UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO ACADÊMICO DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS FERNANDO SOUZA DE CASTRO

DESENVOLVIMENTO DE UMA ARQUITETURA MULTIAGENTE HOLÔNICA CONFIGURÁVEL APLICADA AO CENÁRIO DE SMARTPARKING

DISSERTAÇÃO

PONTA GROSSA

2018

LUCAS FERNANDO SOUZA DE CASTRO

DESENVOLVIMENTO DE UMA ARQUITETURA MULTIAGENTE HOLÔNICA CONFIGURÁVEL APLICADA AO CENÁRIO DE SMARTPARKING

Dissertação apresentada como requisito parcial à obtenção do título de Mestre em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação da Universidade Tecnológica Federal do Paraná – Campus Ponta Grossa.

Área de Concentração: Modelagem e Métodos Computacionais

Orientador: Prof. Dr. Gleifer Vaz Alves
Coorientador: Prof. Dr. André Pinz Borges

PONTA GROSSA
2018

Dedido esse trabalho ao que se intitula como sendo a Verdade: Cristo.

AGRADECIMENTOS

Still writing...

Maybe you are a BDI Agent:
You belief about something,
desire something,
and intend to do something.
Lucas F. S. Castro

RESUMO

CASTRO, Lucas Fernando Souza de. *Desenvolvimento de um sistema multiagente holônico para um estacionamento inteligente utilizando o framework JaCaMo. 2018.* Dissertação de Mestrado, Universidade Tecnológica Federal Do Paraná. Ponta Grossa, 2018.

As características presentes em um estacionamento inteligente apontam a utilização de Sistemas Multiagentes como tendência devido a dinamicidade do domínio. Dentro da área de sistemas multiagentes, a utilização de agentes holônicos proporciona ao sistema um nível maior de adaptação em face das mudanças do ambiente. O presente trabalho propoe o desenvolvimento de uma arquitetura configurável e multiagente holônica aplicada ao cenário de estacionamentos inteligentes desenvolvida por meio do *framework* JaCaMo, o qual define-se como uma ferramenta dividida em três camadas, programação de agentes, artefatos e organizações através do Jason, Cartago e Moise respectivamente. O atual trabalho também visa responder a viabilidade do JaCaMo em aplicações holônicas. Ao longo do desenvolvimento e durante os testes a Arquitetura foi capaz de alocar as vagas de estacionamento e de demonstrar a viabilidade do uso do JaCaMo em uma aplicação holônica, a qual é a arquitetura desse trabalho.
Palavras-chaves: Sistemas multiagentes. Agente holônico . JaCaMo.

ABSTRACT

CASTRO, Lucas Fernando Souza de. *Development of a holonic multiagent system to a smartparking through JaCaMo Framework. 2018.* Thesis, Universidade Tecnológica Federal Do Paraná. Ponta Grossa, 2018.

The characteristics that composes a smart parking point to the use of Multi-Agent Systems as a trend due to the dynamicity of the domain. In the the area of multi-agent systems, the use of holonic agents provides the system a higher level of adaptation before the changes in the parking environment. This work proposes the development of a configurable and holonic multiagent architecture applied to the smart parking scenario powered by JaCaMo framework, which is defined as a tool splitted in three layers: agent programming, artifacts and organizations through Jason, Cartago and Moise respectively. The current work also aims to answer the JaCaMo applicability in holonic context. Throughout the development and during the tests the Architecture was able to assign parking spaces and shows the applicability of using JaCaMo in a holonic application, which is the proposed architecture of this work.

Key-words: Multiagent system. Holonic agent . JaCaMo.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LIST OF LISTINGS

LISTA DE ABREVIATURAS E SIGLAS

BB	*Body-Body*
BDI	*Belief, Desire and Intention*
dMARS	*Distributed multi-agent reasoning system*
HB	*Head-Body*
JaCaMo	Jason + Cartago + Moise
MAPS	*MultiAgent Parking System*
PRS	*Procedural Reasoning System*
SMA	Sistema Multiagente
SMAH	Sistema Multiagente Holônico
SMF	Sistema de Manufatura Flexível

SUMÁRIO

1 INTRODUÇÃO

A utilização de softwares com capacidades autônomas tem ganhado destaque em vista da sua versatilidade diante dos ambientes dinâmicos e complexos que as aplicações apresentam. Em vista disso, sistemas multiagentes (SMA) tem apresentado grandes vantagens em serem aplicados em tais cenários para a resolução de problemas. Um SMA pode ser descrito como um sistema composto por agentes, entidades de software inseridas em um ambiente dinâmico, que possuem um certo grau de autonomia e raciocínio para atingir um objetivo (WOOLDRIDGE, 2009). Esses ambientes são dinâmicos devido a complexidade e quantidade de artefatos tecnológicos que alteram seu estado constantamente, necessitando que os demais artefatos sejam notificados de suas mudanças. Um exemplo de ambiente dinâmico onde SMAs tem sido aplicados, são os estacionamentos inteligentes.

Um estacionamento é dito inteligente quando provê para o seu administrador gerenciar o local de forma automática, controlando assim a alocação de vagas, preços, disponibilidade e até mesmo sua infraestrutura (IDRIS *et al.*, 2009). Em Nápoles, na Itália, um grupo de pesquisadores desenvolveram soluções baseadas em SMAs para a negociação de vagas (NAPOLI; NOCERA; ROSSI, 2014b; NOCERA; NAPOLI; ROSSI, 2014). Além das soluções para estacionamentos, há soluções que visam o desenvolvimento de sistemas multiagentes, como por exemplo o framework JaCaMo, sendo composto pelo interpretador Jason (baseado em AgentSpeak(L), linguagem de desenvolvimento para agentes), pelo framework Cartago que visa o desenvolvimento de artefatos para o ambiente dos agentes e, por fim, a ferramenta Moise cujo objetivo é a programação normativa do SMA (JACAMO, 2011). A programação normativa dos agentes visa o estabelecimento de normas e regras para os agentes, uma vez que estes agentes possuem habilidades sociais para a competição e cooperação no alcance de seus objetivos.

O termo *holon* foi citado inicialmente por Arthur Koester (Arthur Koestler, 1969) para apresentar sua visão de mundo holística. Koester, descreve um *holon* como uma estrutura que é composta por sub-estruturas. Esse conceito pode ser aplicado aos órgãos de um corpo humano, os quais são compostos por células que também podem ser decompostas em outras estruturas, sendo que nenhuma sub-estrutura pode ser entendida sem a estrutura a qual ela está contida. Em 1999, Klaus Fisher, apresentou a ideia do *holon* para o paradigma de agentes (FISHER, 1999). A proposta de Fisher foi aplicar a descrição de estruturas e sub-estruturas para agentes e sub-agentes, em que estes agentes poderiam formar organizações e parcerias para o cumprimento de um objetivo em comum ou até mesmo o compartilhamento de recursos (FISCHER; SCHILLO; SIEKMANN, 2003).

Este documento tem como principal objetivo propor uma arquitetura de sistema multiagente holônico para a alocação e gerenciamento de vagas em estacionamentos inteligentes. A motivação para o desenvolvimento de um sistema para a alocação e gerenciamento das vagas em grandes estacionamentos deve-se ao fato de que estacionamentos que utilizam métodos con-

vencionais (ou não automatizados) para a alocação não satisfazem os motoristas (FRAIFER; FERNSTRöM, 2016). As razões para a não satisfação podem ser explicadas devido ao custo pela busca por uma vaga livre, seu preço e até mesmo a má distribuição das vagas no local. Assim, é necessário a utilização e desenvolvimento de soluções que objetivem uma alocação e gerenciamento dessas vagas de estacionamento. O estacionamento inteligente que será utilizado pelo SMA proposto destina-se a estacionamentos privados (estádios, shoppings, universidades, aeroportos, etc. Os recursos providos pelo SMA são as vagas propriamente ditas e estas sendo gerenciadas, alocadas e utilizadas pelos agentes presentes nesse ambiente. Os agentes são divididos em quatro grupos: (i) agente *PSpace* (responsável pela verificação do *status* da vaga de estacionamento); (ii) agente *Sector* (encarregado do gerenciamento da utilização do setor/andar e dos agentes *PSpace* que o compõe); (iii) agente *Manager* (responsável pelo gerenciamento de toda a infra-estrutura do estacionamento, inclusive dos agentes *Sector* e vaga que o compõe); (iv) agente *IDriver* (representa o usuário que deseja ocupar uma vaga de estacionamento).

A arquitetura para o SMA holônico foi diagramada com base na metodologia Prometheus (PROMETHEUS, 2015) e desenvolvida no *framework* JaCaMo . Destaca-se ainda que o trabalho aqui desenvolvido, está inserido em um grupo de pesquisa denominado GPAS (Grupo de Pesquisa de Agentes de Software - DAINF-PG) com o projeto de pesquisa `MAPS` *(MultiAgent Parking System*). O projeto `MAPS` tem como objetivo principal o desenvolvimento de soluções para estacionamentos inteligentes baseados em SMAs. Em CASTRO; ALVES; BORGES (2017) foi desenvolvido um SMA em JaCaMo para a alocação de vagas em estacionamentos. O SMA desenvolvido é composto por dois grupos de agentes: um agente gerente e vários agentes motoristas, sendo todo o processo de alocação e gerenciamento de vagas realizado por um único agente, o gerente. Todavia, caso o agente gerente viesse a falhar todo o sistema falharia. Além dessa falha, devido ao fato de apenas um agente gerente ser o responsável pela alocação e gerenciamento das vagas, isso poderia gerar uma sobrecarga sobre o agente.

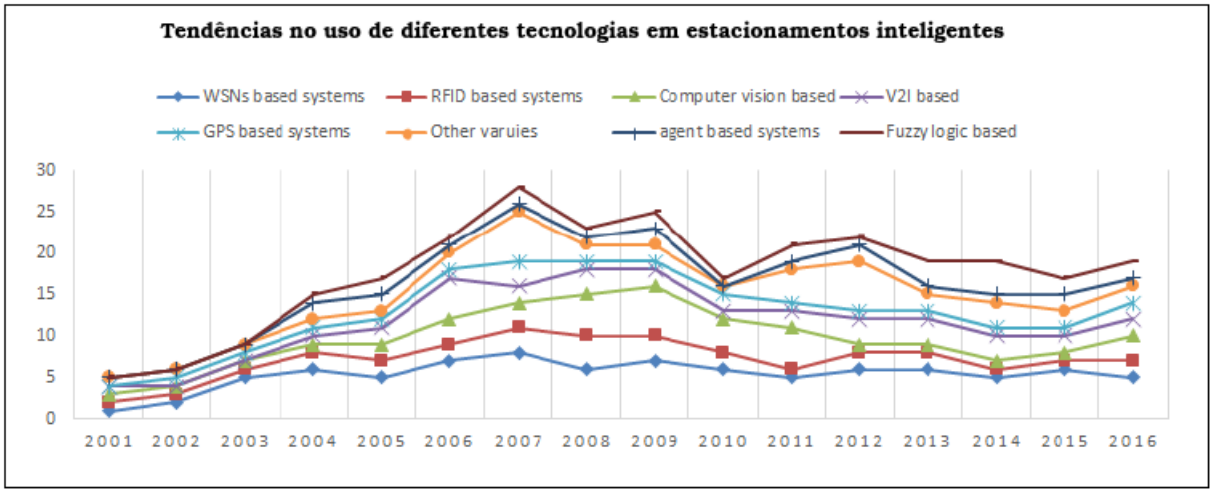
Por fim, diferente do sistema desenvolvido, a arquitetura holônica aqui proposta (`MAPS-HOLO`) não tem como objetivo primário a expansão do Sistema Multiagente previamente desenvolvido, mas sim em um desenvolvimento de toda uma Arquitetura Configurável para Sistemas Multiagentes holônicos com enfoque na alocação de recursos (vagas) para usuários, sendo esses representados por agentes inteligentes. O desenvolvimento da `MAPS-HOLO` foi dividido em duas etapas: O desenvolvimento da arquitetura em uma perspectiva genérica e posteriormente o desenvolvimento por meio do JaCaMo. A principal razão dessa divisão é a motivação para que a `MAPS-HOLO` não fique presa em uma ferramenta específica, mas que possa ser futuramente expandida em outras ferramentas que proporcionem o uso de Sistemas MultiAgentes Holônicos.

Após o desenvolvimento foi verificado que a `MAPS-HOLO` cumpriu os seus objetivos primários e secundários e conseguiu elaborar a seguinte resposta: O JaCaMo é capaz de prover um suporte inicial às aplicações holônicas.

1.1 JUSTIFICATIVA

A utlização de SMAs aplicado a artefatos inteligentes: casas, veículos, cidades, estacionamentos, tem mostrado bons resultados em virtude das características presentes nos agentes (autonomia, habilidade social, proatividade, reatividade) (LESSER, 1995). Especificando ainda mais, no contexto de estacionamento inteligentes, em (FRAIFER; FERNSTRöM, 2016) é apresentado uma tendência da utilização da tecnologia de agentes para o domínio de estacionamentos inteligentes. (Conforme figura 1).

Figura 1 – Tendência de tecnologias para estacionamentos intelientes



Fonte: (FRAIFER; FERNSTRöM, 2016)

A forte tendência pelo uso de tecnologias de sistemas multiagentes em estacionamentos inteligentes explica-se pelo fato da dinamicidade do ambiente de um estacionamento. Nesse local é necessário o monitoramento (sensores) de veículos e decisões a serem tomadas de acordo com essas percepcões. RUSSELL; NORVIG (2004) definem um agente inteligente um software que lê o ambiente ao seu redor mediante sensores e atua por meio de atuadores nesse mesmo ambiente.

Além da utilização de sistemas multiagentes para estacionamentos inteligentes, PěCHOUčEK; MAříK (2008) citam as vantagens da utilização de agentes holônicos perante ambientes dinâmicos em que é requerido dos agentes uma capacidade de agrupamento em estruturas (*holons*) ou organizações para que sejam capazes de atingirem seus objetivos. Além do fator de agrupamento dos agentes, um agente holônico ao estar inserido em um *holon* adquire um nível maior de robustez devido ao suporte que os demais agentes o oferecem (GELBER; SIEKMANN; VIERKE, 1999). No `MAPS-HOLO` , os agentes setores constituirão *holons* , sendo assim, a capacidade de processamento de um agente setor `S1` poderá ter suporte de um agente setor `S2` para que supra uma demanda alta e temporária por vagas em seu local. Sendo assim, a utilização dos *holons* visa o apoio mútuo de diferentes agentes presentes no mesmo *holon* . Essa característica proporciona aos agentes um nível maior de adaptabilidade no ambiente em que estão inseridos.

Os autores em (IDRIS *et al.*, 2009; REVATHI; DHULIPALA, 2012) caracterizam um estacionamento inteligente como sendo um cenário dinâmico, devido aos fatores presentes na

utilização de um estacionamento, por exemplo:

- Entrada e saída de veículos;
- Alocação de vagas;
- Verificação do *status* das vagas (ocupado, reservada e livre);
- Gerenciamento e precificação das vagas;

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O objetivo principal deste trabalho é o desenvolvimento da Arquitetura `MAPS-HOLO` por meio do *framework* JaCaMo .

1.2.2 Objetivos Específicos

Além do objetivo principal, há os objetivos secundários ou específicos que serão necessários serem atingidos para que o sistema aqui proposto seja desenvolvido com êxito:

- Levantar todos os requisitos a serem empregados pela arquitetura para o estacionamento inteligente;
- Modelagem do SMA Holônico no Prometheus;
- Modelagem da Arquitetura em uma perspectiva genérica;
- Programar os diferentes grupos de agentes que compõe a arquitetura por meio do JaCaMo;
- Programar as organizações (holons) e suas normas através do JaCaMo (Moise);
- Desenvolver cenários para teste para o sistema;
- Aplicar os cenários de teste.

1.3 METODOLOGIA

Com base no metamodelo desenvolvido por RIBINO *et al.* (2012), é possível verificar a viabilidade e aplicabilidade do uso do *framework* JaCaMo para SMA Holônicos. Contudo, o

metamodelo apresenta um enfoque no aspecto organizacional do SMA Holônico (Moise). O enfoque do MAPS-HOLO compreende as três camadas do JaCaMo (programação dos agentes - Jason; desenvolvimento dos artefatos - Cartago; programação normativa - Moise). Embora o JaCaMo não apresente o suporte nativo aos *holons* nas três camadas, pode-se apontar algumas características que ele possui que indicam a possibilidade do desenvolvimento do MAPS-HOLO no JaCaMo, sendo elas:

- Artefatos no Cartago: O JaCaMo através do Cartago fornece o desenvolvimento de artefatos que possibitam aos agentes do SMA uma maior flexibilidade e um repertório de ações que podem ser desenvolvidas na linguagem de programação Java;
- Agentes BDI (*Belief, Desire Intention*): Por meio do Jason, os agentes possuem crenças a respeito de si próprios, dos demais agentes e do ambiente.
- Programação normativa: Através do Moise, é possível programar as normas que regem o SMA. Além disso, é possível estabelecer grupos de agentes, missões, objetivos de grupos, sanções, restrições de acesso. Por fim, será através do Moise que os grupos (*holons*) serão estabelecidos;

É importante destacar que o MAPS-HOLO não possui como objetivo principal a extensão do SMA desenvolvido previamente em JaCaMo (CASTRO; ALVES; BORGES, 2017). Embora a extensão desse SMA possa ser elencada como uma contribuição do atual trabalho, o MAPS-HOLO apresenta uma nova estratégia para alocação e gerenciamento de vagas em um estacionamento inteligente através de um Sistema Multiagente Holônico. Essa nova estratégia apresenta quatro grupos de agentes: *IDriver* , *PSpace* , *Sector* e *Manager* . Além disso, oferece alternativas diante das eventuais falhas que os agentes podem ter durante a execução do sistema através de uma auto reorganização dos agentes mediante o uso dos *holons* .

Finalmente, destaca-se a existência de uma linguagem de programação desenvolvida para o desenvolvimento de sistemas multiagentes com suporte nativo a agentes holônicos denominada SARL (RODRIGUEZ; GAUD; GALLAND, 2014). Embora, a linguagem SARL possa oferecer um suporte nativo a agentes holônicos, um dos objetivos do trabalho aqui proposto é utilizar os *holons* com o *framework* JaCaMo.

1.4 ESTRUTURA DO DOCUMENTO

O restante deste documento está divido da seguinte maneira: o capítulo apresenta os trabalhos correlatos ao objetivo principal desse trabalho; o capítulo 3 apresenta os conceitos gerais de sistemas multiagentes; o capítulo 4 apresenta os conceitos ampliados de Sistemas Multigentes a SMAs Holônicos; o capítulo 5 apresenta as ferramentas que serão utilizadas para o desenvolvimento deste trabalho, o framework JaCaMo e a metodologia Prometheus; o capítulo 6 a arquite-

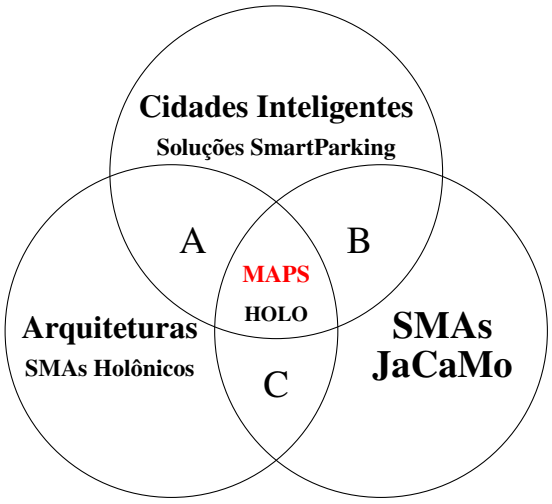
tura MAPS-HOLO em uma perspectiva genérica; capítulo o desenvolvimento do MAPS-HOLO por meio do JaCaMo ; no capítulo 8 os resultados e discussões e por fim no capítulo 9 a conclusão.

2 TRABALHOS CORRELATOS

Esse capítulo visa a descrição de trabalhos correlatos aos temas empregados pelo MAPS-HOLO , tais como: Cidades Inteligentes e *Smart Parkings*; Arquiteturas de Sistemas Multiagentes Holônicos e Sistemas Multiagentes desenvolvidos no *framework* JaCaMo. Para busca de trabalhos relacionados ao MAPS-HOLO , foram realizadas buscas baseadas em palavras chaves vinculadas ao tema do atual trabalho em algumas bases de dados, como: *IEEE Xplore, SpringerLink, ACM Library e ScienceDirect* e também em anais de eventos da área (AAMAS (Autonomous Agents and Multiagent Systems), PAAMS e HoloMAS (Holonic Multiagent Systems)).

Com a finalidade de ilustrar as áreas de abrangência do MAPS-HOLO , a figura 1 apresenta um diagrama de Venn compreendendo os principais temas do sistema aqui proposto:

Figura 2 – Áreas de Abrangência do MAPS-HOLO



Fonte: Autoria Própria

Cidades Inteligentes - Soluções Smart Parking

O termo cidade inteligente abriga diversos sub-termos, devido ao fato de uma cidade possuir diversos elementos que a compõem, não sendo diferente para uma cidade inteligente. (CARAGLIU; BO; NIJKAMP, 2011) atribuem significados ao termo *smartcity* o qual pode ser decomposto em mais seis termos, sendo eles: *smart economy, smart mobility, a smart environment, smart people; smart living, e smart governance* ou economia inteligente, mobilidade inteligente, ambiente inteligente, pessoas inteligentes, convívio inteligente e governança inteligente respectivamente. Além disso, elenca algumas cidades européias que já possuem tais características e como essas podem afetar os profissionais atuantes nestas cidades. No contexto do projeto, o termo *smart parking* enquadra-se em três categorias: *smart mobility, smart living* e *smart environment*. Em específico, o termo *smart parking* é definido como um estacionamento

que utilizam tecnologias que procuram viabilizar e automatizar as rotinas diárias de um estacionamento (REVATHI; DHULIPALA, 2012).

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

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

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

tura de informações da cidade como as informações e distâncias das vagas (NOCERA; NAPOLI; ROSSI, 2014).

Sistemas Multiagentes utilizando JaCaMo e Holônicos

A utilização do *framework* JaCaMo implica na utilização de seus diferentes componentes para o desenvolvimento de SMAs. O JaCaMo é composto pelo interpretador Jason (programação de agentes), pelo Cartago (programação dos artefatos do ambiente) e por fim pelo Moise para a programação normativa do SMA. A utilização do JaCaMo não obriga a utilização das três ferramentas simultâneas mesmo que haja uma interdependência entre elas. Em (PERSSON *et al.*, 2014) os autores apresentam um SMA desenvolvido em JaCaMo para um modelo de governança *machine-to-machine* (M2M) aplicado um estacionamento inteligente. Os autores desse trabalho colocam um enfoque no modelo organizacional baseado no Moise para a atribuição e verificação das vagas a motoristas. Embora esse trabalho seja semelhante ao sistema aqui proposto, há algumas diferenças, sendo elas: (i) O MAPS-HOLO tem o enfoque nas três dimensões do SMA e nas três ferramentas propostas pelo JaCaMo; (ii) Domínio da aplicação: O trabalho aqui proposto destina-se a estacionamentos privados, não à estacionamentos urbanos ou públicos como do trabalho citado; (iii) O SMA aqui proposto além de possuir o enfoque nos *holons* , há o aspecto da integração com o usuário final do sistema (o próprio motorista) e não uma arquitetura M2M.

Outro exemplo de SMA com JaCaMo, é o desenvolvimento de uma casa inteligente. Os autores, ZHANG *et al.* apresentam em (ZHANG *et al.*, 2014) um modelo de SMA para o controle do consumo elétrico em uma residência, onde através do uso de agentes é possível o monitoramento sem que ocorram picos de consumo elétrico. Por fim, em 2017 os autores em (FERAUD; GALLAND, 2017) apresentam uma pesquisa inicial de diferentes *frameworks* para o desenvolvimento de SMAs, sendo que o JaCaMo é citado como uma das três ferramentas mais apropriadas ao desenvolvimento de sistemas multiagentes. Além da tendência da utilização de sistemas multiagentes para estacionamentos inteligentes, juntamente com o *framework* JaCaMo, em (LEITAO; MARIK; VRBA, 2013) os autores apresentam os sistemas holônicos propensos a serem os protagonistas no área de pesquisa de sistemas multiagentes, devido ao fato da integração de holons (organizações) juntamente com os agentes autônomos.

Finalmente, em (RIBINO *et al.*, 2012) é apresentado um metamodelo de sistema holônico dirigido por regras (programação normativa). RIBINO *et al.* apresentam um meta-modelo desenvolvido no JaCaMo, onde um modelo organizacional baseado no Moise (MOISE, 2002) é extendido do modelo normativo já existente para sistemas multiagentes holônicos (COSSENTINO *et al.*, 2010). A motivação dos autores é devido ao fato de que o ASPECS não abrange as tarefas a serem realizadas pelas organizações e nem as regras para assegurar a realização proveita destas atividades. Em contrapartida, o Moise possui o suporte. Sendo assim, os autores desenvolveram

um metamodelo de programação normativa para os sistemas multiagentes holônicos no Moise. Entretanto, o trabalho visou o enfoque no Moise e não apresentou o desenvolvimento dos agentes holônicos propriamente ditos em Jason.

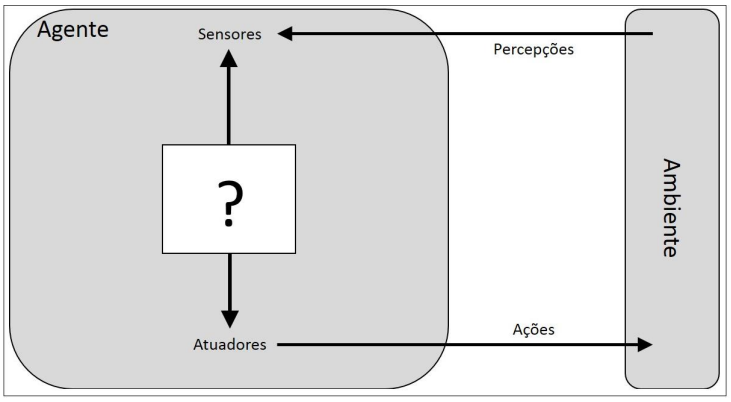
3 SISTEMAS MULTIAGENTES

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

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

Além disso, RUSSELL; NORVIG (2004) definem que um agente é capaz de perceber seu ambiente por meio de sensores e de agir sobre esse ambiente por meio de atuadores. Por exemplo, em um ser humano são os olhos e ouvidos os sensores e mãos, boca, pernas e outras partes do corpo como atuadores ou até mesmo sensores. A figura 3 ilustra um agente interagindo com o ambiente por meio dos atuadores.

Figura 3 – Agente interagindo com o ambiente por meio de sensores e atuadores



Fonte: Adaptado de (RUSSELL; NORVIG, 2004)

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

Finalmente, um SMA é um conjunto de agentes que compartilham o mesmo ambiente e são comunicáveis entre si, porém, cada agente pode ter um grau maior de influência que um outro agente no mesmo ambiente, caracterizando-se assim a possibilidade de níveis de hierarquia dentro do sistema multiagente (WOOLDRIDGE, 2009).

A seguir são apresentadas oito características que um agente pode possuir, ainda que um agente não apresente todas essas características simultâneamente, ao menos deve apresentar um nível de autonomia.

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

  Isso pode implicar um problema em um sistema multiagente, onde a cooperação entre os agentes pode vir a falhar devido ao fato de que eles podem escolher não cooperar com os demais para atingir seus próprios objetivos ao invés de cooperar. Sendo assim, é necessário determinar qual nível de autonomia o agente deve possuir afim de que o SMA possa usufruir da cooperação entre os agentes (LESSER, 1995).

- **Proatividade**: Define a capacidade de um agente tomar a iniciativa para alcançar um objetivo que lhe foi delegado. Difere-se do conceito de objetos, onde espera-se que passivamente que seja invocado por algum método para que assim uma tarefa seja realizada (BORDINI; HüBNER; WOOLDRIDGE, 2007).

- **Reatividade**: Para Vasant Honavar (1999), a reatividade de um agente é a sua capacidade de reagir ao ambiente em que está inserido. Essa reação pode ser através de um reflexo do agente (mais rápida) ou através de uma reação com um certo grau de cognitividade (mais lenta);

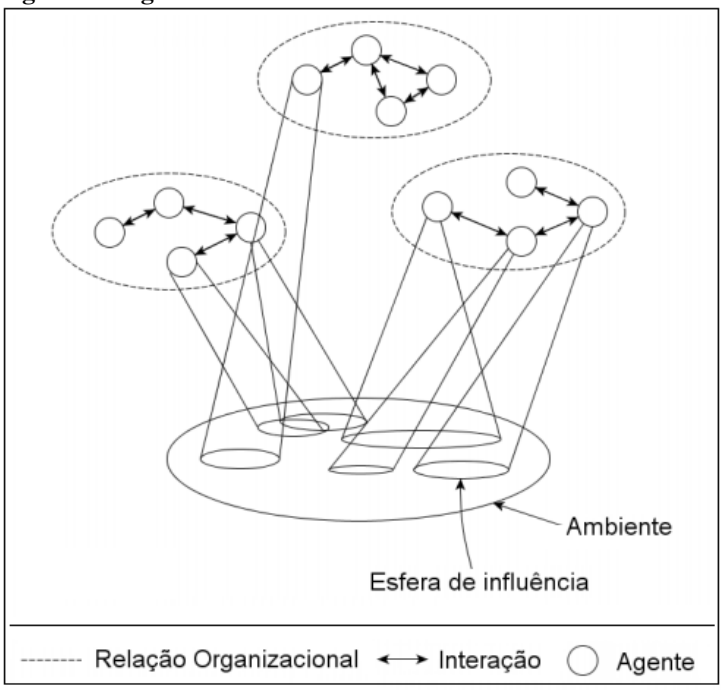
- **Habilidade Social**: O processo de comunicação entre os agentes é um ponto fundamental em um sistema multiagente, visto que, diferentes agentes precisam de informações a todo momento sobre o ambiente e de outros agentes. Essas informações, na maioria das vezes, é proveniente da troca de mensagens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

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

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

Como citado no início desse capítulo, diferentes agentes podem estar inseridos no mesmo ambiente com diferentes níveis de influência nesse ambiente, caracterizando um ambiente de cooperação-competição, gerando assim, áreas de conflito entres esses mesmos agentes. A figura 4 ilustra essa característica.

Figura 4 – Agentes com áreas de influência



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

Além das características de autonomia, reatividade, proatividade e habilidade social, NWANA (1996) apresenta outras propriedades que caracterizam um agente:

- **Raciocínio e aprendizado**: O agente pode possuir uma base de conhecimento a respeito de si próprio e do ambiente em que está situado. Sua capacidade de raciocínio e aprendizado da-se então baseado nas inferências que ele pode fazer de acordo com o seu conhecimento. Para realizar tais inferências é necessário que o agente saiba buscar informações relevantes. Além disso, de acordo com o ambiente em que está inserido, o agente deve verificar se as informações da sua base condizem com as condições do ambiente;
- **Mobilidade**: Capacidade do agente transitar entre diferentes ambientes e *workspaces*. Além disso, o agente pode ser capaz de transitar entre diferentes plataformas;

- **Personalidade**: O agente pode adotar características de uma personalidade humana que influenciam no seu comportamento no ambiente e com os demais agentes;
- **Adaptabilidade**: Vinculado a autonomia do agente, uma vez que esse pode se adaptar de acordo com novas condições do ambiente ou interações com os demais agentes do sistema.

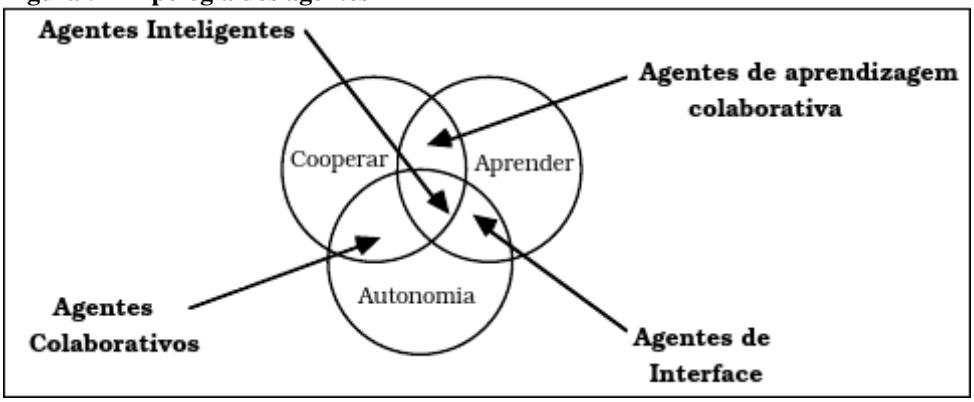
3.1 TIPOLOGIA DE AGENTES

A tipologia dos agentes visa os classificá-los de acordo com suas propriedades, sendo estas citadas na seção anterior. Características como autonomia, habilidade social, personalidade e entre outras estão presentes nos agentes. Entretanto, em alguns agentes um conjunto dessas características pode estar mais evidente do que em outros em virtude do seu tipo e/ou do ambiente em que está inserido. Além disso, um agente pode ser descrito como um agente:

- Mobilidade: Habilidade do agente se mover em um determinado meio (e.g. rede, workspaces). Um agente móvel é o que possui tal habilidade; Agente estático permanece em um único meio;
- Deliberativos ou reativos: Os agentes deliberativos possuem um estado interno, um modelo de raciocínio e eles cooperam entre si no planejamento e negociação para atingir um objetivo comum. Diferente dos deliberativos, os agentes reativos não possui um estado interno simbólico, um modelo simbólico de raciocínio, eles agem através de uma reação ou resposta a uma mudança no ambiente em que estão inseridos (NWANA, 1996; WOOLDRIDGE, 2009).

Com base nestas características, a figura 5 ilustra quatro tipos de agentes: agentes colaborativos, agentes de aprendizagem colaborativa, agentes de interface e agentes inteligentes (*truly*).

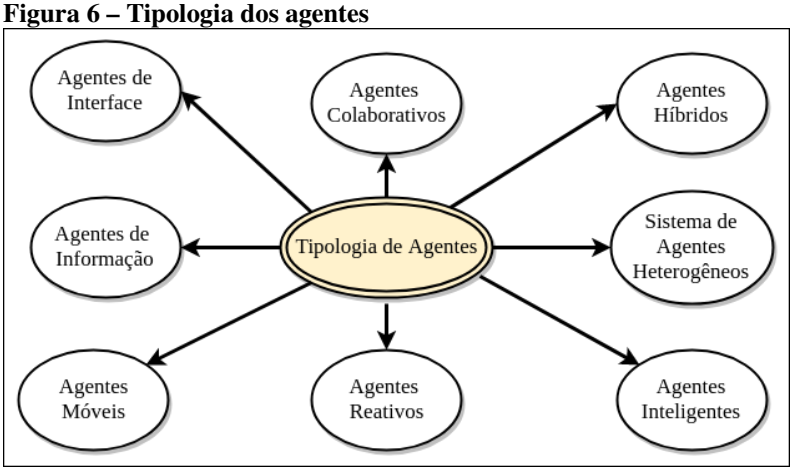
Figura 5 – Tipologia dos agentes



Fonte: Adaptado de (NWANA, 1996)

Um agente colaborativo é descrito como um agente que coopera com os demais para atingir um objetivo, sendo necessário a autonomia deste agente para cooperar com os demais. Já o agente de aprendizagem colaborativa, é aquele que coopera com os demais, além disso, através do processo de cooperação há o fator de aprendizagem com os demais agentes. O agente de interface possui a ênfase em autonomia e aprendizado. Por fim, o agente inteligente é visto como o *truly smart agent* ou verdadeiramente inteligente, uma vez que esse agente é capaz de aprender, cooperar e ter autonomia. Com base nisso, (NWANA, 1996) apresenta sete tipologias para os agentes ilustrada na figura. 6.

Figura 6 – Tipologia dos agentes



Fonte: Adaptado de (NWANA, 1996)

- **Agentes de Interface:** Diferente do agente colaborativo em que os agentes colaboram entre si, o agente de interface colabora com o usuário dando ênfase na autonomia e aprendizado. Um exemplo de agente de interface pode ser denotado como um agente como um assistente pessoal, em que ele observa as ações do usuário (autonomia) e até imita-as, aprende com elas (aprendizado) e assim consegue cumprir tarefas para o seu proprietário. Além disso, este agente pode aprender de acordo com os *feedbacks* que o usuário fornece a este agente.

- **Agentes móveis:** Geralmente aplicavéis a ambientes onde tais agentes podem transitar entre redes, redes WANs por exemplo. Um agente móvel pode interagir com outros sistemas por uma rede externa a fim de obter novas informações para o seu proprietário. Tal transitividade desses agentes requerem um grau de autonomia e cooperação entre os diferentes agentes situados em redes/meios distintos;

- **Agentes de Informação:** Agente focado na coleta, armazenamento e gerenciamento de informações. O ambiente em que está inserido contém uma número demasiado de informações necessárias a serem analisadas e gerenciadas por esse agente. Um exemplo desse tipo de agente são os *internet bots* responsáveis pela varredura de websites de maneira autônoma com o objetivo de catalogar e gerenciar informações contidas em sítios.

- **Agentes reativos:** Um agente que não possui uma representação interna do ambiente em que está inserido. Este tipo de agente geralmente apenas age através de uma reação imediata a uma mudança no ambiente que ele se encontra. Diferente do agente deliberativo, o agente reativo tem uma tolerância maior a falhas devido a sua falta de conhecimento a respeito do meio em que está contido.

- **Agentes Híbridos:** Cada tipo de agente apresenta seus pontos fortos e fracos. Em diferentes domínios, diferentes tipos de agentes serão empregados. Contudo, um agente pode possuir mais de uma tipologia, este agente poderá ser um agente colaborativo e reativo. Sendo assim, um agente hibrido pode possuir duas ou mais tipologias a fim de atuar em um domínio que requira tais tipologias;

- **Sistema de agentes heterogêneos:** Sistema composto por dois ou mais agentes, sendo esses agentes sendo de dois ou mais tipos (híbridos). Um dos desafios desta tipologia é a troca de mensagens, uma vez que os agentes são de tipologias distintas. Uma solução para isso é a utilização de um *transducer* (Software de tradução de mensagens) entre os agentes.

3.2 ARQUITETURA BDI

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

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

- Crença: São as informações que o agente possui sobre o mundo ou ambiente em que está situado. Essa informação poderá estar desatualizada ou imprecisa.

- Desejos: Define-se por ser todos os possíveis estados que o agente pode desejar alcançar. Entretanto, ter um desejo não implica que o agente automaticamente irá satisfazê-lo ou agir conforme isto. Desejo é uma influência em potencial nas ações do agente. Outro ponto que é fundamental é o fato que um desejo pode ser incompatível com outro desejo. De modo geral, desejos são todas as opções que o agente possui a fim de atingir um objetivo.

- Intenções: São os estados que o agente decidiu seguir em frente, ou até mesmo os objetivos que são delegados a este agente. De modo geral, um agente analisa todas os seus

desejos para atingir um objetivo. Ao escolher este desejo, este mesmo desejo torna-se uma intenção. Portanto, um agente com um objetivo a ser cumprido, analisa os seus desejos, escolhe uma ou mais intenções a fim de atingir e completar esse objetivo.

No código 1 é apresentado um exemplo básico de um agente baseado no modelo BDI. O agente inicialmente acredita que o agente é feliz e tem como objetivo dizer *"hello"*. Na linha 2 é utilizado o operador !, o qual indica um objetivo a ser atingido. Na linha 3 é exibido o agente executando a intenção de dizer *"hello"*. Na linha 3 é apresentado um plano, o qual é descrito com detalhes na seção 5.1.3.3.

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

Código 1 – Exemplo de um agente BDI,

Para (BORDINI; HüBNER; WOOLDRIDGE, 2007) o conceito de desenvolver sistemas computacionais em termos de noções mentais, tais como: crença, desejo e intenção é o componente chave para o modelo BDI. Esse conceito pode-se entender como um novo paradigma: programação orientada a agentes (AOP), o qual se baseia em alguns argumentos, tais como:

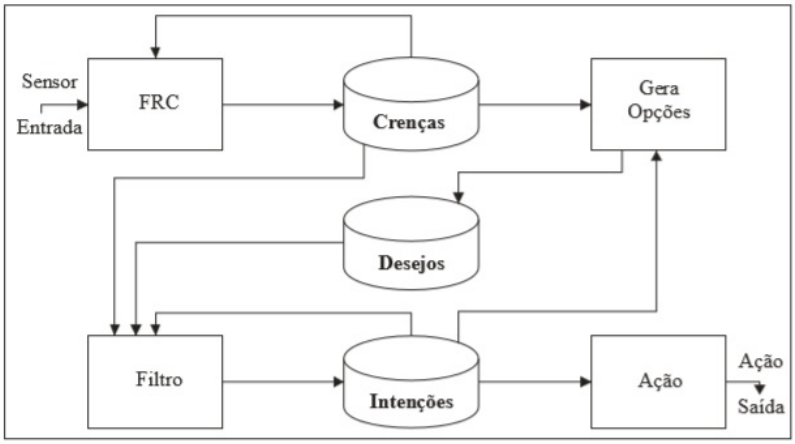
- Esse paradigma proporciona uma maneira não-técnica para tratar de sistemas complexos. Para nós, não é necessário nenhum treinamento formal para entender o sistema ou a maneira mental de raciocionar, pois isso é uma parte da nossa capacidade linguística do dia-dia.

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

Dessa forma, agentes BDI são programas inseridos em um ambiente totalmente dinâmico, onde continuamente recebem estímulos provindos desse ambiente, realizando ações (intenções)

baseadas no seu estado mental e seu conhecimento sobre o mundo (crenças), bem como a todo momento analisando as opções disponíveis (desejos) a fim de atingir o objetivo a ele delegado. Na figura 7 é ilustrado a arquitetura genérica do modelo BDI.

