza o estacionamento 20 vezes para obter um histórico da utilização do motorista ao longo do funcionamento do estacionamento. Quando o agente gerente notifica todos os agentes motoristas, todos requisitam vaga ao mesmo tempo com o objetivo de simular o pior cenário possível.

Cada vez que um agente motorista consegue uma vaga, permanece cinco segundos nela. Em seguida, o motorista deixa a vaga e sai do estacionamento. Após dez segundos de desocupar a vaga o motorista requisita vaga novamente.

No quadro 10 estão todos os 15 cenários testados. Cada cenário apresenta uma configuração inicial do MAPS-NORMS. As variáveis alteradas nos cenários foram a quantidade que cada comportamento de motorista existe no conjunto agentes motoristas e o nível de restrição que o agente gerente configura no estacionamento.

A cada três cenários no quadro 10 a configuração do comportamento do motorista é igual, o que muda é o nível de restrição. Nos três primeiros cenários, a quantidade de agente com cada comportamento é doze. Nos cenários quatro até seis o conjunto de agente RTD é o maior conjunto. Nos cenários de sete até nove o maior conjunto de comportamentos é o NTD, de dez até doze é o CSR e os três últimos tem o maior conjunto com comportamento FSF.

Quadro 10 – Cenários simulados.

Cenário	RTD	NTD	CSR	FSF	Nível de Restrição
1	12	12	12	12	Baixo
2	12	12	12	12	Normal
3	12	12	12	12	Alto
4	18	10	10	10	Baixo
5	18	10	10	10	Normal
6	18	10	10	10	Alto
7	10	18	10	10	Baixo
8	10	18	10	10	Normal
9	10	18	10	10	Alto
10	10	10	18	10	Baixo
11	10	10	18	10	Normal
12	10	10	18	10	Alto
13	10	10	10	18	Baixo
14	10	10	10	18	Normal
15	10	10	10	18	Alto

Fonte: Autoria Própria

Foram avaliados o tempo de espera e o grau de confiança (GC). O tempo de espera é o tempo que um motorista demora para receber uma vaga após requisitá-la. Calcula-se a partir da equação 6.1. O tempo que ganha vaga é subtraído do tempo que o motorista requisitou.

$$tempoEspera = tempoGanhaVaga - tempoRequisitaVaga \tag{6.1}$$

Cada agente obteve ao final da simulação o tempo médio de espera. Obtido a partir da equação 6.2, que é a média aritmética do tempo de espera que o motorista leva ao utilizar o estacionamento *n* vezes, sendo *n* um número inteiro positivo.

$$mediaTempoEspera = \frac{\sum_1^n tempoEspera}{n} \tag{6.2}$$

O quadro 11 apresenta os resultados obtidos em que cada linha estão os dados de cada cenário da simulação. A primeira coluna tem a identificação do cenário, na segunda está o tempo médio de espera de todos os motoristas, da terceira até a sexta coluna estão as respectivas médias aritmética dos tempos médios de espera dos motoristas em cada um dos quatro comportamentos. A sétima coluna identifica o nível de restrição de cada cenário.

O RTD tem em todos os cenário o menor tempo de espera para obter uma vaga e NTD possui em todos os cenários o tempo de espera notadamente maior que os outros motoristas. Este resultado mostra que o MAPS-NORMS sem saber previamente o comportamento do motorista conseguiu dar prioridade aos motoristas com comportamento que colaboraram com o sistema.

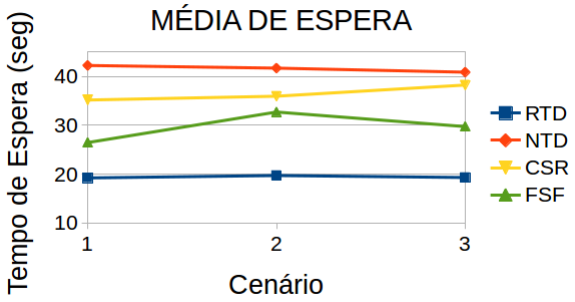
Quadro 11 – Tempo Médio de Espera (em segundos).

