PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

AGENTSPEAK(PY) - INTERPRETADOR AGENTSPEAK(L) PARA PYTHON

ANDRÉ LUIZ LEONHARDT DOS SANTOS

Trabalho de Conclusão I apresentado como requisito parcial à obtenção do grau de Bacharel em Sistemas de Informação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Felipe Meneguzzi

Porto Alegre
2016

RESUMO

O desenvolvimento de sistemas multiagentes tem possibilitado a descrição, a solução e o entendimento de grandes sistemas distribuídos. Ao longo dos anos, linguagens de programação de agentes surgiram com o propósito de facilitar o desenvolvimento destes sistemas. Estas linguagens permitem criar agentes com comportamento baseado em crenças, desejos e intenções, e que interajam com um ambiente complexo. O AgentSpeak(L) está entre as linguagens mais utilizadas, cuja implementação mais comum é o JASON. Entretanto, apesar de suportar diversas extensões da linguagem AgentSpeak(L) o desempenho deste interpretador é limitado. Propomos, então, o desenvolvimento de um interpretador da sintaxe do AgentSpeak(L) para Python, visando eficiência computacional.

Palavras-Chave: Agentes. Sistemas Multiagentes. AgentSpeak(L). Interpretadores. Python.

ABSTRACT

The development of multi-agent systems has made possible the description, solution, and understanding of large-scale distributed systems. Over the years, agent programming languages have emerged with the aim of easing the development of such systems. Such languages allow the creation of agents whose behavior is based on beliefs, desires, and intentions, and which interact with a complex environment. AgentSpeak(L) features among the most widely used languages, in which JASON is its most common implementation. Although this interpreter supports different language extensions, it has poor computational efficiency. Thus, we propose the development of an interpreter of the AgentSpeak(L) syntax for Python, aiming to improve its computational efficiency.

Keywords: Agents. Multi-Agent Systems. AgentSpeak (L). Interpreters. Python.

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE SIGLAS

AAMAS – *International Conference on Autonomous Agent & Multi-Agent Systems*

BDI – *Belief-Desire-Intention*

BRF – *Belief Revision Function*

DMARS – *Distributed Multi-Agent Reasoning System*

IA – Inteligência Artificial

JADEX – *JADE Extension*

JASON – *Java-based AgentSpeak interpreter used with SACI for multi-agent distribution Over the Net*

MAS – *Multi-Agent System*

MGU – *Most General Unifier*

PRS – *Procedural Reasoning System*

UML – *Unified Modeling Language*

US – *User Story*

SUMÁRIO

1. INTRODUÇÃO

Inteligência Artificial (IA) é o campo de estudo da ciência da computação que tem como objetivo construir máquinas e programas de computador com comportamento inteligente. Estes programas, quando interagem entre si de forma autônoma, formam sistemas multiagentes, ou *Multi-Agent Systems* (MAS). Os MAS têm recebido atenção especial pela sua capacidade de explorar as características presentes em cenários com sistemas distribuídos massivos e abertos, como a internet, o tráfego aéreo e terrestre e a nossa sociedade [20, p. xi].

Estes cenários criaram a necessidade de descrever sistemas com um ambiente complexo, em que diversos agentes interagem entre si. Isso impulsionou o desenvolvimento de ferramentas e linguagens de programação próprias para este tipo de tarefa. Entre as linguagens amplamente utilizadas está o AgentSpeak(L), cuja implementação mais comum é o interpretador JASON. Desta forma, é possível criar agentes com comportamento baseado em crenças, desejos e intenções, e que interajam com um ambiente complexo. Neste caso, os agentes são descritos por meio do AgentSpeak(L), ao passo que o ambiente é desenvolvido utilizando a linguagem Java [4].

Tendo em vista que muitos dos cenários de MAS possuem uma complexidade elevada, é necessário que o interpretador em questão seja enxuto e eficiente. No caso do JASON, testes recentes apontam que o interpretador utiliza, em média, 18,2 vezes mais processamento que em abordagens usuais em C++. Nestes testes, o modelo implementado com o JASON obteve 61% de tempo de utilização da CPU, enquanto que o mesmo modelo em C++ atingiu 4% [16, p. 59].

Neste projeto, propomos o desenvolvimento de um interpretador da sintaxe do AgentSpeak(L) para Python. A linguagem Python é adequada, neste sentido, devido as suas características, como a simplicidade no uso de estruturas de dados, o foco em produtividade e legibilidade e a sua facilidade na integração com bibliotecas desenvolvidas em C, C++ e Fortran, que permitem um desempenho mais eficiente em funcionalidades críticas [9, p. 2].

Este documento está organizado em 6 capítulos, sendo o primeiro esta introdução. O Capítulo 2 apresenta as noções de agente, os tipos de ambientes em que eles interagem, as maneiras como eles são construídos para raciocinar, a arquitetura BDI e os sistemas multiagentes. O Capítulo 3 introduz as definições da sintaxe do AgentSpeak(L), o ciclo de raciocínio de um agente descrito utilizando a linguagem e as particularidades do Jason. O Capítulo 4 apresenta os analisadores léxico e sintático para a construção de interpretadores, bem como o conceito de unificação e substituição de variáveis. O Capítulo 5 contém os objetivos deste trabalho. O Capítulo 6 descreve como este trabalho será construído.