Figura 7 – Arquitetura BDI genérica



Fonte: Adaptado de (WOOLDRIDGE, 2009)

Onde o módulo FRC da figura 7 (Função de Revisão de Crenças) recebe informações provindas do ambiente ao qual o agente está inserido, podendo assim ler e atualizar a base de crenças do agente. Com as alterações do ambiente, pode-se atribuir novos objetivos ao agente. Já a função "Gera Opções" é responsável pela elaboração de novos desejos (estados) que o agente irá possuir e verificar se esses estados serão atingidos, bem como as intenções com as quais o agente já está comprometido. A função "Filtro"é utilizada para atualizar o conjunto de intenções de acordo com as crenças que o agente possui.

3.3 PARADIGMAS ORGANIZACIONAIS DE SISTEMAS MULTIAGENTES

A arquitetura de um sistema multiagente descreve como os agentes se organizam e interagem entre si no ambiente em que estão inseridos (SHEHORY, 1998). Além disso, ela define as características estruturais dos agentes que compoêm este sistema. Alguns autores definem a classificação dos agentes como uma arquitetura de agentes (RUSSELL; NORVIG, 2004) e (WOOLDRIDGE, 2009), outros como tipologia de agentes (NWANA, 1996) (Ver seção 3.1).

Outro fator a ser elucidado na definição de uma arquitetura de agente é o seu aspecto organizacional. O aspecto organizacional apresenta os características organizacionais de um sistema multiagente a respeito do comportamento dos agentes em diferentes situações, também denominado como paradigma organizacional de um SMA (HORLING; LESSER, 2004). A escolha de um determinado paradigma organizacional varia de acordo com o domínio da aplicação. Além disso, há domínios que requerem a utilização de mais de um paradigma a fim de reduzir a complexidade do SMA a ser empregado.

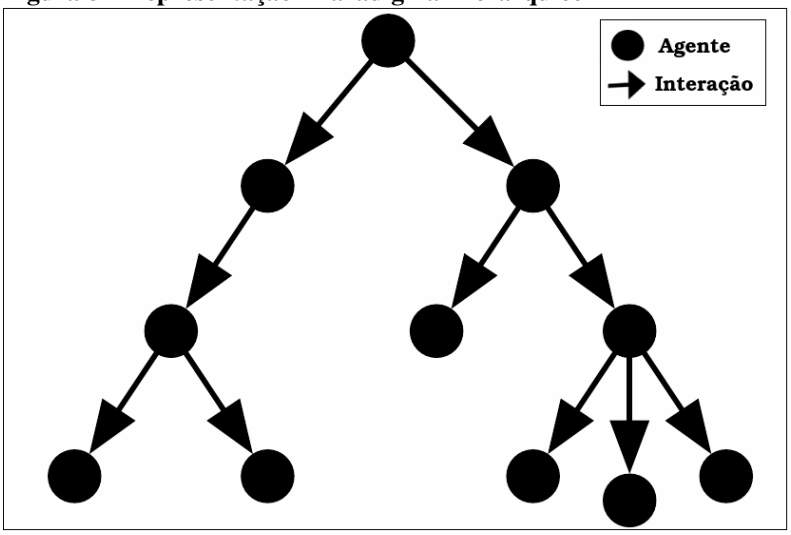
Os autores em (SHEHORY, 1998) apresenta o modelo *flat* ou plano como sendo um

modelo organizacional. Por sua vez outros autores como (HORLING; LESSER, 2004) não menciona como um modelo ou paradigma organizacional devido a sua simplicidade. O modelo *flat* descreve os agentes com o mesmo grau de influência sobre todos os demais, o que implica que cada agente pode interagir diretamente com qualquer outro agente ao seu alcance. Todavia, os agentes podem dinamicamente estabelecer estruturas organizacionais temporárias a fim de executarem uma determinada tarefa. Por fim, dado que os agentes podem interagir com qualquer outro agente em seu ambiente, isso pode gerar um alto grau de *overhead* devido ao grande número de mensagens trocadas entre diferentes agentes (e.g. *via broadcast*). Nas seções a seguir são apresentado os paradigmas hierárquico e de mercado e no próximo capítulo o paradigma holônico.

3.3.1 Paradigma Hierárquico

No modelo organizacional hierárquico os agentes ficam dispostos na estrutura de uma árvore. A figura 8 ilustra como os agentes ficam dispostos nessa estrutura.

Figura 8 – Representação - Paradigma Hierárquico



Fonte: Adaptado de (HORLING; LESSER, 2004)

Devido aos agentes estarem dispostos em um formato de árvore, os agentes em níveis superiores possuem uma visão mais global de outros agentes, de forma que os agentes mais próximos a raiz (ou a própria raiz) possuem um maior grau de influência sobre os agentes sob eles. Sendo assim, os agentes mais próximos as folhas tem um grau menor de influência sobre os demais agentes. Outro fator é a inderdependência dos agentes para a comunicação para comunicação e cooperação. Os agentes inferiores dependem dos seus superiores para trocarem e enviarem mensagens a outros agentes presentes no ambiente.

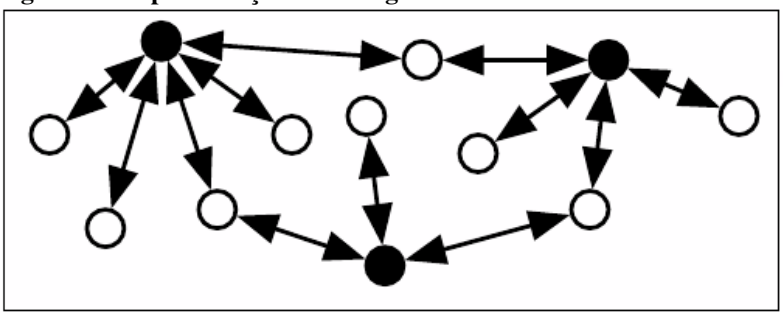
Uma das vantagens de um SMA utilizar o modelo hierárquico é o baixo o índice de *overhead* de mensagens, uma vez que os agentes possuem a informação da rota para o agente

destino da mensagem. Porém, há o fator falha como desvantagem, pois, o agente superior (*apex agent*) da hierarquia pode vir a falhar vindo a ruir toda a estrutura. Há soluções que visam a solução deste ponto falha do agente (*apex*), como a eleição de agentes inferiores para que seja escolhido um novo *apex*. Contudo, a solução da eleição pode ser muito custosa para o SMA, pois toda a estrutura deverá ser reoordenada. Uma outra solução, é a instanciação de um novo agente para o *apex*. Esse ponto central de falha do paradigma hierárquico tem sido um dos principais motivos a pesquisadores desenvolverem novos paradigmas para diferentes domínios. O paradigma holônico deriva do paradigma hierárquico, porém como uma nova visão de como os agentes se organizam. Na próxima seção é descrito as características dos sistemas multiagentes holônicos bem como o seu modelo de interação.

3.3.2 Paradigma Mercado

O paradigma de mercado é ilustrado na figura 9. Nesse paradigma os agentes são dispostos entre compradores (círculos brancos) e vendedores (círculos pretos) e realizam negociações entre si em busca de recursos. As negociações podem ser através de barganha ou através de leilões. Uma forma de negociação entre agentes é através do protocolo *Contract Net* (SMITH, 1980), onde através de *rounds* de negociação, tanto o vendedor quanto o comprador podem chegar em um acordo por um determinado preço de um produto ou recurso.

Figura 9 – Representação - Paradigma Mercado



Fonte: (HORLING; LESSER, 2004)

4 SISTEMAS MULTIAGENTES HOLÔNICOS

Nesse capítulo é apresentado uma breve introdução da utilização dos *holons* , seus conceitos e suas similaridades com os agentes. Por fim, é descrito os agentes holônicos e suas formas de organização.

4.1 INTRODUÇÃO

A crescente utilização de Sistemas Multiagentes em diferentes domínios de aplicação deve-se a esses sistemas proporcionam um nível de autonomia aos agentes. Porém, tal autonomia em um grande conjunto de agentes pode vir a tornar-se um problema, uma vez que é necessário a coordenação e organização dos agentes que compõe o sistema. O conceito de organização em um Sistema Multiagente é descrita como entidades em que os agentes agrupam-se para atingir um objetivo em comum em uma determinada forma de organização ou governo (paradigma organizacional). Dentre as formas de organização de um Sistema Multiagente, o modelo que tende a ser o mais utilizado é o hierárquico (ver figura 8), devido a sua simplicidade de implementação e aplicação em diferentes cenários (ODELL; NODINE; LEVY, 2005).

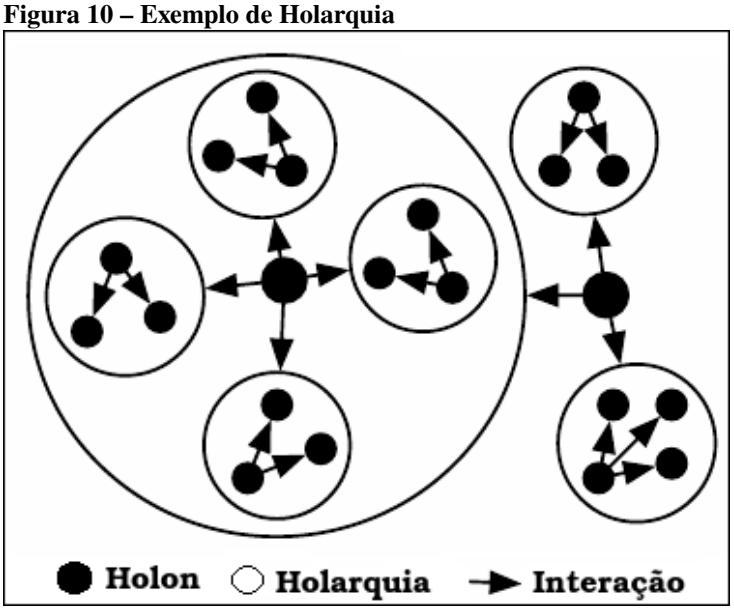
Derivada da forma hierárquica (ou paradigma organizacional hierárquico), o paradigma holônico visa estruturar o sistema multiagente em organizações denominadas de *holons* , os quais juntos formam as holarquias. O termo *holon* foi inicialmente cunhado por Arthur Koestler em sua obra *The Ghost in the Machine* em 1969. O objetivo de Koester foi explicar que o universo é formado simultaneamente por partes e todos, ou um todo que é composto por partes e as partes que compõem o todo. O termo grego *holon* significa ***holos*** ou inteiro. Já o sufixo ***on*** significa parte.

O conceito de *holon* pode ser aplicado a uma sociedade em que cada indivíduo é uma parte do todo, que é a sociedade propriamente dita. Um outro exemplo pode ser o próprio corpo humano, em que o *holos* é o organismo e o *on* são os orgãos, tecidos, fluídos, etc. As próprias partes podem ser consideras como sendo um todo e uma parte simultâneamente. Por exemplo, o coração humano, é um *on* do *holos* organismo. Porém, o próprio coração é um *holos* ao analisarmos o seu interior (ventrículos, tecidos, etc.). Essa composição de todo e parte de uma holarquia, ou diversas delas, é ilustrado na figura 10, onde é apresentado um conjunto de holarquias compostas por seus respectivos *holons* .

(GELBER; SIEKMANN; VIERKE 1999) apresentam que as holarquias e os *holons* possuem vantagens em comparação a hierarquias ou até o mesmo o modelo *flat* de organização, sendo elas:

- Robustez em face a mudanças no ambiente externo e interno de uma holarquia;

Figura 10 – Exemplo de Holarquia



Fonte: Adaptado de (HORLING; LESSER, 2004)

- Eficiência na alocação e uso de recursos por parte dos *holons* ;
- Adaptação em frente a mudanças no ambiente em que os *holons* e as holarquias estão inseridas;

A partir da definição de Koester dos *holons* para a sociedade e ao próprio universo, o termo *holon* foi inicialmente utilizado em Sistemas de Manufatura (SM), onde cada etapa da produção de um determinado produto (estoque, produção, entrega, etc.) é abstraído para um *holon* . Sendo assim, cada *holon* é responsável por uma etapa de produção da indústria, cunhando assim o termo Sistema de Manufatura Holônico (SMH). As vantagens do uso de um SMH é devido as características que um *holon* possui (autonomia e cooperação). A característica autônoma de um *holon* denota-se como a sua capacidade de criar e controlar a execução de seus próprios planos e estratégias para que possa cumprir seus objetivos. Por fim, a cooperação de um *holon* é a capacidade de interagir e auxiliar os demais *holons* de sua holarquia para que possam atingir objetivos em comum.

Além das características de autonomia e cooperação, um *holon* possui habilidades sociais de interação com os demais *holons* , reatividade a estímulos provindos da holarquia (ou ambiente), e até mesmo proatividade (BOTTI; GIRET, 2008; GIRET; BOTTI, 2004). Por fim, o *holon* é recursivo, ou seja, dentro de um *holon* podem haver outros *holons* . Assim, com base nessa recursividade, define-se que:

- ***Super holon***: *Holon* que agrega dentro de si mesmo todos os demais *holons* presentes no sistema;
- ***Holon composto***: *Holon* que possui diversos *holons* acoplados em seu interior;
- ***Holon atômico***: *Holon* que não possui nenhum *holon* em seu interior;

- ***Holarquia***: Sistema de *holons* que podem cooperar para atingir um objetivo em comum. Além desse objetivo, os *holons* dentro de uma holarquia podem compartilhar recursos. A holarquia além de agrupar os *holons* , ela define as regras básicas de cooperação e interação dos seus *holons* mediante um contrato.

Diferente dos Sistemas de Manufatura Holônicos, a utilização dos *holons* também foi aplicada aos Sistemas Multiagentes. As propriedades que caracterizam um *holon* são similares a um agente: autonomia, reatividade, proatividade e habilidades sociais. Todavia, há características que os diferenciam, por exemplo a recursividade. Embora existam trabalhos relacionados a agentes recursivos, ainda assim há diferenças no modo em que um agente recursivo utiliza tal propriedade do que um *holon* . As principais diferenças dessa propriedade é a aplicação da recursividade propriamente dita, uma vez que as aplicações de um *holon* são mais no âmbito teórico na organização de sistemas, ao passo que nos agentes tal recursividade deve ser implementada em nível de código. Finalmente, podemos observar então a similaridade dos agentes com os *holons* , uma vez que eles compartilham algumas propriedades. Em virtude disso, GIRET; BOTTI em *Holons and Agents* apresentam as diversas compatibilidades entre os *holons* e agentes. A conclusão que os autores chegam é que um *holon* é uma espécie de agente.

> Nós podemos dizer que um *holon* é um caso especial de agente. No nível mais baixo de implementação, ambos modelos (*holon* e agente) são simplesmente blocos de código executável com um fluxo de dados de acordo com uma implementação específica. Conceitualmente, o *holon* é um tipo especial de agente. (GIRET; BOTTI, 2004, 5)

4.2 HOLONS E AGENTES

O termo agente holônico deriva da junção entre agente e *holons* . Um agente holônico pode ser composto por múltiplos agentes ou sub-agentes. Inicialmente, o conceito foi citado em (FISHER, 1999) por Klaus Fischer como agentes de estrutura holônica, onde tais agentes possuem a capacidade de comunicação com o ambiente (exterior) e também com os seus sub-agentes (interior). Um agente holônico pode ser observado em duas perspectivas. Primeiramente, a perspectiva interna desse agente visa demonstrar as relações de comunicação e cooperação que ele possui com os seus sub-agentes. A segunda perspectiva é a exterior em que esse agente composto por seus sub-agentes é visto como um único agente perante os demais agentes holônicos.

Além da definição de que um agente holônico pode ser composto por outros agentes, um agente holônico pode ser atômico, ou seja, ele não possui agentes acoplados em seu interior. Sendo ele atômico, esse agente tem a autonomia de escolha para que possa formar um agente holônico composto com os demais agentes presentes no sistema, formando assim um *holon* . Assim, um *holon* no contexto de um Sistema Multiagente Holônico (SMAH) é uma organização que é composta por agentes holônicos. A motivação para que os agentes holônicos possam for-

mar um *holon* é a busca em atingir um objetivo em comum ou até mesmo o compartilhamento de recursos. A seguir são apresentadas algumas definições que são aplicadas ao passo que agentes holônicos atômicos passam a formar um *holon* .

- **Autonomia:** Os agentes holônicos ao participarem de um *holon* formam uma única entidade. Na perspectiva do ambiente, o *holon* interage com o ambiente com as mesmas características de autonomia de um agente. Contudo, aos agentes ingressarem em um determinado *holon* , os agentes abrem mão de parte da sua autonomia para se submeterem a autonomia presente do *holon* , porém, esses agentes mantém a liberdade de saírem do *holon* ;
- **Comportamento baseado em objetivos:** O agente ao vincular-se a um *holon* atenta-se aos objetivos globais deste *holon* . Porém, os objetivos que o agente tem para cumprir não devem conflitar com os objetivos globais do *holon* . Por fim, o agente pode cumprir paralelamente os seus objetivos particulares e os objetivos do *holon* ;
- **Capacidade ampliada através do grupo:** As capacidades do *holon* são extendidas na medida que o número de agentes presentes cresce. Um *Super Holon* podem ter a sua disposição habilidades que nenhum outro agente poderia executar sozinho;
- **Crenças:** Um *holon* possui crenças sobre ele mesmo, assim como crenças do ambiente em que está inserido. Da mesma forma, os agentes que estão vinculados a esse *holon* podem possuir crenças diferentes das do *holon* .
- **Racionalidade limitada:** O *holon* possui *sub holons*, assim como está contido em um *Super Holon*. Assim, o *super-holon* determina as diretrizes sobre os recursos que os *sub-holons* devem possuir;
- **Comunicação:** A comunicação para os agentes holônicos é um fator importante para a sua autonomia. Há a comunicação dentro de um *holon* , em que os agentes podem se comunicar diretamente entre si, ou através de uma mediação do *holon* . Além disso, há a comunicação entre os *holons* é mediada pelos canais de comunicação proporcionados pelo *Super Holon*;
- **Contrato:** Estabelece as normas e/ou objetivos que a organização holônica possui. Os agentes ao formarem ou ingressarem nesse *holon* devem estar de acordo com o que é estabelecido no contrato.

4.2.1 Definição Formal

FISCHER; SCHILLO; SIEKMANN (2003) afirmam que um SMA é dito holônico quando composto por um conjunto $\mathcal{A}_t$ de agentes, conjunto $\mathcal{H}_t$ de todos os *holons* no tempo

t e é definido na seguinte forma:

Definição 1 (Sistema Multiagente Holônico)**.** *Um Sistema Multiagente $\mathcal{MAS}$ que contém holons é chamado de Sistema Multiagente Holônico. O conjunto $\mathcal{H}$ de todos os holons do SMA é definido recursivamente:*

- *para cada $a \in \mathcal{A}_t, h = (\{a\}, \{a\}, \emptyset) \in \mathcal{H}$, i.e. cada agente instanciado constitui um holon atômico, e*
- *$h = (\texttt{Cabeça}, Subholons, C) \in \mathcal{H}$, onde os Subholons $\in 2^{\mathcal{H}} \backslash \emptyset$ é o conjunto dos holons que participam em h, $\texttt{Cabeça} \subseteq$ Subholons é o conjunto não vazio que representa o holon para o ambiente e é responsável pela coordenação das ações dentro do holon. $C \subseteq \texttt{Contratos}$[1] define a relação dentro do holon e é acordada por todos os holons $h' \in$ Subholons no momento de entrada no holon h.*
- *Dado o holon $h = (\texttt{Cabeça}, \{h_1, ..., h_n\}, C)$ denotamos $h_1, ..., h_n$ os Subholons de h, e h o Superholon de $h_1, ..., h_n$. O conjunto $\texttt{Corpo} = \{h_1, ..., h_n\} \backslash \texttt{Cabeça}$ (Complemento de $\texttt{Cabeça}$ em h) é o conjunto dos Subholons que não são permitidos para representar o holon h. Os holons são permitidos para participarem em diferentes Subholons ao mesmo tempo, desde que não entrem em contradição com os $\texttt{Contratos}$ dos respectivos Superholons.*

Definição 2 (Delegação de tarefas - IntraHolon)**.** *Delegação de tarefas entre subholons h_1 e h_2 de um holon h é parte da cooperação dos holons para atingir o objetivo de h.*

Definição 3 (Holarquia)**.** *Um holon que não é atômico é denominado uma holarquia. Uma holarquia é uma organização composta por todos seus Subholons aninhados, sendo que esses subholons podem ser holarquias. Por fim, toda holarquia possui um holon cabeça que é o responsável pelos demais holons ($\texttt{Corpo}$).*

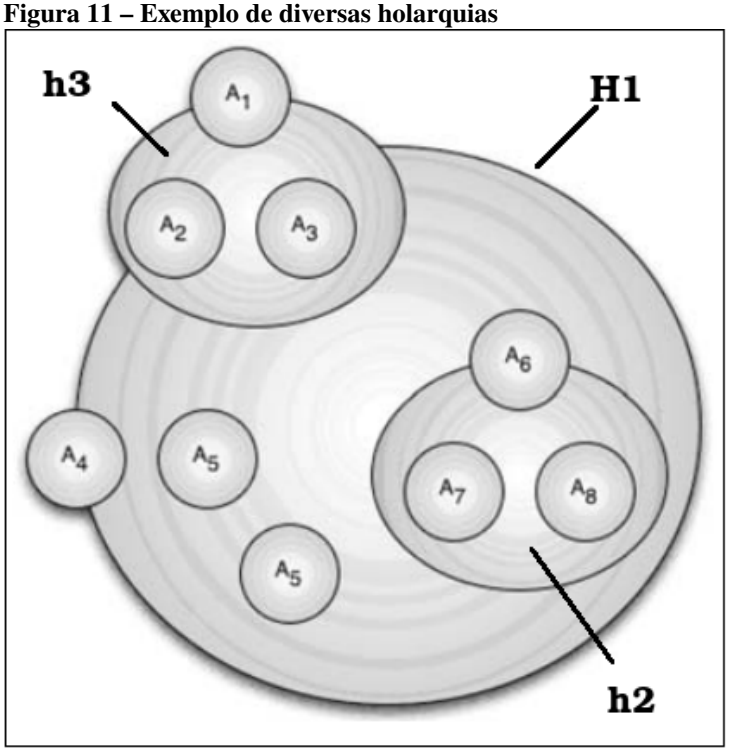
A figura 11 apresenta um exemplo de diversas holarquias e *holons* atômicos. O círculo maior representa a holarquia $H1$ que contém todos os demais *holons* $\{A_1, ..., A_8\}$. A Holarquia $H1$ é denominada também como *Superholon* $H1$ e possui dois *holons* cabeça $\{A_1, A_4\}$. O *holon* A_4 é atômico, já o A_1 é o *holon* cabeça da holarquia $h3$. Além disso, os *holons* $\{A_1, A_2, A_3, A_6, A_7, A_8\}$ pertecem a duas holarquias simultaneamente $h3$ e $h2$.

4.2.2 Organização Holônica

A utilização de diferentes *holons* para a formação de uma holarquia, faz-se necessário a definição de como os *holons* interagem entre si. Na seção anterior foi definido que dentro de

[1] Estabelece as normas de utilização do *holon* . Além disso, atravé do Contrato é possível estabelecer os objetivos comuns do *holon* .

Figura 11 – Exemplo de diversas holarquias



Fonte: Adaptado de (SCHILLO; FISCHER, 2004)

uma holarquia há um *holon* ou mais responsáveis pelos demais *holons* que compõe a holarquia. Entretanto, há a possibilidade de diferentes configurações de como os *holons* cooperam entre si. A seguir são descritas três formas de organização de uma holarquia: conjunto de agentes autônomos, junção dos agentes e sociedade moderada (FISCHER; SCHILLO; SIEKMANN, 2003).

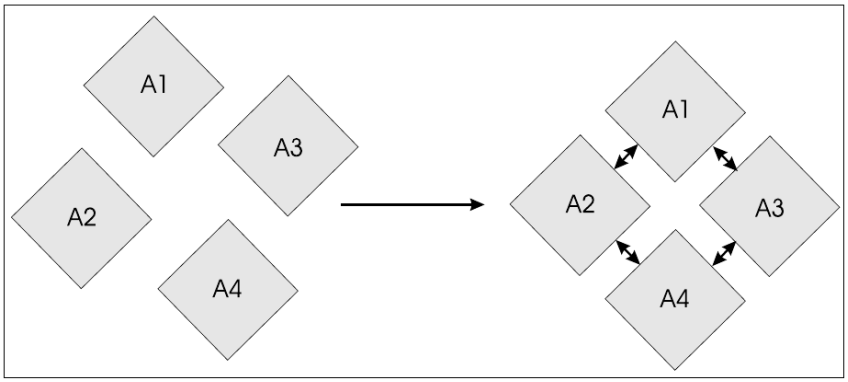
4.2.2.1 Holon como um conjunto de agentes autônomos

Nesse modelo de organização os agentes continuam com o mesmo nível de autonomia, ou seja, nenhum agente que compõe o *super-holon* perde autonomia. Além disso, o *super-holon* criado a partir desses agentes é visto como uma entidade composta pelos agentes que estão comprometidos com o *holon* e não como um único agente. A figura 12 ilustra esse modelo.

Para o correto funcionamento desse modelo, os agentes que compõe o *super-holon* possuem um contrato entre si para que seja possível a cooperação e interação entre eles para que possam atingir um objetivo em comum. A maneira mais transparente para que isso ocorra é mediante a comunicação explícita pelo envio de mensagens entre os agentes para que possam manter a cooperação mútua.

A representação formal para esse modelo pode ser denotada de maneira similar a um SMA, sendo descrito como um *holon* $h = (\{A_1, A_2, A_3, A_4\}, \{A_1, A_2, A_3, A_4\}, C_{contrato\_autnomos})$

Figura 12 – Exemplo de diversas holarquias

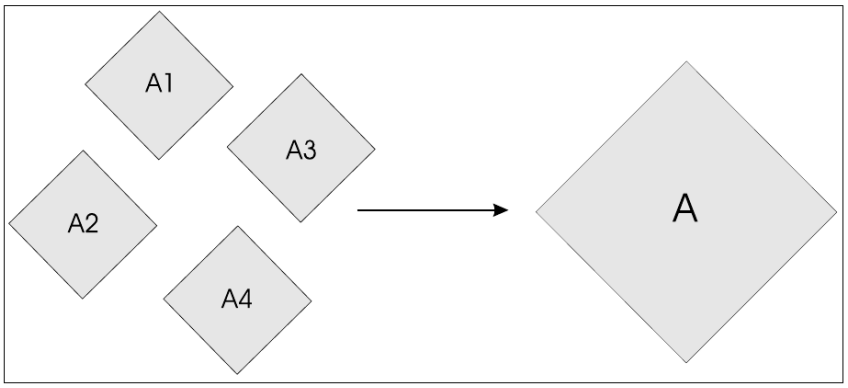


Fonte: (FISCHER; SCHILLO; SIEKMANN, 2003)

4.2.2.2 Diferentes agentes agrupam-se em único agente

Diferente do modelo anterior em que *super-holon* continuava a ser visto como um conjunto de agentes, nesse modelo os agentes agrupam-se (*merge*) para formar um novo agente. Nesse modelo, é importante que os agentes compartilhem da mesma estrutura e arquitetura interna para que possam ser compatíveis para a formação de um novo agente como um *super-holon* . Destaca-se aqui a vantagem da utilização de agentes BDI, uma vez que através da união das crenças, desejos e intenções dos sub-agentes é possível formar um novo agente que possui as características daqueles que o formaram. Além disso, nesse modelo os agentes que agrupam-se para a formação do novo agente desistem completamente de sua autonomia. Consequentemente, o novo agente é considerado um *holon* atômico $h = (\{A\}, \{A\}, C_{fundir})$. A figura 13 apresenta o processo de quatro agentes unindo-se para a formação de um único agente.

Figura 13 – Exemplo de diversas holarquias

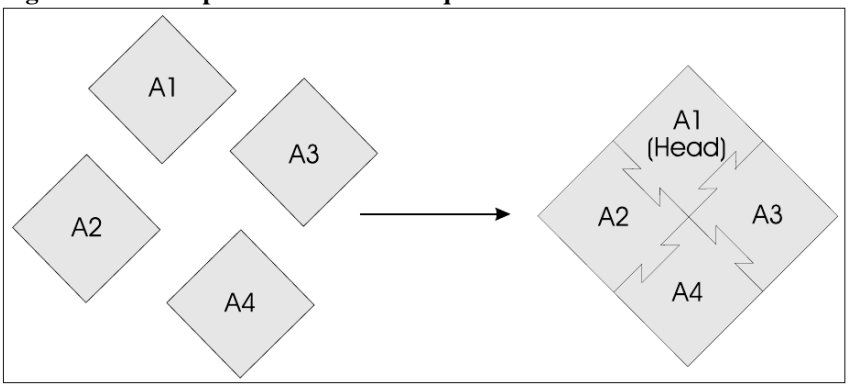


Fonte: (FISCHER; SCHILLO; SIEKMANN, 2003)

4.2.2.3 Holon como uma sociedade moderada

Os dois modelos apresentados acima ocupam dois extremos nos modelos de organização holônica. No primeiro modelo os agentes mantém o seu nível de autonomia e mantém as suas estruturas para formarem um *super-holon* com o intuito de cooperar e compartilhar recursos. Já o segundo modelo os agentes deixam a sua autonomia e suas estruturas para a formação de um novo agente a partir de sub-agentes. Assim, diferente desses dois modelos, na sociedade moderada os agentes abrem mão de parte de sua autonomia para o *super-holon* . Nesse modelo, um dos agentes é considerado o agente cabeça[2], o qual é responsável pela coordenação e gerenciamento dos demais agentes que compõe o *holon* . A figura 14 apresenta o modelo de sociedade moderada com a utilização de um agente cabeça.

Figura 14 – Exemplo de diversas holarquias



Fonte: (FISCHER; SCHILLO; SIEKMANN, 2003)

A definição formal para esse modelo por ser denotada como o holon h sendo $= (\{A_1\}$ $\{A_1, A_2, A_3, A_4\}, C_{sociedade})$. A escolha do agente cabeça na formação do *super-holon* pode ser realizada através de uma instanciação de um novo agente para que esse coordene os demais agentes, ou até mesmo a escolha de um determinado agente do conjunto $\mathcal{A}_t$. Porém, há mais de uma forma para que a escolha de um agente de $\mathcal{A}_t$ possa ser efetuada, podendo ser:

- Escolha predestinada: Alguns agentes são marcados no processo de desenvolvimento para que sejam futuros agentes cabeça;
- Eleição: Os agentes que irão compor o *super-holon* votam para a escolha do agente cabeça;
- Teste: É verificado em tempo de execução qual agente holônico possui mais aptidão para tornar-se um agente cabeça.

A organização holônica através da sociedade moderada será o modelo utilizado pela arquitetura aqui proposta (`MAPS-HOLO`) devido a suas propriedades apresentarem semelhanças ao modelo de domínio que ela estará sendo empregada. Por exemplo, a utilização de um agente

[2] Nesse exemplo é utilizado apenas um agente cabeça, entretanto pode haver casos da existência de dois ou mais agentes cabeça.

setor responsável pela coordenação e gerenciamento dos seus respectivos agentes vaga. Assim, o agente setor é o agente cabeça do *super-holon* vagas.

O próximo capítulo apresenta algumas ferramentas para o desenvolvimento de Sistemas Multiagente. Por fim, o capítulo **??** demonstra como os conceitos de agentes holônicos serão empregados na arquitetura `MAPS-HOLO` .

5 FERRAMENTAS PARA SISTEMAS MULTIAGENTES

Nesse capítulo é apresentado algumas ferramentas para o desenvolvimento de Sistemas Multiagentes. Primeiramente, é apresentado o *framework* JaCaMo e suas ferramentas (Jason + Cartago + Moise). Por fim, é descrito brevemente a metodologia Prometheus.

5.1 JACAMO

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

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

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

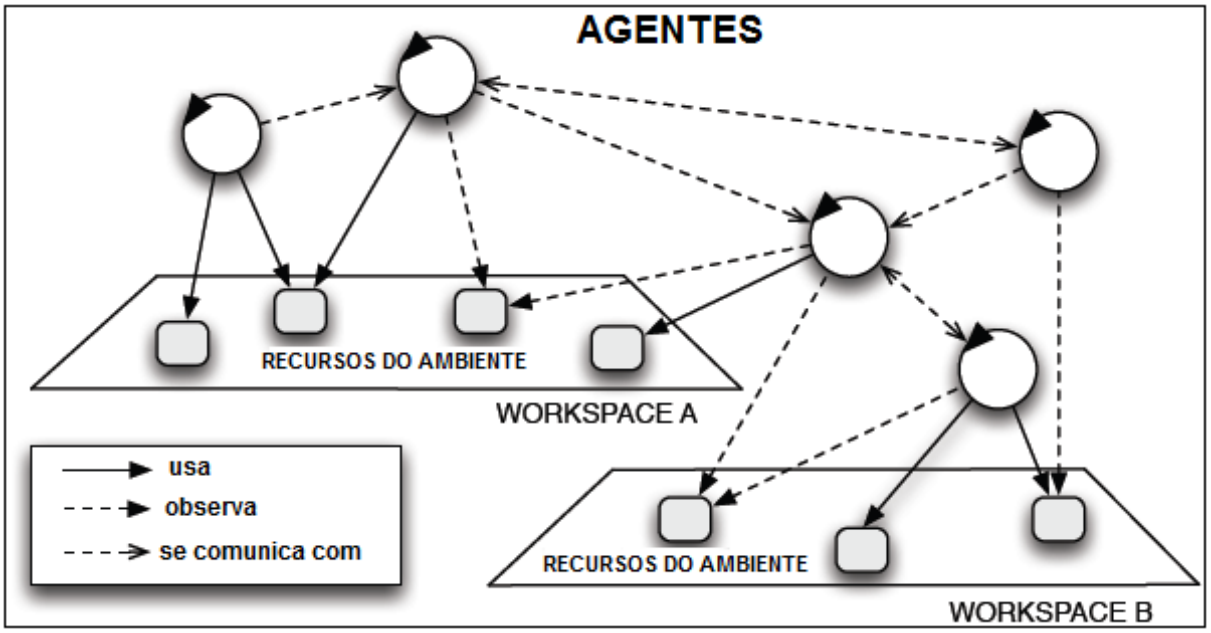
O modelo de programação JaCa (**Ja**son + **Ca**rtago) tem como finalidade proporcionar o desenvolvimento de agentes cooperando e interagindo em um ambiente comum. Para o desenvolvimento dos agentes é utilizado a linguagem Jason e para os artefatos do ambiente em que os agentes estão inseridos utiliza-se o Cartago. A figura 15 ilustra a perspectiva do modelo JaCa.

5.1.1 Abordagem do framework

Ampliando as dimensões do modelo JaCa, um SMA desenvolvido em JaCaMo equivale-se a um sistema multiagente composta pela ferramenta Moise, que é um modelo organizacional baseado em regras, grupos e missões (MOISE, 2002). No nível do ambiente em que os agentes estão inseridos é utilizado um ambiente compartilhado-distribuído baseado em artefatos no Cartago e por fim os agentes BDI programados em Jason. A figura 16 ilustra a abordagem geral do *framework* JaCaMo .

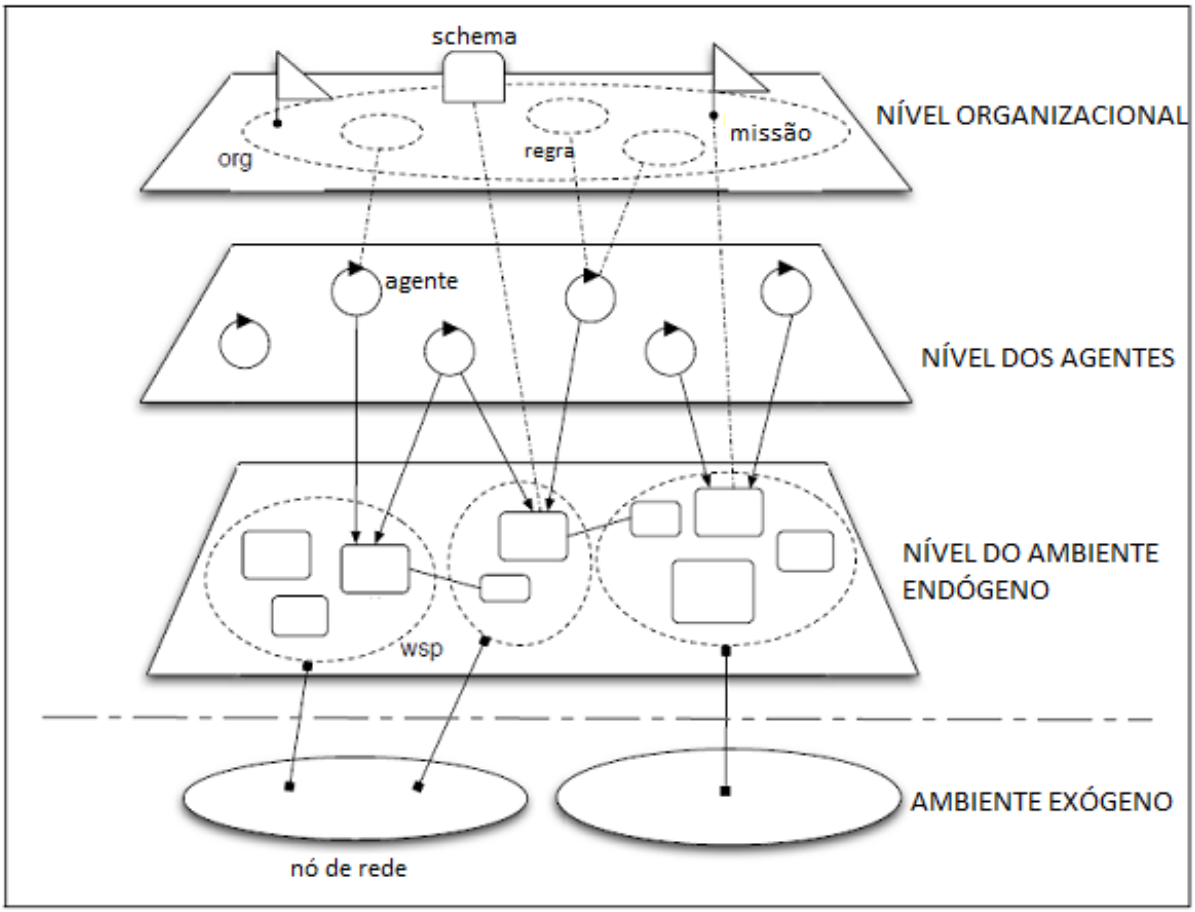
Cada uma das ferramentas que compõem o *framework* JaCaMo possui o seu próprio conjunto de abstrações para a programação, bem como, seu próprio modelo e meta-modelo de

Figura 15 – Arquitetura JaCa



Fonte: Adaptado de (JACAMO, 2011)

Figura 16 – JaCaMo - Abordagem Geral



Fonte: Adaptado de (JACAMO, 2011)

programação. Portanto, para o *framework* considerou-se como peça fundamental a definição do modelo global de programação, tornando assim possível a integração de todas as abstrações

disponíveis em cada plataforma, Jason, Cartago e Moise.

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

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

Por outro lado, na dimensão do ambiente, cada instância do ambiente Cartago, na figura 17 a entidade "*Work Environment*" é composto por um ou mais entidades de *workspace*. Onde cada *workspace* é formado por um ou mais artefatos. Um artefato provê um conjunto de operações e propriedades observáveis, definindo assim uma interface de uso de artefatos (RICCI; PIUNTI; VIROLI, 2011) (JACAMO, 2011). A entidade *Operation* é responsável pela atualização das propriedades observáveis e eventos especificamente observáveis. Finalmente, a entidade *Manual* é utlizada para representar a descrição de funcionalidades oferecidas por um artefato.

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

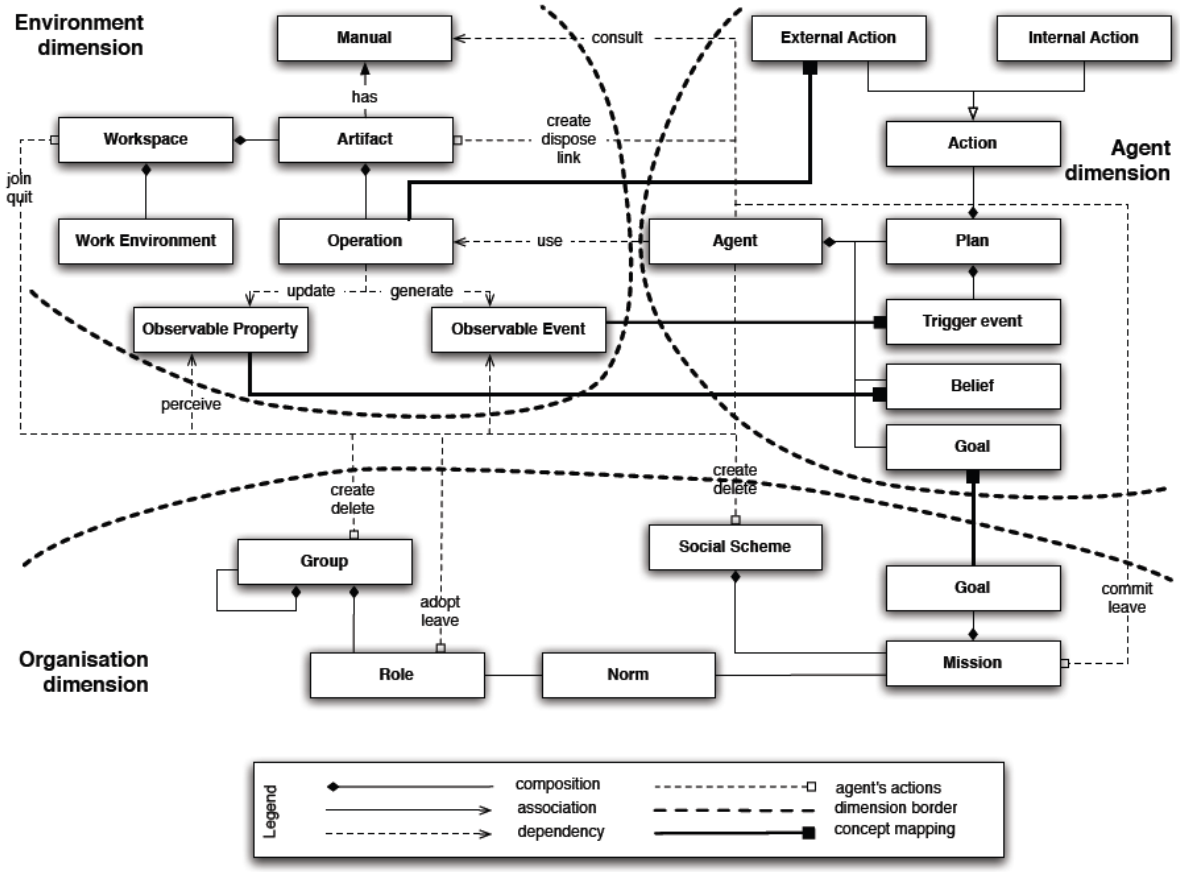
- Especificação estrutural é descrita pelo grupo e pelas regras das entidades. Onde ambas definem diferentes grupos de agentes e sub-grupos dentro da organização;
- Especificação funcional é definida pelo esquema social, missão e entidades objetivo. O esquema social define os objetivos da organização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

5.1.2 AgentSpeak(L)

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

O *framework* PRS foi desenvolvido para aplicações embarcadas em ambientes dinâmicos e de tempo de real, bem como para aplicações militares e industriais (INGRAND; GEORGEFF; RAO, 1992). A figura 18 ilustra a arquitetura de um agente baseado na arquitetura BDI e no *framework* PRS.