Cenário	Todos os Comportamentos	RTD	NTD	CSR	FSF	Nível de Restrição
1	30	19	42	35	26	Baixo
2	32	19	41	35	32	Normal
3	32	19	40	38	29	Alto
4	30	22	40	36	23	Baixo
5	33	26	41	38	26	Normal
6	32	16	41	36	34	Alto
7	28	14	40	35	23	Baixo
8	30	16	40	35	28	Normal
9	31	17	40	37	29	Alto
10	30	15	40	34	29	Baixo
11	30	17	41	36	27	Normal
12	29	11	40	36	29	Alto
13	33	21	44	41	27	Baixo
14	32	18	41	38	30	Normal
15	32	19	41	40	28	Alto

Fonte: Autoria Própria

O gráfico na Figura 28 representa o tempo de espera (em segundos) que os conjuntos de motoristas têm para conseguir uma vaga. Os comportamentos RTD e NTD são os comportamentos que menos variaram com a mudança do nível de restrição. O tempo de espera dos agentes com comportamento FSF aumenta quando é introduzido a punição por ocupar uma vaga vazia no cenário 2.

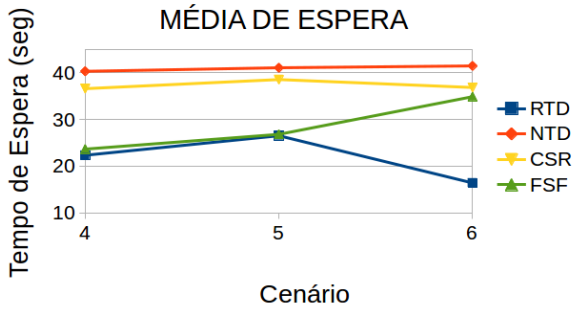
Figura 28 – Média de Espera para cenários: 1, 2 e 3.



Fonte: Autoria própria

O gráfico 29 apresenta a média de espera dos motoristas no cenário quatro, cinco e seis. A maior variação ocorre com os agentes RTD e FSF. Como os agentes RTD são maioria nesses cenários, nos cenários 4 e 5 os agentes com pontuação alta estão em maior número e acabam esperando mais tempo e ficam próximos do comportamento FSF, porém quando a restrição aumenta no cenário 6 a pontuação atualizada por este cenário faz com que os agentes do comportamento RTH esperem bem menos tempo que o FSF.

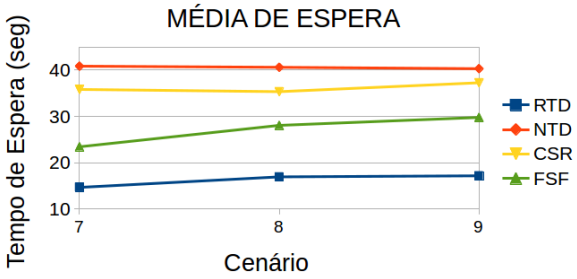
Figura 29 – Média de Espera para cenários: 4, 5 e 6.



Fonte: Autoria própria

O gráfico 30 apresenta os cenários em que o comportamento NTD é maioria, a média de espera dos agentes com comportamento NTD e CSR permanecem semelhantes aos gráficos anteriores, porém o tempo de espera dos agentes com comportamento RTD e FSF reduzem, por estarem em menor quantidade e serem selecionados mais vezes.

Figura 30 – Média de Espera para cenários: 7, 8 e 9.

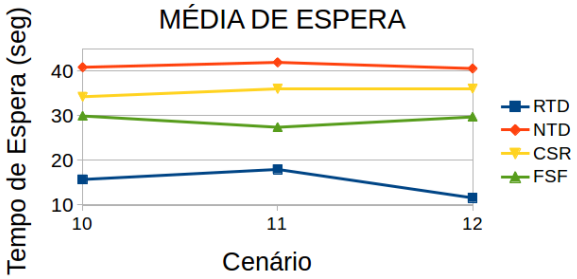


Fonte: Autoria própria

O gráfico 31 apresenta o cenário em que o maior conjunto de agentes tem comportamento CSR, assim escolhendo aleatoriamente a vaga que vai ocupar. Os agentes com comportamento CSR têm sua pontuação pouco alterada de acordo com o nível de restrição.

O comportamento RTD se mantém esperando menos que o restante dos comportamentos e NTD se mantém como o comportamento que espera por mais tempo para receber uma vaga. Quando o nível de restrição é alto os motoristas do comportamento RTD esperam menos tempo que nos outros níveis, pois como a bonificação e a punição são maiores, eles são favorecidos por escolher estacionar em suas vagas reservadas. A menor média de espera do motorista RTD ocorre no cenário 12.