2. AGENTES

Um agente é um sistema computacional que *percebe* o *ambiente* em que se encontra e é capaz de realizar *ações* de forma *autônoma* para alcançar seus objetivos [21]. Os agentes percebem o ambiente através de sensores e realizam suas ações utilizando atuadores [15, p. 34], conforme é ilustrado na Figura 2.1.

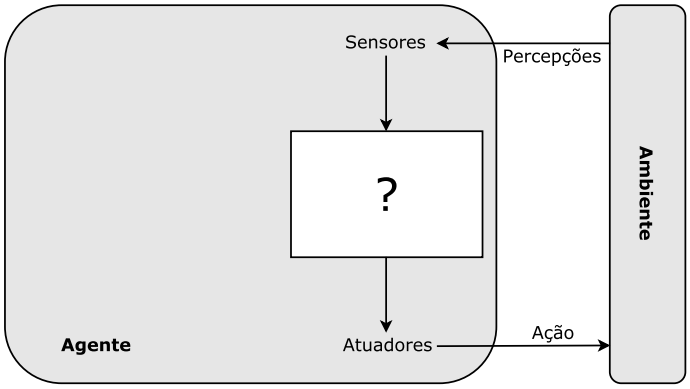

Figura 2.1 – Interação de um agente e seu ambiente [15].

Wooldridge e Jennings [21] também caracterizam agentes com as seguintes propriedades:

- Autonomia: operam sem a intervenção direta de humanos ou outros agentes e possuem controle total sobre suas ações e estado interno.
- Habilidade Social: interagem com outros agentes, humanos ou computacionais.
- Reatividade: percebem e reagem às alterações do ambiente em que estão inseridos.
- Pró-atividade: além de atuarem em resposta às alterações ocorridas em seu ambiente, apresentam um comportamento orientado a objetivos, tomando a iniciativa quando julgarem apropriado.

2.1 Ambiente

O ambiente em que o agente obtém as informações e realiza suas ações pode, segundo Russel e Norvig [15, p. 41], ser classificado de acordo com suas propriedades. O ambiente é totalmente observável quando o agente obtém informações completas e atualizadas dos estados do ambiente através de suas percepções. Em contrapartida, o ambiente é parcialmente observável quando os sensores não conseguem obter as informações destes estados devido a ocorrência de ruídos ou imprecisão nas informações.

O ambiente é determinístico quando é possível saber exatamente o próximo estado do ambiente, levando-se em conta o estado atual e a ação executada pelo agente. Caso contrário,

o ambiente é não-determinístico e, neste caso, não é possível saber com precisão o resultado de uma ação. O ambiente é estático quando não há mudanças nos estados do ambiente enquanto um agente está deliberando, apenas quando uma ação é executada por um agente. Enquanto que, um ambiente é dinâmico quando o agente precisa continuar observando os estados do ambiente enquanto ele está decidindo a ação que será realizada.

O ambiente é discreto quando existe um número contável de ações, percepções e estados do ambiente. Por sua vez, um ambiente é contínuo quando este número torna-se incontável. As propriedades dos ambientes são importantes para entendermos como os agentes devem ser projetados. Quanto mais complexo for o ambiente, mais complexo deve ser o agente.

2.2 Arquitetura

O comportamento de um agente é descrito por um programa que, através de processos internos, mapeia uma sequência de percepções em uma ação [15, p. 46]. A especificação de como são construídos estes processos internos é denominada de arquitetura do agente. De acordo com Maes [7, p. 115], uma arquitetura propõe como a construção de um agente pode ser dividida em um conjunto de componentes que interagem entre si para prover uma resposta de como as percepções e o estado interno do agente irão determinar sua ação.

Russel e Norvig [15, p. 47] destacam quatro tipos básicos de arquiteturas de agentes:

1. Agente Reativo Simples
2. Agente Reativo Baseado em Modelos
3. Agente Baseado em Objetivos
4. Agente Baseado em Utilidade

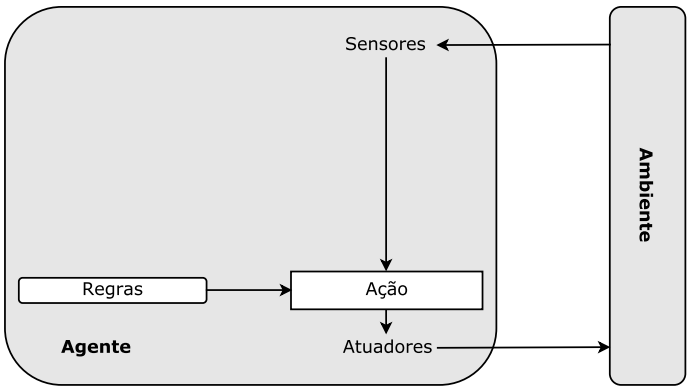

Figura 2.2 – Agente reativo simples [15].

O agente reativo simples executa suas ações baseado apenas na percepção atual, ignorando todo o histórico de percepções anteriores. Isso torna a implementação desse tipo de agente mais

fácil, pois o seu programa é composto por uma série de estruturas condicionais do tipo *if-then-else* que contém as regras para traduzir as percepções em uma ação, conforme ilustra a Figura 2.2. No entanto, um comportamento mais complexo pode gerar uma implementação mais extensa.