Figura 17 – JaCaMo - Meta-Modelo



Fonte: (JACAMO, 2011)

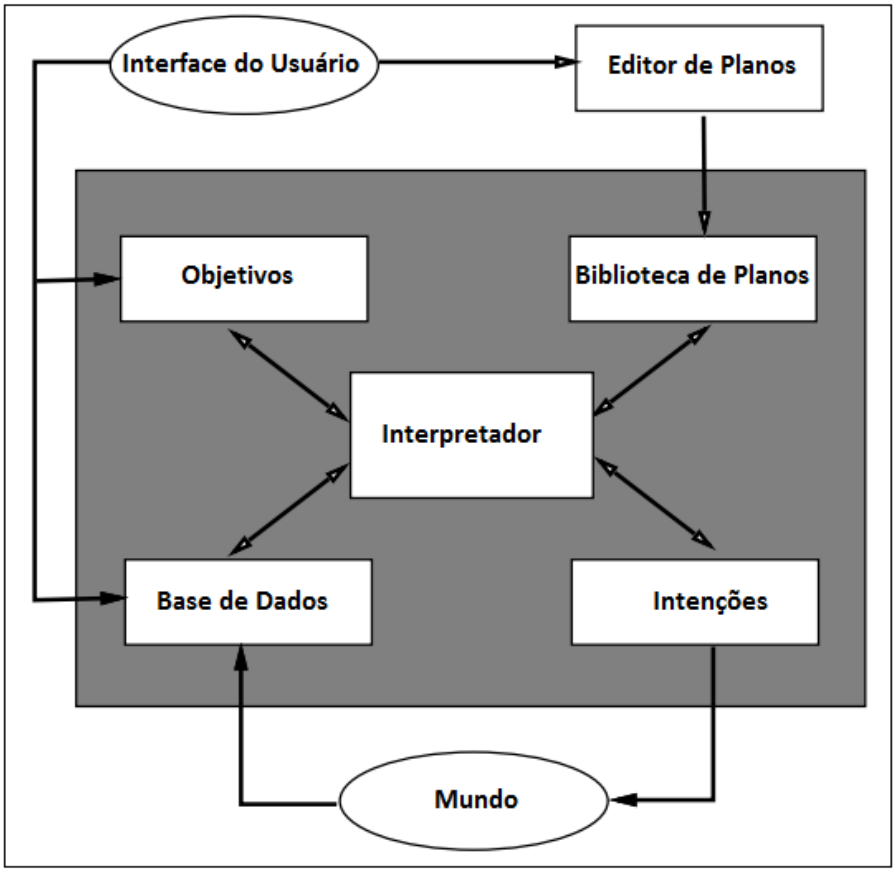
Esta arquitetura ilustrada na figura 18 consiste em uma base de dados que contém as informações sobre o mundo (ambiente), objetivos a serem atingidos, planos que descrevem como esses objetivos podem ser realmente alcançados e as sequências de ações que devem ser tomadas para tal através das intenções. O interpretador é fundamental, pois é através dele que a interação entre todos os módulos é possível. A comunicação do mundo com o interpretador utilizando a base de dados é responsável para que o sistema perceba e tenha conhecimento sobre o mundo em que está inserido e suas mudanças, visto que esse mundo é altamente dinâmico.

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

Características da linguagem

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

Figura 18 – Arquitetura de um agente baseado em BDI:PRS



Fonte: Adaptado de (MYERS, 1993)

tais como:

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

5.1.3 Jason

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

A interpretação do programa agente efetivamente determina o ciclo de raciocínio do agente. Um agente constantemente está recebendo informações sobre o ambiente através das

suas percepções e reagindo de forma a responder a esses estímulos. Inicialmente, o agente possui planos pré-programados em sua base de dados, de como reagir a esses estímulos, porém a escolha de que maneira o agente irá reagir é realizada de maneira autônoma (BORDINI; HüBNER; WOOLDRIDGE, 2007).

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

5.1.3.1 Crenças

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

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

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

Na figura 19 são ilustrados os tipos de termos(objetos) que o Jason utiliza para representar os predicados.

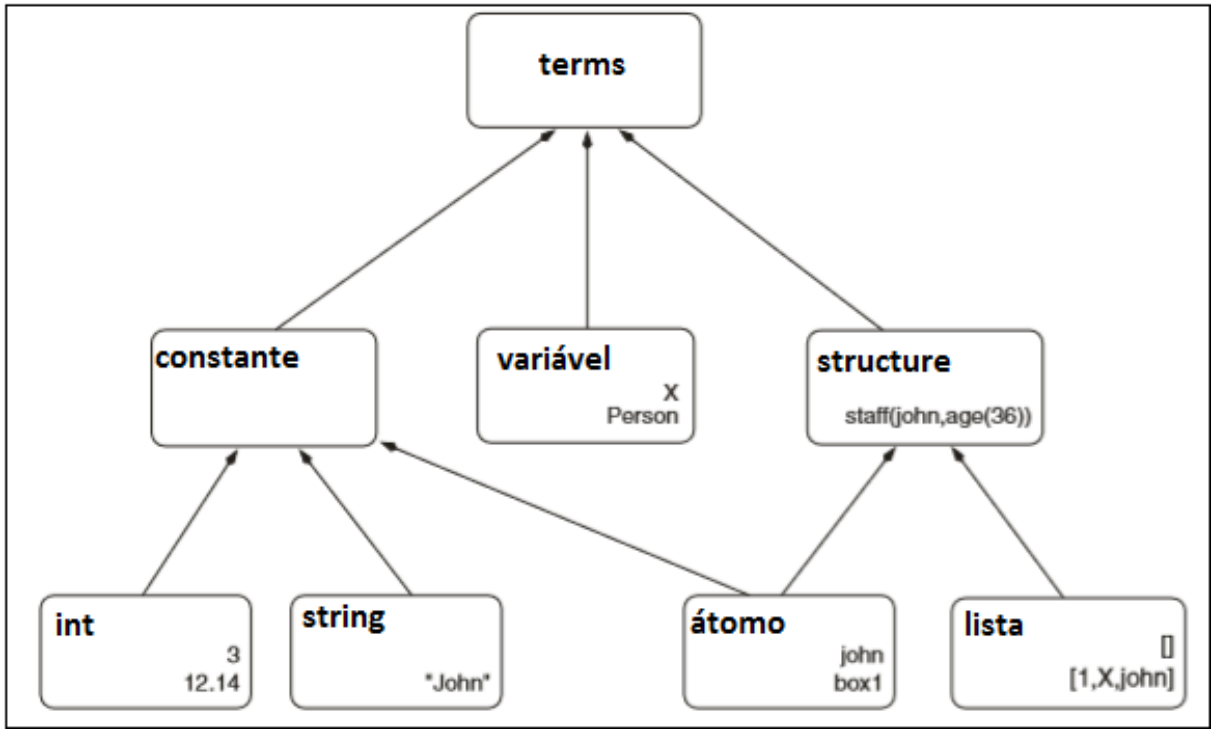
Devido as informações que o agente recebe vindas de um agente no ambiente ou do próprio ambiente, a linguagem Jason possui um recurso de anotações, onde é possível identificar a origem de cada informação recebida. As anotações são identificadas e delimitadas utilizando os colchetes.

1. **porcentaUso(0.5)[source(manager)]:** Agente crê que no seu setor 50% das vagas estão ocupadas. A informação dessa crença é fornecida pelo agente *manager*.

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

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

Figura 19 – Tipos de termos do AgentSpeak no Jason



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

2. **vagasOcupadas(30,A1)** Agente acredita que há trinta vagas ocupadas no setor A1.

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

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

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

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

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

Negação forte

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

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

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

5.1.3.2 Objetivos

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

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

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

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

5.1.3.3 Planos

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

EVENTOGATILHO : CONTEXTO ← CORPO

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

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

Tabela 1 – Definição dos eventos em planos

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

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

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

Tabela 2 – Uso dos literais no contexto

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

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

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

Ações Internas

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

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

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

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

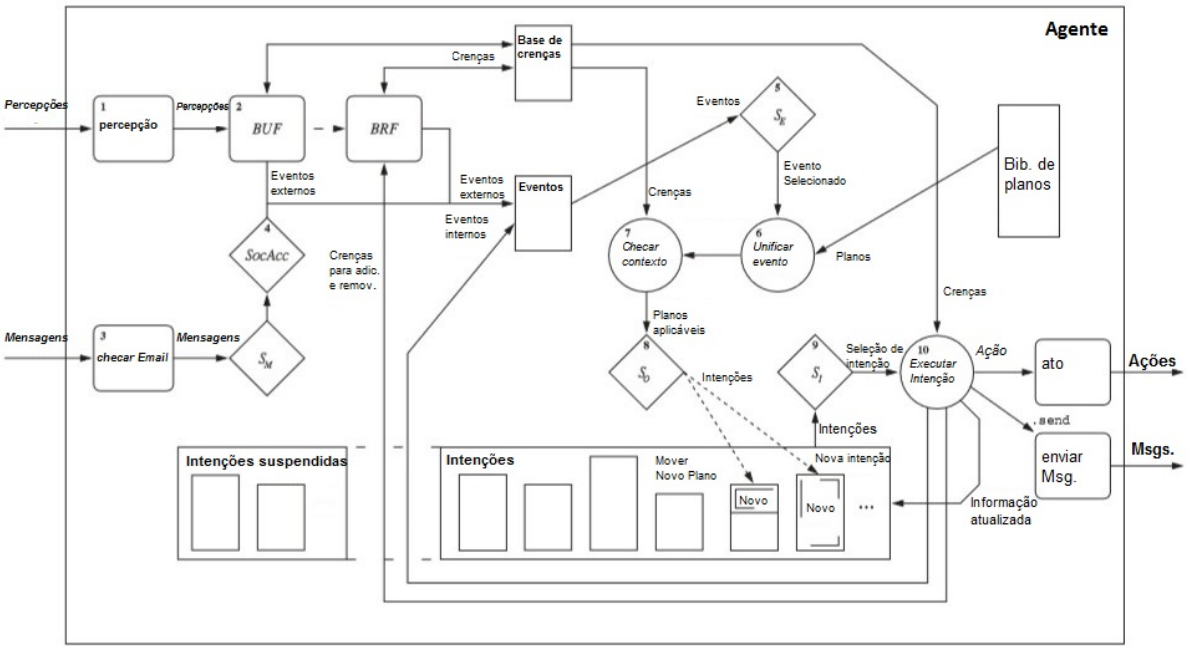
Código 2 – Agente em Jason com ação interna e definição de um plano

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

5.1.3.4 Interpretador

O interpretador da linguagem Jason executa um agente com base em um ciclo de raciocínio que pode ser descrito em dez etapas. A figura 20 ilustra de modo geral esse ciclo.

Figura 20 – Ciclo de raciocínio - Linguagem Jason



Fonte: (BORDINI; HüBNER; WOOLDRIDGE, 2007)

- **Percepção do ambiente**: Os agentes necessitam constantemente avaliar o ambiente para assim alterar e validar sua base de crenças;
- **Atualização da base de crenças**: De acordo com as percepções que o agente teve sobre o ambiente, é necessário atualizar a base de crenças. De modo padrão, o agente insere tudo em sua base de crenças sobre o ambiente, porém isso é personalizável, visto que em um ambiente complexo podem haver informações desnecessárias para um agente em particular;
- **Comunicação entre os agentes**: Envio e recebimento de mensagens entre os agentes. Cada agente possui uma caixa de entrada de mensagens. Sendo assim, o agente necessita verificar se há novas mensagens disponíveis;
- **Escolha das mensagens**: No ambiente em que o agente está inserido pode haver inúmeros agentes, e o número de mensagens pode crescer exponencialmente. Assim, nem todas as mensagens recebidas por um agente podem ser interessantes a ele. (BORDINI; HüBNER; WOOLDRIDGE, 2007) definem tais mensagens como "socialmente aceitáveis", onde o agente irá fazer uma seleção das mensagens;
- **Escolha de eventos**: Os eventos necessitam ser processados para que os agentes executem as suas ações e assim alcançar os objetivos. Porém, a cada ciclo de raciocínio o Jason seleciona apenas um evento para ser tratado;

- **Lista de planos possíveis**: Outro fator importante no ciclo é a escolha do plano que o agente irá executar. O agente necessita verificar todos os planos possíveis para determinado evento e decidir se o plano é relevante ou não;
- **Determinação dos planos utilizáveis**: Sendo todos os planos relevantes apontados, o Jason verifica os planos que realmente podem ser utilizados de acordo com o contexto;
- **Escolha do plano a ser executado**: Uma vez que todos os planos possíveis foram analisados e depois verificados se são utilizáveis, escolhe-se um plano a ser executado. De modo padrão, o Jason escolhe o primeiro plano da lista dos possíveis, porém há uma enorme linha de pesquisa na área que determina qual é o melhor método para a escolha de planos (HORTY; POLLACK, 2001).
- **Escolha da intenção a ser executada**: O agente possui uma lista completa de intenções a serem executadas e nessa etapa é selecionada qual intenção será executada;
- **Execução da intenção**: Esta etapa depende de como a fórmula da intenção está sendo executada, podendo ser categorizada em até seis diferentes tipos, sendo eles:
  - Ação do ambiente;
  - Objetivos a serem atingidos;
  - Objetivos de teste;
  - Notas mentais;
  - Ações internas;
  - Expressões.

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

5.1.4 Cartago

Common ARTifact infrastructure for AGents Open environments, Cartago, é um *framework* que possibilita desenvolver e executar ambientes virtuais para sistemas multiagentes. Cartago é baseado no meta-modelo de agentes e artefatos (A&A) para modelar, o qual é baseado e implementar sistemas multiagentes (CArtAgO, 2006).

Além disso, o *framework* é capaz de isolar o desenvolvimento do sistema multiagente em duas camadas, a programação dos agentes e a programação do ambiente, apenas estabelecendo uma interface comum entre os agentes e o ambiente para que possa haver um nível sociabilidade entre eles. Com isso, o Cartago não é dependente de nenhuma linguagem para agentes

em específico, pois seu enfoque é no ambiente. No caso da linguagem Jason, há um suporte específico de integração das ferramentas, como no caso do projeto JaCa (Jason + Cartago) e também no caso do *framework* utilizado no desenvolvimento desse trabalho, o JaCaMo .

5.1.4.1 Workspaces

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

5.1.4.2 Repertório de Ações do Agente e Artefatos

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

5.1.4.3 Artefatos Padrões

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

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

5.1.5 Integração Jason-Cartago

Um dos principais objetivos da plataforma JaCaMo é a integração das diferentes ferramentas Jason, Cartago e Moise. Contudo, é possível a utilização das três ferramentas de forma

independente, ou até mesmo realizar integrações entre elas de maneira manual. Com a utilização do JaCaMo essa integração torna-se menos complexa e mais direta. Com base na figura 15, pode-se observar a integração de diferentes agentes inseridos em um mesmo ambiente com diversos artefatos. Através da integração Jason-Cartago os agentes podem invocar ações diretamente ao Cartago, sem a necessidade de importação e exportação de parâmetros. Outro fator, a implementação dos artefatos em Cartago é baseada na linguagem Java, tornando assim o sistema multiagente mais versátil e robusto, visto a aplicabilidade da linguagem Java.

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

1. Criação do artefato:

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

Código 3 – Agente Jason instanciando um artefato

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

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

2. *Lookup* no artefato:

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

Código 4 – Lookup no artefato em Jason

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

5.1.6 Moise

A programação normativa do SMA em JaCaMo é realizada através do Moise+. O modelo Moise+ deriva do modelo previamente denominado por Moise e desenvolvido por HANNOUN *et al.*, o qual estabeleceu um modelo organizacional com a finalidade de normatizar um Sistema Multiagente para que a visão da organização esteja centrada no sistema. Os agentes deste sistema poderiam consultar essas normas e agirem de acordo com elas, mas além disso, o modelo MOISE pode reorganizar os agentes com a finalidade de manter o objetivo a ser cumprido (HANNOUN *et al.*, 2000).

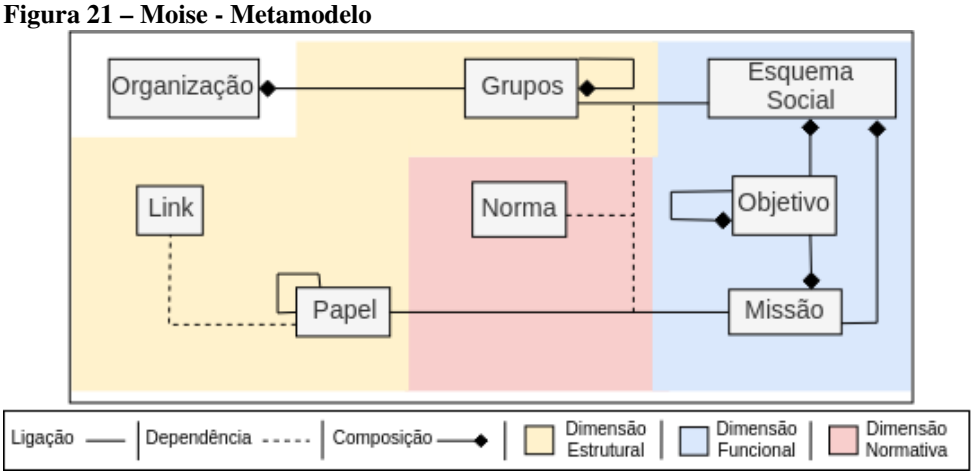
Para a organização dos agentes, o Moise apresenta três níveis ou dimensões distindas, sendo elas:

- **Dimensão Estrutural (DE):** Encarregada pelo estabelecimento da estrutura da organização (os papéis que a compõe). Essa dimensão é dividida em três níveis: individual, social e coletiva.
  - Individual: Caracteriza o comportamento que um agente possui de si próprio e suas restrições;
  - Social: Manifesta o comportamento do agente em relação aos demais agentes presentes no ambiente;
  - Coletiva: Estabele os agentes que possuem propriedades semelhantes;
- **Dimensão Funcional (DF):** A dimensão funcional visa a especificação da coordenação dos agentes do SMA. Além disso, determina a forma e a ordem em que as ações dos agentes serão realizadas em prol da eficiência do sistema para que seja atingido o objetivo ou meta global do sistema através de missões. Por exemplo, no caso do problema do `MAPS-HOLO` , a dimensão funcional irá organizar os agentes de forma com que as suas ações estejam coordenadas para que a alocação das vagas ocorra de maneira eficiente.
- **Dimensão Normativa (DN):** Nessa dimensão é realizada a especificação normativa, a qual visa estabelecer a relação entre a dimensão estrutural e funcional. Na Especificação normativa é estabelecido por exemplo quais as missões (dimensão funcional) tem um determinado papel (dimensão estrutural) comprometido ou obrigado a executar.

A Figura 21 ilustra o metamodelo do Moise+, assim como as dimensões e seus respectivos componentes.

A seguir é descrito os componentes da Figura 21 através das suas respectivas dimensões.

Figura 21 – Moise - Metamodelo



Fonte: Adaptado de (HüBNER; SICHMAN; BOISSIER, 2008)

5.1.6.1 Dimensão Estrutural

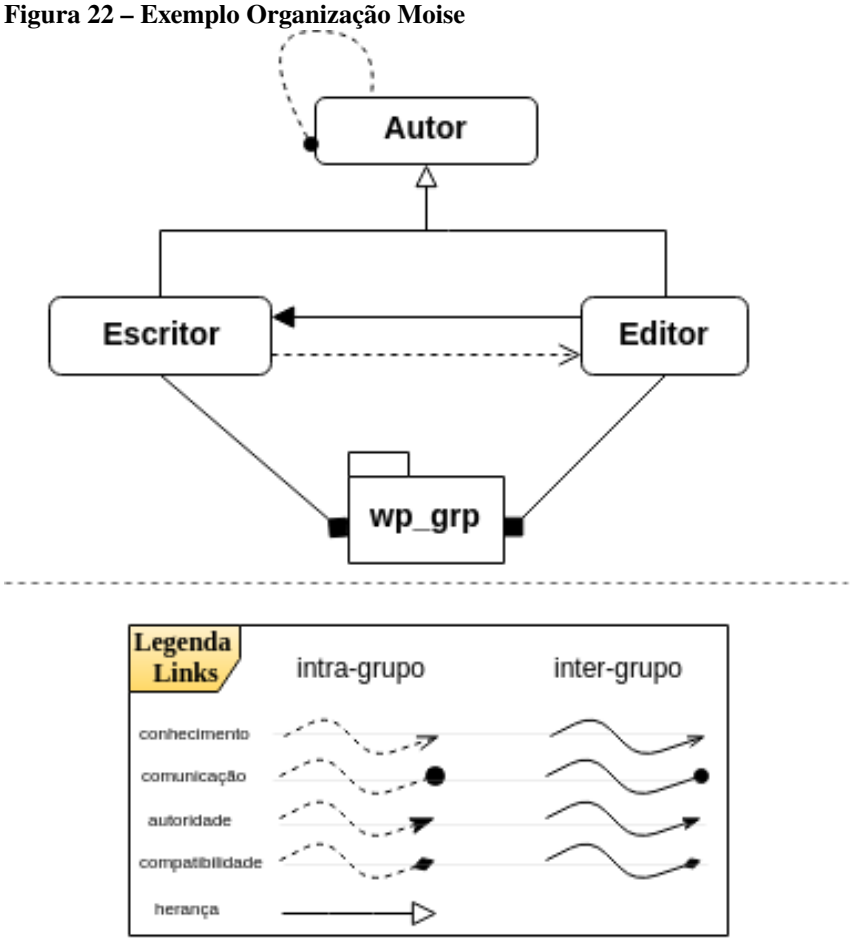
- **Papel:** Descrito como uma classe de comportamentos que podem permitir ou restringir as ações dos agentes do SMA. Esses comportamentos são definidos na Dimensão Estrutural, onde ao agente performar um papel, ele estará sujeito as restrições desse papel. O papel é considerado em dois níveis: individual e coletivo. No nível individual diz respeito as ações que o agente poderá desempenhar ou que estará impedido de realizá-las. O coletivo diz respeito as restrições no nível social do agente com os demais do sistema. Além disso, há a relação de herança-composição entre os papéis, onde um papel poderá herdar as restrições e comportamentos de outro papel e da mesma forma um papel poderá ser composto por sub-papéis.

- **Link:** Link ou ligação entre os papéis é o relacionamento entre os papéis na forma de ligações, onde um papel exerce sobre o outro de modo autoritário (*authority*), comunicativo (*communication*), conhecimento ((*aqcuaintance*)), herança (*extends*), composto ()*composed*), compatibilidade (*compatibility*), intergrupo e intragrupo (*intergroup e intra-group*). Com o objetivo de exemplificar esses relacionamentos, a Figura 22 apresenta uma organização que tem por objetivo a elaboração de um artigo e composta por três papéis: Autor, Editor e Escritor. Abaixo também é mostrado a codificação desses relacionamentos e suas implicações.

  – **Conhecimento :** Relacionamento que permite o conhecimento sobre um agente X a outro agente Y. No Código 5 é apresentado o papel Escritor com relacionamento de conhecimento para o Editor. Sendo assim, o Escritor sabe da existência do Editor no sistemna.

```
1            <links>
2                <link from="writer" type="acquaintance"  to="editor"
                 ↪  scope="intra-group" />
```

Figura 22 – Exemplo Organização Moise



Fonte: Autoria Própria

```
3               </links>
```

Código 5 – Exemplo de Relacionamento - Conhecimento

– **Comunicação :** Proporciona ao agente que possui o papel de comunicar-se com outro agente com o intuito da troca de informações. Abaixo é descrito o papel Autor tendo comunicação com si próprio, ou seja, os agentes que desempenham esse papel podem comunicar-se entre si.

```
1               <links>
2                   <link from="author" type="communication" to="author"
                    ↪  scope="intra-group" />
3               </links>
```

Código 6 – Exemplo de Relacionamento - Comunicação

– **Autoridade :** Os papéis que exercem autoridade sobre os demais, possuem controle e domínio sobre os agentes subordinados. No exemplo do Código 7, o papel Editor possui autoridade sobre o papel Escritor.

```
<links>
    <link from="editor" type="authority"     to="writer"
    ↪  scope="intra-group" />
</links>
```

Código 7 – Exemplo de Relacionamento - Autoridade

– **Herança :** No relacionamento de herança, os agentes que herdam (*extends*) outro papel, assumem as características e restrições desse papel. Assim, no Código 8 os papéis Escritor e Editor herdam as características (e.g relacionamentos) do papel Autor.

```
<role-definitions>
    <role id="author" />
    <role id="writer"> <extends role="author"/> </role>
    <role id="editor"> <extends role="author"/> </role>
</role-definitions>
```

Código 8 – Exemplo de Relacionamento - Herança

– **Composição:** Um grupo de agentes é descrito também como um grupo de agentes que desempenham um papel, sendo a quantidade desses agentes podendo ser limitada a um determinado valor ou intervalo. No Código 9 é descrito que o papel Escritor deve possuir pelo menos um agente participante e no máximo cinco. Já o Editor deve possui no mínimo e máximo um agente.

```
<roles>
    <role id="writer" min="1" max="5" />
    <role id="editor" min="1" max="1" />
</roles>
```

Código 9 – Exemplo de Relacionamento - Composição

– **Compatibilidade :** Relacionamento que permite que um determinado agente que desempenha um papel X possa desempenhar outro papel Y, pois X e Y são compativeis. Com isso, como no exemplo do Código 10, um agente que perfoma o papel Editor poderá assumir também o papel de Escritor.

```
<formation-constraints>
    <compatibility from="editor" to="writer"
    ↪  type="compatibility" scope="intra-group"
    ↪  bi-dir="true"/>
</formation-constraints>
```

Código 10 – Exemplo de Relacionamento - Compatibilidade

– **Intergrupo e intragrupo:** Relacionamento de restrição de escopo entre os grupos ou internamente a eles. No intergrupo o papel aplica-se ao relacionamento entre os

grupos, uma vez que o sistema pode possui N grupos na mesma organização. Já no relacionamento de intragrupo a restrição do papel destina-se à apenas o seu grupo em particular. No código 11 é apresentado o caso de um relacionamento intergrupo e intragrupo, respectivamente.

```
1            <links>
2                <link from="editor" type="authority"     to="writer"
                 ↪  scope="inter-group" />
3            </links>
4
5            //
6
7            <links>
8                <link from="editor" type="authority"     to="writer"
                 ↪  scope="intra-group" />
9            </links>
```

Código 11 – Exemplo de Relacionamento - Intergrupo

- **Grupo:** Entidade organizacional a qual é descrita como um conjunto de papéis que possuem um objetivo em comum a ser alcançado (objetivo global). Para cumprir o objetivo global, cada papel possui os seus objetivos particulares ou locais. Ao passo que os objetivos locais são cumpridos, o objetivo global é alcançado. Na especificação do grupo é inserido a cardinalidade dos papéis (relacionamento de composição) e também os relacionamentos entre os papéis. Uma organização pode possuir n-grupos, sendo cada um deles podendo possui m-subgrupos.

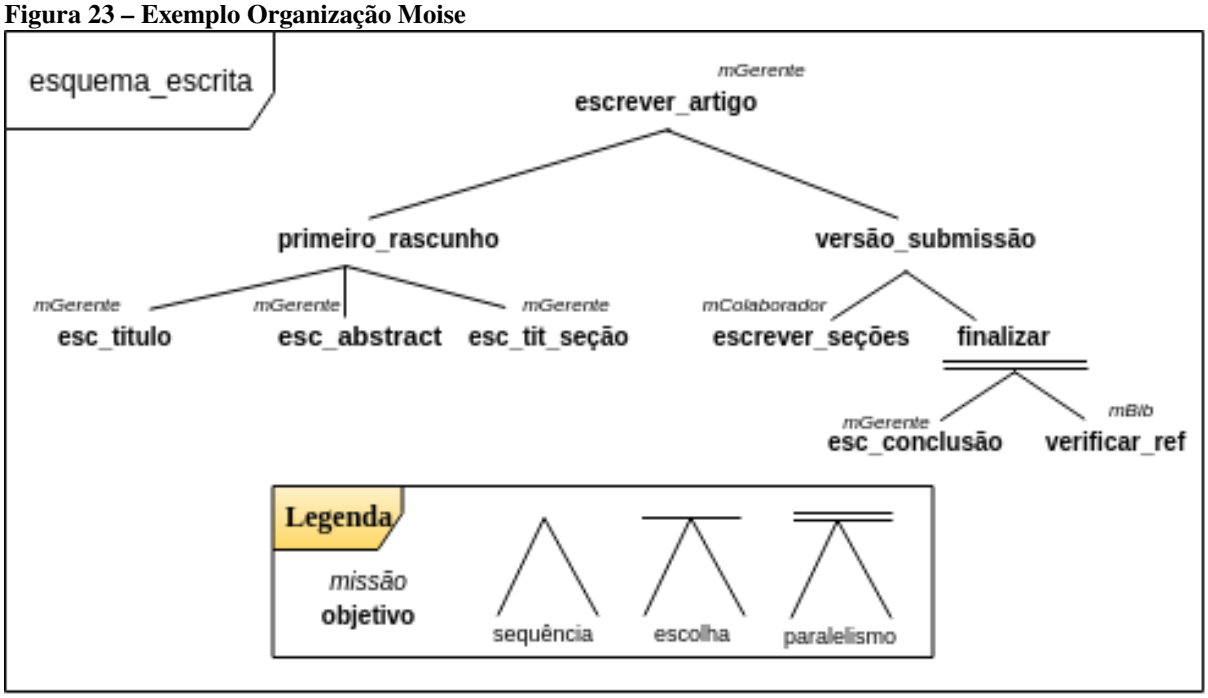
5.1.6.2 Dimensão Funcional

- **Esquema Social:** Descrito como um conjunto global de objetivos em direção ao cumprimento de um propósito, o esquema social no Moise é relacionado a um grupo em específico, que por sua vez delega aos agentes que desempenham os seus papéis para que assim cumpram os objetivos propostos no esquema social. A Figura 23 ilustra o Esquema Social ESQUEMA_ESCRITA. Esse esquema tem como finalidade a escrita de um artigo. Para que o esquema seja cumprido, são descritos missões e objetivos.

- **Missão:** Conjunto local de objetivos que buscam o cumprimento de uma tarefa específica. A missão é composta por objetivos que ao serem cumpridos, a missão será cumprida. Na Figura 23 são apresentadas três missões, sendo elas:

  - MGERENTE: Missão de Gerência, responsável pela administração da escrita do artigo. O Código 12 descreve a missão que é composta pelos objetivos: ESC_TITULO,

Figura 23 – Exemplo Organização Moise



Fonte: Autoria Própria

ESC_ABSTRACT,ESC_TIT_SEÇÃO,ESC_CONCLUSÃO e que deve ser executada no mínimo e máximo uma vez.

```
1  <mission id="mManager" min="1" max="1">
2       <goal id="esc_titulo"/>
3       <goal id="esc_abstract"/>
4       <goal id="esc_tit_secao"/>
5       <goal id="esc_conclusao"/>
6       <goal id="escrever_artigo"/>
7   </mission>
```

Código 12 – Código da Missão - mGerente

– MCOLABORADOR: Missão de Colaboração é responsável pela colaboração no trabalho do artigo, onde é composta por apenas um objetivo: ESCREVER_SECOES. O Código 13 descreve a missão, a qual é executada no mínimo uma e no máximo cinco vezes.

```
1  <mission id="mColaborador" min="1" max="5">
2      <goal id="esc_secoes"/>
3  </mission>
```

Código 13 – Código da Missão - mColaborador

– MBIB: Missão de Bibliografia, tem como objetivo a verificação das referências do artigo. O Código 14 descreve a missão.

```
<mission id="mBib" min="1" max="1">
            <goal id="esc_referencias"/>
        </mission>
```

Código 14 – Código da Missão - mBib

- **Objetivo:** Descrito como sendo um estado onde um agente almeja estar ou conquistar. Os objetivos podem ser classificados como locais ou globais. Um objetivo global é do escopo de todo o sistema representado por um Esquema Social. O objetivo local é descrito como um objetivo pontual ou particular que um agente que desempenha um papel está relacioado em cumprir esse objetivo. De forma resumida, há o Esquema Social (Objetivo Global) e missões (Conjunto de objetivos locais). Um objetivo do Moise está diretamente ligado ao fato do JaCaMo ser composto pelo Jason, Cartago e Moise. Sendo assim, os objetivos do Moise são representados e implementados por meio do Jason.

5.1.6.3 Dimensão Normativa

- **Norma:** Permite ou obriga um determinado papel a cumprir uma missão em específico, as normas visam a normatização da organização do SMA, onde os agentes são comprometidos por meio das normas a cumprirem as suas respectivas missões (HüBNER; SICHMAN; BOISSIER, 2002). Uma norma segue o seguinte padrão de sintaxe:

  ID NORMA,PAPEL,TIPO,MISSÃO,DEADLINE)

  - **ID Norma:** Identificador único da norma;
  - **Papel:** Qual papel é responsável/alvo da norma;
  - **Tipo:** Obrigação ou permissão;
  - **Missão:** Qual missão a norma é responsável em permitir ou obrigar a um determinado papel;
  - **Deadline:** Tempo limite para que a norma seja cumprida ou válida.

  O Código 15 apresenta as normas vigentes no Organização para escrever o artigo.

```
<normative-specification>
    <norm id = "n1" role="editor" type="permission" mission="mGerente" />
    <norm id = "n2" role="escritor" type="obligation" mission="mBib"
    ↪  time-constraint="1 day" />
    <norm id = "n3" role="escritor" type="obligation" mission="mColaborador"
    ↪  time-constraint="1 day" />
</normative-specification>
```

Código 15 – Código das Normas - Escrita do Artigo

No Código 15 são apresentadas três normas: N1,N2,N3. A primeira norma, `n1`, permite o papel do Editor a estar comprometido com a missão `mGerente`. A segunda, impõe a obrigação do Escritor com a missão `mBib`. Por fim, a última norma é responsável pela imposição da obrigação ao Escritor na missão de colaboração (`mColaborador`).

5.1.7 Configurações do Sistema Multiagente no JaCaMo por meio da linguagem JCM

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

- Instanciação dos agentes;
- Crenças iniciais para os agentes (Agente *Builder*);
- *Lookup* dos artefatos *IDriverTools*;
- Definição da organização (*maps_holo_org*);
- Definição do grupo *maps_holo_grp* e os agentes iniciais que desempenharão os papéis de `r_builder` e `r_observer`;
- Configuração do Esquema Social de construção do SMA (*build_scheme*).

Tais funcionalidades descritas acima, poderiam ter sido implementadas sem a utilização do arquivo JCM, porém, com o objetivo de tornar o SMA mais flexível, robusto e otimizado foi utilizado o arquivo de controle. No código 17 é apresentado o arquivo JCM.

```
1   mas mAPS_HOLO{
2           agent driver_john : idriver.asl
3       focus: artifacts.IDriverTools
4       agent observer
5           agent builder{
6                   beliefs: nSectors(5),
7                            nPSpaces(20)
8       }
9           organisation maps_holo_org : maps_holo_org.xml {
10                  group maps_holo_grp : maps_holo_group {
11                          responsible-for: bScheme
12                          players: builder r_builder
```

```
13                                          observer r_observer
14                                          debug
15                  }
16                  scheme bScheme: build_scheme{
17                          debug
18                  }
19          }
20  }
```

Código 16 – Exemplo de código JCM

No código 16 é apresentado o SMA com três agentes: `driver_john, observer e builder`. O agente `driver_john` é uma instância do agente do tipo `IDriver` definido no arquivo `idriver.asl`. O agente `builder` é o responsável pela construção inicial do sistema, tendo as crenças iniciais da quantidade de setores e vagas. Por fim, na linha 9 é instanciada a organização `maps_holo_org` a qual é definida no arquivo `maps_holo_org.xml` (Vide Apêndice no Capítulo 9). Dentro da definição da organização é definido o grupo e qual Esquema Social é responsável, bem como os agentes iniciais participantes desse grupo. Na linha 16 é definido a instância do Esquema Social de construção do SMA.

5.2 METOLOGIA PROMETHEUS

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

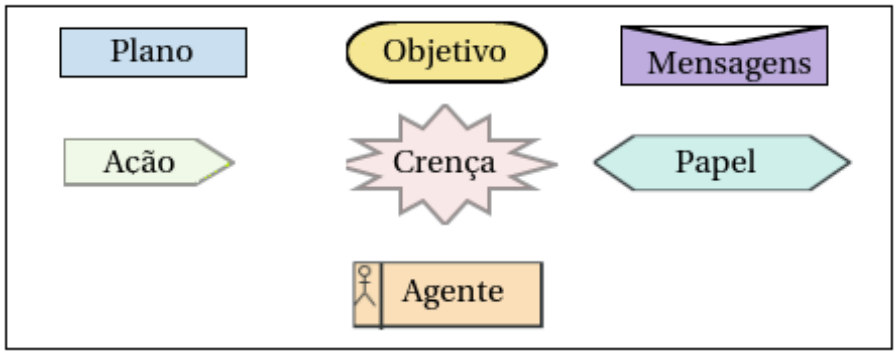
1. **Especificação do sistema:** Composta por identificar objetivos dos agentes, desenvolver casos de uso, descrever ações, percepções e interações dos agentes.

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

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

   O desenvolvimento do atual trabalho utilizou apenas a fase de Design Detalhado, onde será mostrado no capítulo 6 a arquitetura holônica multiagente. Com o objetivo de ilustrar o sistema e as interações entre os agentes, a Figura 24 ilustra os componentes utilizados e sua representação gráfica a ser exibida.

Figura 24 – Elementos Prometheus- Fase: Design Detalhado



Fonte: Autoria Própria

5.3 MODELO DE NEGOCIAÇÃO - CONTRACT NET PROTOCOL

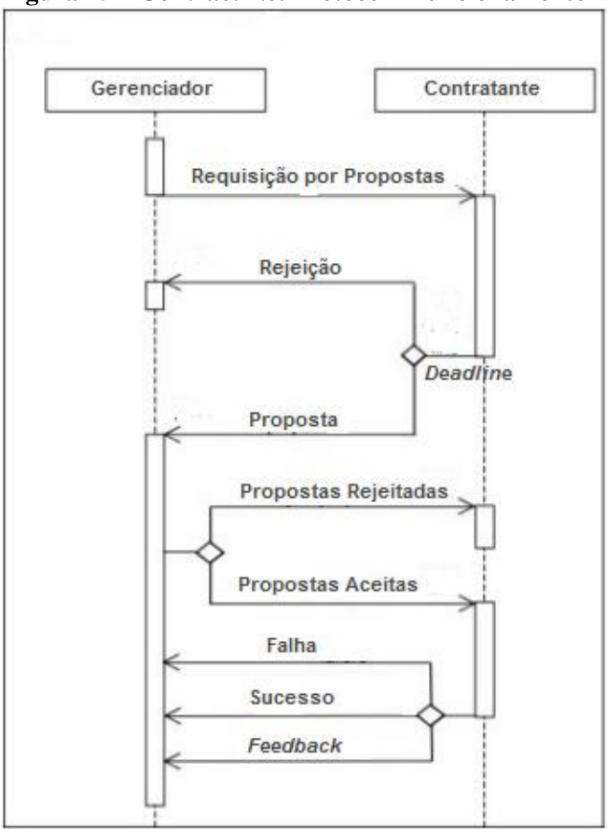
O modelo *Contract Net Protocol* é um protocolo de negociação de tarefas e recursos na forma de licitação. Em 2002 foi padrizado pela Foundation for Intelligent Physical Agents (FIPA) para Sistemas Multiagentes e desde então tem sido utilizado para a negociação de recursos entre os agentes de uma forma simples e prática. O modelo desse protocolo dá-se basicamente por dois grupos de agentes: Gerenciador e Contratantes. O Gerenciador inicia o processo de negociação oferecendo recursos para que sejam requisitados pelos Contratantes. Ao recurso estar disponível para ofertas, os Contratantes enviam suas propostas em um determinado intervalo de tempo. Esse intervalo também pode ser denotado como uma quantidade de rounds de negociação. Após esse período, o Gerenciador avalia as ofertas recebidas, podendo aceitá-las ou rejeitá-las. Após aceitar a oferta, o Contratante envia ao fim da negociação uma mensagem de sucesso, falha ou de *feedback*.. A Figura 25 ilustra o processo da negociação entre o Gerenciador e o Contratante.

Dentro da etapa de negociação, as propostas devem ser aceitas ou rejeitadas de acordo com um parâmetro estabelecido previamente pelo Gerenciador. Para isso, a função de utilidade proporciona um cálculo de satisfação em um índice normatizado (0-1) onde de acordo com uma configuração prévia de pesos α_i e de parâmetros onde é possível posicionar quais ofertas atingem um valor de desempenho ou satisfação esperado (`THRESHOLD`).