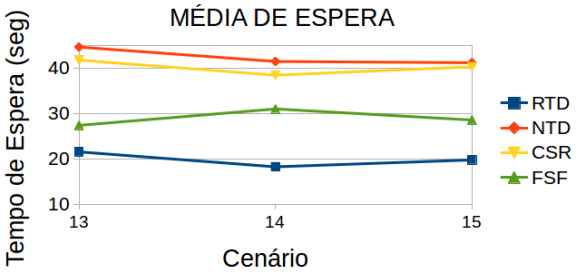
Figura 31 – Média de Espera para cenários: 10, 11 e 12.



Fonte: Autoria própria

O gráfico 32 apresenta os cenários em que existe mais motoristas com comportamento FSF. O tempo de espera do comportamento aumenta quando é introduzido a punição por ocupar vaga vazia que não está reservada, porém diminui quando o cenário tem restrição alto. O comportamento da espera dos motoristas CSR é semelhante ao NTD. O motorista que menos espera por vaga é o motorista com comportamento RTD.

Figura 32 – Média de Espera para cenários: 13, 14 e 15.



Fonte: Autoria própria

O GC foi obtido a partir do último GC que foi atualizado pelo PC para cada motorista, pois o objetivo é comparar com o GC inicial de 499. Conforme utilizam o estacionamento o PC incrementa ou decrementa este valor e assim mostrar como este valor ficou no final da utilização do estacionamento. O quadro 12 apresenta a média de GC dos motoristas de cada comportamento nos cenários testados.

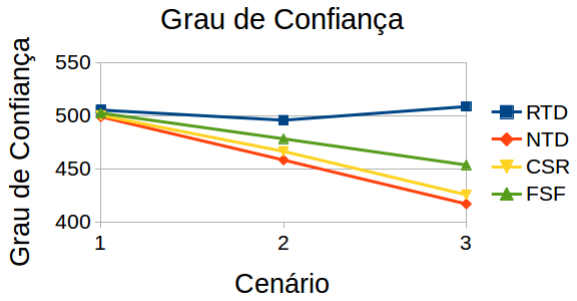
Quadro 12 – Grau de Confiança

Cenário	Todos os Comportamentos	RTD	NTD	CSR	FSF	Nível de Restrição
1	502	505	499	500	502	Baixo
2	475	495	458	466	478	Normal
3	451	508	417	425	453	Alto
4	503	508	499	500	504	Baixo
5	473	491	458	463	480	Normal
6	455	523	417	432	447	Alto
7	502	507	499	500	503	Baixo
8	477	498	458	464	487	Normal
9	451	499	417	423	462	Alto
10	502	507	499	500	502	Baixo
11	482	510	459	467	492	Normal
12	463	535	417	427	469	Alto
13	501	504	499	499	502	Baixo
14	475	500	457	463	477	Normal
15	444	483	418	422	453	Alto

Fonte: Autoria Própria

O gráfico 33 apresenta o grau de confiança nos três primeiros cenários. O GC do comportamento RTD se mantém maior que o restante. Enquanto os outros comportamentos decrescem conforme o cenário aumenta a bonificação e a punição.

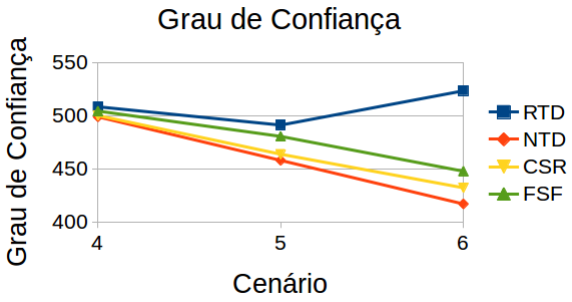
Figura 33 – Média do grau de confiança para cenários: 1, 2 e 3.



Fonte: Autoria própria

O gráfico 34 apresenta o GC dos cenários quatro, cinco e seis em que o maior conjunto de motoristas é o que tem comportamento RTD. O GC dos motoristas com comportamento RTD aumenta em relação ao gráfico anterior. A redução do GC dos outros comportamentos se mantém semelhante ao gráfico anterior.

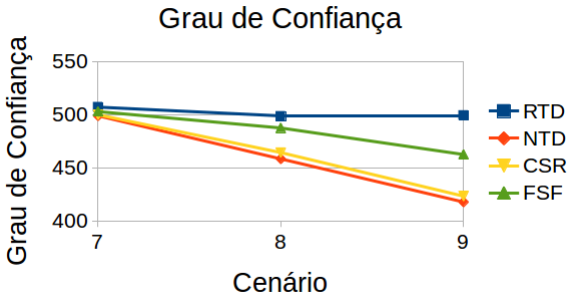
Figura 34 – Média do grau de confiança para cenários: 4, 5 e 6.