Diferentemente do agente reativo simples, o agente baseado em modelos possui um estado interno que depende do histórico de percepções e que reflita os aspectos não observados no estado atual. As informações armazenadas neste estado interno são utilizadas no processo de escolha da ação de forma complementar às percepções, permitindo assim, lidar com ambientes parcialmente observáveis. A Figura 2.3 ilustra como a percepção atual é combinada com o estado interno antigo para gerar a descrição atualizada do estado atual.

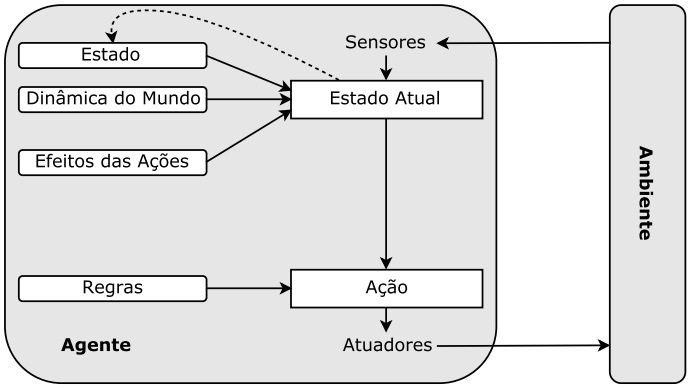

Figura 2.3 – Agente reativo baseado em modelos [15].

O agente baseado em objetivos utiliza informações de estados em que o agente deseja chegar e, combinado com informações sobre os resultados de ações possíveis, escolhe as ações necessárias para alcançar o seu objetivo, conforme ilustrado na Figura 2.4.

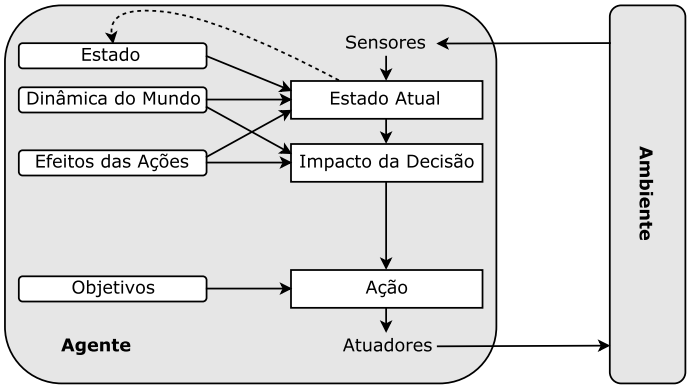

Figura 2.4 – Agente baseado em objetivos [15].

Objetivos isolados não são suficientes para gerar comportamentos adequados na maioria dos ambientes pois criam uma distinção binária dos estados. Diante disso, o agente baseado em utilidade mapeia o quão bom um estado é em um número real para que, posteriormente, realize decisões racionais nos casos em que houver objetivos conflitantes ou quando existirem muitos objetivos pretendidos pelo agente, conforme mostra a Figura 2.5.

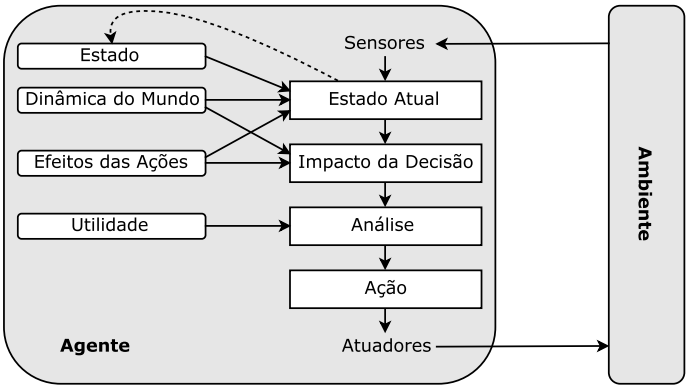

Figura 2.5 – Agente baseado em utilidade [15].

Entender o comportamento de cada uma das arquiteturas é importante no momento da escolha do modelo mais adequado para a resolução de um problema.

2.3 Belief-Desire-Intention

Uma das abordagens mais comuns para a construção de agentes baseados em objetivos é através de estados mentais semelhantes ao raciocínio prático humano como crenças, desejos e intenções. Este modelo de raciocínio é conhecido como *Belief-Desire-Intention* (BDI).

De acordo com Wooldridge [19, p. 7], as crenças, ou *beliefs* correspondem ao conhecimento que o agente possui em relação ao ambiente em que se encontra. Um agente pode ter crenças sobre o mundo, sobre outros agentes, sobre interações com outros agentes e crenças sobre suas próprias crenças, podendo ser, inclusive, incompletas ou incorretas. Os desejos, ou *desires*, representam os estados objetivos que o agente deseja alcançar, motivando o agente a agir de forma a realizar estes objetivos. Enquanto que as intenções, ou *intentions*, são os desejos que o agente está comprometido em alcançar e determinam o processo de raciocínio prático, o qual leva o agente à ação.