A fim de que seja possível avaliar de modo bidirecional as ofertas, ou seja, as ofertas que são enviadas pelo Contrante ao Gerenciador e também as ofertas de recursos que o próprio Gerenciador coloca a disposição. Por exemplo, no caso da `MAPS-HOLO` haverá os clientes que irão requisitar um recurso. Ao passo que o Sistema oferta, ele deve avaliar a oferta que está fazendo ao cliente e da mesma forma o cliente deverá avaliar a oferta que está recebendo. Essa avaliação em ambos os lados (Cliente e Sistema) será realizada através da função utilidade, sendo assim, a equação 5.1 avalia a oferta disponibilizada pelo Sistema, ou pelo Gerenciador. Na equação 5.1, seja:

Contextualizando a Equação 5.1 ao `MAPS-HOLO` temos:

Figura 25 – Contract Net Protocol - Funcionamento



Fonte: Adaptado de (FIPA..., 2002)

- α_i**:** Valor do peso (0.0 - 1.0) para os parâmetros (preço e tempo de permanência). A soma dos valores peso deve ser 1.
- $q_{i,k}$**:** Valor do atributo da oferta (preço($2-$10) | tempo de permanência (15-360min));
- $min_j(q_{i,j})$**:** Valor mínimo do atributo (preço - $2) | tempo de permanência (15min);
- $max_j(q_{i,j}$**:** Valor máximo do atributo (preço - $10) | tempo de permanência (360min);

$$UtilidadeSistema(oferta_{recurso}(k)) = \sum_{i=1}^{n}(\alpha_i * \frac{q_{i,k} - min_j(q_{i,j})}{max_j(q_{i,j}) - min_j(q_{i,j})}) \tag{5.1}$$

Já a equação 5.2 avalia a oferta disponibilizada pelo Sistema, ou Gerenciador, a fim de que o Cliente possa então aceitar ou recusar a oferta.

Contextualizando a Equação 5.2 ao `MAPS-HOLO` temos:

- β_i**:** Valor do peso (0.0 - 1.0) para os parâmetros (preço e tempo de permanência). A soma dos valores peso deve ser 1. Os demais itens da equação 5.2 são os mesmos da 5.1.

$$UtilidadeCliente(oferta_{recurso}(k)) = 1 - \left[\sum_{i=1}^{n}(\beta_i * \frac{q_{i,k} - min_j(q_{i,j})}{max_j(q_{i,j}) - min_j(q_{i,j})})\right] \tag{5.2}$$

6 MAPS HOLO - MODELAGEM DA ARQUITETURA EM UMA PERSPECTIVA GENÉRICA

Esse capítulo apresenta a arquitetura MAPS-HOLO em uma perspectiva genérica, onde é possível observer a arquitetura sem o viés da aplicação através do *framework* JaCaMo. Assim, através da arquitetura genérica é possível implementá-la em uma linguagem de sistemas multiagentes que forneça suporte aos agentes e suas organizações.

O capítulo na seção 6.1 apresenta uma introdução ao MAPS-HOLO e a motivação para o desenvolvimento do sistema. A seção 6.2 apresenta uma visão geral da arquitetura do MAPS-HOLO , a seção 6.3 o modelo de negociação utilizado, seção 6.4 os casos de uso da Arquitetura e por fim a seção 6.5 com as considerações finais do capítulo.

6.1 INTRODUÇÃO

O principal objetivo deste trabalho é o desenvolvimento de uma arquitetura de um sistema multiagente baseado no paradigma organizacional holônico. O SMAH a ser desenvolvido será aplicado ao cenário de um estacionamento inteligente. O termo inteligente ou *smart* (*smart parking*) vem da característica de que o estacionamento possui tecnologias empregadas para que o seu uso e gerenciamento ocorra de uma maneira automatizada (CARAGLIU; BO; NIJKAMP, 2011). Os estacionamentos compreendidos como sendo inteligentes podem ser públicos ou privados. Os estacionamentos públicos serão denotados como sendo os estacionamentos de vias, ruas, avenidas, etc. Assim, os privados serão aqueles em que estão alocados em uma determinada região e possuem acesso controlado mediante o pagamento ou alocação. Como delimitação da atual proposta, os estacionamentos compreendidos pelo MAPS-HOLO serão os privados.

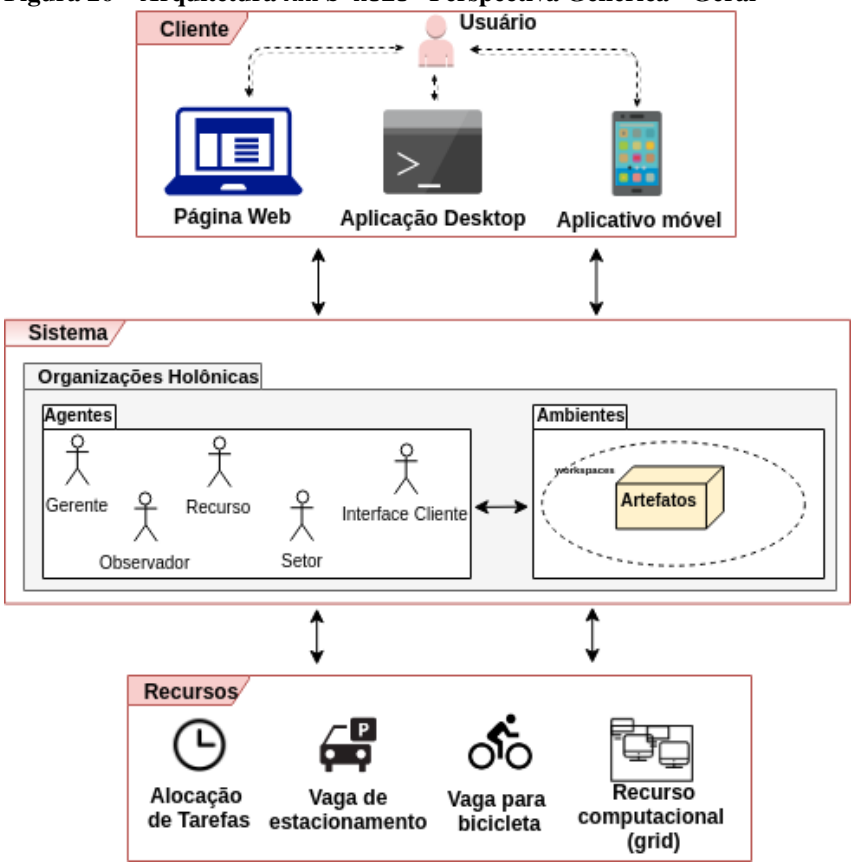
O desenvolvimento deste atual trabalho está situado dentro de um projeto de pesquisa denominado MAPS - *MultiAgent Parking System*, ou Sistema Multiagente de Estacionamento. O objetivo principal deste projeto destina-se ao desenvolvimento de soluções utilizando sistemas multiagentes para estacionamentos inteligentes. Em (CASTRO; ALVES; BORGES, 2017) desenvolvemos um SMA baseado em JaCaMo para a alocação de vagas em um estacionamento inteligente com base em valores de confiança. Os agentes que compunham o ambiente foram organizados em dois grupos: agentes motoristas e agente gerente. O gerente era o agente responsável por todo o processo de alocação e gerenciamento dessas vagas, e o motorista era o agente responsável pela requisição e uso da vaga provida pelo SMA. O SMA desenvolvido foi baseado no paradigma hierárquico, onde o agente ápice era o agente gerente e os demais agentes motoristas os seus subordinados. Entretanto, ao passo que o número de agentes motoristas crescia, existia um gargalo no agente gerente, uma vez que esse agente era o responsável pelo gerenciamento de todo o SMA. Assim, caso o gerente viesse a falhar, todo o SMA falharia.

Em vista de uma solução sobre a questão do gargarlo dos agentes Motorista e Gerente,

bem como o caso de possíveis falhas, realizamos a proposta do desenvolvimento não apenas de um Sistema Multiagente, mas sim de uma Arquitetura Multiagente Holônica e Configurável aplicada ao cenário dos estacionamentos inteligentes (CASTRO; BORGES; ALVES, 2018). Diferente do SMA, na Arquitetura MAPS-HOLO nós propomos uma Arquitetura que pode ser implementada não exclusivamente por meio do *framework* JaCaMo, mas podendo ser em outra linguagem que possui suporte ao desenvolvimento de SMAs. Assim, esse capítulo apresenta a descrição do MAPS-HOLO em uma perspectiva genérica. No próximo capítulo será apresentado a implementação realizada por meio do JaCaMo.

6.2 VISÃO GERAL

Figura 26 – Arquitetura MAPS-HOLO - Perspectiva Genérica - Geral



Fonte: Autoria Própria

A MAPS-HOLO tem como finalidade a alocação de recursos por um determinado valor relacionado a um tempo de utilização. Os recursos contemplados pela arquitetura podem ser os mais variados, desde que estejam de acordo com a premissa de serem alocáveis a um custo C e um tempo T. Na Figura 26 é ilustrado a Arquitetura em uma perspectiva geral, onde é apresentado quatro tipo de recursos que seguem a premissa de utilização: vaga de estacionamento, vaga de bicicleta, tempo computacional (Grid) e alocação de tarefas [1].

[1] Embora a perspectiva do atual capítulo seja genérica, o principal enfoque da atual implementação da MAPS-HOLO nesse

Módulos

A fim de que a Arquitetura possa contemplar a alocação de recursos, a MAPS-HOLO foi divida em três módulos: Cliente, Sistema e Recursos.

- **Cliente:** Módulo responsável pela instânciação de plataformas a fim de os usuários tenham acesso à Arquitetura, sendo assim, o usuário poderá acessar através de um Aplicativo para smartphone, uma página WEB ou até mesmo do console (shell) do seu computador [2];

- **Sistema :** O núcleo da MAPS-HOLO , sendo responsável pelo gerenciamento das requisições provindas da módulo Cliente e pelas alocação do módulo Recurso. Esse módulo é composto por organizações holônicas, agentes inteligentes(Recurso, Setor, Interface Cliente, Gerente e Observador), artefatos e workspaces. Os elementos do módulo Sistema visam a sinergia do Arquitetura como um todo, por meio dos agentes representando os Recursos e Clientes e da utilização dos artefatos juntamente com as organizações;

- **Recursos :** Agrega os recursos contemplados pela Arquitetura, onde esses são representados pelos agentes Recurso. Destaca-se o uso da Arquitetura não exclusivamente a um recurso por vez, mas sim podendo fornecer aos Clientes vários recursos de uma única vez, pois em um nível mais abstrato todos os recursos (Vaga de estacionamento, tempo computacional(grid),vaga de bicicleta e alocação de tarefas) podem ser descritos todos como recursos alocáveis por um custo C a um tempo T.

6.2.1 Visão de Sistema

Em uma perspectiva de Sistema, a Arquitetura é divida em duas frentes: Centralizada e Descentralizada. Como descrito no capítulo 4, em um sistema holônico pode haver várias maneiras de organização dos agentes dentro de uma holarquia. Na MAPS-HOLO é empregada duas formas de organização: cabeça-corpo (centralizada) e corpo-corpo (descentralizada) ***(head-body (HB) e body-body (BB))***.

- Organização Head-Body: Nessa organização há a presença de um agente *head* em cada holarquia como líder, sendo os demais agentes os agentes *body* submissos ao agente *head* da sua holarquia;

- Organização Body-Body: Organização onde não há a presença central de um agente *head* como líder, sendo os demais agentes *body* seus próprios líderes. Além disso, nesse modelo de

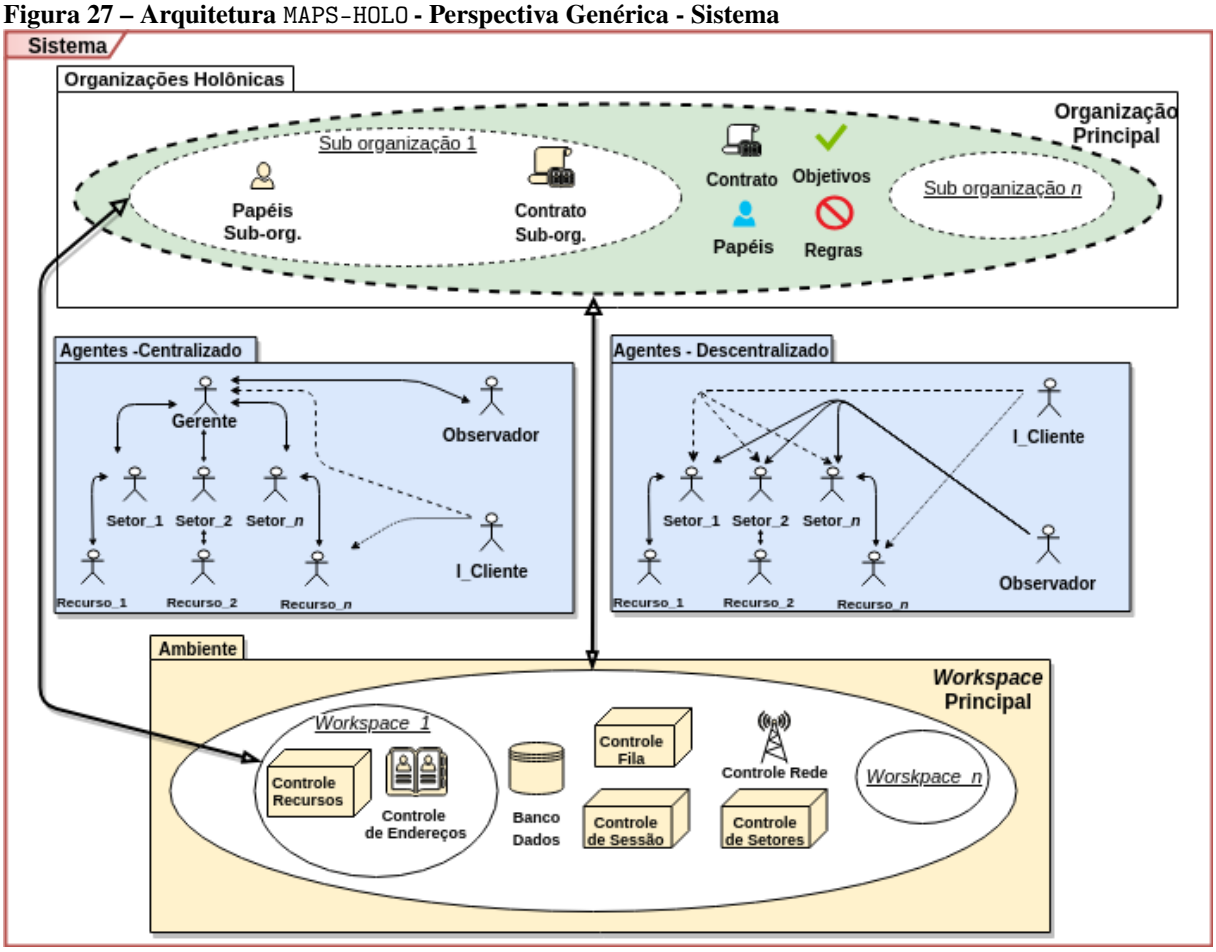
trabalho serão vagas de estacionamento

[2] Está disponível a implementação via shell-console da Arquitetura em: www.github.com/lcastropg/maps_holo_client. A versão do aplicativo ainda está em fase de prototipação

organização há a possibilidade de lideranças locais, diferente do modelo *head-body* onde há a liderança global.

A Figura 27 apresenta a visão de Sistema da MAPS-HOLO e também a ilustração das duas formas de organização suportadas pela Arquitetura.

Figura 27 – Arquitetura MAPS-HOLO - Perspectiva Genérica - Sistema



Fonte: Autoria Própria

Conforme a Figura 27, no módulo das Organizações Holônicas é descrito os seguintes itens:

- **Organização Principal:** Descreve a organização holônica global, onde por meio dela as demais organizações estarão aninhadas. Nessa organização os agentes serão dispostos da seguinte forma:
  - Agente cabeça: Gerente;
  - Agente corpo: Setores;
- **Sub-Organização 1-n:** Suborganizações aninhadas na Organização Principal, onde definem papéis da sub-organização e contratos. Nessa organização os agentes serão dispostos da seguinte forma:

  - Agente cabeça: Setor;
  - Agente corpo: Recursos;

- **Papel:** Podendo ser global (Organização Principal) ou local (Sub-organizações), os papéis definem as características que os agentes desempenham. Os papéis serão: ICliente, Gerente, Setor, Recurso e Observador;

- **Contrato:** Definem as normas de utilização dos recursos (preços e tempo máximo de utilização). O Contrato que se encontra na Organização Principal estabelece os preços e tempo para toda a Holarquia. Contudo, podem haver Contratos locais para cada sub-organização, assim esses podem prevalecem em frente ao Contrato da Organização Principal. Destaca-se aqui a opção de cada sub-organização, podendo essa escolher seguir o Contrato da Organização Principal, estabelecer o próprio Contrato local ou até mesmo realizar uma média entre os contratos;

- **Regras:** Estabelecem as regras de uso do sistema (Similar a Dimensão Normativa do Moise), onde obrigam ou permitem a realização dos objetivos;

- **Objetivos:** Além dos objetivos de cada agente, a organização possui objetivos globais de todo o sistema;

O Módulo do Sistema será divido em dois módulos: Agentes - Centralizado e Agentes - Descentralizado. Como os agentes são os mesmos, mas com diferentes interações, assim abaixo é descrito os agentes e as suas diferentes interações nos módulos centralizado e descentralizado:

- **Gerente:** Presente apenas no módulo centralizado, é o agente responsável pela administração dos agentes Setores e Recursos sendo o agente cabeça da holarquia principal. Ele é o responsável por receber as requisições dos agentes I_Cliente e pelas alocações de recursos aos mesmos. Ao receber uma requisição, realiza uma requisição aos Setores a fim de quem o recurso seja alocado ao ICliente.

- **Observador:** Verifica o estado em que o sistema encontra (utilização e falhas). Ao longo da execução, envia mensagens para os agentes Gerente e Setores para checar se os agentes estão em bom funcionamento. Ao verificar falhas toma medidas para que o sistema não tenha interrupções ou até mesmo finalizar. Uma falha é apontada quando um agente não responde as mensagens enviadas, assim o agente ou está sobrecarregado ou em estado de falha. A fim de verificar os agentes Recurso, o Observador ao verificar o Setor solicita a esse que verifique também os seus agentes Recurso. O Sistema pode apresentar cinco níveis de falhas, dos quais três emitem alertas (*Yellow Alert, Orange Alert e Red Alert*). Abaixo é listado as falhas, seus respectivos alertas e suas soluções:

  - Falha de agente Gerente: Mudança de estrutura no Sistema para Descentralizado. Todos os agentes Setores são notificados. Caso haja uma negociação em andamento, o agente ICliente reinicia a negociação;

  - Falha de agente Recurso: Notificação de outro agente Recurso para que abrigue, assuma as responsabilidades e recursos do agente Recurso falhado;
  - Falha de todos os agentes Recurso: Falha que emite o *Yellow Alert*, pois todos os agentes Recurso de uma holarquia Setor falharam. Essa falha aponta para uma provável falha também na holarquia. A solução para isso é a instanciação de um novo agente Recurso (Recurso_Emergência) para que assuma todos os recursos e responsabilidades dos agentes Recurso dessa holarquia. Essa holarquia não irá aceitar novas requisições para a alocação de recursos, apenas irá sustentar as requisições que já estavam previamente alocadas.
  - Falha de agente Setor: Falha que emite o *Orange Alert*. Todos os agentes Recurso dessa holarquia serão remanejados para outra holarquia, sendo agora movidos de Setor.
  - Falha de todos os agentes Setores: Falha grave que aponta que o Sistema está comprometido, assim é emito o *Red Alert*. É instanciado um novo agente Setor (Setor_Emergência) para que assuma todos os demais setores e seus respectivos agentes Recursos. O sistema entra em estado de suspensão e não aceitará novas requisições para nenhum setor, pois irá apenas sustentar as requisições já alocadas.

- **Setor:** Possui duas características que o difere dos demais agentes, pois o agente Setor é cabeça e corpo simultâneamente, pois está presente em duas organizações holárquicas (Principal e sub); Cada agente Setor possui a sua própria organização holárquica, sendo esse o cabeça. No modo centralizado ele depende das requisições provindas do Gerente para que possa alocar seus recursos. Já no modo descentralizado, ele possui interação direta com o agente I_Cliente para receber as requisições.

- **Recurso:** Estando situado dentro da organização holárquica de um Setor, o agente Recurso é um agente corpo que é submisso ao seu Setor. Esse agente controla um recurso em específico, verificando seu estado (Ocupado, Reservado e Livre);

- **ICliente:** Agente de interface responsável por requisitar o recurso desejado pelo Cliente (Usuário do sistema). No modo centralizado o agente ICliente negocia apenas com o agente Gerente, não tendo nenhum contato com os Setores. Ao receber o recurso alocado, passa ter visão apenas do seu Setor e Recurso. Já no modo descentralizado, o ICliente tem acesso a todos os setores para que possa negociar com todos eles simultâneamente.

Finalmente, com base na Figura 27 o módulo Ambiente apresenta os Workspaces e Artefatos do sistema. Os workspaces são espaços de trabalho que são mapeados como os ambientes providos para as organizações e sub-organizações. Dentro de cada Workspace há diferentes artefatos, os quais proporcionam aos agentes funcionalidades externas as quais eles não possuem nativamente. Os artefatos da `MAPS-HOLO` em sua perspectiva genérica são:

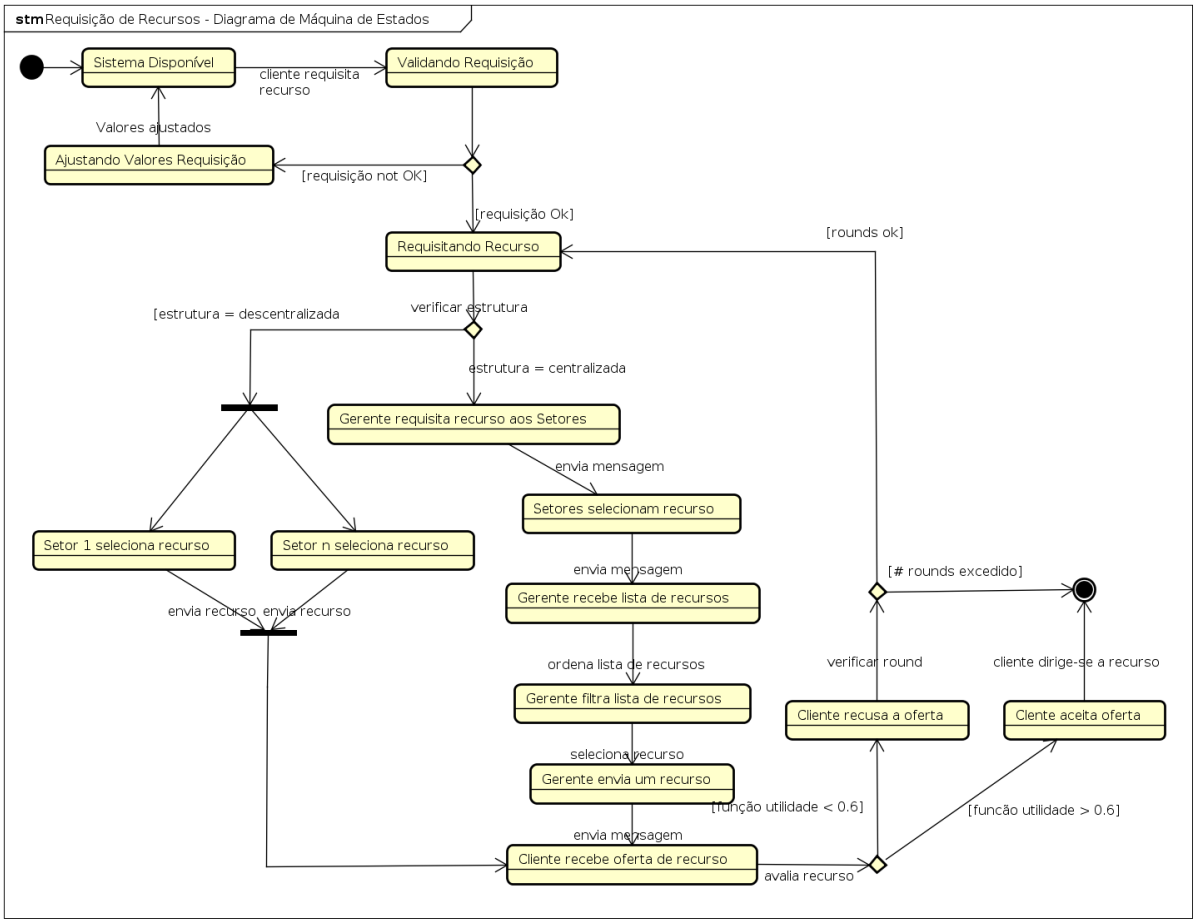
- **Controle de Recursos:** Responsável por fornecer estrutura de controle de acesso aos recursos e de utilização dos mesmos em tempo de execução;
- **Controle de Endereços:** Devido ao fato de haver vários níveis de organizações e também de aninhamento de agentes, assim é necessário traduzir o endereço de um agente, pois o mesmo pode não se encontrar no seu endereço original;
- **Banco de Dados:** Capaz de fornecer o armazenamento dos dados dos usuários, recursos, setores de forma persistente;
- **Controle de Fila:** Uma vez que a quantidade de recursos é limitada, consequentemente pode haver uma demanda maior que a oferta. Assim, há uma fila gerenciada pelo Controle de Fila;
- **Controle de Sessão:** Regula o histórico de negociação dos IClientes com o Gerente/Setores a fim de que não ocorra ofertas repetidas. Também é utilizado em caso de falhas para que seja recuperado o status da negociação;
- **Controle de Setores:** Responsável por fornecer uma estrutura para o gerenciamento dos Setores. Além disso, através desse controle é possível a verificação do uso de cada Setor;
- **Controle de Rede:** Uma vez que a utilização da camada de rede pelos agentes não é proporcionada por todas as linguagens de agentes, um artefato se torna requisito a fim de que os agentes possam comunicar via rede. No atual trabalho é empregado na comunicação entre os IClientes e Usuários.

6.3 MODELO DE NEGOCIAÇÃO

Uma vez que o objetivo do `MAPS-HOLO` é a alocação de recursos para usuários, assim se faz necessário estabelecer um modelo de negociação para que os recursos então sejam alocados aos usuários. Devido ao enfoque principal do atual trabalho não focar na negociação dos agentes, mas sim na modelagem e implementação da Arquitetura `MAPS-HOLO` , foi escolhido o modelo *Contract Net Protocol* por proporcionar ao sistema a negociação dos recursos de forma satisfatória e prática. Sendo o `MAPS-HOLO` configurável, ou seja, a Arquitetura pode ser configurada de maneira automática ou manual na forma *Head-Body* ou *Body-Body*, assim o modelo de negociação foi adaptado para ambos os cenários da arquitetura. A Figura 28 ilustra um diagrama de estados modelado para o modelo de negociação da Arquitetura.

Um ponto de diferença entre as negociações nos dois modos de configuração da Arquitetura é como os *rounds* são contabilizados. No modo *head-body* o *round* é incrementado a cada interação do ICliente com o Gerente após a requisição ser realizada. Cada interação é feita apenas uma oferta de recurso, pois o ICliente tem apenas contato com o Gerente. Já no modo

Figura 28 – Arquitetura MAPS-HOLO - Modelo de Negociação - Diagrama: Máquina de Estados



Fonte: Autoria Própria

body-body há múltiplos *rounds* para cada Setor. Por exemplo, dado os Agentes: ICliente1, SetorA, SetorB, SetorC e Gerente. Nesse exemplo o limite de rounds é três. A seguir é descrito como a negociação ocorre nas duas configurações.

- *Head-Body:*

  1. ICliente1 requisita ao Gerente um recurso (**Round: 1**);
  2. Gerente requisita aos Setores A,B,C um recurso;
  3. Setores selecionam os recursos com base em suas funções utilidade e enviam as ofertas ao Gerente;
  4. Gerente armazena as ofertas em um buffer (tamanho 3)
  5. Gerente calcula a sua função utilidade nas ofertas recebidas dos Setores;
  6. Gerente envia uma oferta do buffer;
  7. ICliente recebe a oferta e calcula a função utilidade;
  8. Caso aceite a oferta: IDriver recebe o recurso (**Round: 1**);

9. Caso recuse a oferta: Gerente envia uma nova oferta do buffer (**Round: 2**) (Caso o buffer fique vazio, o Gerente repete o passo 2). Se o IDriver recusar por mais duas vezes as ofertas (**Round: 4**), a negociação será encerrada.

- *Body-Body:*

1. ICliente1 requisita a todos os Setores um recurso (**Round: 1_A, Round: 1_B, Round: 1_C**);
2. SetorA seleciona um recurso com base na função utilidade e envia ao ICliente1;
3. ICliente1 calcula função utilidade e recusa a oferta do SetorA, assim SetorA envia uma nova oferta (**Round: 2_A, Round: 1_B, Round: 1_C**);
4. SetorC seleciona um recurso com base na função utilidade e envia ao ICliente1;
5. ICliente1 calcula função utilidade e recusa a oferta do SetorC, assim SetorC envia uma nova oferta (**Round: 2_A, Round: 1_B, Round: 2_C**);
6. SetorB seleciona um recurso com base na função utilidade e envia ao ICliente1;
7. ICliente1 calcula função utilidade e aceita a oferta do SetorB, logo ICliente1 encerra as negociações com os demais setores e dirige-se ao recurso do SetorB (**Round: 2_A, Round: 1_B, Round: 2_C**);
8. A negociação nesse modo é encerrada apenas quando os rounds em **todos** os setores forem encerrados;

6.4 CASOS DE USO

A fim de ilustrar as funcionalidades disponíveis para todos os agentes da `MAPS-HOLO`, abaixo é apresentado diagramas de caso de uso para cada agente.

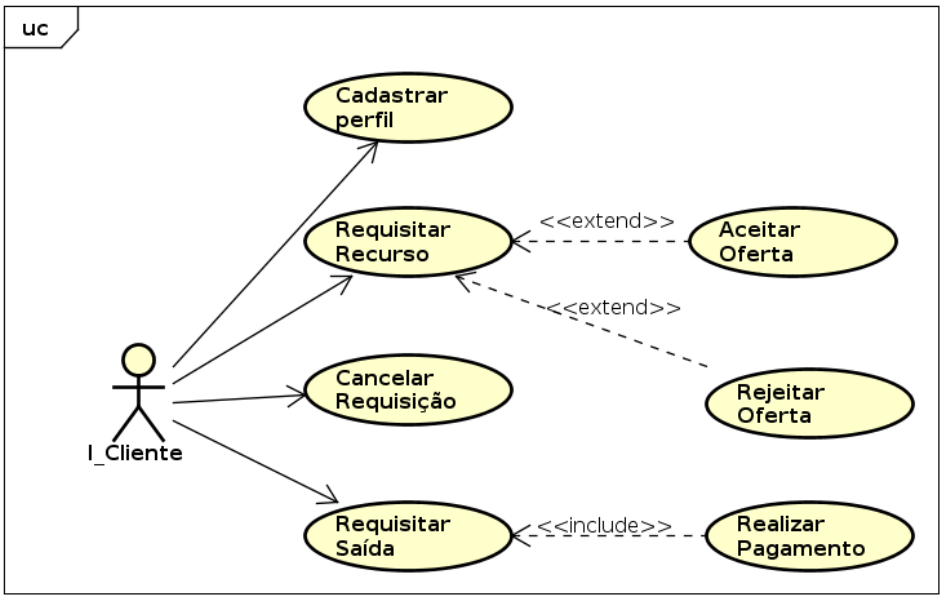
6.4.1 Agente ICliente

O diagrama de caso de uso do ICliente é exibido na figura 29.

Casos de Uso:

- **Cadastrar Perfil:** O ICliente poderá cadastrar seu perfil com suas preferências de utilização. Essas preferências são o valor de pagamento, tempo desejado de utilização do recurso e os respectivos pesos de preferência para esses valores;
- **Requisitar Recurso:** O agente poderá requisitar o recurso com base nas informações armazenadas em seu perfil. Após a requisição, o ICliente aguarda o recebimento de uma oferta, podendo **Aceitar a Oferta** ou **Rejeitar a Oferta**;

Figura 29 – Diagrama de Caso de Uso - Agente: ICliente



Fonte: Autoria Própria

- **Cancelar Requisição:** Enquanto aguarda o processo de negociação, o ICliente poderá cancelar a requisição e consequentemente a negociação;
- **Requisitar Saída:** O ICliente após o tempo desejado de utilização poderá requitar a saída do sistema. Após a requisição o ICliente **Realizará o pagamento** pelo recurso.

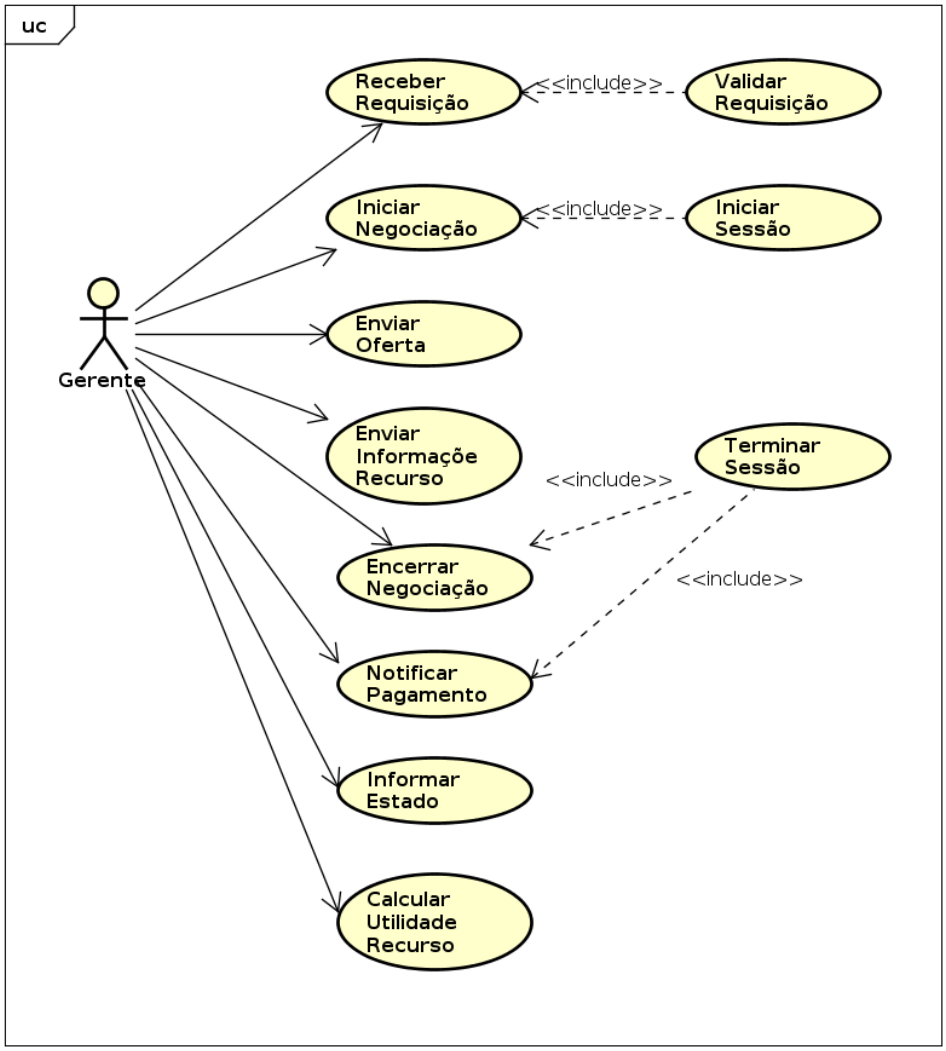
6.4.2 Agente Gerente

O diagrama de caso de uso do Gerente é exibido na figura 30.

Casos de Uso:

- **Receber Requisição:** O Gerente poderá receber uma requisição provinda de um ICliente. Após isso ele **Valida a Requisição** para verificar se os parâmetros de entrada (perfil do ICliente) está correto;
- **Iniciar Negociação:** Gerente inicia o processo de negociação enviando uma requisição aos Setores e consequentemente a **Sessão é Iniciada**;
- **Enviar Oferta:** Envia oferta do buffer para o ICliente;
- **Enviar Informações Recurso:** Ao passo que o ICliente aceita uma oferta, o Gerente envia maiores informações sobre o recurso (e.g localidade);
- **Encerrar Negociação:** Caso os rounds sejam encerrados ou o ICliente cancelou a requisição, a negociação é encerrada e a **Seção é Finalizada**;

Figura 30 – Diagrama de Caso de Uso - Agente: Gerente



Fonte: Autoria Própria

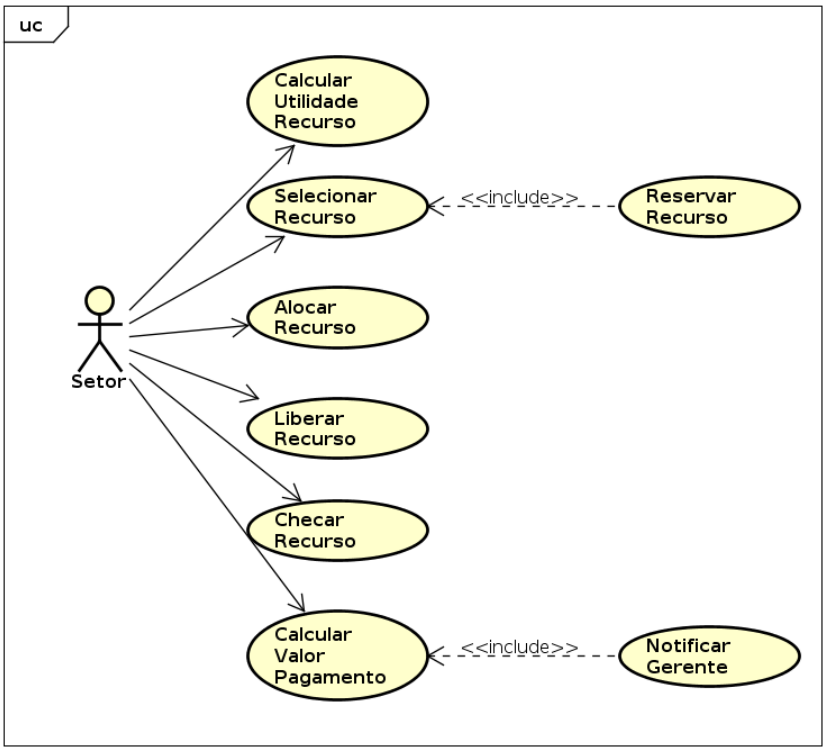
- **Notificar Pagamento:** Após o ICliente requisitar a saída, o Gerente notifica-o sobre o valor do pagamento e a **Seção é Finalizada**;
- **Informar Estado:** Envia o estado que se encontra (falha ou ok) para o Agente Observador;
- **Calcular Utilidade Recurso:** Ao receber as ofertas dos Setores, o Gerente calcula a função utilidade para esses recursos de acordo com os seus parâmetros (pesos).

6.4.3 Agente Setor - HeadBody

O diagrama de caso de uso do Setor - *Head-Body* é exibido na figura 31. Casos de Uso:

- **Calcular Utilidade Recurso:** Setor calcula a função utilidade para os agentes Recurso com base em seus parâmetros (pesos);

Figura 31 – Diagrama de Caso de Uso - Agente: Setor *Head-Body*



Fonte: Autoria Própria

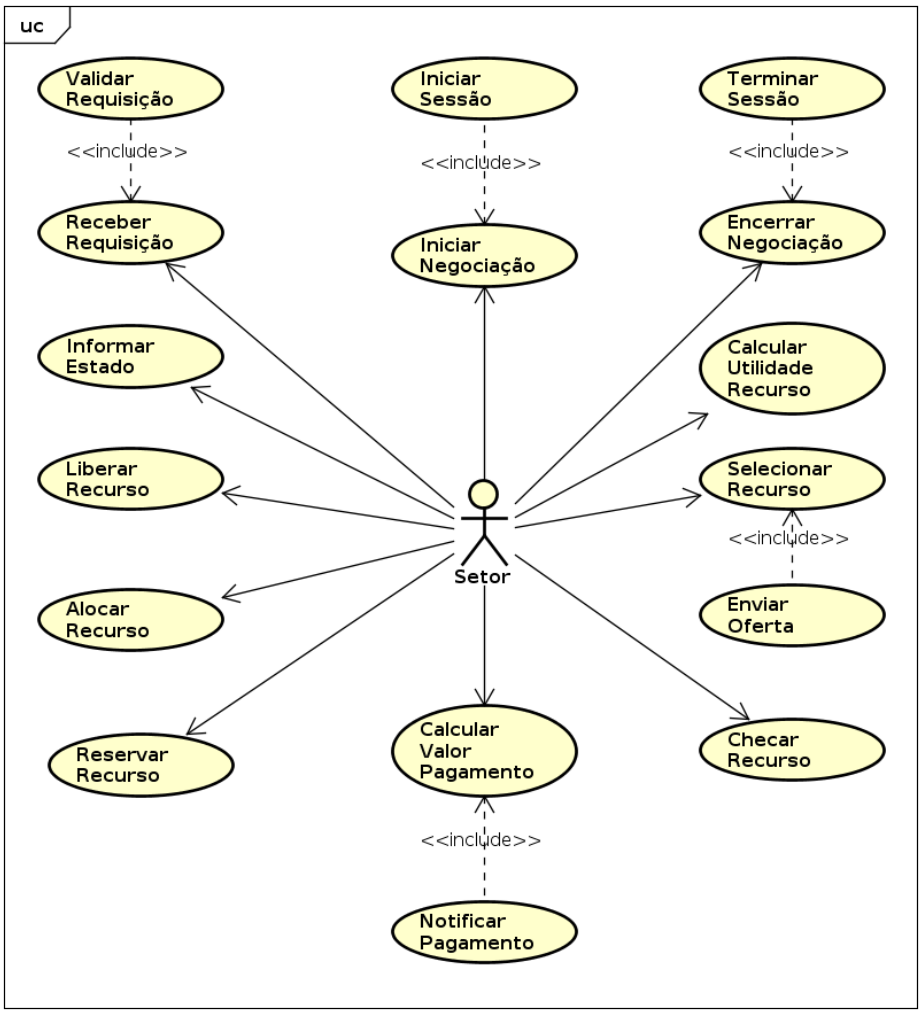
- **Selecionar Recurso:** Ao receber uma requisição do Gerente, o Setor seleciona um Recurso a ser alocado;
- **Reservar Recurso:** Após o ICliente ter aceitado a oferta, o Recurso é setado pra Reservado até que o ICliente informe a sua chegada/utilização do Recurso;
- **Alocar Recurso:** Ao passo que ICliente chega no local do Recurso ou inicia a utilização, o Recurso é setado para Alocado/Ocupado;
- **Liberar Recurso:** Ao passo que o ICliente realiza a requisição para sair, o Gerente solicita ao Setor para que libere o Recurso;
- **Checar Recurso:** Dado um intervalo de tempo, o Setor manda uma mensagem *ping* para o Recurso com a finalidade de checar o seu estado;
- **Calcular Valor Pagamento:** Após a requisição de saída, o Setor calcula o valor a ser pago pelo Recurso de acordo com o tempo de utilização;

6.4.4 Agente Setor - BodyBody

O diagrama de caso de uso do Setor - *Body-Body* é exibido na figura 32.