Fonte: Autoria própria

O gráfico 35 apresenta os cenários em que o comportamento NTD possui mais motoristas que os outros comportamentos. O comportamento RTD tem um decremento no GC no cenário com nível de restrição alto, porque com mais motoristas que ocupam vaga que não são reservadas a eles, os motoristas com comportamento RTD são prejudicados ao tentar ocupar vagas reservadas. Os motoristas de comportamento FSF aumentam o GC neste caso.

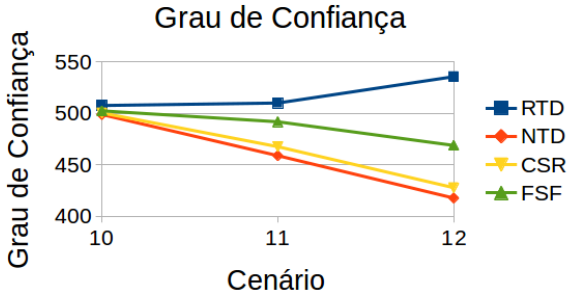
Figura 35 – Média do grau de confiança para cenários: 7, 8 e 9.



Fonte: Autoria própria

O gráfico 36 apresenta o comportamento do motorista que ocupa uma vaga aleatoriamente tendo o conjunto maior que os outros comportamento. Os motoristas de comportamento CSR tem seu GC diminuído no cenário com restrição alto, porém se mantém maior que os motoristas com comportamento NTD.

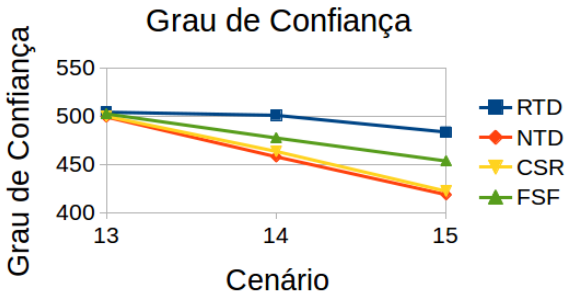
Figura 36 – Média do grau de confiança para cenários: 10, 11 e 12.



Fonte: Autoria própria

O gráfico 37 apresenta o GC dos motoristas e em sua maioria composta por motoristas com comportamento FSF.

Figura 37 – Média do grau de confiança para cenários: 13, 14 e 15.



Fonte: Autoria própria

Como mostrado nos gráficos de 33 até 37 existe um padrão de comportamento do gráfico. No nível de restrição baixo os quatro tipos de comportamentos estão com valores bem próximos e depois (nos outros dois níveis) o RTD permanece praticamente igual, ao passo que os outros três sempre decrescem.

O comportamento NTD sempre mantém o GC menor que os outros comportamentos. O comportamento CSR tem redução semelhante ao NTD.

O comportamento FSF tem menor redução de GC que NTD e CSR pois tem mais chances de estacionar numa vaga que o PC reservou, pois o PC verifica qual vaga está livre da primeira até a última, semelhante a estratégia de FSF. Além disso, o motorista FSF pode estacionar numa vaga reservada para NTH e CSR que não utilizaram, se o nível de restrição é o baixo, o motorista FSF não recebe punição.

7 CONCLUSÃO

Este capítulo apresenta as conclusões e os trabalhos futuros que podem dar continuidade ao desenvolvimento do projeto MAPS. O SMA MAPS-NORMS estende as funcionalidades dos agentes e do ambiente do MAPS com o objetivo principal de implementar a organização normativa.

7.1 CONCLUSÕES

O sistema MAPS-NORMS obteve uma organização que controla as ações que os agentes realizam para utilizar um estacionamento. A organização definiu os papéis que os agentes motoristas e gerente assumiram para participar do SMA. Porém, para a organização não extinguir a autonomia do agente motorista, permitiu-se à ele escolher qual vaga ocupar.

A centralização do controle do estacionamento no artefato *Parking Controller* fez com que o agente gerente tivesse uma redução de sua autonomia em relação ao MAPS, pois o agente gerente participava diretamente da negociação de cada vaga. No MAPS-NORMS cabe apenas atividades relacionadas a criação dos artefatos e comunicação para o agente motorista.

Uma das principais mudanças para permitir que a organização normativa fosse testada, foi a mudança do artefato de crença para a representação de artefato. Isto permitiu que o motorista que recebesse uma vaga pudesse estacionar em outra vaga que não fosse destinada à ele. Assim, a autonomia dos motoristas foi aumentada pois não era necessário a participação do gerente no processo de estacionar um veículo na vaga.