O processo de raciocínio em um agente BDI genérico é ilustrado na Figura 2.6 e possui 7 componentes principais. O conjunto de crenças contém as informações que o agente possui sobre o seu ambiente atual. A função de revisão de crenças determina um novo conjunto de crenças a partir da entrada percebida e das crenças atuais do agente. A função de geração de opções determina as opções disponíveis para o agente, tendo como base suas crenças sobre seu ambiente e suas intenções atuais. O conjunto de desejos contém as ações possíveis disponíveis para o agente. A função de filtro representa o processo de raciocínio prático do agente e determina as suas intenções, tendo como base suas crenças, desejos e intenções atuais. O conjunto de intenções atuais representa o subconjunto de desejos com os quais o agente se comprometeu a alcançar. Por fim, a função de seleção de ações determina uma ação que será executada, tendo como base as intenções atuais [18, p. 31].

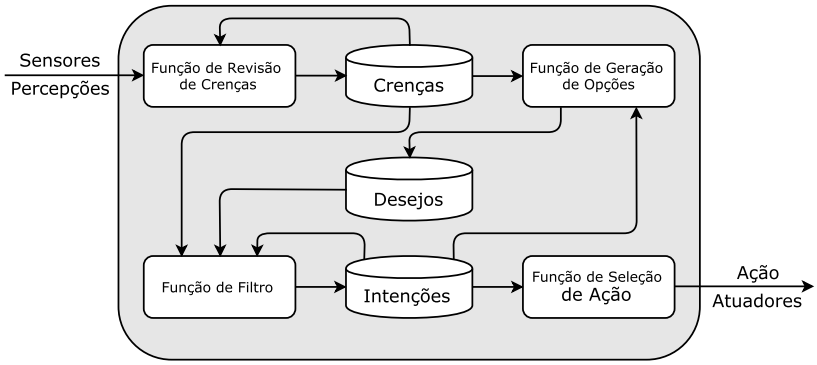

Figura 2.6 – Diagrama de uma arquitetura BDI genérica [19].

O processo de raciocínio da arquitetura BDI, descrito nessa seção, é importante pois constitui a base para as implementações que serão apresentadas a seguir.

2.3.1 Implementações da arquitetura BDI

Existem inúmeras ferramentas e linguagens que permitem o desenvolvimento de agentes utilizando a arquitetura BDI [8]. Abaixo, são apresentadas algumas destas ferramentas, que foram utilizadas como base para a criação da linguagem AgentSpeak(L), tema deste trabalho.

Procedural Reasoning System

O ***P****rocedural* ***R****easoning* ***S****ystem* (PRS) é uma arquitetura BDI genérica para representação e raciocínio de ações e procedimentos em ambientes dinâmicos. O PRS, desenvolvido em LISP pelo *Stanford Research Institute International*, foi inicialmente utilizado no sistema de controle reativo de viagens espaciais da *National Aeronautics & Space Administration* (NASA).

Segundo Georgeff e Lansky [6], a arquitetura do PRS pode ser subdividida em componentes periféricos e de raciocínio, conforme ilustrado na Figura 2.7. Os componentes periféricos atuam diretamente no ambiente e correspondem aos sensores; ao módulo monitor que traduz informações dos sensores em crenças para o agente; ao módulo gerador de comando que traduz ações primitivas em comandos para os atuadores; e aos atuadores em si.

Por sua vez, os componentes que realizam o raciocínio correspondem ao banco de dados, com as crenças atuais e fatos sobre o mundo, expressos em lógica de primeira ordem; à biblioteca de planos, com as representações simbólicas explicitas de crenças, desejos e intenções; ao conjunto de objetivos atual, com os desejos que são consistentes e que podem ser atingidos; à estrutura de intenção, com uma pilha dos planos selecionados em tempo de execução; e, por fim, ao interpretador, que gerencia o sistema, selecionando os planos aptos em resposta aos objetivos e crenças do sistema.

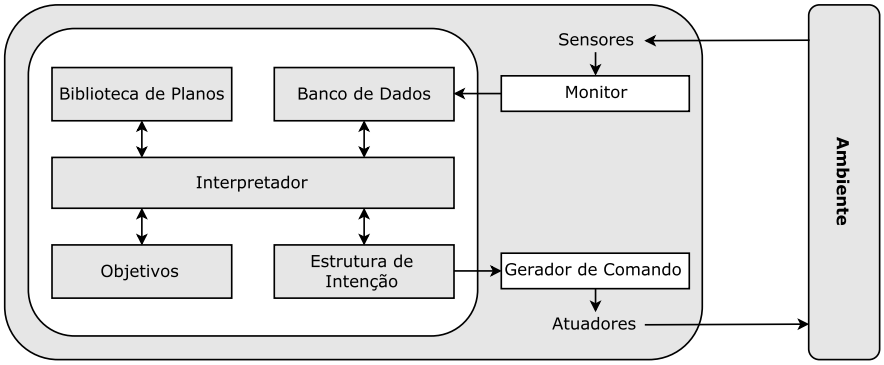

Figura 2.7 – Arquitetura do PRS [6].

Distributed Multi-Agent Reasoning System

O ***D**istributed **M**ulti-**A**gent **R**easoning **S**ystem* (dMARS) é uma evolução da arquitetura PRS para construção de sistemas de domínios dinâmicos onde há conhecimento incerto e complexo. O dMARS foi desenvolvido em C++ pelo *Australian Artificial Intelligence Institute* como uma ferramenta comercial voltada para aplicações nas áreas de telecomunicações, viagens espaciais e tráfego aéreo.