Casos de Uso:

Figura 32 – Diagrama de Caso de Uso - Agente: Setor *Body-Body*



Fonte: Autoria Própria

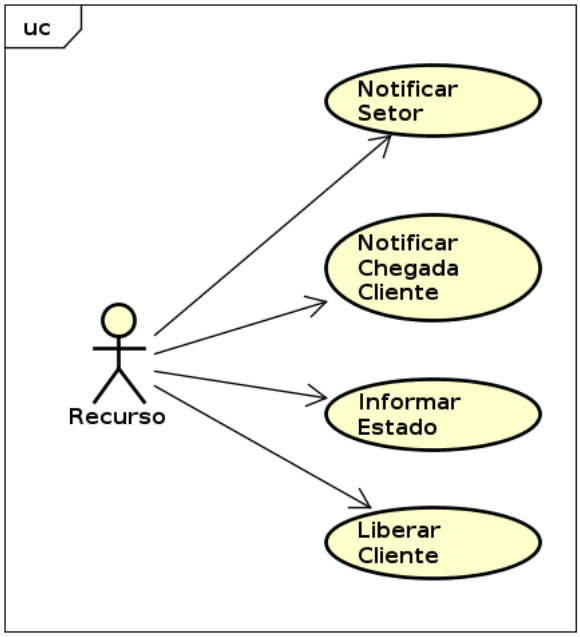
- **Receber Requisição:** O Setor poderá receber uma requisição provinda de um ICliente. Após isso ele **Valida a Requisição** para verificar se os parâmetros de entrada (perfil do ICliente) está correto;
- **Iniciar Negociação:** O Setor inicia o processo de negociação ao validar a requisição. A seguir, a **Sessão é iniciada**;
- **Encerrar Negociação:** Caso os rounds sejam encerrados ou o ICliente cancelou a requisição, a negociação é encerrada e a **Seção é Finalizada**;
- **Calcular Utilidade Recurso:** Após a inicialização de todo o sistema, o Setor calcula o valor da função utilidade de todos os Recursos;
- **Selecionar Recurso:** Após receber uma requisição de vaga do ICliente, o Setor seleciona um Recurso de acordo com a função utilidade e assim **Envia a oferta** ao ICliente;
- **Enviar Informações Recurso:** Ao passo que o ICliente aceita uma oferta, o Setor envia maiores informações sobre o recurso (e.g localidade);

- **Calcular Valor Pagamento:** Após a requisição de saída, o Setor calcula o valor a ser pago pelo Recurso de acordo com o tempo de utilização e assim **Notifica** o ICliente sobre o pagamento;
- **Reservar Recurso:** Após o ICliente ter aceitado a oferta, o Recurso é setado pra Reservado até que o ICliente informe a sua chegada/utilização do Recurso;
- **Alocar Recurso:** Ao passo que ICliente chega no local do Recurso ou inicia a utilização, o Recurso é setado para Alocado/Ocupado;
- **Liberar Recurso:** Após o ICliente informar que deixou o local do Recurso, o Recurso é setado para Livre;
- **Informar Estado:** Envia o estado que se encontra (falha ou ok) para o Agente Observador;

6.4.5 Agente Recurso

O diagrama de caso de uso do Recurso é exibido na figura 33.

Figura 33 – Diagrama de Caso de Uso - Agente: Recurso



Fonte: Autoria Própria

Casos de uso:

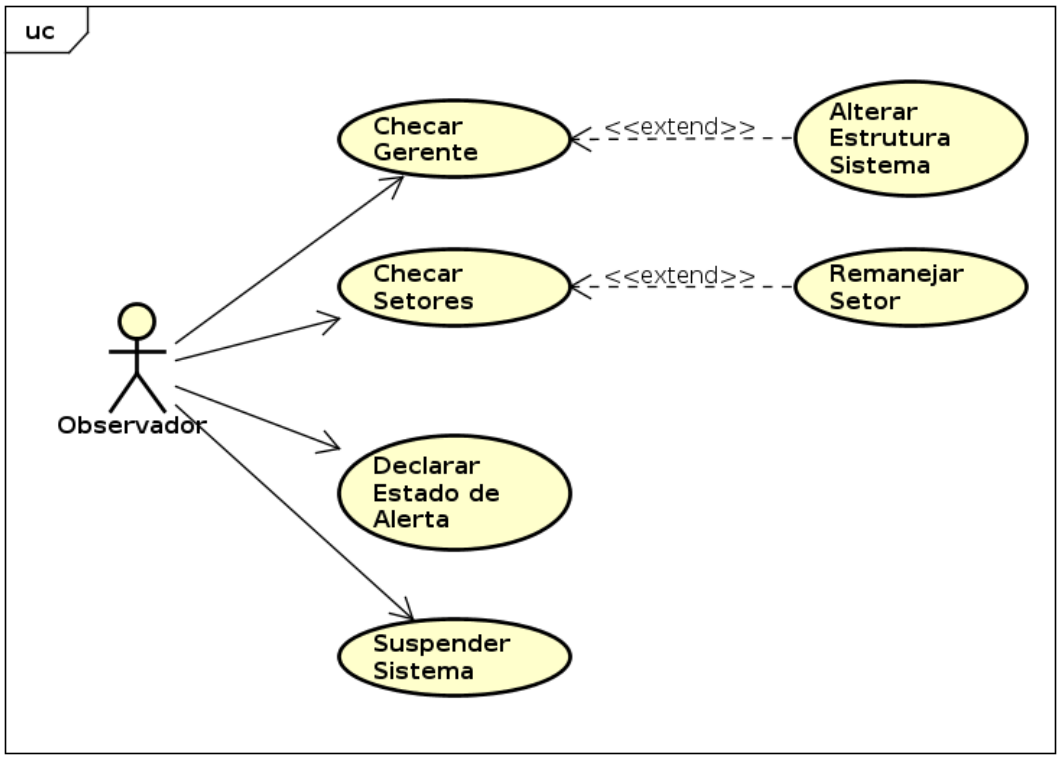
- **Notificar Setor:** Após a iniciação do agente Recurso, ele notifica o Setor sobre seu estado;
- **Notificar Chegada Cliente:** Ao passo que o ICliente chegue ao local ou inicie a utilização, o Recurso notifica o Setor sobre a chegada;

- **Informar Estado:** Informa ao Setor o estado do agente (falha ou ok);
- **Liberar Cliente:** ICliente notifica a saída, o Recurso então finaliza o tempo de utilização e notifica o Setor.

6.4.6 Agente Observador

O diagrama de caso de uso do Observador é exibido na figura 34.

Figura 34 – Diagrama de Caso de Uso - Agente: Observador



Fonte: Autoria Própria

- **Checar Gerente:** Em um intervalo de tempo $t1$ o Observador manda uma mensagem *ping* para o Gerente. Caso após o tempo $t1$ o Gerente não responder com a mensagem *pong*, então caracteriza-se uma falha no sistema, assim o Observador **Altera a Estrutura do Sistema** para *body-body* ;
- **Checar Setores:** Em um intervalo de tempo $t2$ o Observador manda uma mensagem *ping* para os Setores. Caso após o tempo $t2$ algum gerente não responder com *pong*, caracteriza-se falha em um Setor, assim é necessário **Remanejar o Setor** e seus agentes Recurso;
- **Declarar Estado de Alerta:** Após a ocorrência de falhas, o Observador poderá emitir um alerta (*Yellow, Orange e Red Alert*);

- **Suspender Sistema:** Caso ocorra uma falha grave, o Observador suspenderá o sistema para novas requisições.

6.5 CONSIDERAÇÕES FINAIS

A modelagem em uma perspectiva genérica da MAPS-HOLO proporciona uma visão mais abstrata e em alto nível da Arquitetura, sendo por sua vez de grande auxílio para verificar a possibilidade de implementação da mesma Arquitetura em uma plataforma diferente do JaCaMo, por exemplo. Sendo assim, um possível futuro trabalho pode ser o desenvolvimento em uma outra plataforma e então a comparação dessas. Por fim, a perspectiva genérica da Arquitetura possibilita também um vislumbre maior de aplicação dessa não apenas em vagas de estacionamento, mas em diferentes recursos. Assim, a MAPS-HOLO pode servir de modelo para diferentes cenários de aplicação.

7 MAPS HOLO - MODELAGEM E DESENVOLVIMENTO DA ARQUITETURA NO JACAMO

Esse capítulo tem o objetivo de apresentar o desenvolvimento da Arquitetura MAPS-HOLO por meio do *framework* JaCaMo . Além disso, tem como objetivo também demonstrar a possibilidade do desenvolvimento inicial de aplicações holônicas nesse *framework* .

A divisão deste capítulo dá-se da seguinte maneira: a seção 7.1 apresenta uma breve introdução do capítulo, a seção 7.2 a etapa de construção e inicialização, as seções 7.3 e 7.4 o processo de alocação das vagas de estacionamento, as seções de 7.5 à 7.10 no que diz respeito a tolerância as falhas e por fim a seção 7.11 as considerações finais.

7.1 INTRODUÇÃO

O desenvolvimento por meio do JaCaMo compreende três grandes camadas: agentes (Jason), ambiente (Cartago) e organizações (Moise). Além das camadas, é necessário desenvolver a integração entre elas de forma com que o SMAH seja uniforme e otimizado. Assim, faz-se necessário antes do desenvolvimento propriamente dito a modelagem inicial da Arquitetura. Com base nisso, o capítulo 6 ilustra e descreve os principais requisitos da MAPS-HOLO .

Tendo em vista a motivação de tornar claro e simples a apresentação da extensa arquitetura MAPS-HOLO , será adotado nesse documento as seguintes etapas, as quais refletem a ordem de ocorrência no sistema : Construção e instanciação, Alocação das vagas e Restabelecimento de Sistema em caso de Falhas. Na explanação dos códigos da MAPS-HOLO será demonstrado apenas o que for de vital importância para a compreensão do Arquitetura. Além disso, será dado enfoque nos elementos do *framework* JaCaMo que proporcionam o desenvolvimento de um SMAH por meio do Jason, Cartago, Moise e a sinergia existente entre as ferramentas. Finalmente, na seção 7.11 é apresentado uma discussão a respeito da viabilidade do desenvolvimento de aplicações holônicas no JaCaMo .

7.2 ETAPA DE CONSTRUÇÃO E INSTANCIAÇÃO

O *framework* JaCaMo possui um arquivo que configura os agentes iniciais (suas crenças, objetivos, instâncias, etc.), os artefatos do ambiente e a organização. Esse arquivo (citado na seção 5.1.7) é o JCM. Contudo, na implementação da MAPS-HOLO optou-se pela utilização do arquivo .JCM juntamente com um agente construtor, chamado Builder[1]. A escolha da utilização

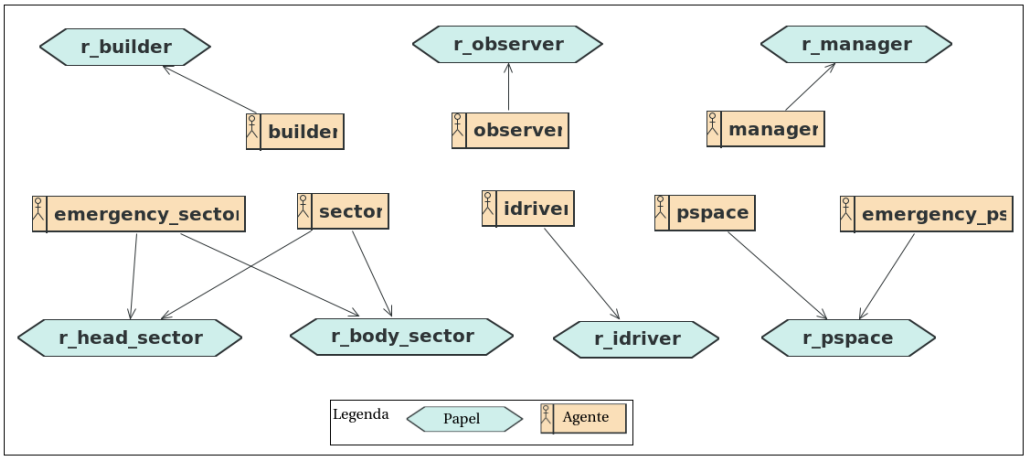
[1] A partir desse capítulo, os agentes estarão em Inglês devido a implementação ser realizada no idioma Inglês. Sendo assim, temos os agentes traduzidos: (ICliente → IDriver), (Gerente → Manager), (Setor → Sector), (Recurso → PSpace), (Observador → Observer) e (Construtor → Builder)

do *Builder* torna o sistema mais flexível e dinâmico, pois pode-se ajustar com os parâmetros iniciais qual será a estrutura de todo o Sistema. Assim, o Builder é o primeiro agente a ser iniciado no sistema e após iniciar, todo o SMAH é construído.

7.2.1 Agentes, Papéis e Grupos

Além do agente *Builder* , outros agentes desempenham papéis importantes no sistema. Na Figura 35 é apresentado todos os agentes e seus respectivos papéis no sistema na perspectiva do Prometheus. Já na Figura 36 são apresentados os mesmos agentes e seus relacionamentos com os grupos na perspectiva do Moise. Na Figura 35 e 36 é ilustrado todos os agentes do sistema que podem aparecer no Sistema.

Figura 35 – Papéis dos Agentes - Perspectiva Prometheus



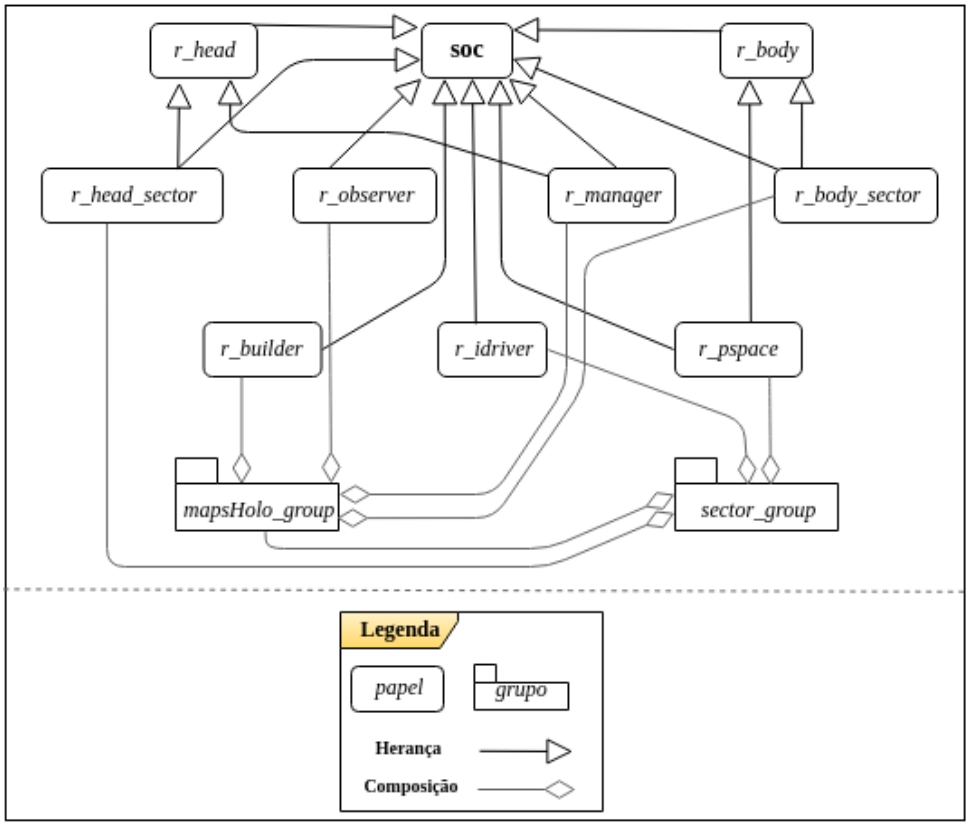
Fonte: Autoria Própria

Nota-se que os agentes `emergency_ps` e `emergency_sector` serão instânciados somente em caso de falhas. Além disso, na Figura 36 é apresentado dois grupos do `MAPS-HOLO`: `mapsHoloGroup` e `sectorGroup`. O grupo `mapsHoloGroup` é o grupo principal ou global de todo o sistema, sendo por sua vez o grupo `sectorGroup` um grupo aninhado, ou um sub-grupo do grupo principal.

Observa-se que ambas Figuras (35 e 36) diferentes papéis são utilizados e que foram previamente descritos no capítulo 6. Porém, na implementação no JaCaMo adotou-se novos quatro papéis, a fim de que seja possível a implementação nos modelos *Head-Body* e *Body-Body*. O papel `r_head` é um papel genérico que define o relacionamento entre autoridade com o papel `r_body`, sendo da mesma forma o papel `r_body` de submissão para com o papel `r_head`. Os papéis `r_head_sector e r_body_sector` substituem o papel de `r_sector`, pois o agente *Sector* desempenha dois papéis diferentes.

Na figura 36 é apresentado dois grupos: `mapsHolo_group` e `sector_group`. Ambos os grupos são denotados como organizações holárquias, ou simplesmente holarquias, pois através deles são possíveis as atribuições de configuração *Head-Body* e *Body-Body*. Para que ainda

Figura 36 – Papéis e Grupos dos Agentes - Perspectiva Moise



Fonte: Autoria Própria

fique mais claro a relação dos grupos do Moise como holarquias, a Figura 37 apresenta os links desenvolvidos entre os papéis para que os agentes possam estabelecer os relacionamentos entre si mediante o desempenho dos papéis.

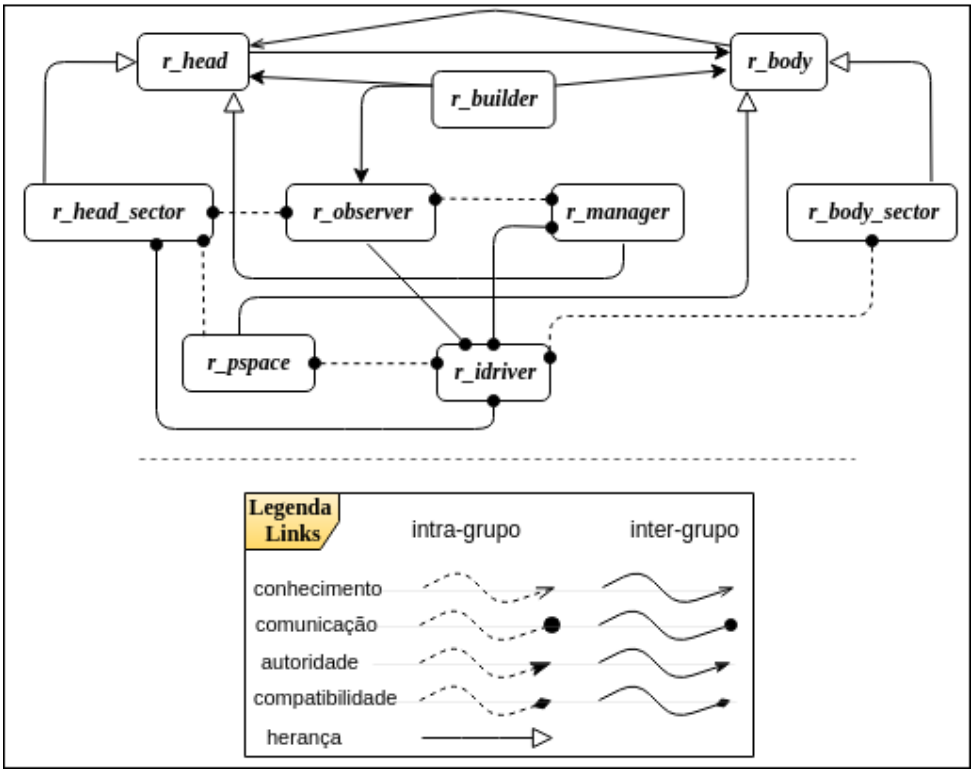
7.2.2 Configuração Inicial - Agente Builder

O Agente *Builder* inicialmente recebe os parâmetros como crenças iniciais para que o sistema seja construído. As crenças iniciais são setadas no código JCM. O Código 17 apresenta o agente builder e a sua configuração inicial. Após o agente *Builder* , o agente *Observer* é também instanciado;

```
1  mas mAPS_HOLO {
2          agent builder{
3                  beliefs: nSectors(5),
4                           nPSpaces(20), //# of parking spaces by sector
5                           structure("BB"),
6                           network(true),
7                           simulation(false)}
8      agent observer}
```

Código 17 – Arquivo JCM - Configuração inicial Agentes Builder e Observer

Figura 37 – Papéis e seus relacionamentos - Moise



Fonte: Autoria Própria

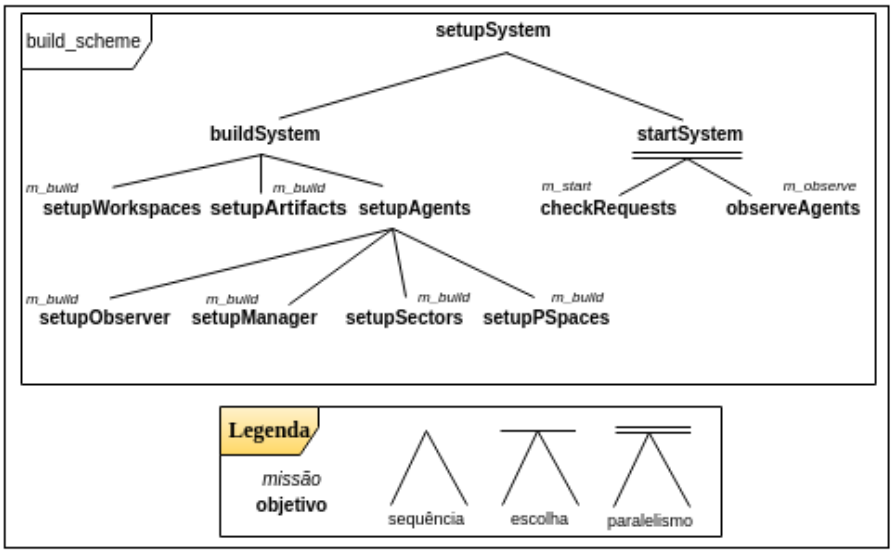
As crenças iniciais do agente *Builder* são as seguintes:

- **nSectors(int):** Informa a quantidade de setores do sistema;
- **nPSpaces(int):** Número de vagas por setor;
- **structure(String):** Qual o tipo da estrutura do sistema *(BB - Body-Body / HB - Head-Body)*
- **network(boolean):** Se o sistema receberá requisições dos usuários via rede;
- **simulation(boolean):** O sistema poderá funcionar em modo simulação (destinado para os testes);

Após a instanciação do agente *Builder* , há objetivos do Sistema a serem atingidos para que o resto da Arquitetura esteja pronta. Para isso, o Moise fornece os objetivos, missões e esquemas sociais da organização para que haja ordem e organização no cumprimento das missões e objetivos. Os papéis, missões, objetivos e esquemas sociais de todo o `MAPS-HOLO` estão disponíveis no Apêndice 9.

Na etapa de construção do sistema, foi elaborado um Esquema Social visando a construção organizada e ordenada de todo o sistema (agentes, artefatos e grupos). Para isso, a Figura 38 mostra o Esquema Social `build_Scheme` e as missões e objetivos nele contidos.

Figura 38 – Build_Scheme - Moise



Fonte: Autoria Própria

Com base na Figura 38, os objetivos estão alocados em suas respectivas missões. Há três principais missões relacionadas nessa primeira etapa: `m_build, m_observe e m_start`. A missão `m_build` possui os objetivos de construção do sistma, a `m_observe` da checagem dos agentes (Gerente, Setores e PSpaces) e `m_start` para a checagem constante de novas requisições provindas do *IDrivers* . No Código 18 é mostrado o código Moise que configura o Esquema Social do `Build_Scheme` e das missões `m_build, m_observe e m_start`. Além disso, é apresentado também a dimensão normativa das missões na Tabela 3.

Tabela 3 – Dimensão normativa - Build_Scheme

ID Norma	Tipo	Papel	Missão
`n_build`	*obligation*	`r_builder`	`m_build`
`n_start`	*obligation*	`r_observer`	`m_start`
`n_observe`	*obligation*	`r_observer`	`m_observe`

Autoria Própria

```
<functional-specification>
    <scheme id= "build_scheme">
        <goal id = "setupSystem">
                <plan operator = "sequence">
                        <goal id = "buildSystem">
                            <plan operator="sequence">
                                <goal id ="setupWorkspaces"/>
                                <goal id ="setupArtifacts"/>
                                <goal id ="setupAgents">
                                        <plan operator = "sequence">
                                                <goal id ="setupObserver"/>
                                                <goal id ="setupManager"/>
                                                <goal id ="setupSectors"/>
                                                <goal id ="setupPSpaces"/>
```

```
                                        </plan>
                                </goal>
                            </plan>
                        </goal>
                        <goal id ="startSystem">
                                <plan operator="parallel">
                                        <goal id ="checkRequests"/>
                                        <goal id ="observeAgents"/>
                                </plan>
                                   </goal>
                </plan>
        </goal>
        <mission id="m_build" min="1" max="1">
                <goal id="setupWorkspaces"/>
                <goal id="setupArtifacts"/>
                <goal id ="setupObserver"/>
            <goal id ="setupManager"/>
             <goal id ="setupSectors"/>
             <goal id ="setupPSpaces"/>
        </mission>
        <mission id="m_start" min="1">
                <goal id="checkRequests"/>
        </mission>

        <mission id="m_observe" min="1">
                <goal id="observeAgents"/>
        </mission>
    </scheme>
    ...
    <normative-specification>
        <norm id="n_build" type="obligation" role="r_builder" mission="m_build"/>
        <norm id="n_start" type="obligation" role="r_observer" mission="m_start"/>
        <norm id="n_observe" type="obligation" role="r_observer"
        ↪  mission="m_observe"/>
    </normative-specification>
```

Código 18 – Esquema Social - Build_Scheme

7.2.3 Agente Builder - Missão m_build

A seguir é descrito as etapas da missão `m_build` por meio da descrição dos códigos e exibição de diagramas por meio do Prometheus;

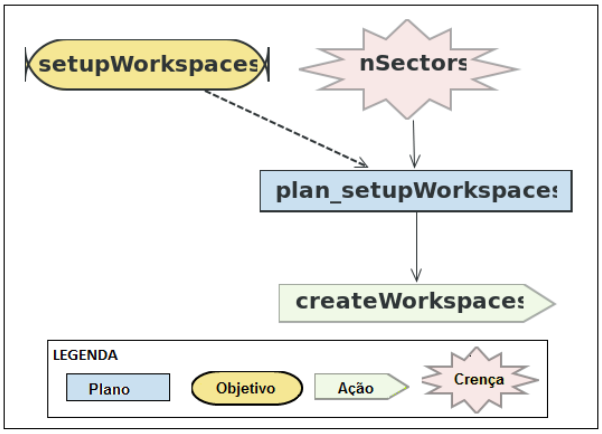
1. **Objetivo - setupWorkspaces:** Destina-se a criação e instanciação de todos os Workspaces dos Setores no Cartago, sendo um workspace destinado para cada Setor. A Figura 39 ap-

resenta o diagrama Prometheus ilustrando o escopo do plano para a configuração dos Workspaces. Por fim, o Código 19 descreve o código em Jason desse plano.

```
1    +!setupWorkspaces : nSectors(NS) <-
2        for (.range(I,0,NS-1) ) {
3            .concat("wsp_sector_",I,WSP_SECTOR);
4            createWorkspace(WSP_SECTOR);
5        }.
```

Código 19 – JaCaMo - Agente *Builder* - setupWorkspaces

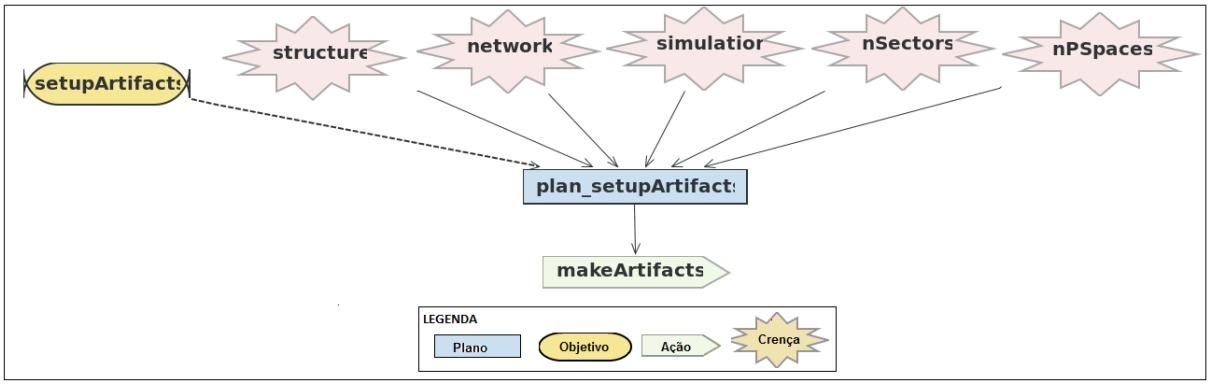
Figura 39 – Diagrama Prometheus - Plano: setupWorkspaces



Fonte: Autoria Própria

2. **Objetivo - setupArtifacts:** Criação e instânciação dos principais artefatos da `MAPS-HOLO`. Abaixo é descrito a função de cada artefato demonstrado no Código 20. A Figura 40 ilustra o plano para o cumprimento desse objetivo. Na linha 1 do Código 20 é apresentado uma regra do Jason, onde através dela é possível recuperar informações da base de crenças e realizar inferências sobre ela.

Figura 40 – Diagrama Prometheus - Plano: setupArtifacts



Fonte: Autoria Própria

```
getInfoSystem(ST,NET,SIMU,NS,NPS) :- structure(ST) & network(NET) &
↪  simulation(SIMU) & nSectors(NS) & nPSpaces(NPS).

+!setupArtifacts : getInfoSystem(ST,NET,SIMU,NS,NPS) <-
        makeArtifact("a_SectorControl","artifacts.A_SectorControl",[],ART_SECT
        ↪  ORCONTROL);
        focus(ART_SECTORCONTROL);

        makeArtifact("a_BDConnection","artifacts.A_BDConnection",[],__);
        makeArtifact("a_SessionControl","artifacts.A_SessionControl",[NS],__);
        makeArtifact("a_IDriverTools","artifacts.A_IDriverTools",[],__);
        makeArtifact("a_QueueControl","artifacts.A_QueueControl",[],__);
        makeArtifact("a_StructureInfo","artifacts.A_StructureInfo",[ST,NET,SIM
        ↪  U,NS,NPS],ART_STRINFO);
        focus(ART_STRINFO);

        if(NET = true){
                makeArtifact("a_DriverConnection","artifacts.A_DriverConnectio
                ↪  n",[],ART_DRIVERCONNECTION);
        }
        if(SIMU = true){
                makeArtifact("a_Simulation","artifacts.A_Simulation",[],ART_SI
                ↪  MULATION);
        }.
```

Código 20 – JaCaMo - Agente *Builder* - setupArtifacts

Tabela 4 – Principais Artefatos da MAPS-HOLO

Artefato	Função
A_SectorControl	Possui um HashMap para armazenagem de informações dos Setores
A_BDConnection	Interface de comunicação com o Banco de Dados
A_SessionControl	Controle de sessão de alocação das vagas. Utilizado a fim de gerenciar as vagas ofertadas e recuperação de histórico de negociação
A_IDriverTools	Proporciona aos IDrivers funcionalidades extras (Cálculo da função utilidade)
A_QueueControl	Controle da fila de IDrivers em caso do Sistema estiver cheio
A_StructureInfo	Informações a respeito da estrutura do Sistema (Head-Body ou Body-Body, # Setores, # PSpaces, % de uso de cada setor, emissão de alertas
A_DriverConnection	Interface de comunicação TCP (Socket) para com os usuários
A_Simulation	Artefato utilizado para simulações (Usuários simulados)

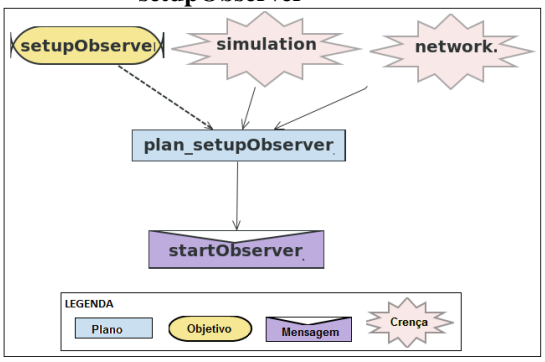
Fonte: Autoria Própria

O Código 20 apresenta os principais artefatos da MAPS-HOLO, os quais fornecem aos agentes Jason funcionalidades que não são nativas a eles. Assim, é possível por meio

do Cartago desenvolver funcionalidades extras que auxiliam esses agentes. A Tabela 4 descreve as funcionalidades dos artefatos exibidos no Código 20.

3. **Objetivo - setupObserver:** Responsável pela iniciação do agente *Observer*. O agente *Observer* já havia sido criado no Código 17, entretanto, ainda não iniciado. Para iniciar, o *Builder* envia uma mensagem ao *Observer* (linha 3), invocando-o plano startObserver. O Código 21 apresenta o plano para o start no *Observer* e a Figura 41 o diagrama Prometheus para esse plano.

Figura 41 – Diagrama Prometheus - Plano: setupObserver



Fonte: Autoria Própria

```
1    +!setupObserver : getInfoSystem(ST,NET,SIMU,__,__)  <-
2            .send(observer,achieve,startObserver(NET,SIMU)).
```

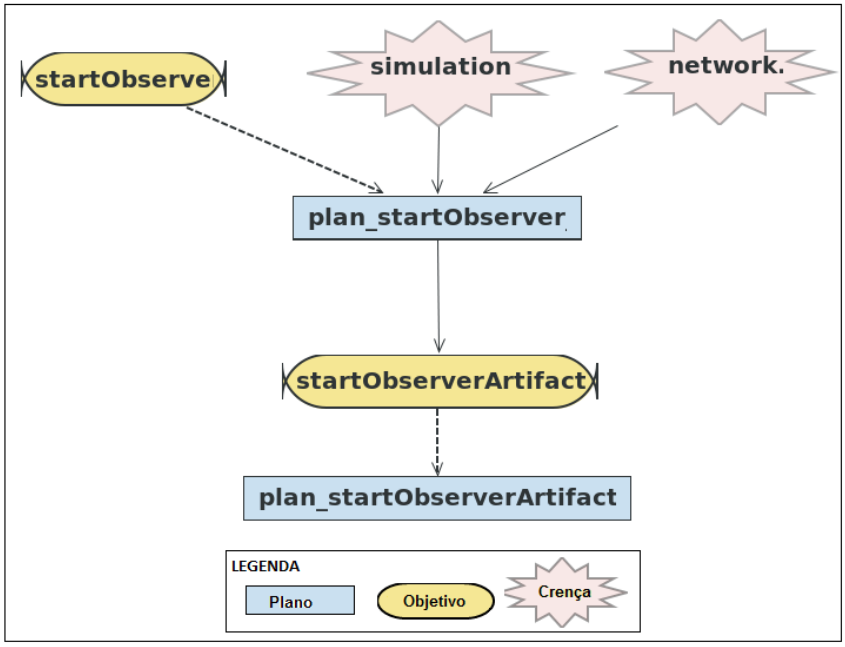
Código 21 – JaCaMo - Agente *Builder* - setupObserver

O Código 22 apresenta o plano do agente *Observer* ao ser invocado pelo agente *Builder*. O plano para o objetivo `startObserverArtifacts` será resumido devido ao seu tamanho. Assim, o Código irá exibir apenas o de vital importância para a compreensão do trabalho [2]. Consequentemente, os próximos agentes a serem detalhados (*Manager, Sector, IDriver e PSpace*) irão apresentar os seus códigos resumidos[3]. Por fim, a Figura 42 ilustra esse plano do *Observer*.

```
1  +!startObserver(NET,SIMU)[source(AG)]<-
2          +simulation(SIMU); +network(NET);
3          !startObserverArtifacs(NET,SIMU).
4  +!startObserverArtifacs(NET,SIMU) <-
5          ...
6      if(NET = true){
7                  lookupArtifact("a_DriverConnection",ART_DRIVERCONNECTION);
8                  focus(ART_DRIVERCONNECTION);
9          }if(SIMU = true){
10                 lookupArtifact("a_Simulation",ART_SIMULATION);
```

[2] O código completo da MAPS-HOLO encontra-se em: www.github.com/lcastropg/mapsholo
[3] O símbolo de reticências (...) nos códigos demonstra que o mesmo foi resumido

Figura 42 – Diagrama Prometheus - Plano: startObserver



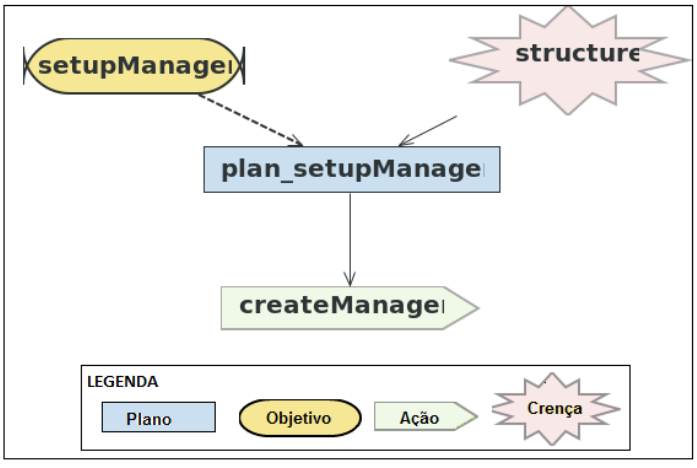
Fonte: Autoria Própria

```
11                  focus(ART_SIMULATION);
12          }.
```

Código 22 – JaCaMo - Agente *Observer* - startObserver

4. **Objetivo - setupManager:** Caso a estrutura setada pelo código 17 for *Head-Body (HB)*, o agente *Manager* será criado e inicializado. Abaixo é descrito o código do agente *Builder* na criação do *Manager* no Código 23 e na Figura 43 a ilustração do diagrama Prometheus desse plano. No Código 24 é apresentado o código do *Manager* na sua inicialização e por fim na Figura 44 o diagrama Prometheus exibindo o plano de inicialização do *Manager* .

Figura 43 – Diagrama Prometheus - Plano: setupManager



Fonte: Autoria Própria

```
1    +!setupManager : structure(ST)[artifact_id(ART)] <-
2        if(ST = "HB"){
3            jacamo.create_agent("manager","manager.asl");
4        }.
```

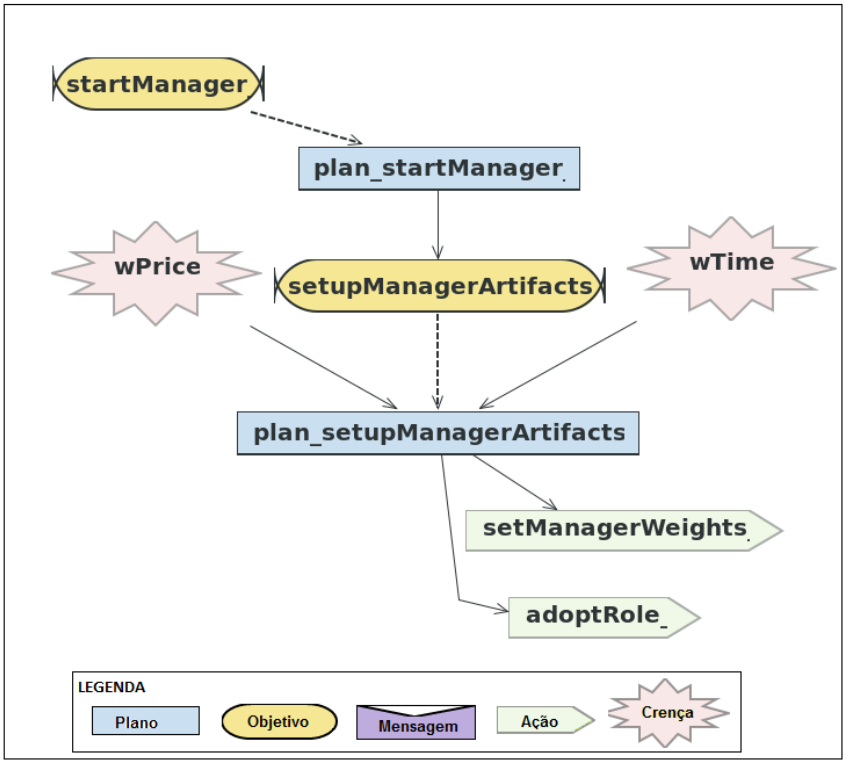
Código 23 – JaCaMo - Agente *Builder* - setupManager

```
1    +!startManager <-
2           joinWorkspace("maps_holo_org",WSP_ORG);
3           !setupManagerArtifacts.
4    +!setupManagerArtifacts : wPrice(WP) & wTime(WT)<-
5           ...
6           setManagerWeights(WP,WT)[artifact_id(ART_SESSIONCONTROL)];
7           adoptRole(r_manager)[artifact_id(ART_MAIN_GROUP)].
```

Código 24 – JaCaMo - Agente *Manager* - startManager

A inicialização do *Manager* possui etapas importantes no funcionamento da Arquitetura no modo *Head-Body*. Na linha 3, o *Manager* recupera da sua base de crenças os seus valores de peso para o preço e tempo (wPrice(WP) e wTime(WT)). Esses valores serão utilizados na etapa de negociação das vagas. Após a recuperação desses valores, o *Manager* configura esses valores de peso no artefato de sessão (Linha 6). Por fim na linha 7, o *Manager* adota o papel `r_manager`, o qual é ilustrado na Figura 36 e 37.