Para o motorista ter a capacidade de ocupar outra vaga que não a reservada à ele, foi necessário implementar diferentes comportamentos que distinguissem claramente que tipo de ação o agente motorista teria em relação ao uso do estacionamento. O comportamento do agente motorista é implementado no próprio agente, os comportamentos são: escolher ocupar a vaga que estava reservada à ele (RTD), escolher ocupar necessariamente outra vaga que não seja a reservada pelo gerente (NTD), escolher ocupar uma vaga aleatoriamente (CSR) e escolher a primeira vaga livre (FSF).

O *Parking Controller* não teve acesso ao comportamento do motorista, por isso todos começam com o mesmo grau de confiança. A partir das ações que o motorista tomava ao utilizar o estacionamento, o *Parking Controller* poderia bonificar ou punir o motorista incrementando ou decrementando o grau de confiança. Quando uma vaga fica livre é selecionado um motorista num conjunto de motoristas que requisitaram vaga com o maior grau de confiança e assim o comportamento do motorista influencia no tempo de espera do motorista.

Nas simulações realizadas, cada agente utilizou o estacionamento por um número determinado de vezes e cada agente tinha seu comportamento inicial determinado e ele mantinha

seu comportamento até o fim do número de vezes que usaria o estacionamento. Existiam três níveis de restrição que o SMA poderia assumir: Baixo, Normal e Alto. Para cada nível de restrição eram dadas as bonificações ou punições proporcional ao nível de restrição.

A partir dos resultados apresentados nos gráficos de média de espera por vaga e também, pelo grau de confiança final de cada comportamento do motorista que dois resultados esperados aconteceram: o motorista com comportamento RTD espera menos tempo que os outros comportamentos em todos os cenários testados e o motorista com comportamento NTH esperou mais tempo para ocupar uma vaga que os outros motoristas em todos os cenários. Além disso, cenários onde o motorista com comportamento RTD era maioria seu tempo de espera aumentou devido a concorrência com motoristas com o mesmo tipo de comportamento.

Outro resultado a ser destacado é o tempo de espera do motorista FSF, que em alguns cenários apresentava a média de tempo de espera muito próxima ao RTH, principalmente em cenários com o nível de restrição baixo, onde não havia punição por ocupar uma vaga vazia. Isso ocorreu devido ao motorista FSF ocupar a primeira vaga vazia que encontrasse e outro motorista ter estacionado em sua vaga.

O motorista com comportamento CSR em todos os cenários esperava menos tempo que o NTD e mais que o restante, uma vez que a probabilidade de escolher a vaga reservada pelo gerente era 5%.

Ainda é possível elencar as principais contribuições deste trabalho:

- Vagas representadas por artefatos, permite ao motorista interagir diretamente com as vagas sem dependência do gerente.
- Aumento da autonomia dos agentes motoristas ao escolherem qual vaga do estacionamento ocupar.
- Registro centralizado do grau de confiança dos motoristas num único artefato. Assim é possível armazenar o histórico do grau de confiança do agente motorista, permitindo uma extração de dados mais detalhada da utilização do estacionamento.
- Percepção do ambiente pelo *Parking Controller* permitindo ao artefato saber quais ações ocorreram no estacionamento em tempo real, por exemplo, quando um vaga avisa ao *Parking Controller* que foi ocupada.
- Implementação de uma organização normativa que estabelece a organização e a coordenação das ações dos agentes do SMA.
- Implementação dos artefatos numa área de trabalho específica, separando os artefatos do estacionamento da área de trabalho comum do SMA.
- Implementação de níveis de restrições diferentes, determinando as ações do motorista com peso diferente para a atualização do grau de confiança.

- Simulação do funcionamento do estacionamento baseado no comportamento dos agentes e na aplicação das normas.

Sendo assim, os objetivos previstos para o desenvolvimento do trabalho foram concluídos e a sua implementação tornou o código mais flexível para futuras extensões.

7.2 TRABALHOS FUTUROS

A partir do desenvolvimento do SMA MAPS-NORMS nota-se possíveis extensões para o projeto MAPS. Dentre os trabalhos futuros estão motoristas com comportamento adaptado, ou seja, quando está com pontuação baixa ele ocupa a vaga reservada pelo gerente para aumentar sua pontuação, quando tiver pontuação alta ele, escolhe a vaga de acordo com a necessidade.