Segundo d'Inverno *et al.* [5], os agentes utilizam a arquitetura BDI através de uma biblioteca de planos, que especifica os cursos de ação que podem ser tomados pelo agente para realizar suas intenções. Cada plano contém um conjunto de componentes que correspondem ao gatilho, que descreve as circunstâncias em que o plano pode ser considerado, geralmente em termos de eventos; ao contexto, ou pré-condição, especifica as circunstâncias iniciais para a execução de um plano; à condição de manutenção caracteriza as circunstâncias que devem permanecer verdadeiras enquanto o plano é executado; e, por fim, ao corpo com objetivos e ações primitivas, que define o curso de ação.

O interpretador é o responsável por manter a operação do agente e executar, continuamente, o seguinte ciclo:

- Observa o mundo e o estado interno do agente. Em seguida, atualiza a fila de eventos para refletir os novos eventos observados.
- Gera novos eventos possíveis (tarefas), a partir de planos encontrados com eventos compatíveis na lista de eventos.
- Seleciona, a partir do conjunto de planos compatíveis, um evento para execução.
- Adiciona o evento escolhido em uma pilha nova ou já existente de intenção, dependendo se o evento é ou não um subobjetivo.
- Seleciona uma pilha de intenção, pega o primeiro plano da pilha e executa o próximo passo do plano atual: se o passo for uma ação, executa esta ação; se for um subobjetivo, envia este para a fila de eventos.

A Figura 2.8 ilustra as interações dos componentes do dMARS descritos acima.

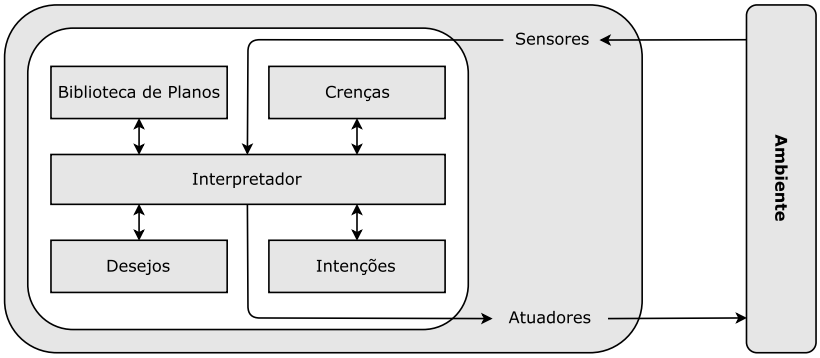

Figura 2.8 – Arquitetura do dMARS [5].

JADE Extension

O ***JADE** e**X**tension* (JADEX) é um *framework* para implementação de agentes utilizando a arquitetura BDI e a plataforma JADE. O JADEX permite a criação de agentes através da descrição dos elementos do BDI em arquivos XML e da utilização da linguagem de programação orientado a objetos *Java*. Segundo Baubach *et al.* [11], os principais componentes do JADEX são:

- Crenças: representam o conhecimento do agente sobre o seu ambiente e sobre si mesmo. Estes conhecimentos são armazenadas em uma base de crenças através de objetos *Java* contendo os fatos percebidos.
- Objetivos: representam os desejos concretos e momentâneos de um agente. Para qualquer objetivo que o agente possua, ele irá engajar-se em ações apropriadas até que ele considere o objetivo como atingido, inatingível ou não mais desejado.
- Planos: representam as ações que um agente irá realizar no ambiente a partir de uma biblioteca de planos. Um plano é composto de um cabeçalho, que especifica as circunstâncias iniciais para a sua execução, e um corpo contendo a sequência de ações a serem realizadas.
- Eventos: representam todas as ocorrências dentro de um agente.
- Capacidade: permite que crenças, objetivos, planos e eventos sejam encapsulados em um módulo reutilizável.

2.4 Sistemas Multiagente

A utilização, na prática, de sistemas com um único agente são raros. Os casos mais comuns são os de agentes convivendo em um ambiente com outros agentes, criando um sistema multiagente [3, p. 5].

Um MAS contém agentes conectados por algum tipo de relacionamento organizacional, e interagem entre si através de algum mecanismo de comunicação. Cada agente possui uma esfera de influência sobre o ambiente em que compartilha com os demais agentes. Estas esferas referem-se à parte do ambiente no qual um agente pode ter o controle parcial ou total e podem coincidir com as de outros agentes [20, p. 105].

O fato das esferas de influência de dois ou mais agentes coincidirem torna a escolha de ações destes agentes mais complicada, pois para que um deles alcance seu objetivo neste ambiente compartilhado é necessário levar em consideração as ações dos demais agentes [3, p. 5]. A Figura 2.9 demonstra as interações entre os agentes e suas esferas de influência no ambiente.