Figura 44 – Diagrama Prometheus - Plano: startManager



Fonte: Autoria Própria

5. **Objetivo - setupSectors e setupPSPaces** Responsável pela criação e inicialização de todos os agentes *Sector* e *PSpace* do MAPS-HOLO . Além disso, nessa etapa é criado também todas as holarquias de cada Setor. O Código 25 é apresentado o código do plano para a criação dos Setores feita pelo agente *Builder* . Já no Código 26 é descrito o código da inicialização dos setores nos seus respectivos agente *Sector* . O Código da inicialização dos Setores destaca-se dos Códigos anteriores, pois é através dele que as organizações holárquicas (sector_groups) serão criadas e aninhadas como sub-holarquias da holarquia principal (maps_holo_group). A tabela 5 apresenta as principais funcionalidades descrita no Código 25.

```
1   getInfoSystem(ST,NET,SIMU,NS,NPS) :- structure(ST) & network(NET) &
    ↪  simulation(SIMU) & nSectors(NS) & nPSpaces(NPS).
2   ...
3   +!setupSectors : getInfoSystem(ST,NET,SIMU,NS,NPS) <-
4   for (.range(I,0,NS-1) ) {
5       .concat("sector_",I,SECTOR_AGENT);
6       jacamo.create_agent(SECTOR_AGENT,"sector.asl");
7       assignSectorSTR(SECTOR_AGENT)[artifact_id(ART_STRINFO)];
8       registerSector(SECTOR_AGENT,WPRICE,WTIME)[artifact_id(ART_SECTORCONTRO
        ↪  L)];
9       .send(SECTOR_AGENT,tell,[nPS(NPS),wPrice(WPRICE),wTime(WTIME),sectorCo
        ↪  de(I)]);
10      .concat("sector_",I,"_grp",SECTOR_GROUP);
11      createGroup(SECTOR_GROUP, sectorGroup, GrArtId);
12      setParentGroup(maps_holo_grp)[artifact_id(GrArtId)];
13  for(.range(J,0,NPS-1)){
14          .concat("pspace_",I,"_",J,PS_AGENT);
15          jacamo.create_agent(PS_AGENT,"pspace.asl");
16          .send(PS_AGENT,tell,[myGroup(SECTOR_GROUP),mySector(SECTOR_AGENT)]
            ↪  );
17      }
18      .send(SECTOR_AGENT,tell,sectorOk(true));
19  }.
20  +!setupPSpaces <-
21      .print("All the PSAgents will be created!").
```

Código 25 – JaCaMo - Agente *Builder* - setupSectors

Um dos pontos importantes em SMAHs é o estabelecimento do Contrato de uma holarquia. O Contrato define as normas de utilização para os agentes presentes na holarquia. Na implementação do MAPS-HOLO o Contrato é definido como os valores-peso para valor e tempo de permanência de cada vaga de estacionamento. Assim, o *Sector* define o Contrato para os agentes *PSpace* , pois sendo ele o *head* da sua holarquia, tem autoridade para definir o Contrato. O Contrato também deve reger as regras da holarquia. No Moise, as regras do Contrato são regidas pela dimensão normativa e estrutural, pois através das normas (obrigação e permissão) e relacionamentos (autoridade, comunicação, etc.) entre os

Tabela 5 – Explanação do Código 25

Linha	Funcionalidade
1	*Regra de inferência (getInfoSystem)*
4-15	*Laço de repetição (i = 0...# Setores)*
6	*Criação de um agente Sector*
7	*Atribuição de dado do agente Sector no artefato (Structure_Info)*
8	*Registro do Setor no artefato (Sector_Control)*
9	*Envio de mensagem ao agente Sector (# PSpaces, weightPrice, weightTime,sectorCode(i)*
11	*Criação da holarquia Setor*
12	*Holarquia Setor fica aninhada na Holarquia Principal*
13-17	*Laço de repetição (j=0...#nPSpaces - 1)*
15	*Criação de um agente PSpace*
16	*Envio de mensagem a um agente PSpace (myGroup(Sector_Group), mySector(Sector_Agent)*
18	*Envio de mensagem a um agente Setor (Informa que todos os seus agentes PSpaces foram criados)*
19	*Plano de criação dos agentes PSpaces. Nota: O objetivo da organização é cumprido nesse plano, contudo os agentes PSpaces já foram criados no plano anterior*

Fonte: Autoria Própria

agentes as normas de utilização de uma holarquia são estabelecidos. Na linha 8 do Código 25 é atribuído pela crença `wPrice e wTime` os valores do Contrato para o *Sector* e sua holarquia.

Destaca-se ainda as linhas $11-12$ do Código 25 as funções `createGroup e setParentGroup`, pois é através delas a criação de grupos no Moise e de atribuir paternidade a grupos. O conceito de paternidade em grupos no Moise possui uma relação direta no estabelecimento da relação *Head-Body* das holarquias, pois o grupo filho torna-se um sub-grupo do grupo pai. Sendo assim, uma holarquia torna-se filha de outra. Assim, através dessas funções é estabelecido o relacionamento de aninhamento das holarquias na `MAPS-HOLO`.

O Código 26 descreve a inicialização do agente *Sector*. No código há a presença de dois objetivos e seus respectivos planos. Na linha 1 o *Sector* é inicializado ao tornar-se participante de dois workspaces: `wsp_sector_I` e `maps_holo_org`. A razão da participação desse agente em dois workspaces deriva-se do fato de que o *Sector* está presente em duas holarquias (`sector_group`, `maps_holo_group`), sendo assim presente em ambos workspaces. Após isso, o *Sector* inicia os seus artefatos, tendo destaque para a linha 9 onde o *Sector* cria e inicializa o seu próprio artefato para o controle e gerenciamento das vagas de estacionamento: `A_PSControl` [4]. Esse artefato tem como finalidade principal fornecer ao agente *Sector* o controle das vagas de estacionamento. As vagas de estacionamento são representadas pelos agentes *PSpace* , porém, as suas informações (disponibilidade, preço, tempo de permanência e endereço) são salvas no `A_PSControl`. Nas linhas $12 - 13$ o

[4] Esse artefato será exibido na Figura 46

Sector através da função `adoptRole` desempenha dois papéis: o `r_head_sector` em sua própria holarquia como o agente *head* e `r_body_sector` como agente *body* na holarquia principal. Por fim, na linha 14 o *Sector* coloca-se como dono/responsável pela sua própria holarquia.

```
1  +!startSector<-
2      .concat("wsp_",Me,WSPNAME);
3      joinWorkspace(WSPNAME,WSP_SECTOR);
4      joinWorkspace("maps_holo_org",WSP_ORG);
5      !setupSectorArtifacts.
6
7  +!setupSectorArtifacts <-
8      ...
9      makeArtifact(PSCONTROLNAME,"artifacts.A_PSControl",[Me,SECTOR_OBJ,NPS],ART
       ↪  _PSCONTROL)[wid(WSP_SECTOR)];
10     focus(ART_PSCONTROL)[wid(WSP_SECTOR)];
11     ...
12     adoptRole(r_head_sector)[artifact_id(ART_GROUP)];
13     adoptRole(r_body_sector)[artifact_id(ART_MAIN_GROUP)];
14     setOwner(Me)[artifact_id(ART_GROUP)];
```

Código 26 – JaCaMo - Agente *Sector* - startSectors

Após a configuração do *Sector*, o agente *PSpace* é inicializado. No Código 27 inicia na linha $2-3$ com a entrada do agente em dois workspaces (workspace da holarquia principal e workspace da holarquia do *Sector* a qual está inserida). Após isso, o *PSpace* passa a desempenhar o papel `r_pspace`.

```
1  +mySector(SECTOR_AGENT)[source(AG)]<-
2          joinWorkspace(WSPNAME,WSP_SECTOR);
3          joinWorkspace("maps_holo_org",WSP_ORG);
4          !setupPSArtifacts;
5      ...
6
7  +!setupPSArtifacts : myGroup(SECTOR_GROUP) <-
8      ...
9          adoptRole(r_pspace)[artifact_id(ART_GROUP)].
```

Código 27 – JaCaMo - Agente *PSpace* - startPSpace

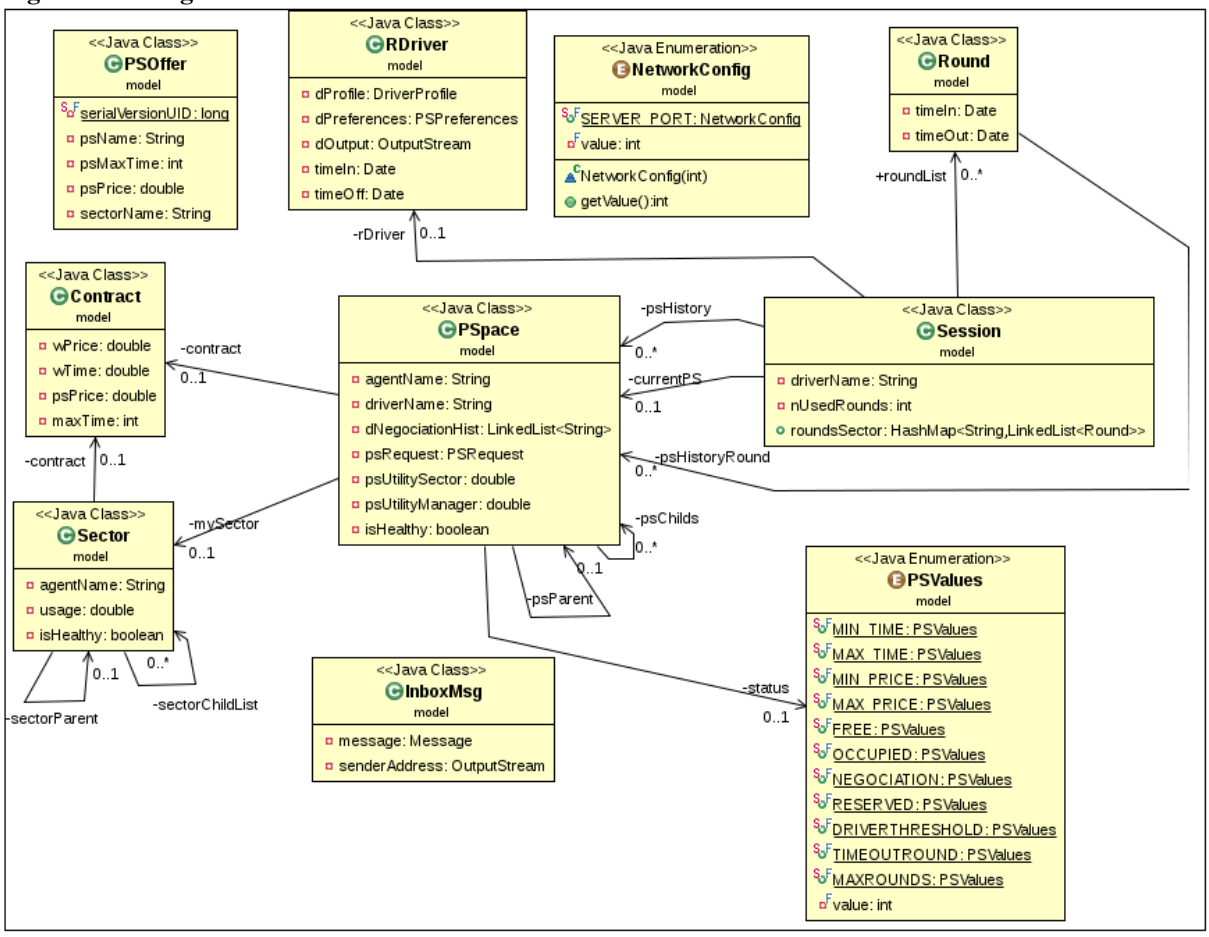
O desenvolvimento da Arquitetura `MAPS-HOLO` foi baseado também no padrão de projeto MVC *(Model-View-Controller)*. Para o atual trabalho, a camada View foi substituita pela camada Artefatos (*Artifacts)*, pois é através dos Artefatos que as classes possuem interação com o restante da Arquitetura. Assim, pode-se sumarizar as camadas desenvolvidas na seguinte maneira:

- **Camada Model:** Classes de abstração das entidades da Arquitetura (`Driver, Sector, PSpace, Round, Contract, InboxMsg, Session.`);

- **Camada Artifact:** Todos os artefatos da Arquitetura;
- **Camada Control:** Classes de controle (`ContractGenerator`,`DBControl` e `T_DriverConnection`).

Com a finalidade de tornar visual as camadas, abaixo é apresentado dois diagramas de classes das camadas **Model e Control** nas Figuras 45 e 47 respectivamente.

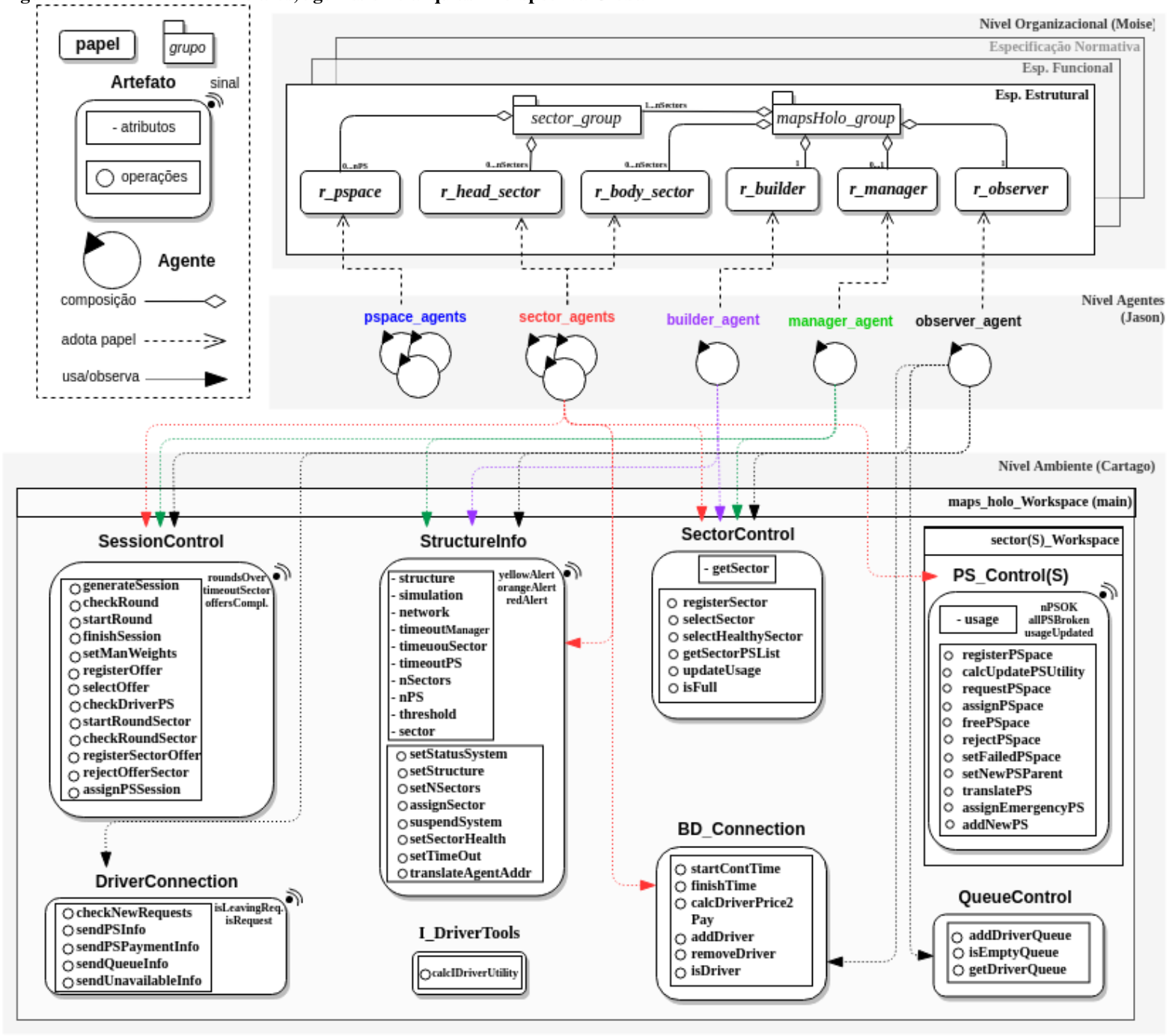
Figura 45 – Diagrama de Classes - Camada Model



Fonte: Autoria Própria

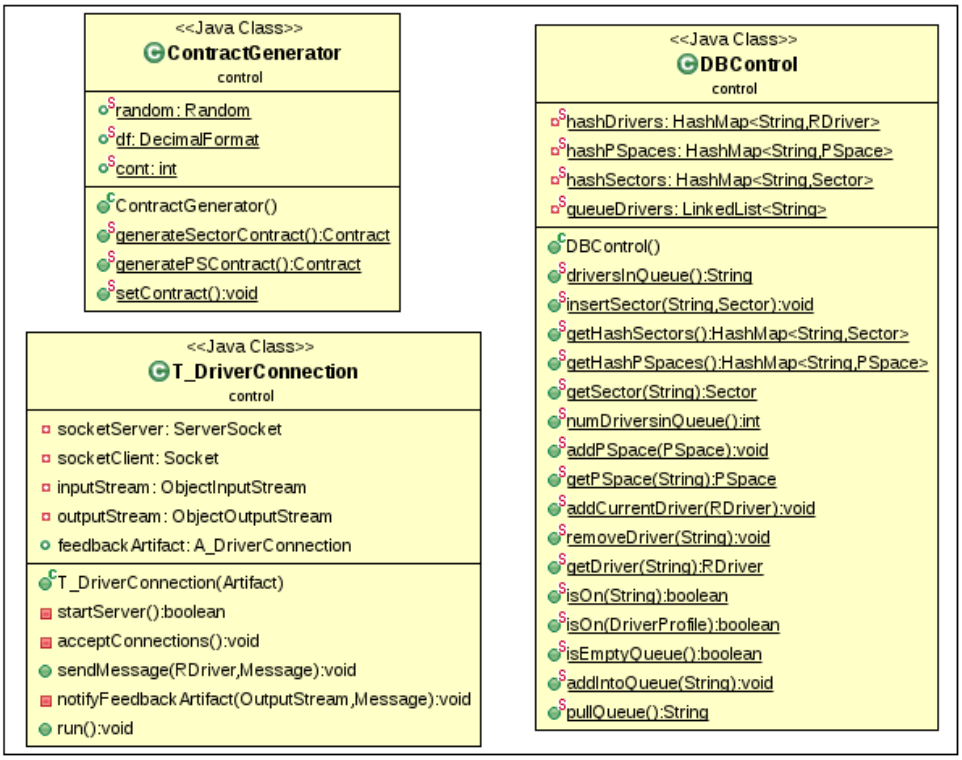
Finalmente, com a finalidade de ilustrar a camada **Artifact** e também a interatividade com os agentes e holarquias, a Figura 46 demonstra todos os artefatos, os agentes iniciais e suas holarquias. Observa-se que a Figura 46 não apresenta os agentes *IDrivers*. Isso ocorre pelo fato de que inicialmente (fase de construção e inicialização) nenhum *IDriver* está presente no sistema.

Figura 46 – MAPS-HOLO - Artefatos, agentes e holarquias - Perspectiva Global



Fonte: Autoria Própria

Figura 47 – Diagrama de Classes - Camada Control



Fonte: Autoria Própria

7.2.4 Agente Observer - Missão m_start

Ao finalizar a etapa de construção de todo o SMAH, o agente *Observer* inicia a missão `m_start` com o objetivo de verificar novas requisições provindas dos usuários do sistema. Note que os usuários nesse momento não são representados pelos *IDrivers*. Os usuários da `MAPS-HOLO` comunicam-se com o sistema via rede através do artefato `A_DriverConnection`, onde é provido um socket TCP para as conexões. Ao receber uma requisição provinda do usuário, o `A_DriverConnection` notifica o *Observer* através de um sinal do Cartago (Vide Figura 46) informa a requisição. As requisições se dividem em duas categorias, sendo elas:

- **isRequest:** Requição para vaga de estacionamento;
- **isLeavingRequest:** Requisição para saída de vaga de estacionamento;

A missão `m_start` compreende o objetivo `checkRequests`, o qual verifica no Artefato de conexão com os usuários se há alguma nova requisição pendente. O Código 28 apresenta a implementação do Agente *Observer* para a checagem das requisições. Por fim, é apresentado no Código 29 o código da função de checagem das requisições do Artefato `A_DriverConnection` no Cartago.

```
1  +!checkRequests <-
2          .print("Waiting for new requests...");
3          checkNewRequests;
4          !!startRequestLoopChecking.
5
6  +!startRequestLoopChecking <-
7          checkNewRequests;
8          .wait(5000); //interval to check new messages
9          !!startRequestLoopChecking.
```

Código 28 – Jason - Agente *Observer* - checkRequests

Na linha 4 do Código é adicionado um segundo objetivo: `!!startRequestLoopChecking`. Esse objetivo tem como função o loop contínuo (intervalo de cinco segundos) para a verificação de novas requisições.

```
1  @OPERATION
2  public void checkNewRequests() {
3      if (this.bufferMessages.size() > 0) {
4          for (InboxMsg inboxMsg : bufferMessages) {
5              switch (inboxMsg.getMessage().getHeader()) {
6              case LEAVING_PS:
7                  driver = (String) inboxMsg.getMessage().getContent();
8                  if (DBControl.isOn(driver)) {
9                      DBControl.getDriver(driver).setdOutput(inboxMsg.getSenderAddress
                       ↪  ());
```

```
10                      signal("isLeavingRequest", driver);
11                  }
12                  break;
13              case REQUEST_PS:
14                  psRequest = (PSRequest) inboxMsg.getMessage().getContent();
15                  if (!DBControl.isOn(psRequest.getdProfile().getDriverName())) {
16                      signal("isRequest", psRequest.getdProfile().getDriverName(),
17                              psRequest.getdPreferences().getPsPrice(),
                                ↪  psRequest.getdPreferences().getPriceWeight(),
18                              psRequest.getdPreferences().getPsTime(),
                                ↪  psRequest.getdPreferences().getTimeWeight(),
19                              inboxMsg.getSenderAddress());
20                  }
21                  break;
22              ...
23          clearBufferMessages();
24      }
25  }}}
```

Código 29 – Cartago - Artefato A_DriverConnection - checkRequests

No código do Artefato `A_DriverConnection` é utilizado um buffer de mensagens (linha 3), devido ao fato de que o *Observer* verifica as requisições a cada cinco segundos, sendo que nesse intervalo mais de uma requisição pode ser recebida. Nas linhas 10 e 16 é onde que o sinal de requisição é enviado ao agente *Observer* .

7.2.5 Agente Observer - Missão m_observer

Essa missão incumbe ao *Observer* a verificação dos agentes *Manager* e *Sectors* a fim de verificar eventuais gargalos ou falhas. A checagem para averiguar as falhas que porventura venham a acontecer, deve-se ao fato de que um agente que tenha falhado não possua mais capacidades computacionais de informar a sua falha. Assim, o *Observer* em um intervalo de tempo T (Configurado no Artefato `StructureInfo`) envia mensagens do tipo *ping* aos agentes. Após o tempo T, o *Observer* verifica se há uma mensagem corresponde ao ping enviado (mensagem *pong*)). Esse procedimento de envio de mensagens *ping* e *pong* como resposta, é comum em ambiente de redes locais onde o *gateway* envia tais mensagens para averiguar o estado dos nós.

Caso após o intervalo T não haja uma mensagem *pong* correspondente ao *ping*, o *Observer* tomará as atitudes necessárias para a correção do SMAH. A partir da seção 7.5 é apresentado os procedimentos tomados pela `MAPS-HOLO` em face das eventuais falhas;

O Código 30 exibe o plano de funcionamento para o envio das mensagens *ping* e posterior checagem das mensagens *pong*.O Código 30 das linhas $1-6$ destina-se ao modo *Head-Body* , e das linhas $8-9$ no modelo *Body-Body* . As linhas $11-17$ realizam o envio das mensagens

ping a todos os setores saudáveis (Que não possuem falhas). A partir da linha 19 até 21 o envio da mensagem *ping* ao *Manager*. Finalmente, das linhas $23 - 40$ é realizada a checagem das mensagens *pong*.

```
1  +!startLoopObserving : obs_hb(TM,TS) <-
2          !sendPingManager;
3          .wait(TM);
4          !sendPingSector;
5          .wait(TS);
6          !checkPong.
7
8  +!startLoopObserving : obs_bb(TS) <-
9          !sendPingSector.
10
11 +!sendPingSector  <-
12         for(sector(S,HEALTH)){
13                 if(HEALTH = true){
14                         .send(S,achieve,ping);
15                         +ping(S);
16                 }
17         }.
18
19 +!sendPingManager<-
20         .send(manager,achieve,ping);
21         +ping(manager).
22
23 +!checkPong <-
24         for(ping(AG)){
25                 if(pong(AG,CODE)){
26                         .print(AG," ok!");
27                         -pong(AG);
28                 }
29                 else{
30                         .print(AG, " failed!!!");
31                         if(AG == "manager"){
32                                 +managerFail;
33                         }
34                         else{
35                                 +sectorFail(AG)
36                         }
37                 }
38                 -ping(AG);
39         };
40         !!startLoopObserving.
```

Código 30 – Jason - Agente *Observer* - Checagem de agentes (Ping e Pong)

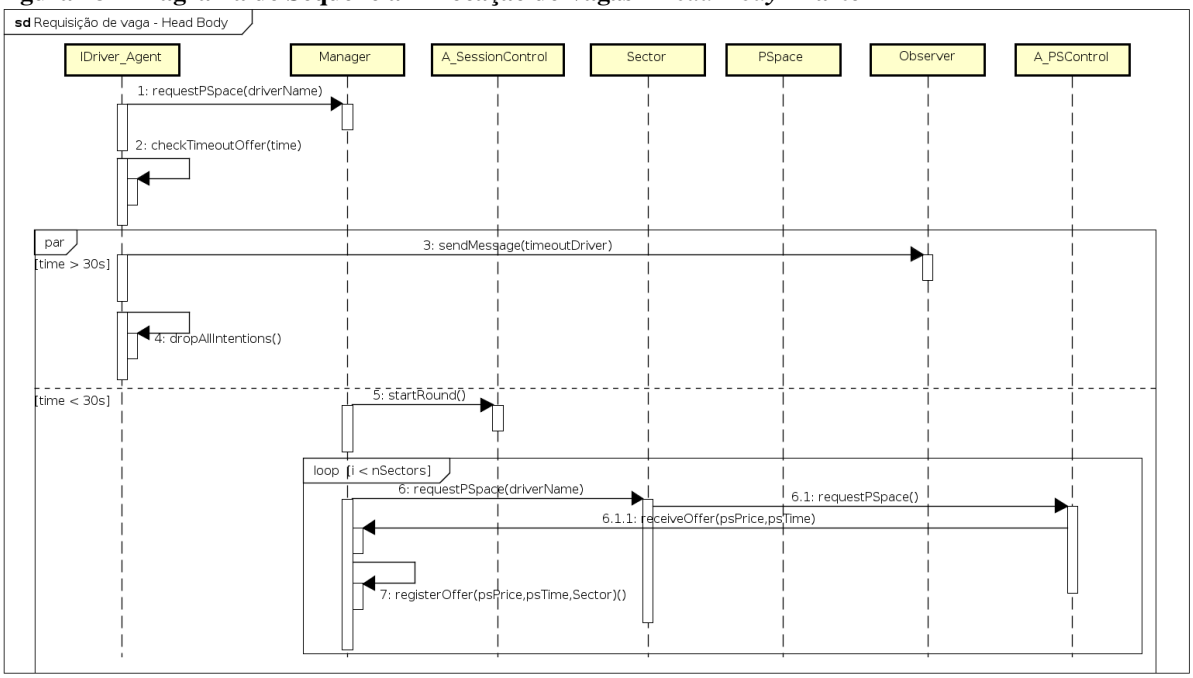
A próxima seção irá demonstrar a utilização dos *IDrivers* no processo de alocação de vagas.

7.3 ALOCAÇÃO DE VAGAS - HEAD-BODY

Essa seção visa a demonstração do processo de alocação das vagas de estacionamento através do modelo *Head-Body* , onde as requisições provindas dos *IDrivers* são enviadas ao *Manager* e este inicia o processo de negociação. Para que seja fácil a compreensão do processo, será exibido nas Figuras 48,49 e 50 um diagrama de sequência divido em três partes ilustrando todas as etapas percorridas pelos agentes para que a negociação pela vaga ocorra. Além dos diagramas, há trechos em que será exibido o código JaCaMo desenvolvido.

7.3.1 Parte 1

Figura 48 – Diagrama de Sequência - Alocação de Vagas - *Head-Body* - Parte 1



Fonte: Autoria Própria

- **1: requestPSpace(driverName):** *IDriver* inicia o processo de negociação ao enviar uma mensagem ao *Manager* requisitando uma vaga;
- **2: checktimeOutOffer(time):** *IDriver* após enviar a requisição de vaga, inicia a contagem do tempo para conseguir uma vaga de estacionamento;
- **3: sendMessageTimeOutDriver:** Caso tenha passado um intervalo T de tempo (Implementação atual: 30 segundos), o *IDriver* recebe mensagem de timeout;
- **4: dropAllIntentions:** Em consequência do passo anterior, o agente *IDriver* encerra todas as suas intenções;

- **5: startRound:** Um novo round de negociação é inicializado;
- **6: requestPSpace(driverName):** *Manager* após receber a requisição de vaga, realiza um *broadcast* com todos os setores a fim de requisitar uma vaga de estacionamento. Os agentes *Sectors* recebem a requisição e selecionam um *PSpace* de acordo com o valor da função utilidade. Após isso, é enviado como resposta ao *Manager* uma oferta de vaga;
- **7: registerOffer(psPrice,psTime,Sector):** O *Manager* recebe a oferta e armazena em seu *buffer* de ofertas;

O Código 31 apresenta o código fonte dos passos 1 à 4.

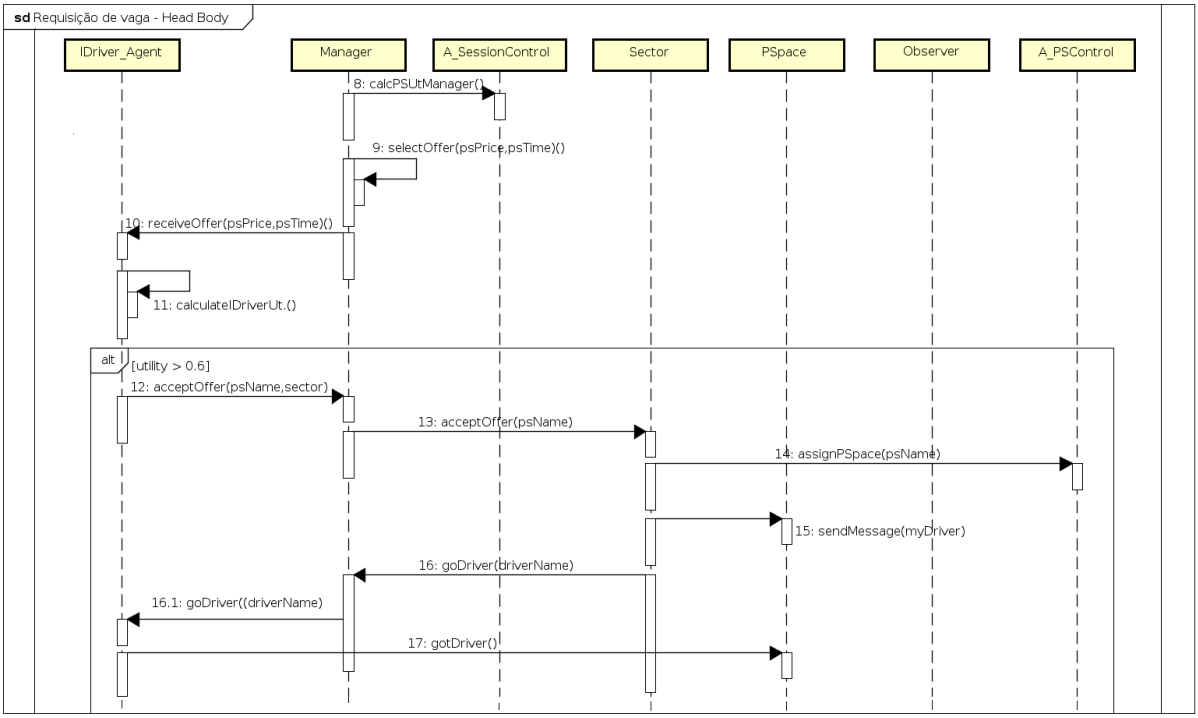
```
1 +!requestPSpace(STR) : STR = "HB" & getInfoDriver(_,_,_,_,DN) <-
2         .send(manager,achieve,requestPSpace(DN));
3         !!checkTimeoutOffer.
4     ...
5 +!checkTimeoutOffer : timeoutOffer(TO)<-
6         .wait(TO);
7         if(not gotPS){
8         .send(observer,tell,timeoutDriver);
9         .drop_all_intentions}.
```

Código 31 – JaCaMo - Agente *IDriver* - requestPSpace

7.3.2 Parte 2

- **8: calculatePSUtilityManager:** O *Sector* durante a fase de construção da `MAPS-HOLO` previamente definiu os valores de utilidade para as vagas de estacionamento. Contudo, no modo *head-body* , é realizado uma segunda seleção para o envio de uma vaga para o *IDriver* . Assim, ao passo que o *Manager* envia a requisição de vaga de estacionamento ao *Sector* e esse seleciona uma *PSpace* de acordo com a sua função utilidade (Previamente definida no Contrato), o *Manager* com base nas ofertas recebidas dos *Sectors*, realiza um segundo filtro nas vagas de estacionamento. Esse segundo filtro pode ser realizado por diferentes motivações, por exemplo: Todos os *sectors* possuem uma alta preferência pelo alto valor das vagas, ao passo que o *IDriver* pode possuir uma alta demanda por valores baixos, o que acarreta problemas para as negociações. Ou por outro lado, o *Manager* pode inferir nas negociações a fim de maximizar ainda mais o lucro global de todo o sistema. Assim, nesse modelo há dois cálculos de função utilidade com diferentes pesos para as preferências, podendo preferenciar o *IDriver* ou o próprio sistema. Na atual implementação, optou-se por pesos equilibrados (0.5 - preço / 0.5 - tempo de utilização).
- **9: selectOffer(psPrice,psTime):** Conforme apresentado no passo 8 :, o *Manager* realiza uma seleção no *buffer* das ofertas um agente *PSpace* a ser ofertado;

Figura 49 – Diagrama de Sequência - Alocação de Vagas - *Head-Body* - Parte 2



Fonte: Autoria Própria

- **10: receiveOffer(psPrice,psTime):** O *IDriver* recebe a oferta selecionada pelo *Manager* ;
- **11: calculateIDriverUtility:** *IDriver* avalia de acordo com a sua função utilidade a satisfação da oferta;
- **12: acceptOffer(psName,sector):** Caso a satisfação seja atingida (threshold), o *IDriver* aceita a oferta;
- **13: acceptOffer(psName,sector):** O *Manager* informa ao *Sector* que o *IDriver* aceitou a oferta;
- **14: assignPSpace(psName):** O *Sector* aloca a vaga de estacionamento;
- **15: sendMessage(myDriver):** O *Sector* notifica o agente *PSpace* a respeito da alocação e informa qual será o seu *IDriver* ;
- **16: goDriver(driverName):** O *Sector* informa ao *Manager* que a vaga está liberada. O *Manager* informa ao *IDriver* a localidade da vaga e o autoriza a dirigir-se até a vaga;
- **17: gotDriver:** O *IDriver* notifica a chegada na vaga de estacionamento ao *PSpace* ;

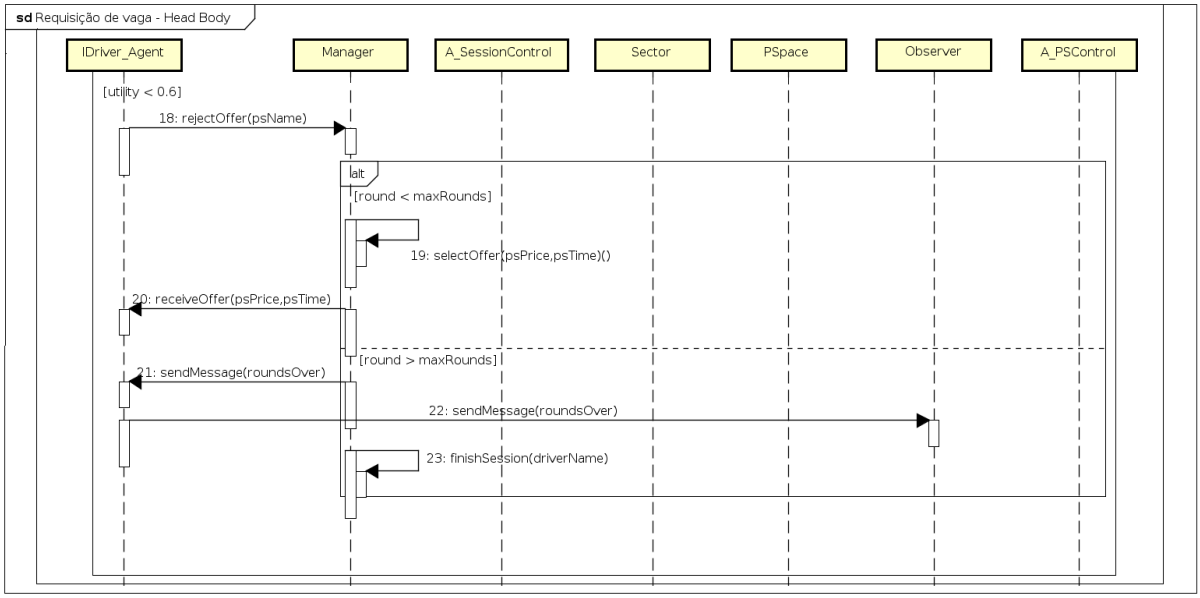
O Código 32 apresenta o código do *IDriver* nos passos 10 à 12. Destaca-se as linhas $7 - 15$, onde o *IDriver* verifica se aceita ou não oferta recebida. Nota-se que esse plano `+!receiveOffer` só é executado se o *IDriver* não aceitou nenhuma vaga e seus rounds não finalizam(`not gotPS & not roundsOver`). A atual implementação do `MAPS-HOLO` considerou o valor de `THRESHOLD` de **0.6** (60% de satisfação).

```
1  +!receiveOffer(PSNAME,PSPRICE,PSTIME,SECTORNAME)[source(AG)] : not gotPS & not
   ↪  roundsOver <-
2          +receivedOffer;
3          ?getInfoDriver(DP,DPW,DT,DTW,DN);
4          ?threshold(THRESHOLD);
5          .my_name(Me);
6          calculateIDriverUtility(DPW,DTW,DP,DT,PSPRICE,PSTIME,IDUTILITY);
7          if(IDUTILITY >= THRESHOLD){
8                  +gotPS;
9                  .print("I will accept the offer! (",PSNAME,") - UT: ",IDUTILITY);
10                 .send(AG,achieve,acceptOffer(PSNAME,SECTORNAME));
11         }
12         else{
13                 .print("I will reject the offer! (",PSNAME,") - UT: ",IDUTILITY);
14                 .send(AG,achieve,rejectOffer(PSNAME,SECTORNAME));
15         }
16 .
```

Código 32 – JaCaMo - Agente *IDriver* - receiveOffer

7.3.3 Parte 3

Figura 50 – Diagrama de Sequência - Alocação de Vagas - *Head-Body* - Parte 3



Fonte: Autoria Própria

- **18: rejectOffer(psName):** Caso o valor de `THRESHOLD` seja inferior a 0.6, o *IDriver* rejeita a oferta;

- **19: selectOffer(psPrice,psTime):** O *Manager* seleciona uma nova oferta em seu *buffer* de ofertas. Caso o *buffer* esteja vazio, o *Manager* repete o passo 6.
- **20: receiveOffer(psPrice,psTime):** O *Manager* envia então uma nova oferta ao *IDriver* . O *IDriver* repete o passo 11, pois é necessário avaliar a oferta;
- **21: sendMessage(roundsOver):** Caso o *IDriver* rejeite uma quantidade de ofertas maior que o número de rounds permitidos, a negociação é encerrada. Assim, o *Manager* notifica o *IDriver* que a negociação foi encerrada;
- **22: sendMessage(roundsOver):** *Manager* notifica o *Observer* para que ele notifique o usuário via rede (via `A_DriverConnection`) informando que a negociação foi encerrada;
- **23: finishSession(driverName):** Com a negociação encerrada, a sessão do *IDriver* é finalizada.