Uma possível extensão da dimensão estrutural do MOISE é aumentar o número papéis dos agentes no estacionamento, por exemplo, uma hierarquia de subgerentes que controlariam partes do estacionamento e um gerente que fosse responsável pelas atividades dos subgerentes.

No sistema desenvolvido foi utilizado apenas um estacionamento pois existe uma área de trabalho que contém os artefatos do estacionamento, mas pode ser estendido para existir mais de um estacionamento a partir da criação de várias áreas de trabalho, desta forma permitindo a generalização e a multiplicidade de artefatos e grupos.

Outra alternativa que estende este trabalho é um estacionamento onde não exista o papel do agente gerente, onde somente os motoristas interajam por vagas, haja vista que os artefatos vagas e cancela são utilizadas pelos motoristas sem a dependência do gerente.

Por não haver o controle a saída de veículos, pode-se estender e realizar o controle de pagamento, e assim o faturamento do estacionamento de acordo com as normas implementadas. Também é possível implementar no agente motorista a possibilidade de enviar ao gerente a informação que está saindo do estacionamento antes de chegar no veículo, e assim, o gerente conseguir alocar aquela vaga com mais eficiência por ter o conhecimento prévio que aquela vaga logo estará disponível.

REFERÊNCIAS

AGRE, Philip E; CHAPMAN, David. Pengi: An implementation of a theory of activity. In: **AAAI**. [S.l.: s.n.], 1987. v. 87, n. 4, p. 286–272.

AUSTIN, John L. **How to do Things with Words**. [S.l.]: Cambridge: Harvard University Press, 1962.

BELLIFEMINE, Fabio Luigi; CAIRE, Giovanni; GREENWOOD, Dominic. **Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)**. [S.l.]: John Wiley & Sons, 2007. ISBN 0470057475.

BIGUS, Joseph P. *et al.* Able: A toolkit for building multiagent autonomic systems. **IBM Systems Journal**, IBM, v. 41, n. 3, p. 350–371, 2002.

BILAL, Mustapha *et al.* Multi-agent based governance model for machine-to-machine networks in a smart parking management system. In: IEEE. **Communications (ICC), 2012 IEEE International Conference on**. [S.l.], 2012. p. 6468–6472.

BOELLA, Guido; TORRE, Leendert Van Der; VERHAGEN, Harko. Introduction to normative multiagent systems. **Computational & Mathematical Organization Theory**, Springer, v. 12, n. 2-3, p. 71–79, 2006.

BOISSIER, Olivier *et al.* Multi-agent oriented programming with jacamo. **Science of Computer Programming**, Elsevier, v. 78, n. 6, p. 747–761, 2013.

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.

BRATMAN, Michael E; ISRAEL, David J; POLLACK, Martha E. Plans and resource-bounded practical reasoning. **Computational intelligence**, Wiley Online Library, v. 4, n. 3, p. 349–355, 1988.

BRIOT, Jean-Pierre. **Principes et architecture des systèmes multi-agents**. [S.l.]: Hermès Science, 2001.

CASTELFRANCHI, Cristiano. Commitments: From individual intentions to groups and organizations. In: **ICMAS**. [S.l.: s.n.], 1995. v. 95, p. 41–48.

CASTELFRANCHI, C.; FALCONE, R. Principles of trust for mas: Cognitive anatomy, social importance, and quantification. In: **Proceedings of the 3rd International Conference on Multi Agent Systems**. Washington, DC, USA: IEEE Computer Society, 1998. (ICMAS '98), p. 72–. ISBN 0-8186-8500-X. Disponível em: <http://dl.acm.org/citation.cfm?id=551984.852234>.

CASTRO, Lucas Fernando Souza de. **Modelagem e implementação de um sistema multiagente utilizando a plataforma JaCaMo para alocação de vagas em um estacionamento inteligente**. Tese (Doutorado) — Universidade Tecnológica Federal do Paraná, 2015.

CHOU, Shuo-Yan; LIN, Shih-Wei; LI, Chien-Chang. Dynamic parking negotiation and guidance using an agent-based platform. **Expert Systems with Applications**, Elsevier, v. 35, n. 3, p. 805–817, 2008.

DEMAZEAU, Yves; MULLER, Jean-Pierre (Ed.). **Decentralized A.I. 2**. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., 1991. ISBN 0444890513.

DORIGO, Marco; BLUM, Christian. Ant colony optimization theory: A survey. **Theoretical computer science**, Elsevier, v. 344, n. 2, p. 243–278, 2005.