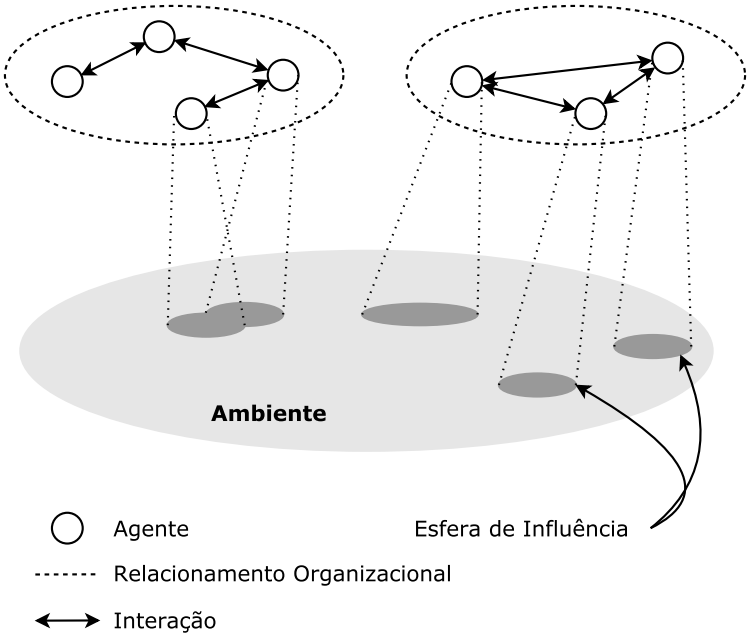

Figura 2.9 – Estrutura típica de um sistema multiagente [3].

3. AGENTSPEAK(L)

O AgentSpeak(L) é uma linguagem de programação orientada a agentes baseada em lógica de primeira ordem, com eventos e ações. Rao [12] utilizou como base implementações da arquitetura BDI, como o PRS e o dMARS, para formalizar sua semântica operacional em uma linguagem abstrata de programação. Segundo o autor, o desenvolvedor utiliza a notação do AgentSpeak(L) para descrever o estado atual do agente, de seu ambiente e de outros agentes, (crenças); os estados que o agente deseja atingir, baseado em estímulos externos e internos (desejos); e os planos que satisfazem um dado estímulo (intenções).

3.1 Sintaxe

Um programa desenvolvido em AgentSpeak(L) é especificado por um conjunto de crenças, planos, eventos ativadores e um conjunto de ações básicas que o agente executa no ambiente. Rao [12] introduz as noções básicas para especificação desses conjuntos a partir das definições descritas a seguir.

Para um melhor entendimento das definições, o autor apresenta como exemplo uma rodovia que contém 4 faixas adjacentes e carros que podem trafegar, de sul a norte, em qualquer uma das faixas, conforme ilustrado na Figura 3.1. Quando o lixo jogado pelos motoristas aparece em uma das faixas, um robô recolhe os resíduos e os deposita em uma lixeira. Por motivos de segurança, um robô não pode estar na mesma faixa que um carro.

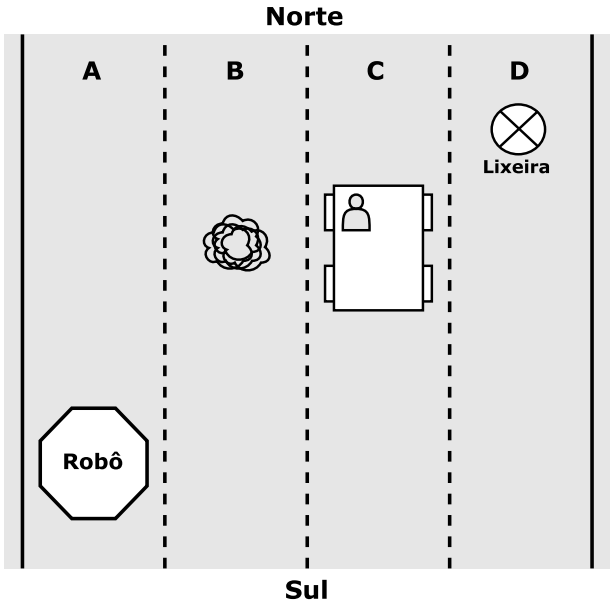

Figura 3.1 – Rodovia com faixas para limpeza de lixo [12].

Definição 1 *Se b é um símbolo de predicado e $t_1, \ldots, t_n$ são termos, então $b(t_1, \ldots, t_n)$ ou $b(\vec{t})$ é um átomo de crença, ou belief atom. Uma crença, ou belief, é uma instância de um átomo de crença, enquanto que um conjunto de crenças é uma coleção de crenças.*

A Definição 1 permite relacionar os átomos de crença com as configurações das faixas e localizações do robô, dos carros, do lixo e da lixeira, como por exemplo, `adjacente(X, Y)`, `localizacao(robo, X)`, `localizacao(lixeira, X)` e `localizacao(carro, X)`. As crenças, neste caso, são as instâncias `adjacente(a, b)`, `adjacente(b, c)`, `adjacente(c, d)`, `localizacao(robo, a)`, `localizacao(lixeira, d)` e `localizacao(carro, c)`.

Em AgentSpeak(L) são utilizadas as notações $\&$ para $\wedge$, *not* para $\neg$, e $<-$ para $\leftarrow$. Por convenção, variáveis são escritas em letra maiúscula e constantes em letra minúscula.

Definição 2 *Se g é um símbolo de predicado e $t_1, \ldots, t_n$ são termos, então $!g(t_1, \ldots, t_n)$ ou $!g((\vec{t})$ são objetivos de realização, ou achievement goal; e $?g(t_1, \ldots, t_n)$ ou $?g(\vec{t})$ são objetivos de teste, ou test goal.*

Os objetivos de realização expressam que o agente quer alcançar um estado no ambiente onde o predicado associado ao objetivo é verdadeiro, e são identificados pelo prefixo '!'. Os objetivos de teste, representados pelo prefixo '?', são aqueles que o agente quer testar sempre que o predicado associado é uma crença verdadeira.