7.4 ALOCAÇÃO DE VAGAS - BODY-BODY

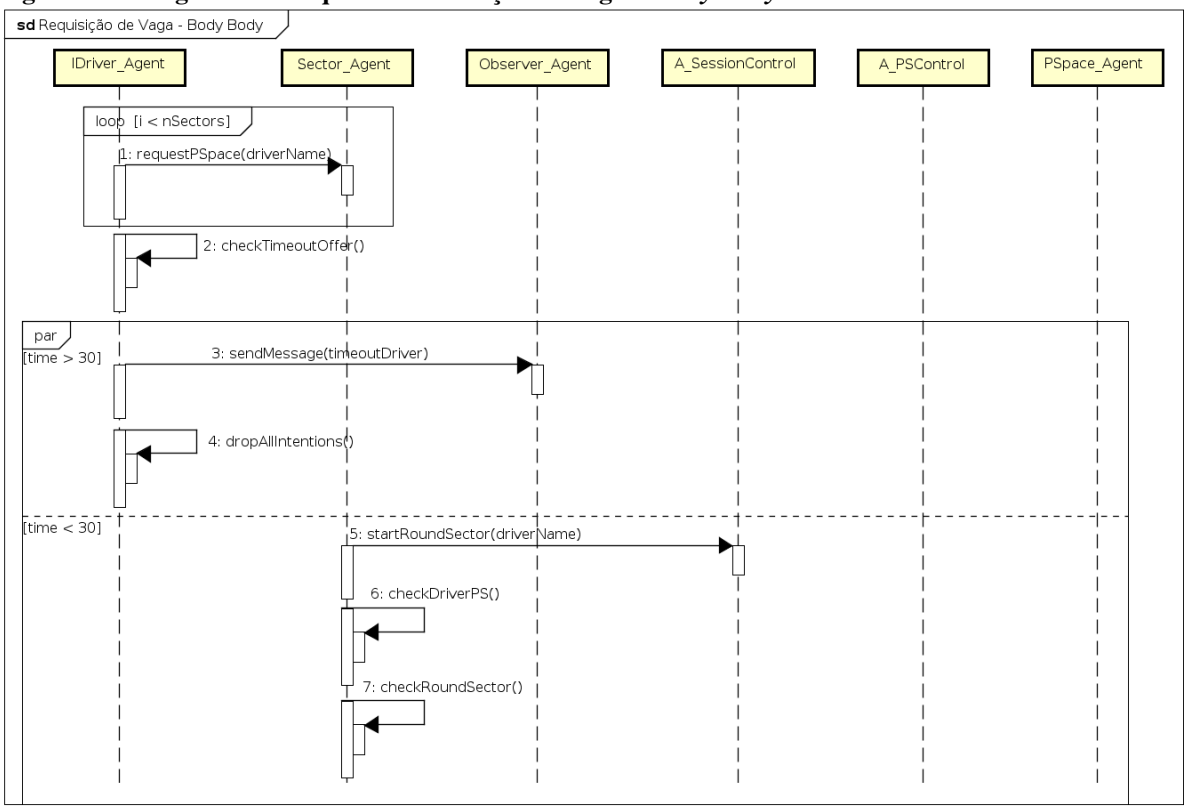
Diferente do modelo *Head-Body* , o *Body-Body* proporciona ao *IDriver* o acesso direto aos *Sectors* para a negociação sem a utilização de um agente central. As Figuras 51 e 52 apresentam um diagrama de sequência dividido em duas partes para tornar a visualização da interação dos agentes e artefatos mais clara.

7.4.1 Parte 1

- **1: requestPSpace(driverName):** *IDriver* envia a requisição de vaga a todos os *Sectors*.
- **2: checktimeOutOffer(time):** *IDriver* após enviar a requisição de vaga, inicia a contagem do tempo para conseguir uma vaga de estacionamento;
- **3: sendMessage(timeoutDriver):** Caso atinja o valor de timeout, o *IDriver* notifica o *Observer* sobre o seu timeout;
- **4: dropAllIntentions:** Cancela todas as intenções;
- **5: startRoundSector(driverName):** Após o *Sector* receber a requisição por vaga, ele inicia um novo round particular para a negociação;
- **6: checkDriverPS:** *Sector* checa se o *IDriver* já não obeteve sucesso em uma negociação com outro *Sector* ;

- **7: checkRoundSector:** *Sector* verifica se a quantidade de rounds utilizados na negociação já não ultrapassou os rounds permitidos.

Figura 51 – Diagrama de Sequência - Alocação de Vagas - *Body-Body* - Parte 1



Fonte: Autoria Própria

O Código 33 apresenta a codificação *IDriver* do passo 1. Na linha 3 é realizado um laço de repetição para todos os setores sem falhas no sistema.

```
1 +!requestPSpace(STR) : STR = "BB" & getInfoDriver(_,_,_,_,DN) <-
2     //DN = driverName
3         for(sector(S,HEALTH)){
4                 if(HEALTH == true){
5                         .send(S,achieve,requestPSpace(DN));
6                 };
7         }.
```

Código 33 – Jason - Agente *IDriver* - requestPSpace (*Body-Body*)

O código 34 apresenta um trecho do código do artefato `A_SessionControl` onde verifica se o *IDriver* já conseguiu uma vaga na negociação corrente. Caso o *IDriver* ainda não possua nenhuma vaga alocada, o Setor do passo 6 envia a oferta de vaga.

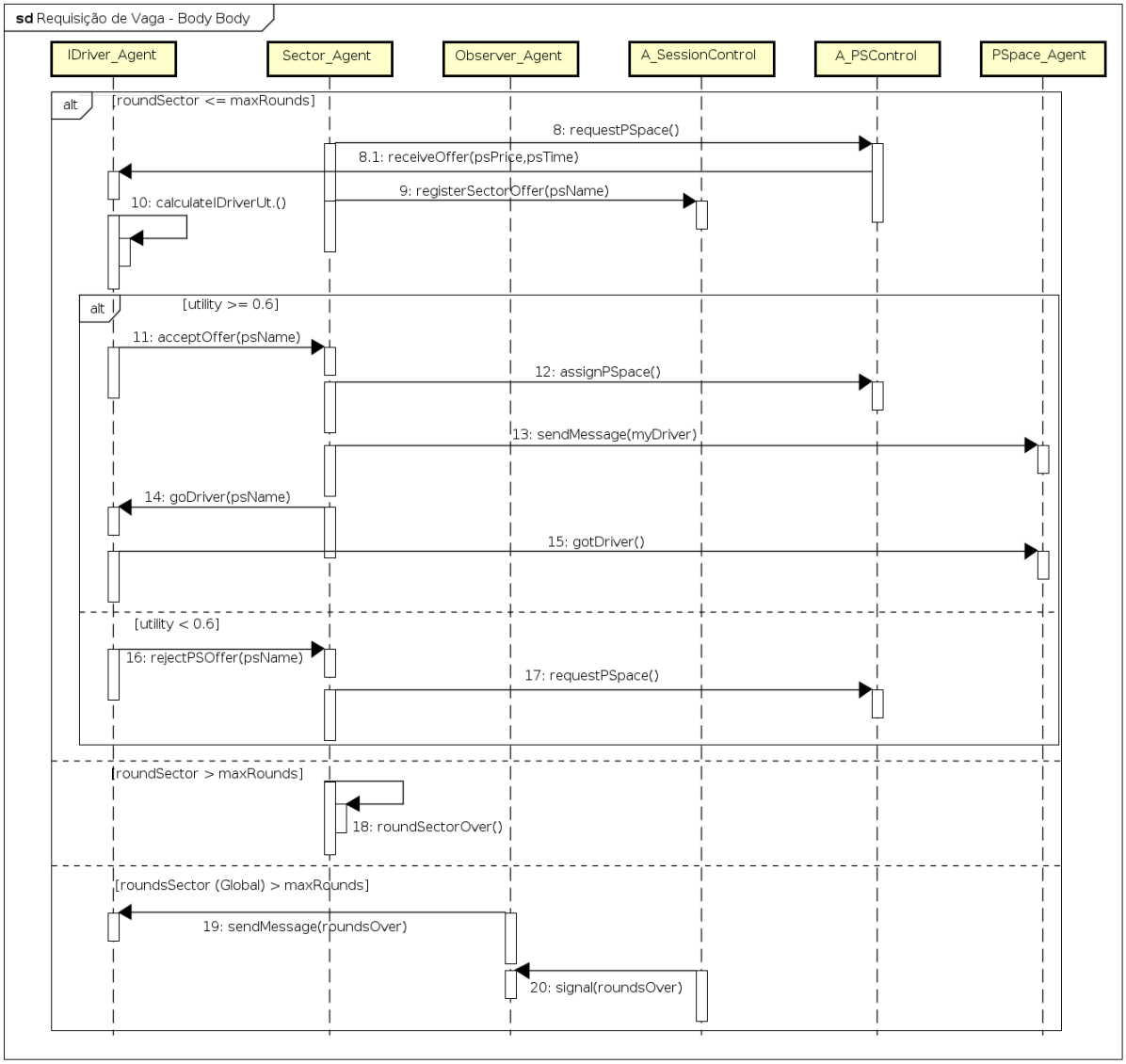
```
1 @OPERATION
2 public void checkDriverPS(String dName, OpFeedbackParam<Boolean> isPSDriver){
```

```
3       isPSDriver.set(hashSessions.get(dName).getCurrentPS() == null ? false : true);
4   }
```

Código 34 – Cartago - Artefato A_SessionControl - checkDriverPS(*Body-Body*)

7.4.2 Parte 2

Figura 52 – Diagrama de Sequência - Alocação de Vagas - *Body-Body* - Parte 2



Fonte: Autoria Própria

- **8: requestPSpace:** O *Sector* solicita ao artefato `A_PSControl` uma vaga de estacionamento disponível. Caso positivo, envia ao *IDriver* uma oferta de vaga.
- **9: registerSectorOffer(psName):** A oferta feita no passo anterior deve ser registrada no `A_SessionControl` para seja armazenada no histórico de negociações desse *IDriver* ;

- **10: calculateIDriverUtility:** *IDriver* avalia de acordo com a sua função utilidade a satisfação da oferta;
- **11: acceptOffer(psName):** Caso a satisfação seja atingida (threshold), o *IDriver* aceita a oferta;
- **12: assignPSpace:** O *Sector* aloca a vaga de estacionamento;
- **13: sendMessage(myDriver):** O *Sector* envia mensagem ao *PSpace* correspondente a vaga de estacionamento com as informações do *IDriver* (myDriver);
- **14: goDriver(psName):** O *Sector* autoriza o deslocamento do *IDriver* para a localidade da vaga de estacionamento;
- **15: gotDriver:** *IDriver* informa sua chegada na vaga de estacionamento;
- **16: rejectPSOffer(psName):** Caso a satisfação não seja atingida (threshold), o *IDriver* rejeita a oferta;
- **17: requestPSpace:** Conforme o passo 16, o *Sector* repete o passo 8;
- **18: roundSectorOver:** Caso o número de rounds do *Sector* exceda o valor máximo de rounds, o *Sector* encerra a sua negociação com o *IDriver* . Aqui observa-se um ponto importante de diferença com o modelo *Head-Body* . No modelo com a figura do *Manager* havia uma única via de rounds, um número mais limitado devido a visibilidade do sistema pelo *IDriver* , pois a negociação era de um para um. Agora pois no modelo *Body-Body* a negociação é de um para muitos. Assim, há uma quantidade maior de rounds;
- **19: sendMessage(roundsOver):** Contudo, mesmo nesse modelo os rounds podem ser todos excedidos. Ao passo que todos os *Sectors* encerram seus respectivos rounds, a negociação com o *IDriver* é encerrada;
- **20: signal(roundsOver):** *IDriver* recebe o sinal de que a sua negociação com todos os *Sectors* foi encerrada.

O Código 35 apresenta a codificação do artefato `A_PSControl` para o passo 8. Destaca-se as linhas $5 - 6$, onde na linha 6 é selecionado vagas de estacionamento que estejam livres e que não foram ofertadas ainda ao *IDriver* corrente. Por fim na linha 11 a vaga é previamente alocada, pois é configurada como estando em negociação devido ao fato de que o *IDriver* poderá aceitá-la. Caso contrário, a vaga tornará a estar a livre.

```
1 @OPERATION
2 public void requestPSpace(String dName, OpFeedbackParam<Boolean> isPS,
  ↪  OpFeedbackParam<String> psName,
3       OpFeedbackParam<Double> psPrice, OpFeedbackParam<Integer> psTime) {
```

```
4       isPS.set(false);
5       for (PSpace ps : psList) {
6           if (ps.getStatus().equals(PSValues.FREE) &&
            ↪  !ps.isOnHistoryNegociation(dName)) {
7               psName.set(ps.getAgentName());
8               psPrice.set(ps.getContract().getPsPrice());
9               psTime.set(ps.getContract().getMaxTime());
10              ps.setDriverName(dName);
11              ps.setStatus(PSValues.NEGOCIATION);
12              ps.insertDriverInNegociation(dName);
13              isPS.set(true);
14              nUsedPS++;
15              updateUsage();
16              break;
17          }
18      }
19  }
```

Código 35 – Cartago - Artefato A_PSControl - requestPSpace(*Body-Body*)

7.5 FALHAS DOS AGENTES

Uma das características de um SMAH é a possibilidade de aninhar agentes dentro de outros agentes, sendo um agente adotado por outro. A terminologia usada nas seções ao invés do termo aninhado, será **agente pai e filho**. O agente **filho** estará aninhado no agente **pai**. Uma vez que os agentes BDI são compostos de crenças, desejos e intenções, um agente que falhou pode não ter mais capacidades para que possa atingir seus desejos por meio de suas intenções. Uma forma equivocada de solucionar a falha dos agentes seria o simples fato de aninhar um agente que tenha falhado dentro de outro saudável, pois o agente falho pode não haver nem mesmo condições computacionais de responder aos estímulos do seu agente pai.

A estratégia adotada na `MAPS-HOLO` foi modificada da premissa original do conceito de aninhamento completo dos agentes, pois essa premissa não tem como implicação principal o fator de agentes falhos. Assim, a alternativa usada e testada na Arquitetura foi o aninhamento parcial dos agentes. Esse aninhamento parcial não engloba todas as características (crenças, desejos e intenções) do agente dentro de outro, ou em outras palavras todas as características do filho dentro do agente pai. No aninhamento parcial da `MAPS-HOLO` somente as crenças são aninhadas, assim todas as informações do ambiente e do próprio agente filho está no agente pai. A motivação de não transferir os desejos e intenções se deve ao fato de que ambos agentes (pai e filho) possuem as mesmas intenções e desejos. Assim, torna-se desnecessária a transferência de todo o agente, mas somente o essencial para que ainda sim o agente filho possa ser visto pelo sistema como um agente real e em funcionamento e não um agente que tenha falhado.

Isso possui implicações diretas no sistema, uma vez que um agente tenha falhado o SMAH não o enxerga como falho, mas sim como existente e em funcionamento. Novamente isso gera um outro problema no sistema, o endereçamento. Um agente falho visto como não-falho pelo SMAH, não responderá aos estímulos e invocações do sistema. Assim, faz-se necessário que o agente pai do agente filho responda os estímulos destinados ao filho. Por fim, para solucionar essa questão foi desenvolvido um método de tradução de endereços dos agentes filho para os seus respectivos pais. A seguir é descrito como a `MAPS-HOLO` trabalha em face das falhas que eventualmente podem ocorrer.

7.5.1 Alertas

Em virtude das diferentes falhas que podem aparecer no sistema e dos seus diferentes impactos no SMAH, foi desenvolvido três alertas que são emitos pelo artefato `A_StructureInfo`.

- ***Yellow alert:*** Falha pontual no sistema onde um alerta é emitido. Um exemplo dessa falha é quando o *Manager* falha e a estrutura do sistema necessita de mudança;
- ***Orange alert:*** Falha média onde a utilização de agentes auxiliares será utilizada;
- ***Red alert:*** Falha grave e o sistema é suspenso para novas requisições;

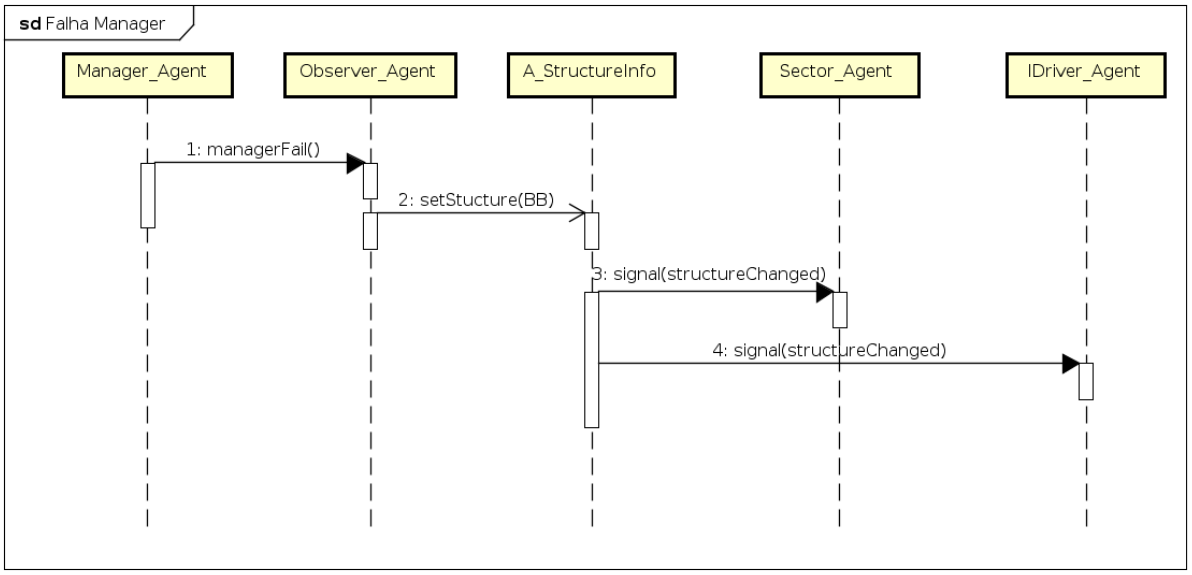
Uma falha na `MAPS-HOLO` não implica necessariamente na falha física ou lógica do agente, mas também na sua sobrecarga. As falhas podem ser detectadas de duas formas:

- Verificada pelo próprio agente: O agente verifica seu estado de falha e notifica o *Observer* a respeito da sua falha para que ele tome as devidas providências. O *PSpace* ao falhar notifica o *Sector* sobre sua falha, pois o mesmo não tem acesso direto ao *Observer* ;
- Verificada pelo agente *Observer*: Verifica periódicamente os agentes por meio de mensagens *ping*;

7.5.2 Agente Manager

A falha do agente *Manager* requer que o sistema altere sua estrutura de organização para o modelo *head-body* . Ao perceber a falha, o *Observer* altera estrutura do sistema mediante o artefato `StructureInfo`. Ao ser alterado, o artefato emite um sinal (`structureChanged`), assim os agentes correntes alteram seu estado mental interno para se adaptar ao novo modelo. A Figura 53 apresenta um diagrama de sequência, o qual ilustra as etapas requeridas para essa mudança.

Figura 53 – Diagrama de Sequência - Falha Agente *Manager*



Fonte: Autoria Própria

- **1: managerFail:** *Manager* notifica *Observer* sobre sua falha;
- **2: setStructure(BB):** *Observer* altera a estrutura do sistema para *body-body* ;
- **3: signal(structureChanged):** *Sectors* alteram seu estado mental para *body-body* ;
- **4: signal(structureChanged):** *IDrivers* que estão em fase de negociação são notificados e reiniciam a sua negociação no modelo *body-body* .

O Código 36 apresenta o código fonte do *IDriver* ao ser notificado da mudança de estrutura.

```
1 +structureChanged : structure(ST) & not gotPS <-
2         .print("Structure changed... Changing my mind!");
3         .drop_all_intentions;
4         !requestPSpace(ST).
```

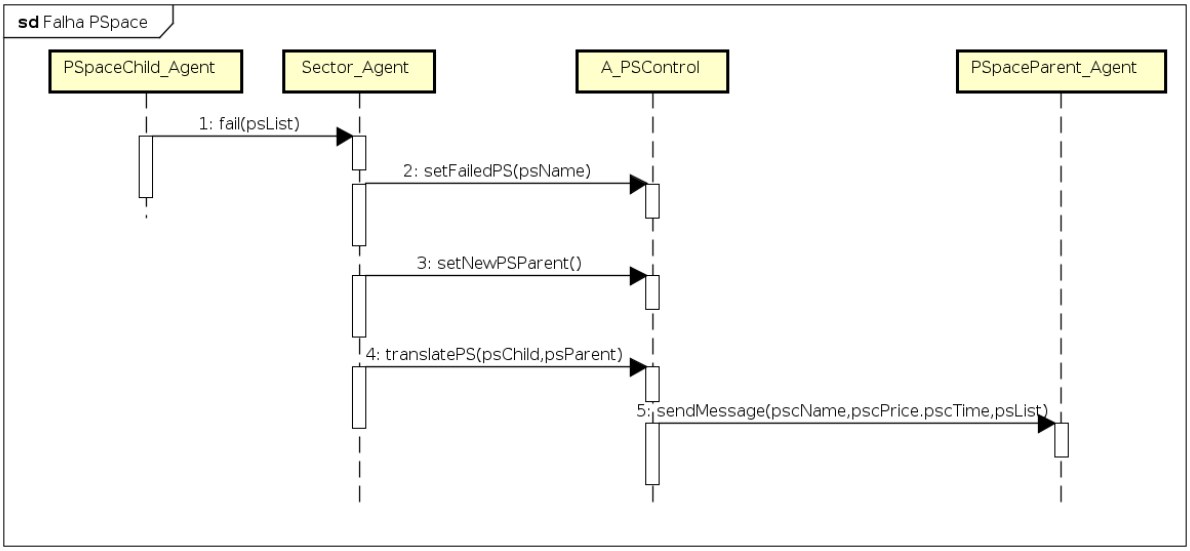
Código 36 – Jason - Agente *IDriver* - signal(structureChanged) (*Head-Body*)

7.5.3 Falha: Agente PSpace

Ao falhar, o *PSpace* torna-se um agente filho e o seu *Sector* seleciona um agente pai. A Figura 54 apresenta as etapas seguidas pela `MAPS-HOLO` para solucionar a falha de um agente *PSpace* . Durante a falha, o Esquema Social `psFail_Scheme` é utilizado (Figura 55)

- **1: fail(psList):** *PSpace* falho (filho) notifica o seu *Sector* a respeito da falha. Observa-se que um agente filho ao falhar pode possuir agentes que previamente já falharam. Assim

Figura 54 – Diagrama de Sequência - Falha *PSpace*



Fonte: Autoria Própria

através da variável psList é enviado ao agente pai os demais agentes filho (agentes neto) que estavam aninhados no agente falho corrente;

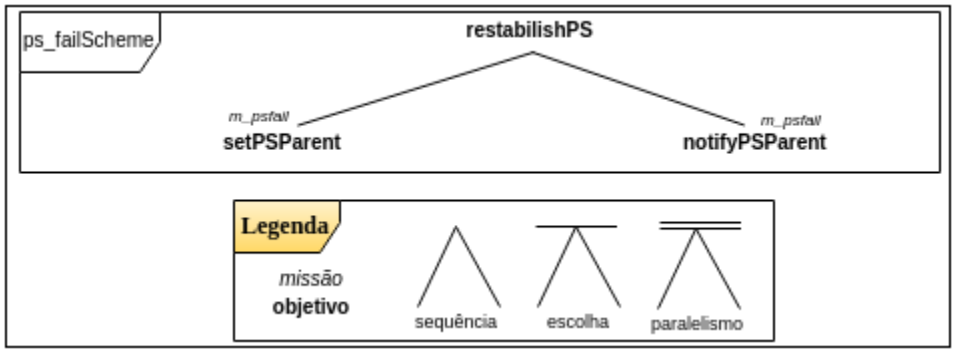
- **2: setFailed(psName):** O *Sector* altera o estado da vaga para falho;
- **3: setNewPSParent:** *Sector* escolhe um outro *PSpace* para ser o agente pai do *PSpace* . Assim, é alterado o endereço do agente filho para o agente pai;
- **4: translatePS(psChild,psParent):** Executa a tradução do endereço do filho a fim de verificar se a mudança de endereço está correta;
- **5: sendMessage(psName,pscPrice,pscTime,psList):** Envia mensagem ao agente *PSpace* pai com as informações (crenças) do agente filho. Nessa mensagem também é enviado caso existam os demais filhos do novo agente filho (agentes neto). Nota-se que pode haver n-níveis de recursão entre os agentes (pai,filho e filhos (netos)), tornando-se um problema para o controle desses agentes. Assim, ao passo que um agente filho é adotado por um agente pai, os filhos do agente filho (caso existam (netos)) serão repassados não como agentes neto, mas sim serão adotados como agentes filho no mesmo nível do agente filho.

O Código 37 apresenta a codificação do artefato `A_PSControl` nos passos **3 e 4**. Na linha 3 é apresentado a função `findChildLessPS`, sendo essa responsável por encontrar um *PSpace* com a menor quantidade de filhos estabelecidos. Assim, caso encontrado, esse *PSpace* será o agente pai (`psParent`) do agente filho (`psChild`). Na linha 4 é realizado uma checagem, onde caso seja `null`, significa que nenhum agente *PSpace* está disponível e que todos falharam. Sendo assim, é emitido um alerta a respeito desse *Sector* , pois todos os seus *PSpaces* falharam. A próxima sub-seção irá apresentar como a `MAPS-HOLO` lida com essa situação. Por fim, nas linhas $20 - 26$ é realizada a tradução do endereço do agente filho para o agente pai.

```
1  @OPERATION
2          public void setNewPSParent(String psChildName, OpFeedbackParam<String>
           ↪  psParentName) {
3                  PSpace psChild = getPSpace(psChildName);
4                  PSpace psParent = findChildLessPS(psChildName);
5                  if (psParent == null) {
6                          signal("all_PSBroken");
7                  } else {
8                          if (!psChild.getPsChilds().isEmpty()) {
9                                  for (PSpace psgrandchild : psChild.getPsChilds()) {
10                                         psgrandchild.setPsParent(psParent);
11                                         psParent.addChild(psgrandchild);
12                                 }
13                         }
14                         psChild.setPsParent(psParent);
15                         psParent.addChild(psChild);
16                         psParentName.set(psParent.getAgentName());
17                 }
18         }
19         @OPERATION
20         public void translatePS(String psChildName, OpFeedbackParam<String>
           ↪  psParentName) {
21                 PSpace ps = getPSpace(psChildName);
22                 if(ps.getPsParent() != null)
23                         psParentName.set(ps.getPsParent().getAgentName());
24                         else
25                                 psParentName.set(ps.getAgentName());
26     }
```

Código 37 – Cartago - Artefato A_PSControl - setNewPSParent e translatePS

Figura 55 – Moise - Esquema Social - PSFail_Scheme



Fonte: Autoria Própria

Após a tradução dos endereços, o agente pai passa a responder pelo agente filho. A seguir, no Código 38 é dois planos do agente pai. Na linha 1 é o plano invocado pelo *Sector* para liberar a vaga de estacionamento, contudo, esse plano é em resposta e pelo agente filho. A partir da linha 5 o agente pai é informado de qual ou quais agentes será pai. O processo de armazenagem das crenças dos agentes filhos no pai dá-se pela implementação das listas em Jason.

```
+!leavePSpace(DRIVERNAME,PS_NAME)[source(AG)] : psChildList(PSLIST) &
    .term2string(PS_NAME_T,PS_NAME) & .member(PS_NAME_T,PSLIST) <-
        .print(PS_NAME,": Getting free!");
        -myDriver(DRIVERNAME,PS_NAME).

+psChild(PSC_NAME,PSC_PRICE,PSC_TIME,PSC_LIST) <-
        .print("I manage the ",PSC_NAME," now!");
        .findall(P,psChild(P,_,_,_),PSLIST);

        if(psChildList(L)){
                -psChildList(L);
        }

        if(.list(PSC_LIST)){
                .union(PSC_LIST,PSLIST,NEWLIST);
                +psChildList(NEWLIST);
        }
        else{
                +psChildList(PSLIST);
        }.
```

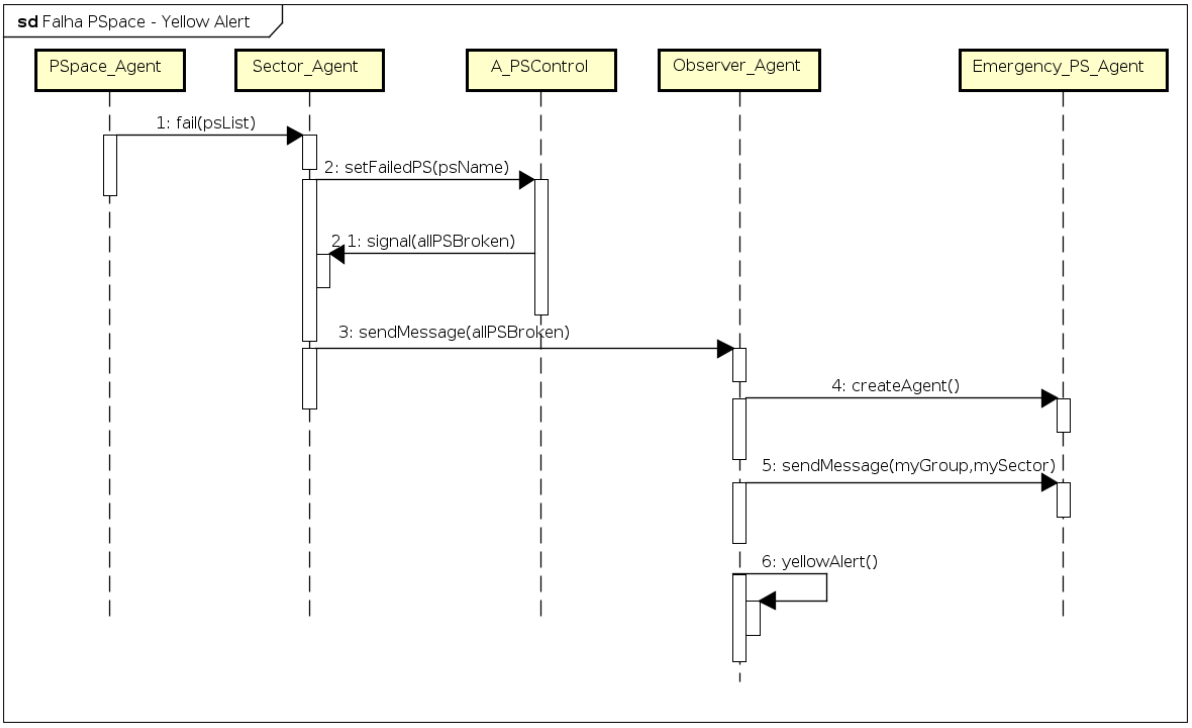
Código 38 – Jason - Agente *PSpace* - Agente pai em resposta pelo filho

7.5.4 Falha: Todos os agentes PSspaces

Ao passo que todos os agentes *PSpaces* de uma holarquia vem a falhar, isso demonstra que há um problema nessa holarquia e em seu agente *head* , o *Sector* . Assim, a medida adotada pela MAPS-HOLO para solucionar essa falha é criação de um agente *PSpace* de emergência. Uma vez que todos os *PSpaces* falharam, essa holarquia não aceitará novas requisições para vaga, mas continuará somente a administrar as alocações correntes (vagas de estacionamento alocadas aos *IDrivers*). Para isso, o agente de emergência será considerado o agente pai de todos os agentes *PSpaces* dessa holarquia. A Figura 56 ilustra um diagrama de sequência com as etapas tomadas pelo sistema diante da falha de todos os *PSpaces* de uma holarquia.

- **1: fail(psList):** Agente *PSpace* declara estado de falha;
- **2: setFailedPS(psName):** *Sector* atribui falha ao agente *PSpace* . Contudo, todos os demas *PSpaces* dessa holarquia falharam. Assim, o sinal (allPSBroken) é gerado;
- **3: sendMessage(allPSBroken):** Diante do sinal gerado, o *Sector* manda uma mensagem ao *Observer* para notificá-lo de que todos os *PSpaces* da holarquia do *Sector* falharam. A holarquia fica fechada para novas requisições;

Figura 56 – Diagrama de Sequência - Falha de todos os *PSpaces* de uma holarquia



Fonte: Autoria Própria

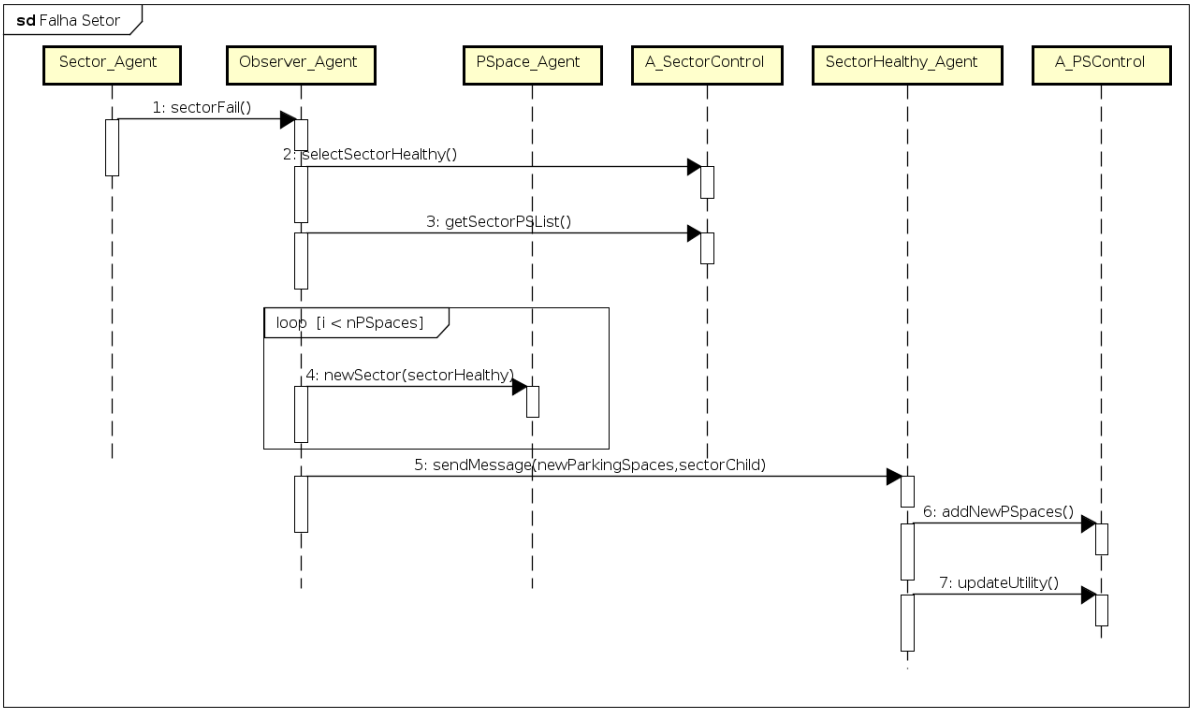
- **4: createAgent :** *Observer* cria um agente de emergência para que assuma as alocações previamente estabelecidas na holarquia;
- **5: sendMessage(myGroup,mySector):** *Observer* notifica ao agente de emergência em qual holarquia irá atuar;
- **6: yellowAlert:** Sistema declara *Yellow Alert*.

7.5.5 Falha: Agente Setor

O agente *Sector* ao falhar possui um procedimento diferente de quando o agente *PSpace* falha. Ao invés de adotar o aninhamento dos agentes, na falha do *Sector* , todos os seus agentes *PSpaces* da sua holarquia são transferidos para outra holarquia de modo que sejam submissos a outro agente *Sector* . Para que isso ocorra é necessário notificar a nova holarquia que irá receber os agentes *PSpaces* , bem como todos os agentes *PSpaces* que serão transferidos. A Figura 57 ilustra as etapas desse processo mediante um diagrama de sequência.

- **1: sectorFail:** *Sector* notifica o *Observer* sobre a sua falha;
- **2: selectSectorHealthy:** *Observer* seleciona outro *Sector* que não possua falhas;
- **3: getSectorPSList:** *Observer* recupera a lista dos *PSpaces* que estão na holarquia do *Sector* que falhou;

Figura 57 – Diagrama de Sequência - Falha do agente *Sector*



Fonte: Autoria Própria

- **4: newSector(sectorHealthy:)** Agente *Observer* notifica todos os *PSpaces* sobre o seu novo *Sector* ;
- **5: sendMessage(newParkingSpaces,sectorChild):** *Observer* envia mensagem ao novo *Sector* com a lista os *PSpaces* do *Sector* que falhou;
- **6: addNewParkingSpaces:** Agente *Sector* novo adiciona no seu artefato `A_PSControl` as informações dos seus novos *PSpaces* ;
- **7: updateUtility:** Agente *Sector* novo atualiza os valores da função utilidade baseado no seu contrato para as *PSpaces* do antigo *Sector* .

O Código 39 apresenta a codificação dos passos $2 - 5$ implementados no agente *Observer* . Destaca-se a linha 10 o laço de repetição do envio das mensagens aos *PSpaces* informando-os da transferência de holarquias e de agente *Sector* . Nota-se também a última linha, 14, onde é informado ao novo *Sector* (Healthy) a respeito dos seus novos integrantes (*PSpaces*) e também do *Sector* que elas provêm.

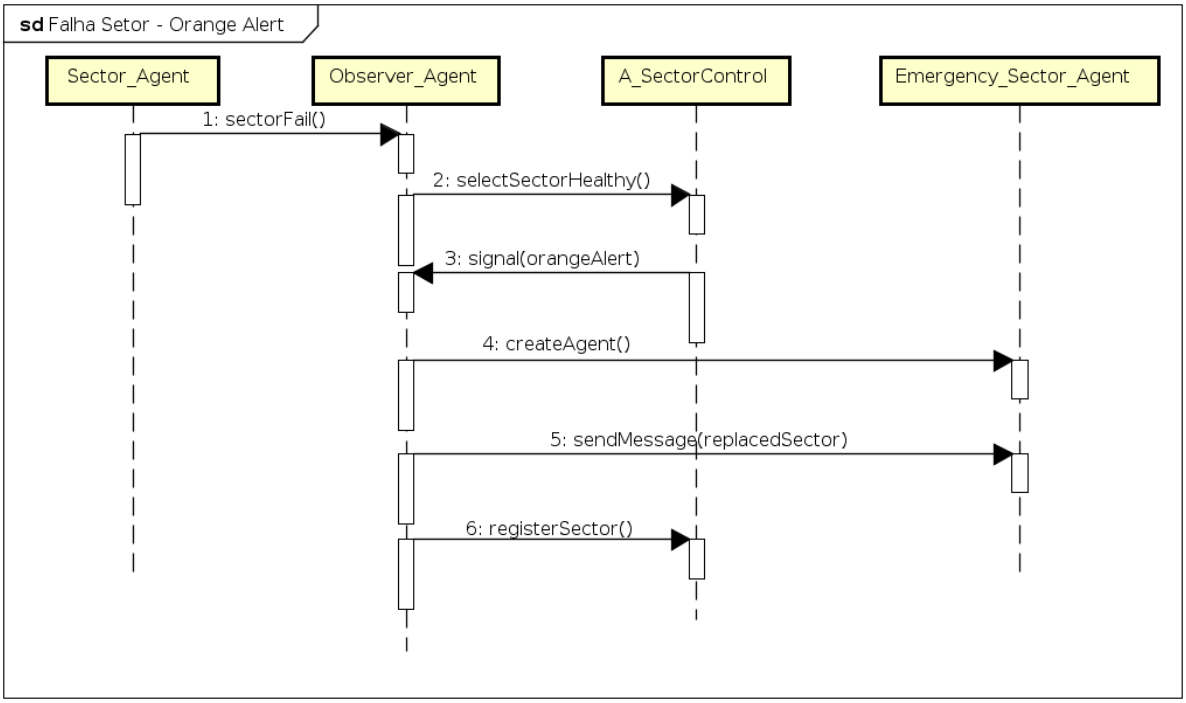
```
1 +sectorFail(SECTOR_NAME,SECTOR_CODE) <-
2         .print("The ",SECTOR_NAME," failed!");
3         setSectorHealthy(SECTOR_NAME,false);
4         selectHealthySector(SECTOR_NAME,SECTOR_HEALTHY);
5         .print("Notifying the Parking Spaces of ", SECTOR_NAME, "about their new
          ↪  Sector! (",SECTOR_HEALTHY,")");
6
```

```
7         //receive the old sector ps list
8         getSectorPSList(SECTOR_NAME,PSLIST);
9
10        for(.member(PS_AGENT,PSLIST) ) {
11        .send(PS_AGENT,achieve,newSector(SECTOR_HEALTHY));
12      };
13
14    .send(SECTOR_HEALTHY,tell,[newParkingSpaces(PSLIST),sectorChild(SECTOR_NAME)]).
```

Código 39 – Jason - Agente *Observer* - sectorFail

7.5.6 Falha: Agente Todos os agentes Setores

Figura 58 – Diagrama de Sequência - Falha de todos os agentes *Sectors*



Fonte: Autoria Própria

Finalmente, pode ocorrer em que todos os agentes *Sectors* venham a falhar, indicando assim uma falha em todo o sistema. Com essa falha, o Sistema todo fica indisponível para novas requisições de vagas de estacionamento e é emitido o *Orange Alert*. No entanto, há *IDrivers* correntes no Sistema e que não devem ser afetados por essas falhas. Assim, é criado um agente de Emergência para os Setores chamado de *Emergency_sector*. Esse agente funcionará apenas para o envio das mensagens aos *IDrivers* , pagamentos e finalização das sessões.

Em uma última instância, esse agente de emergência pode vir a falhar, o que demonstra uma falha completa no SMAH, não restrita apenas aos agentes, artefatos e holarquias, mas ao sistema todo. Assim, o sistema emite o último alerta, o *Red Alert* e encerra as atividades. A

Figura 58 ilustra através de um diagrama de sequências com as etapas em caso de todos os *Sectors* falharem.

- **1: sectorFail:** *Sector* informa a falha ao *Observer* ;
- **2: selectSectorHealthy:** *Observer* tenta selecionar um outro setor para transferir os *PSpace* do agente *Sector* falhado;
- **3: signal(orangeAlert):** Não há outros *Sectors* disponíveis, pois todos falharam. Assim é emitido o *Orange Alert*;
- **4: createAgent:** *Observer* cria o agente *Emergency_Sector*;
- **5: sendMessage(replacedSector)** Agente *Observer* envia ao *Emergency_Sector* qual foi o último *Sector* em funcionamento para que agora assuma a posição de cabeça (o *Emergency_Sector*) nessa holarquia;
- **6: registerSector:** Agente *Emergency_Sector* é registrado no artefato `A_SectorControl`.