ESTEVA, Marc *et al.* Ameli: An agent-based middleware for electronic institutions. In: IEEE COMPUTER SOCIETY. **Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1**. [S.l.], 2004. p. 236–243.

FERBER, Jacques; GUTKNECHT, Olivier; MICHEL, Fabien. From agents to organizations: an organizational view of multi-agent systems. In: **Agent-Oriented Software Engineering IV**. [S.l.]: Springer, 2003. p. 214–230.

FERGUSON, Innes A. **TouringMachines: An architecture for dynamic, rational, mobile agents**. Tese (Doutorado) — University of Cambridge UK, 1992.

FICHMAN, Flamínio; DUARTE, Fábio; NETO, João C. A circulação restrita de carros em centros urbanos. **Parking Brasil**, n. VI, p. 14, 02 2016.

FININ, Tim *et al.* Kqml as an agent communication language. In: **Proceedings of the Third International Conference on Information and Knowledge Management**. New York, NY, USA: ACM, 1994. (CIKM '94), p. 456–463. ISBN 0-89791-674-3. Disponível em: <http://doi.acm.org/10.1145/191246.191322>.

FISMAN, Raymond; MIGUEL, Edward. Corruption, norms, and legal enforcement: Evidence from diplomatic parking tickets. **Journal of Political Economy**, JSTOR, v. 115, n. 6, p. 1020–1048, 2007.

GEORGEFF, Michael P.; LANSKY, Amy L. Reactive reasoning and planning. In: **Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 2**. AAAI Press, 1987. (AAAI'87), p. 677–682. ISBN 0-934613-42-7. Disponível em: <http://dl.acm.org/citation.cfm?id=1863766.1863818>.

GONÇALVES, Wesley RC; ALVES, Gleifer Vaz. Smart parking: mecanismo de leilão de vagas de estacionamento usando reputação entre agentes. WESAAC, 2015.

GUTKNECHT, Olivier; FERBER, Jacques. The madkit agent platform architecture. In: SPRINGER. **Workshop on Infrastructure for Scalable Multi-Agent Systems at the International Conference on Autonomous Agents**. [S.l.], 2000. p. 48–55.

HANNOUN, Mahdi *et al.* Moise: An organizational model for multi-agent systems. In: **Advances in Artificial Intelligence**. [S.l.]: Springer, 2000. p. 156–165.

HÜBNER, J. F. **Organização de Sistemas Multiagentes**. Tese (Doutorado) — Escola Politécnica, USP, São Paulo, 2003.

HÜBNER, Jomi F. *et al.* Instrumenting multi-agent organisations with organisational artifacts and agents. **Autonomous Agents and Multi-Agent Systems**, v. 20, n. 3, p. 369–400, 2010. ISSN 1573-7454. Disponível em: <http://dx.doi.org/10.1007/s10458-009-9084-y>.

HÜBNER, Jomi Fred; SICHMAN, Jaime Simão. Organização de sistemas multiagentes. **III Jornada de Mini-Cursos de Inteligência Artificial (JAIA'03)**, v. 8, p. 247–296, 2003.

HÜBNER, Jomi Fred *et al.* Moise tutorial. Citeseer, 2010.

HÜBNER, Jomi F; SICHMAN, Jaime S; BOISSIER, Olivier. Developing organised multiagent systems using the moise+ model: programming issues at the system and agent levels. **International Journal of Agent-Oriented Software Engineering**, Inderscience Publishers, v. 1, n. 3-4, p. 370–395, 2007.

KAHAN, James P; RAPOPORT, Amnon. **Theories of coalition formation**. [S.l.]: Psychology Press, 2014.

KARATSUBA, Anatolii; OFMAN, Yu. Multiplication of multidigit numbers on automata. In: **Soviet physics doklady**. [S.l.: s.n.], 1963. v. 7, p. 595.

KESSLER, Aaron M. Elon musk says self-driving tesla cars will be in the us by summer. **The New York Times. http://mobile. nytimes. com/2015/03/20/business/elon-musk-says-selfdriving-tesla-cars-will-be-in-the-us-by-summer.html**, 2015.

KOSTER, Andrew; KOCH, Fernando; BAZZAN, Ana LC. Incentivising crowdsourced parking solutions. In: **Citizen in Sensor Networks**. [S.l.]: Springer, 2014. p. 36–43.

LEMAÎTRE, Christian; EXCELENTE, Cora B. Multi-agent organization approach. In: **Proceedings of II Iberoamerican Workshop on DAI and MAS**. [S.l.: s.n.], 1998.