No exemplo da rodovia, a limpeza da faixa "b" pode ser alcançada pelo objetivo `!limpar(b)`. Enquanto que, para verificar se o carro está na faixa "c", é validada a crença `?localizacao(carro, c)`.

Definição 3 *Se $b(\vec{t})$ é um átomo de crença, e $!g(\vec{t})$ e $?g(\vec{t})$ são objetivos, então $+b(\vec{t})$, $-b(\vec{t})$, $+!g(\vec{t})$, $+?g(\vec{t})$, $-!g(\vec{t})$ e $-?g(\vec{t})$ são eventos ativadores.*

Um evento ativador define quais eventos podem iniciar a execução de um plano, e podem ser de dois tipos: interno ou externo. O primeiro ocorre quando o evento é gerado pela execução de um plano, enquanto que o segundo é gerado pelas atualizações de crenças que resultam da percepção do ambiente. Os eventos, internos ou externos, são adicionados no conjunto de eventos E, apresentado na Definição 8. Eventos ativadores são relacionados com a adição e remoção de estados mentais, como crenças e objetivos, e são representados pelos prefixos '$+$' e '-'.

Utilizando o exemplo apresentado por Rao, `+localizacao(lixo, X)` cria uma crença que existe lixo em uma faixa `X` e `+!limpar(X)` cria um objetivo para limpar a faixa `X`.

Definição 4 *Se a é um símbolo de ação e $t_1, \ldots, t_n$ são termos de primeira ordem, então $a(t_1, \ldots, t_n)$ ou $a(\vec{t})$ é uma ação.*

Por exemplo, se `mover` é um símbolo de ação, `mover(X, Y)` é uma ação que resulta em um estado do ambiente em que o robô não está mais na faixa `X` e está na faixa `Y`.

Definição 5 *Se e é um evento ativador, $b_1, \ldots, b_m$ são crenças, e $h_1, \ldots, h_n$ são objetivos ou ações, então $e : b_1 \wedge \cdots \wedge b_m \leftarrow h_1; \ldots; h_n$ é um plano.*

Planos fazem referências a ações básicas que um agente é capaz de executar em seu ambiente. Um plano é formado por um cabeçalho, representado por um evento ativador e um contexto separados por ':', e um corpo, conforme a estrutura abaixo:

```
evento_ativador :  contexto <- corpo.
```

O evento ativador especifica as circunstâncias em que um plano deve ser executado, isto é, a adição ou remoção de uma crença ou objetivo. O contexto de um plano especifica as crenças que devem estar no conjunto de crenças de um agente quando o plano é ativado. O corpo do plano é a sequencia de objetivos que o agente deve executar ou testar, e as ações que o agente deve executar. Caso o corpo do plano esteja vazio, devemos utilizar a expressão `true`.

Ainda de acordo com o nosso exemplo, criaremos um plano que é ativado quando o lixo aparece em uma das faixas. Se o robô está na mesma faixa que o lixo, será realizada a ação `pegar` o lixo, seguida pelo o objetivo de mudar sua localização para a mesma faixa da lixeira e de realizar a ação de `jogar` o lixo na lixeira, conforme o plano do Código 3.1.

```
1  +localizacao(lixo, X) : localizacao(robo, X) &
2                          localizacao(lixeira, Y)
3                        <- pegar(lixo);
4                           !localizacao(robo, Y);
5                           jogar(lixo, lixeira).
```

Código 3.1 – Plano P1.

Por sua vez, a criação de um plano para que o robô mude de faixa, adquirindo um objetivo para mover para uma localização `X`, depende de 2 cenários. No primeiro, o robô já está na localização `X` e não precisa realizar nenhuma ação. Nesse caso, o corpo do plano para esse evento possui valor verdadeiro (`true`), conforme descrito no Código 3.2.

```
1  +!localizacao(robo, X) : localizacao(robo, X) <- true.
```

Código 3.2 – Plano P2.

No segundo cenário, o robô encontra-se em uma faixa diferente da localização `X`. Nesse caso, o robô precisa encontrar a faixa adjacente que não possua carros e, em seguida, mover-se para ela, conforme o plano do Código 3.3.

```
1   +!localizacao(robo, X) : localizacao(robo, Y) &
2                            (not (X = Y)) &
3                            adjacente(Y, Z) &
4                            (not (localizacao(carro, Z)))
5                          <- mover(Y, Z);
6                             +!localizacao(robo, X).
```

Código 3.3 – Plano P3.

Definição 6 *Um agente é dado pela tupla* $\langle E, B, P, I, A, S_E, S_O, S_I \rangle$.

Os elementos da tupla da Definição 6 correspondem, respectivamente, a um conjunto de eventos E, um conjunto de crenças B, um conjunto de planos P, um conjunto de intenções I, e a um conjunto de ações A. As funções de seleção S_E, S_O e S_I selecionam, respectivamente, um evento do conjunto E, um plano aplicável do conjunto de planos aplicáveis, e uma intenção do conjunto I.

Definição 7 *O conjunto* I *é uma coleção de intenções com uma pilha de planos parcialmente instanciados. Uma intenção é representada por* $[p_1 \ddagger ... \ddagger p_z]$*, onde* p_1 *é o último elemento da pilha, enquanto que* p_z *é o primeiro. Os demais elementos da pilha são representados pelo símbolo* $\ddagger$.