7.6 CONSIDERAÇÕES FINAIS

O dsenvolvimento da `MAPS-HOLO` por meio do *framework* JaCaMo trouxe grandes desafios em diferentes etapas e em diferentes níveis. Uma vez que o JaCaMo é composto basicamente por três níveis: agentes, ambiente e organização é possível uma gama maior de funcionalidades para um SMA, entretanto, para um desenvolvimento de uma SMA Holônico alguns ajustes e desafios foram enfrentados.

Primeiramente, ao aprofundar os conhecimentos em ambos conceitos: SMA Holônicos e JaCaMo, aumentou a percepção da possibilidade da implementação da `MAPS-HOLO` no JaCaMo. No entanto, o fator determinante da viabilidade foi o conceito do aninhamento parcial dos agentes, pois se fosse utilizado o conceito de aninhamento total, seria necessário modificar e implementar algumas novas funcionalidades no JaCaMo. Sendo assim, com o aninhamento parcial dos agentes, foi possível utilizar o conceito de agente pai-filho de maneira satisfatória no Jason. No Cartago a flexibilidade é maior, pois há centenas de estruturas de dados fornecidas para que o SMAH ficasse robusto. Por fim, o Moise foi o grande responsável pelo fator holônico organizacional da `MAPS-HOLO` devido as funcionalidades nativas do Moise, como os papéis e os seus relacionamentos, as organizações, os grupos, sub-grupos e a relação entre eles. Finalmente, após o desenvolvimento e posterior testes, é possível afirmar que o JaCaMo possui um suporte inicial ao desenvolvimento de aplicações holônicas. O suporte não é pleno devido ao objetivo do *framework* não englobar necessariamente os agentes holônicos, porém, com alguns ajustes esse suporte pode ampliar-se e até tornar-se pleno.

8 RESULTADOS E DISCUSSÃO

Nesse capitulo será apresentado as simulações realizadas na MAPS-HOLO em diferentes cenários de utilização. As simulações tiveram como objetivo a verificação do funcionamento da MAPS-HOLO e a comparação dos modelos implementados para alocação das vagas e do processo de recuperação das falhas. Os índices aqui demonstrados estarão configurados em dois blocos: Comparação do modo *Head-Body versus Body-Body* , analisando assim a troca de mensagens e funções utilidade. O segundo bloco de comparação será para a análise do método desenvolvido na MAPS-HOLO para a questão das falhas nos agentes via aninhamento parcial ou transferência de agentes. Para comparar esse método, foi realizado uma comparação básica com o método *default* do JaCaMo para a substituição dos agentes, a clonagem dos agentes.

8.1 CONFIGURAÇÃO DOS CENÁRIOS - COMPARAÇÃO DOS MODELOS

Nessa seção será realizado a comparação entre os modelos implementados pela MAPS-HOLO , o *Head-Body* e *Body-Body*.

A Tabela 6 apresenta 10 cenários distintos com a presença de *IDrivers* . Durante a demonstração do sistema nos capítulos anteriores, foi mencionado o artefato A_Simulation. Esse artefato será utilizado a fim de que os *IDrivers* possam ser simulados como usuários reais. O artefato A_Simulation tem a mesma funcionalidade do artefato A_DriverConnection, o envio das requisições de vaga e de saída de vaga para o sistema. Sendo assim, a utilização dos *IDrivers* simulados possuem a maior se não exata semelhança com *IDrivers* que representam usuários reais.

Tabela 6 – Cenários - Comparação dos modelos

Cenário	# IDrivers	#Vagas	#Setores
1	300	200	4
2	300	150	2
3	200	100	4
4	200	100	2
5	200	100	1
6	100	50	5
7	50	25	5
8	50	25	1
9	20	10	2
10	20	10	1

Fonte: Autoria Própria

Além dos cenários, há a configuração dos pesos que os Setores irão utilizar para o cálculo das suas respectivas funções utilidade. A Tabela 7 apresenta os cinco setores utilizados e seus respectivos pesos.

Tabela 7 – Setores - Atribuição dos pesos

Setor	Peso: Preço	Peso: Tempo permanência
1	0.5	0.5
2	0.8	0.2
3	0.2	0.8
4	0.6	0.4
5	0.4	0.6

Fonte: Autoria Própria

Tabela 8 – Comparativo entre modelos - Função Utilidade

	Head-Body		Body-Body	
Cenário	**F.Utilidade Sistema**	**F.Utilidade Cliente**	**F.Utilidade Sistema**	**F.Utilidade Cliente**
1	0.47	0.75	0.83	0.78
2	0.91	0.79	0.86	0.79
3	0.45	0.79	0.86	0.74
4	0.9	0.78	0.75	0.79
5	0.98	0.74	0.95	0.78
6	0.73	0.85	0.77	0.85
7	0.33	0.73	0.68	0.85
8	0.78	0.75	0.78	0.83
9	0.31	0.64	0.61	0.79
10	0.63	0.75	0.84	0.83
Média	0.649	0.757	0.793	0.803

Fonte: Autoria Própria

8.2 COMPARAÇÃO DO MODELO HEAD-BODY VS BODY-BODY

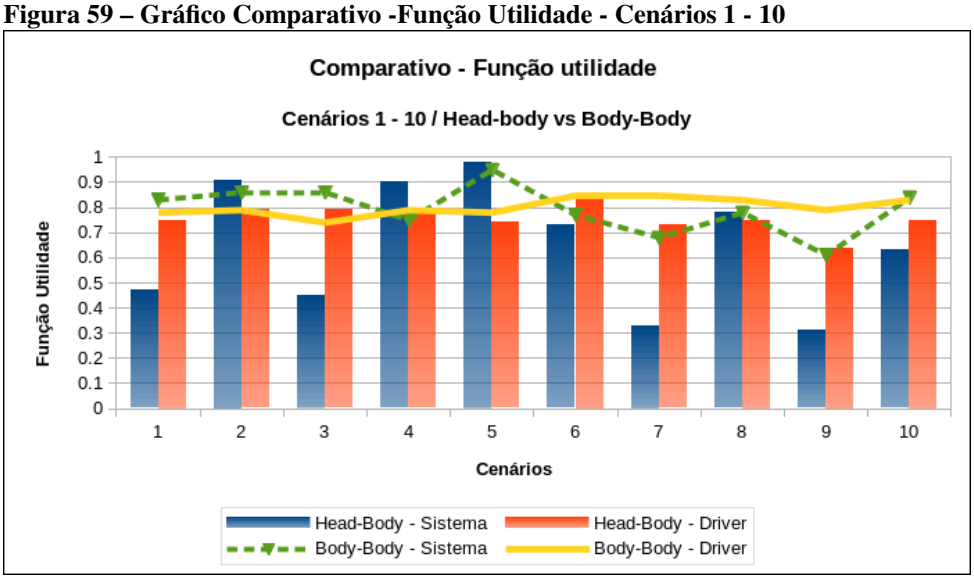
As tabelas a seguir apresentam um comparativo entre os modelos implementados na `MAPS-HOLO`.

8.2.1 Função Utilidade

No comparativo da função utilidade, será comparado os resultados em dois domínios (Cliente e Sistema). Nota-se que quanto mais próximo o valor de 1, maior será a satisfação. Por fim, é apresentado uma média da função utilidade levando em conta os 10 cenários da tabela 6. A Tabela 8 apresenta os resultados obtidos após a simulação nos 10 cenários em ambos modelos. A Figura 59 ilustra um gráfico com os resultados da Tabela 8 sumarizados.

O gráfico exibido pela figura 59 demonstra uma maior satisfação dos usuários no modelo *Body-Body* em média, pois em alguns cenários, como o 2 e o 4 tiveram resultados inferiores ao *Head-Body* . Na média o modelo descentralizado se saiu melhor e na maioria dos cenários, porém isso se deve ao fato de como o processo de negociação foi desenvolvido em ambos modelos. No modelo *Head-Body* o *IDriver* negociava apenas com o *Manager* e tinha o número de

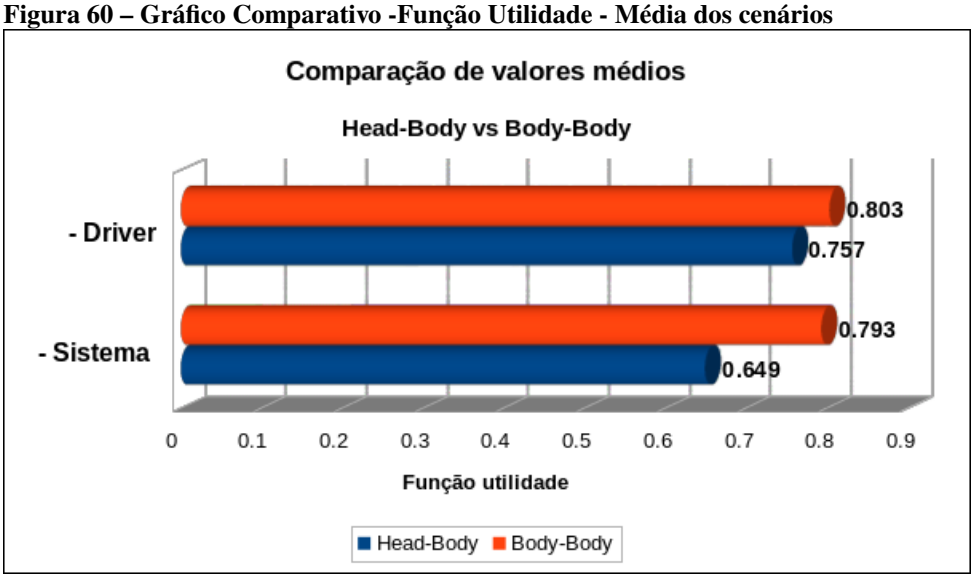
Figura 59 – Gráfico Comparativo -Função Utilidade - Cenários 1 - 10



Fonte: Autoria Própria

rounds determinado pelo número de interações com o *Manager*. Embora o *Manager* também tinha interferência no processo de selecionar as ofertas para gerar um equilíbrio, o modelo *Body-Body* mostrou ser mais vantajoso tanto para o Sistema (lucro e giro de motoristas), mas principalmente para os Clientes. Para ficar ainda mais claro, o gráfico da Figura 60 apresenta a média de ambos modelos apontando a vantagem consolidade no quesito função utilidade para o *Body-Body*.

Figura 60 – Gráfico Comparativo -Função Utilidade - Média dos cenários



Fonte: Autoria Própria

Tabela 9 – Comparativo entre modelos - Troca de Mensagens

	Mensagens Trocadas	
Cenário	**Head - Body**	**Body-Body**
1	5329	2671
2	4178	2766
3	3628	1714
4	2747	1817
5	2183	1601
6	2045	833
7	990	418
8	537	422
9	244	183
10	219	165
Média	2210	1259

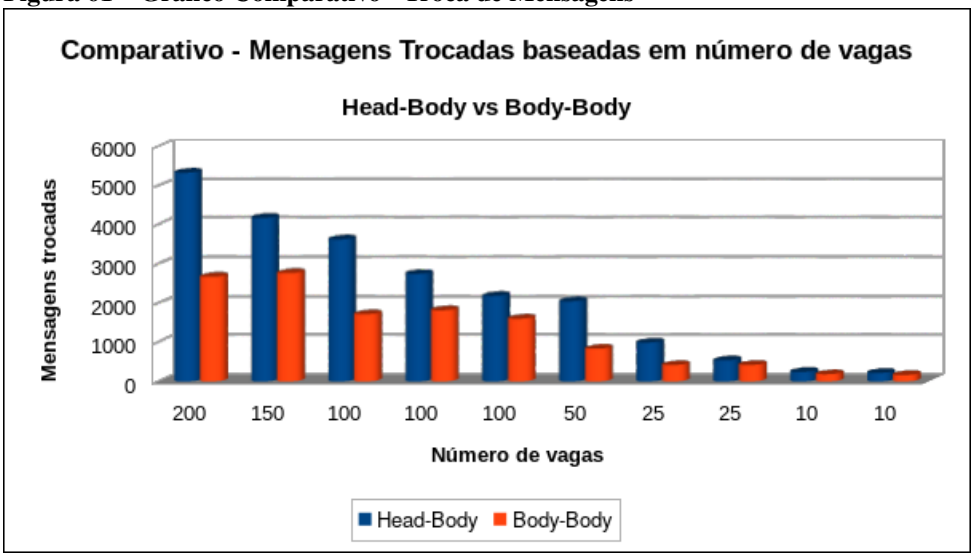
Fonte: Autoria Própria

8.2.2 Troca de Mensagens

O último quesito a ser avaliado entre os modelos seráa troca de mensagens entre os agentes. As trocas contabilizadas são entre os agentes participantes da negociação. Nesse quesito as negociações que não foram bem sucedidas entraram na contabilidade. A tabela 9 lista as trocas de mensagens nos modelos da `MAPS-HOLO` nos cenários previamente apresentados.

Em conjunto com a tabela 9, a Figura 61 apresenta em um gráfico a troca de mensagens e a diferença da quantidade de mensagens entre os modelos *Head-Body* e *Body-Body* .

Figura 61 – Gráfico Comparativo - Troca de Mensagens



Fonte: Autoria Própria

Com base na Tabela 9 e no gráfico 61, o modelo *head-body* consome praticamente em todos os cenários o dobro das mensagens enviadas pelo modelo *body-body*. Isso se deve ao fato da presença de um agente centralizador em que recebe todas as requisições e as retransmite com certas alterações. A facilidade de um agente central torna mais fácil a implementação e o

controle do SMAH, porém, tendo em vista o número de mensagens massivamente maior que o modelo descentralizado, podemos concluir nessa primeira parte de testes que sim o modelo *body-body* apresenta maiores vantagens, não em todos os cenários, mas na média com uma larga vantagem. Não apenas em trocas de mensagens, mas na função utilidade exibida no gráfico dos cenários na Figura 59, o Sistema possui maior satisfação, bem como o Cliente. Como desenvolvedor, vale ressaltar a dificuldade muitas vezes do desenvolvimento descentralizado de um SMA, pois há uma quantidade maior de elementos a serem controlados em vista de um agente central. No entanto, diante dos benefícios disponíveis pelo modelo *body-body* , a escolha em um cenário de alocação de recursos por um tempo e custo, uma boa escolha será o modelo descentralizado.

8.3 COMPARAÇÃO MÉTODO `MAPS-HOLO` VS CLONAGEM DE AGENTES

A segunda etapa de testes do `MAPS-HOLO` diz respeito a perfomance e flexibilidade do SMAH diante das falhas que possam vir a ocorrer no sistema. Com o objetivo de comparar a estratégia adotada nesse trabalho, será realizado uma etapa de testes em cinco cenários distintos. Nesses cenários todos os agentes (*Sector, PSpace e Manager*) irão falhar em algum instante do sistema. Para verificar a performance do Sistema, foi desenvolvido funções de captura de tempo em milisegundos para averiguar em quanto tempo o Sistema estaria recuperado da eventual falha. Para a comparação, será analisado o método de aninhamento parcial do `MAPS-HOLO` com o método nativo do JaCaMo, o qual é a clonagem de agentes. Destaca-se que a clonagem de agentes do JaCaMo provém do método de clonagem de objetos do Java.

A Tabela 10 lista os cenários que serão empregados nos testes.

Tabela 10 – Cenários - Comparação dos métodos: `MAPS-HOLO` vs clonagem de agentes

Cenário	Vagas	Setores
1	150	5
2	100	5
3	100	4
4	80	2
5	50	1

Fonte: Autoria Própria

Com base nos cenários da Tabela 10 e após rodar todas as ocorrências de falha em todos os cenários, a Tabela 11 apresenta os tempos de recuperação do sistema em milisegundos.

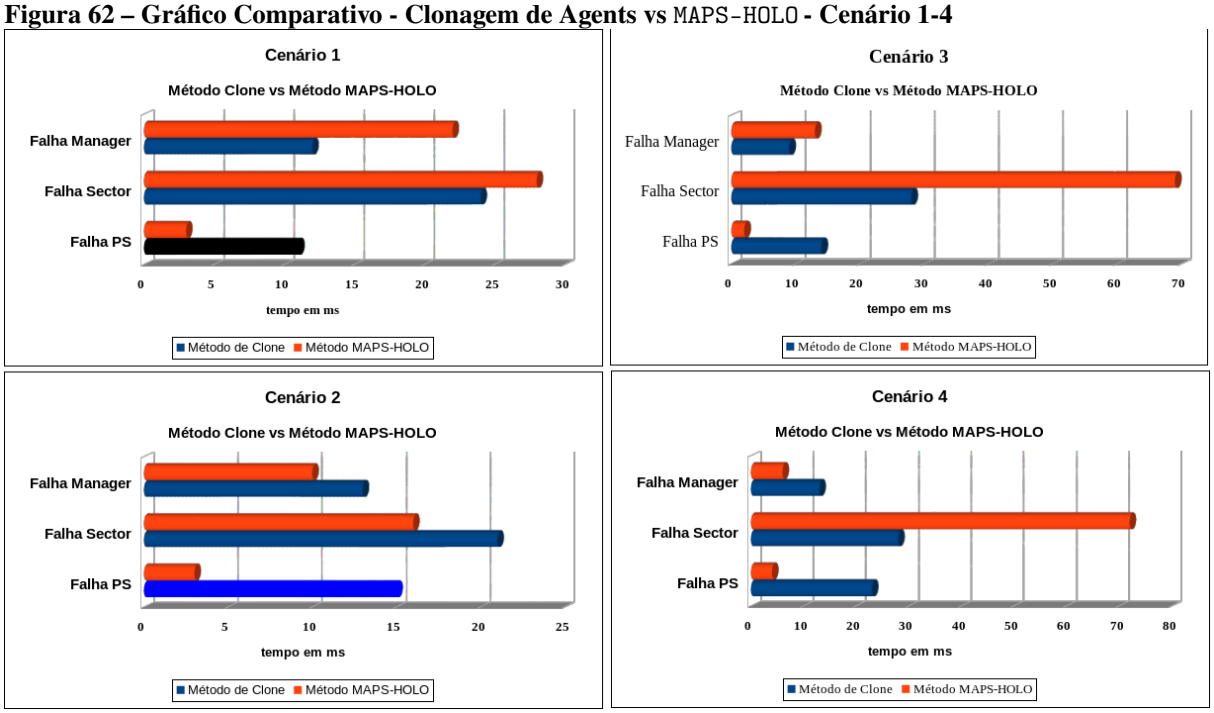
Corroborando com os resultados da Tabela 11, abaixo é apresentado os gráficos de cada cenário a fim de que torne-se mais claro a comparação entre os dois modelos.

Tabela 11 – Resultados - Comparação dos métodos: MAPS-HOLO vs clonagem de agentes

	Método Clone Tempo de recuperação falha			Método MAPS-HOLO Tempo de recuperação falha		
Cenário	**PSpace**	**Setor**	**Manager**	**PSpace**	**Setor**	**Manager**
1	11	24	12	3	28	22
2	15	21	13	3	16	10
3	14	28	9	2	69	13
4	23	28	13	4	72	6
5	69	36	8	4	50	5
Média Local	26.4	27.4	11	3	46.25	12.75
Média Global		**21.6**			**20.6**	

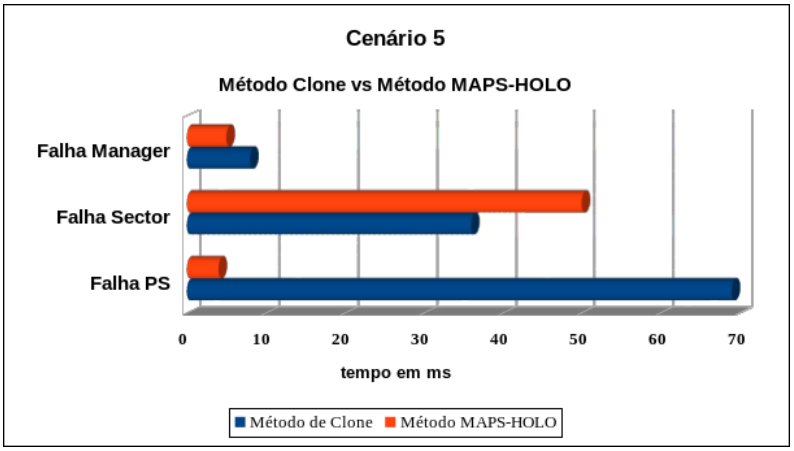
Fonte: Autoria Própria

Figura 62 – Gráfico Comparativo - Clonagem de Agents vs MAPS-HOLO - Cenário 1-4



Fonte: Autoria Própria

Figura 63 – Gráfico Comparativo - Clonagem de Agents vs MAPS-HOLO - Cenário 5

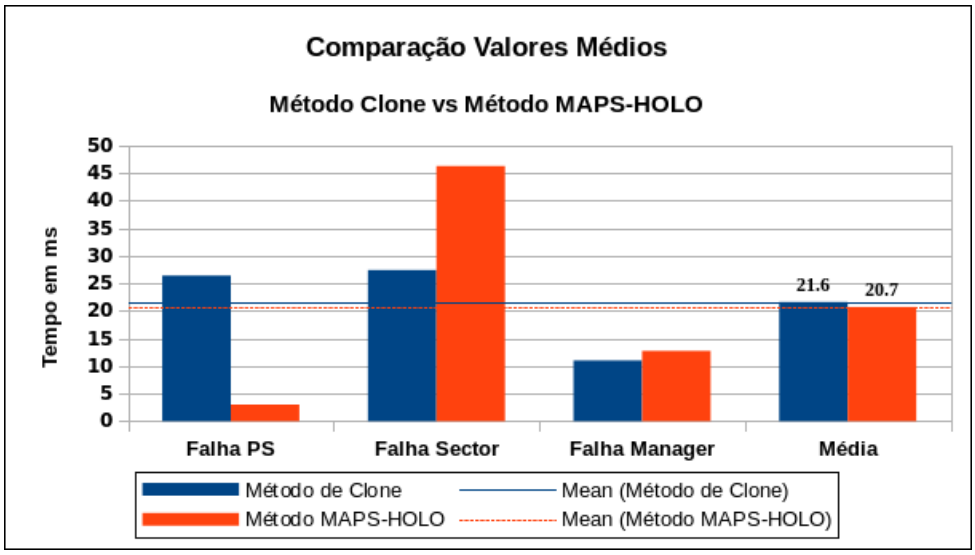


Fonte: Autoria Própria

8.3.1 Comparativo entre os métodos - Média dos Cenários

Com base nos gráficos das Figuras 62 e 63, o gráfico da Figura 64 apresenta o gráfico final com os valores médios do tempo de recuperação do SMAH diante das diferentes falhas.

Figura 64 – Gráfico Comparativo - Clonagem de Agents vs `MAPS-HOLO` - Média dos Cenários



Fonte: Autoria Própria

Após os dois blocos de bateria de testes do `MAPS-HOLO`, inicialmente comparando o sistema com ele mesmo em suas diferentes configurações, o modelo *Body-Body* apresentou uma vantagem considerável dinate do modelo *head-body*. Por fim, nesse último bloco o Sistema foi comparado com ele mesmo, porém utilizando duas abordagens distintas. A abordagem desenvolvida na Arquitetura do `MAPS-HOLO` não é nativo do JaCaMo, mas sim foi implementado utilizando os recursos que o JaCaMo oferece, como a programação dos agentes, artefatos e organizações. O método de clonagem embora presente no JaCaMo, está presente também via programação Java.

Os resultados obtidos na segunda etapa de testes testifica o conceito de que o JaCaMo possui um suporte inicial para o desenvolvimento de aplicações holônicas, mesmo que os recursos providos pelo JaCaMo não tenham a motivação primária de trabalhar com os agentes holônicos. Porém, com os resultados obtidos, podemos inferir a vantagem do método de aninhamento parcial dos agentes em frente as falhas que ocorrem no Sistema.

Os tempos de recuperação do JaCaMo estão em milisegundos, o que mostra que mesmo no pior cenário, o JaCaMo em qualquer um dos métodos apresenta facilidades para a recuperação do Sistema para que não ocorra a interrupção e suspensão do Sistema por qualquer eventual falha. Tornando os resultados mais específicos, o método da `MAPS-HOLO` mostrou-se superior na média em aproximadamente 10 à 15%. É necessário mencionar também a probabilidade da ocorrência das falhas, pois devido ao número maior de agentes *PSpace*, a probabilidade de falha neles é maior. Portanto, nos agentes *PSpace* o método da `MAPS-HOLO` se mostrou praticamente

9 vezes mais eficiente do que o método de clonagem dos agentes. Porém, na falha dos agentes *Manager* e *Sector* a clonagem de agentes saiu-se levemente em vantagem (1 milisegundo). Assim concluímos que ambos os métodos são eficientes e proveitosos nos cenários, e que o JaCaMo possui muito ainda a oferecer com a metodologia holônica de aninhamento dos agentes.

9 CONCLUSÃO

O projeto MAPS baseia-se no contexto de estacionamentos inteligentes, que por sua vez estão inseridos em cidades inteligentes, as quais são descritas como cidades que possuem tecnologias com a finalidade de facilitar e automatizar inúmeros sistemas que a compõe, como: governo, população, segurança, meio ambiente, trânsito, etc. A atual proposta de trabalho apresentada neste documento é uma ampliação de projetos de pesquisa anteriormente desenvolvidos sob a sigla MAPS (*MultiAgent Parking System*). Por sua vez, o MAPS-HOLO propôs a modelagem e o desenvolvimento de toda uma arquitetura configurável para Sistemas Multiagentes Holônicos.

A utilização do JaCaMo para o desenvolvimento do MAPS-HOLO foi decidido desde o início primeiramente devido a familiaridade com a ferramenta. Diante dessa questão, houve desafios a serem enfrentados e soluções foram criadas para esses desafios, uma vez o JaCaMo não possui suporte nativo aos agentes holônicos. Destaca-se a implementação dos Contratos, o Artefato de Endereços e tradução dos endereços dos agentes e outras tantas funcionalidades que proporcionaram ao JaCaMo o vislumbre de um suporte inicial às aplicações holônicas. O modelo holônico, não necessariamente o de agentes, está presente em diversas áreas da pesquisa, como: Engenharia da Produção na indústria 4.0, Biologia e outras áreas, sendo assim há uma demanda por aplicações holônicas. O suporte que o JaCaMo proveu ao MAPS-HOLO esta em estado inicial, porém com perspectivas de até ampliação da arquitetura para outros domínios e recursos.

Por fim, a Arquitetura MAPS-HOLO destaca-se por não estar restringida à apenas vagas de estacionamento, mas para qualquer recurso alocável. Além disso, a Arquitetura provê mecanismos de negociação, alocação e tolerância a falhas dos agentes. Sendo assim, podemos destacar alguns futuros trabalhos que podem ser desenvolvidos através dessa primeira versão da MAPS-HOLO como: i)Desenvolvimento de novos mecanismos de negociação (e.g leilão); ii) Aplicação dos agentes recursos em plataformas embarcadas; iii) Desenvolvimento de um ambiente gráfico para a Arquitetura; iv) Criação e edição automática dos Contratos dos setores mediante o uso dos mesmos. Esses foram apenas alguns exemplos, pois a Arquitetura teve como propósito ser flexível e expansível para que futuras pesquisas possam ser desenvolvidas sobre e sob ela em breve.

REFERÊNCIAS

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

Arthur Koestler. The ghost in the machine. Arkama Books, 1969.

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

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.

BOTTI, Vicente; GIRET, Adriana. Holonic manufacturing systems. **ANEMONA: A Mulit-agent Methodology for Holonic Manufacturing Systems**, p. 7–20, 2008. Disponível em: <https://link.springer.com/chapter/10.1007/978-1-84800-310-1_2>.

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

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

CASTRO, Lucas Fernando SOUZA DE; ALVES, Gleifer VAZ; BORGES, André PINZ. Using trust degree for agents in order to assign spots in a Smart Parking. **ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal**, v. 6, n. 2, p. 45–55, jun. 2017. ISSN 2255-2863. Disponível em: <http://campus.usal.es/~revistas_trabajo/index.php/2255-2863/article/view/ADCAIJ2017624555>.

CASTRO, Lucas Fernando Souza de; BORGES, André Pinz; ALVES, Gleifer Vaz. Developing a smart parking solution based on a Holonic Multiagent System using JaCaMo Framework. In: **Anais do XII Workshop-Escola de Sistemas de Agentes, seus Ambientes e apliCações - WESAAC 2018**. Fortaleza - CE: [s.n.], 2018. XII, p. 226–231. ISBN 2177-2096.

COSSENTINO, Massimo *et al.* ASPECS: an agent-oriented software process for engineering complex systems. **Autonomous Agents and Multi-Agent Systems**, v. 20, n. 2, p. 260–304, 2010. Disponível em: <http://www.springerlink.com/index/436p7ll3n8443637.pdf>.

FERAUD, Maxime; GALLAND, Stéphane. First comparison of SARL to other agent-programming languages and frameworks. In: **International Workshop on Agent-based Modeling and Applications with SARL (SARL 2017)**. [S.l.]: Elsevier, 2017.

FIPA Contract Net Interaction Protocol Specification. 2002. Disponível em: <http://www.fipa.org/specs/fipa00029/SC00029H.html>.

FISCHER, Klaus; SCHILLO, Michael; SIEKMANN, Jörg. Holonic multiagent systems: A foundation for the organisation of multiagent systems. In: **HoloMAS**. Springer, 2003. p. 71–80. Disponível em: <http://link.springer.com/content/pdf/10.1007/b11833.pdf#page=83>.

FISHER, Klaus. Agent-based design of holonic manufacturing systems. **Robotics and autonomous Systems**, v. 27, n. 1-2, p. 3–13, 1999. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0921889098000797>.

FRAIFER, Muftah; FERNSTRöM, Mikael. Investigation of Smart Parking Systems and their technologies. In: **Thirty Seventh International Conference on Information Systems. IoT Smart City Challenges Applications (ISCA 2016), Dublin, Ireland**. [s.n.], 2016. p. 1–14. Disponível em: <http://iot-smartcities.lero.ie/wp-content/uploads/2016/12/Investigation-of-Smart-Parking-Systems-and-their-technologies.pdf>.

GELBER, Christian; SIEKMANN, Jorg; VIERKE, Gero. **Holonic multi-agent systems**. [S.l.], 1999. Disponível em: <http://www.dfki.uni-kl.de/dfkidok/publications/TM/07/01/tm-07-01.pdf>.

GIRET, Adriana; BOTTI, Vicente. Holons and agents. **Journal of Intelligent Manufacturing**, v. 15, n. 5, p. 645–659, 2004. Disponível em: <http://www.springerlink.com/index/T286MU16R6423073.pdf>.

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

HANNOUN, Mahdi *et al.* MOISE: An organizational model for multi-agent systems. In: MONARD, Maria Carolina; SICHMAN, Jaime Simão (Ed.). **Advances in Artificial Intelligence**. [S.l.]: Springer Berlin Heidelberg, 2000. p. 156–165. ISBN 978-3-540-44399-5.

HORLING, Bryan; LESSER, Victor. A survey of multi-agent organizational paradigms. **The Knowledge Engineering Review**, v. 19, n. 4, p. 281–316, 2004. Disponível em: <https://www.cambridge.org/core/journals/knowledge-engineering-review/article/a-survey-of-multi-agent-organizational-paradigms/A07BCCB1379F001DE995F3E5476EE4AB>.

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

HüBNER, Jomi Fred; SICHMAN, Jaime Simão; BOISSIER, Olivier. A model for the structural, functional, and deontic specification of organizations in multiagent systems. In: BITTENCOURT, Guilherme; RAMALHO, Geber L. (Ed.). **Advances in Artificial Intelligence**. Springer Berlin Heidelberg, 2002. v. 2507, p. 118–128. ISBN 978-3-540-00124-9 978-3-540-36127-5. Disponível em: <http://link.springer.com/10.1007/3-540-36127-8_12>.

HüBNER, Jomi Fred; SICHMAN, Jaime Simao; BOISSIER, Olivier. **MOISE+ tutorial**. [S.l.]: Citeseer, 2008.

IDRIS, M.Y.I. *et al.* Car Park System: A Review of Smart Parking System and its Technology. **Information Technology Journal**, v. 8, n. 2, p. 101–113, fev. 2009. ISSN 18125638. Disponível em: <http://www.scialert.net/abstract/?doi=itj.2009.101.113>.

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

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

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

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

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

LEITAO, Paulo; MARIK, Vladimir; VRBA, Pavel. Past, Present, and Future of Industrial Agent Applications. **IEEE Transactions on Industrial Informatics**, v. 9, n. 4, p. 2360–2372, nov. 2013. ISSN 1551-3203, 1941-0050. Disponível em: <http://ieeexplore.ieee.org/document/6319392/>.

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

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

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

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

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

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

NWANA, Hyacinth S. Software agents: An overview. **The knowledge engineering review**, v. 11, n. 3, p. 205–244, 1996. Disponível em: <https://www.cambridge.org/core/journals/knowledge-engineering-review/article/software-agents-an-overview/66832B4C8D509136A8E1AC2E61EB70D0>.

ODELL, James; NODINE, Marian; LEVY, Renato. A metamodel for agents, roles, and groups. In: ____. **Agent-Oriented Software Engineering V: 5th International Workshop, AOSE 2004, New York, NY, USA, July 19, 2004. Revised Selected Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 78–92. ISBN 978-3-540-30578-1. Disponível em: <https://doi.org/10.1007/978-3-540-30578-1_6>.

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

PERSSON, Camille *et al.* A multi-agent based governance of machine-to-machine systems. In: **Proceedings of the 2014 International Workshop on Web Intelligence and Smart Sensing**. New York, NY, USA: ACM, 2014. (IWWISS '14), p. 25:1–25:2. ISBN 978-1-4503-2747-3. Disponível em: <http://doi.acm.org/10.1145/2637064.2637112>.

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

PěCHOUčEK, Michal; MAříK, Vladimír. Industrial deployment of multi-agent technologies: review and selected case studies. **Autonomous Agents and Multi-Agent Systems**, v. 17, n. 3, p. 397–431, dez. 2008. ISSN 1387-2532, 1573-7454. Disponível em: <http://link.springer.com/10.1007/s10458-008-9050-0>.

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

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

RIBINO, Patrizia *et al.* A norm-governed holonic multi-agent system metamodel. In: **International Workshop on Agent-Oriented Software Engineering**. Springer, 2012. p. 22–39. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-642-39866-7_2>.

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

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

RODRIGUEZ, Sebastian; GAUD, Nicolas; GALLAND, Stephane. SARL: A General-Purpose Agent-Oriented Programming Language. In: . IEEE, 2014. p. 103–110. ISBN 978-1-4799-4143-8. Disponível em: <http://ieeexplore.ieee.org/document/6928174/>.

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

SCHILLO, Michael; FISCHER, Klaus. A Taxonomy of Autonomy in Multiagent Organisation. In: GOOS, Gerhard *et al.* (Ed.). **Agents and Computational Autonomy**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. v. 2969, p. 68–82. ISBN 978-3-540-22477-8 978-3-540-25928-2. DOI: 10.1007/978-3-540-25928-2_6. Disponível em: <http://link.springer.com/10.1007/978-3-540-25928-2_6>.

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

SHEHORY, Onn M. **Architectural properties of multi-agent systems**. Carnegie Mellon University, The Robotics Institute, 1998. Disponível em: <http://www.ri.cmu.edu/pub_files/pub1/shehory_onn_1998_1/shehory_onn_1998_1.pdf>.

SMITH, R. G. The contract net protocol: High-level communication and control in a distributed problem solver. **IEEE Transactions on Computers**, C-29, n. 12, p. 1104–1113, Dec 1980. ISSN 0018-9340.

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

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

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

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

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

APÊNDICE A - CÓDIGO MOISE DA ORGANIZAÇÃO - MAPSHOLO ORG

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="http://moise.sourceforge.net/xml/os.xsl" type="text/xsl" ?>
<organisational-specification
    id="maps_holo_org"
    os-version="0.8"
    xmlns='http://moise.sourceforge.net/os'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://moise.sourceforge.net/os
                        http://moise.sourceforge.net/xml/os.xsd' >

<structural-specification>
    <role-definitions>
            <role id = "r_head"/>
            <role id = "r_body"/>
            <role id="r_observer"/>
        <role id="r_builder"/>
                <role id="r_manager"> <extends role= "r_head"/> </role>
                <role id="r_head_sector"> <extends role= "r_head"/> </role>
                <role id="r_body_sector"> <extends role= "r_body"/> </role>
                <role id="r_pspace"> <extends role= "r_body"/> </role>
                <role id="r_idriver"/>
    </role-definitions>

    <group-specification id="maps_holo_group">
        <roles>
            <role id="r_observer" min="1" max="1"/>
            <role id="r_builder" min="1" max="1"/>
            <role id="r_manager" min="0" max="1"/>
            <role id="r_body_sector" min="0" max="9999"/>
        </roles>

        <links>
           <link from="r_builder" to="r_manager" type="authority"
            ↪  scope="inter-group" bi-dir="false"/>
           <link from="r_builder" to="r_head_sector" type="authority"
            ↪  scope="inter-group" bi-dir="false"/>
           <link from="r_builder" to="r_body_sector" type="authority"
            ↪  scope="inter-group" bi-dir="false"/>
           <link from="r_builder" to="r_observer" type="authority"
            ↪  scope="inter-group" bi-dir="false"/>
           <link from="r_builder" to="r_pspace" type="authority"
            ↪  scope="inter-group" bi-dir="false"/>
           <link from="r_builder" to="r_idriver" type="authority"
            ↪  scope="inter-group" bi-dir="false"/>
        </links>
```

```
40
41          <subgroups>
42                  <group-specification id="sectorGroup">
43                      <roles>
44                              <role id="r_idriver"  min="0" max="999999"/>
45                          <role id="r_pspace" min="0" max="999999"/>
46                          <role id="r_head_sector" min="1" max="2"/>
47                  </roles>
48                          <links>
49                               <link from="r_head_sector" to="r_pspace"
                                 ↪  type="authority"  scope="intra-group"
                                 ↪  bi-dir="false"/>
50                               <link from="r_pspace" to="r_head_sector"
                                 ↪  type="acquaintance"  scope="intra-group"
                                 ↪  bi-dir="false"/>
51                               <link from="r_pspace" to="r_head_sector"
                                 ↪  type="communication"  scope="intra-group"
                                 ↪  bi-dir="false"/>
52                          </links>
53                        </group-specification>
54          </subgroups>
55      </group-specification>
56  </structural-specification>
57
58  <functional-specification>
59      <scheme id= "build_scheme">
60          <goal id = "setupSystem">
61                  <plan operator = "sequence">
62                          <goal id = "buildSystem">
63                              <plan operator="sequence">
64                                  <goal id ="setupWorkspaces"/>
65                                  <goal id ="setupArtifacts"/>
66                                  <goal id ="setupAgents">
67                                          <plan operator = "sequence">
68                                                  <goal id ="setupObserver"/>
69                                                  <goal id ="setupManager"/>
70                                                  <goal id ="setupSectors"/>
71                                                  <goal id ="setupPSpaces"/>
72                                          </plan>
73                                  </goal>
74                              </plan>
75                          </goal>
76                          <goal id ="startSystem">
77                                  <plan operator="parallel">
78                                          <goal id ="checkRequests"/>
79                                          <goal id ="observeAgents"/>
80                                  </plan>
81                                     </goal>
82                  </plan>
```

```
83          </goal>
84
85          <mission id="m_build" min="1" max="1">
86                  <goal id="setupWorkspaces"/>
87                  <goal id="setupArtifacts"/>
88                  <goal id ="setupObserver"/>
89              <goal id ="setupManager"/>
90               <goal id ="setupSectors"/>
91               <goal id ="setupPSpaces"/>
92          </mission>
93
94          <mission id="m_start" min="1">
95                  <goal id="checkRequests"/>
96          </mission>
97
98          <mission id="m_observe" min="1">
99                  <goal id="observeAgents"/>
100         </mission>
101     </scheme>
102
103     <scheme id= "psFail_scheme">
104         <goal id = "restabilishPS">
105                 <plan operator = "sequence">
106                         <goal id = "setPSParent"/>
107                         <goal id ="notifyPSParent"/>
108                 </plan>
109         </goal>
110
111         <mission id="m_psFail" min="0">
112                 <goal id="setPSParent"/>
113                 <goal id="notifyPSParent"/>
114         </mission>
115     </scheme>
116
117     <scheme id= "sectorFail_scheme">
118         <goal id = "restabilishSector">
119                 <plan operator = "sequence">
120                         <goal id = "notifyObserver"/>
121                         <goal id ="notifySectorPS"/>
122                 </plan>
123         </goal>
124
125         <mission id="m_notifyObserver" min="0">
126                 <goal id="notifyObserver"/>
127         </mission>
128
129         <mission id="m_notifySectorPS" min="0">
130                 <goal id="notifySectorPS"/>
131         </mission>
```

```
132     </scheme>
133 </functional-specification>
134 
135 <normative-specification>
136     <norm id="n_build" type="obligation" role="r_builder" mission="m_build"/>
137     <norm id="n_start" type="obligation" role="r_observer" mission="m_start"/>
138     <norm id="n_observe" type="obligation" role="r_observer" mission="m_observe"/>
139     <norm id="n_psFailHead" type="permission" role="r_head_sector"
        ↪  mission="m_psFail"/>
140     <norm id="n_psFailBody" type="permission" role="r_body_sector"
        ↪  mission="m_psFail"/>
141 </normative-specification>
142 
143 </organisational-specification>
```

Código 40 – Organização Moise - MAPSHOLO ORG

APÊNDICE B - CÓDIGO JACAMO - MAPSHOLO

```
mas mAPS_HOLO {

        agent observer

        agent builder{
                beliefs: nSectors(5),
                         nPSpaces(20), //# of parking spaces by sector
                         structure("BB"),
                         network(true),
                         simulation(false)
        }

        organisation maps_holo_org : maps_holo_org.xml {
                group maps_holo_grp : maps_holo_group {
                        responsible-for: bScheme
                        players: builder r_builder
                                         observer r_observer
                                         debug
                }

                scheme pFailScheme : psFail_scheme{
                        debug
                }

                scheme sectorFailScheme : sectorFail_scheme{
                        debug
                }

                scheme bScheme: build_scheme{
                        debug
                }
        }
}
```

Código 41 – Código JaCaMo - JCM