LIN, Trista. **Smart Parking: Network, Infrastructure and Urban Service**. Tese (Doutorado) — INSA Lyon, 2015.

LUGO, Gustavo Giménez; HÜBNER, Jomi Fred; SICHMAN, Jaime Simao. Representaçao e evoluçao de esquemas sociais em sistemas multi-agentes: Um enfoque funcional. **Anais do III Encontro Nacional de Inteligência Artificial**, p. 1237–1246, 2001.

MASSONET, Philippe; DEVILLE, Yves; NÈVE, Cédric. From aose methodology to agent implementation. In: ACM. **Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1**. [S.l.], 2002. p. 27–34.

MEIJER, Albert; BOLÍVAR, Manuel Pedro Rodríguez. Governing the smart city: a review of the literature on smart urban governance. **International Review of Administrative Sciences**, SAGE Publications, v. 82, n. 2, p. 392–408, 2015.

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**. [S.l.]: Springer, 2014. p. 98–109.

PADGHAM, Lin; WINIKOFF, Michael. Prometheus: A methodology for developing intelligent agents. In: SPRINGER. **International Workshop on Agent-Oriented Software Engineering**. [S.l.], 2002. p. 174–185.

____. **Developing intelligent agent systems: A practical guide**. [S.l.]: John Wiley & Sons, 2005.

PETROLO, Riccardo; LOSCRÌ, Valeria; MITTON, Nathalie. Towards a smart city based on cloud of things, a survey on the smart city vision and paradigms. **Transactions on Emerging Telecommunications Technologies**, Wiley Online Library, 2015.

PYNADATH, David V; TAMBE, Milind. An automated teamwork infrastructure for heterogeneous software agents and humans. **Autonomous Agents and Multi-Agent Systems**, Springer, v. 7, n. 1-2, p. 71–100, 2003.

RAO, Anand S. Agentspeak (l): Bdi agents speak out in a logical computable language. In: **Agents Breaking Away**. [S.l.]: Springer, 1996. p. 42–55.

RICCI, Alessandro; VIROLI, Mirko; OMICINI, Andrea. Cartago: A framework for prototyping artifact-based environments in mas. In: **Proceedings of the 3rd International Conference on Environments for Multi-agent Systems III**. Berlin, Heidelberg: Springer-Verlag, 2007. (E4MAS'06), p. 67–86. ISBN 978-3-540-71102-5. Disponível em: <http://dl.acm.org/citation.cfm?id=1759343.1759348>.

RUSSELL, Stuart J.; NORVIG, Peter. **Artificial Intelligence: A Modern Approach**. 2. ed. [S.l.]: Pearson Education, 2003. ISBN 0137903952.

SEARLE, John. **The construction of social reality**. [S.l.]: Simon and Schuster, 1995.

SEARLE, John R. **Speech acts: An essay in the philosophy of language**. [S.l.]: Cambridge university press, 1969. v. 626.

SICHMAN, J.S. *et al.* **Multi-Agent Oriented Programming JaCaMo Tutorial**. 2014. Disponível em: <http://jacamo.sourceforge.net/tutorial/>. Acesso em: 25 jun. 2016.

SINGH, Munindar P. Agent communication languages: Rethinking the principles. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 31, n. 12, p. 40–47, dez. 1998. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/2.735849>.

SORICI, Alexandru *et al.* Multi-Agent Oriented Reorganisation within the JaCaMo infrastructure. In: **Proceedings of The Third International Workshop on Iinfraestructures and tools for multiagent systems: ITMAS**. [S.l.: s.n.], 2012. p. 135–148.

TELEKOM. **Deutsche Telekom: MWC 2014: Deutsche Telekom and Pisa start Smart City project**. 2014. Disponível em: <https://www.telekom.com/media/company/216108>. Acesso em: 16 mai. 2016.

VIASSEGURAS. 2014. Disponível em: <http://www.vias-seguras.com/os_acidentes/estatisticas/estatisticas_nacionais>. (Acesso em: 11 jun. 2016).

VIROLI, Mirko *et al.* Infrastructures for the environment of multiagent systems. **Autonomous Agents and Multi-Agent Systems**, Springer, v. 14, n. 1, p. 49–60, 2007.

WEISS, Gerhard (Ed.). **Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence**. Cambridge, MA, USA: MIT Press, 1999. ISBN 0-262-23203-0.

WOOLDRIDGE, Michael; JENNINGS, Nicholas R. Agent theories, architectures, and languages: a survey. In: **Intelligent agents**. [S.l.]: Springer, 1994. p. 1–39.