Definição 8 *O conjunto* E *é uma coleção de eventos com uma tupla* $\langle e, i \rangle$*, onde* e *é um evento ativador e* i *é uma intenção. Um evento externo não é gerado a partir de uma intenção, por isso, utiliza-se nestes casos uma intenção verdadeira* $(+!true : true < -true)$.

As definições descritas abaixo abordam noções de relevância e aplicabilidade de planos. Para um melhor entendimento, é necessário compreender as noções de unificação e substituição de variáveis, apresentadas na Seção 4.3.

Definição 9 *A função* S_E *seleciona um evento do conjunto de eventos* E *e, em seguida, define o conjunto de planos relevantes. Um plano* p *é relevante em relação ao evento* $_E$ *quando existe um Unificador Mais Geral, ou Most General Unifier (MGU), denominado unificador relevante, entre os eventos ativadores contidos no cabeçalho dos planos do conjunto de planos* P.

Por exemplo, assumindo que o evento ativador do evento selecionado do conjunto E é `+!localizacao(robo, b)`. Os planos dos Códigos 3.2 e 3.3 da Definição 5 são relevantes para o evento a partir do unificador relevante `{X/b}`.

Definição 10 *Após a definição do conjunto de planos relevantes, a função S_O seleciona o conjunto de planos aplicáveis. Um plano p é aplicável em relação a um evento $_E$ quando existe uma substituição que, ao ser combinada com o unificador relevante e o contexto do plano, é uma consequência lógica do conjunto de crenças B.*

Esta combinação entre o unificador relevante e a substituição é denominada *unificação aplicável*. O conjunto de planos aplicáveis corresponde aos planos que podem ser utilizados, no contexto atual, para tratar o evento selecionado no ciclo atual.

No mesmo exemplo, considerando o conjunto de crenças `adjacente(a, b)`, `adjacente(b, c)`, `adjacente(c, d)`, `localizacao(robo, a)`, `localizacao(lixo, b)` e `localizacao(lixeira, d)`, a única unificação aplicável é `{X/b, Y/a, Z/b}` referente ao plano do Código 3.3, portanto, este plano é aplicável.

Definição 11 *Se o evento $_E$ escolhido na Definição 9 for externo, a função de seleção S_O escolhe, entre os planos do conjunto de planos aplicáveis, um único plano pretendido. O plano p é um plano pretendido em relação a um evento $_E$ quando existe uma unificação aplicável para o corpo do plano, de modo que $+!true : true \leftarrow true \ddagger (e : b_1 \wedge \cdots \wedge b_m \leftarrow h_1; \ldots; h_n)$. O plano pretendido é usado para criar uma nova intenção que é adicionada no conjunto de intenções I.*

Seguindo o exemplo do autor, o único plano aplicável é o do Código 3.3. Desta forma, o conjunto de intenções I será representado por uma única intenção I.

```
1   [+!localizacao(robo, b) : localizacao(robo, a) &
2                             not (b = a) &
3                             adjacente(a, b) &
4                             not (localizacao(carro, b))
5                           <- mover(a, b);
6                              +!localizacao(robo, b)].
```

Código 3.4 – Conjunto de intenções I.

Definição 12 *No caso do evento $_E$ ser interno, a função de seleção S_O coloca o plano pretendido, para o objetivo recém alcançado, no topo da intenção existente que ativou o evento interno. O plano p é um plano pretendido em relação a um evento $_E$ quando existe uma unificação aplicável para o corpo do plano.*

Definição 13 *A função S_I seleciona e executa uma intenção i contida no topo do conjunto de intenções I. Se a fórmula no corpo de i for um objetivo de realização, um evento do tipo $\langle +!g(\vec{t}), i \rangle$ é adicionado no conjunto de eventos e a intenção que gerou o evento é considerada executada.*

Definição 14 *No caso da fórmula do corpo da intenção i ser um objetivo de teste, o conjunto de crenças é percorrido para encontrar um átomo de crença que unifique com o predicado de teste. Se encontrado, o objetivo é removido do conjunto de intenções, pois foi realizado.*

Definição 15 *Se a fórmula no corpo da intenção i for uma ação a ser realizada pelo agente no ambiente, o interpretador atualiza o estado do ambiente com a ação requerida e remove a ação do conjunto de intenções.*

Definição 16 *Quando todas as fórmulas no corpo de um plano forem executadas, o plano é removido da intenção, assim como o objetivo de realização que o gerou.*

No exemplo, ao executar a intenção I, iremos adicionar a ação `mover(a, b)` no conjunto de ações A e modificar I para remover a ação, conforme abaixo:

```
1   [+!localizacao(robo, b) : localizacao(robo, a) &
2                             not (b = a) &
3                             adjacente(a, b) &
4                             not (localizacao(carro, b))
5                           <- +!localizacao(robo, b)].
```

Código 3.5 – Atualização da intenção após realização da ação.

De acordo com Rao [12], na iteração seguinte, após o robô se mover de `a` para `b`, o ambiente irá enviar para o agente um evento de atualização de crenças para atualizar a localização do robô para `b`. Esta atualização irá adicionar a crença `localizacao(robo, b)` no conjunto de crenças B e o evento `+localizacao(robo, b)` no conjunto de eventos E. Como o evento `+localizacao(robo, b)` não possui um MGU com nenhum cabeçalho entre os planos do conjunto de planos P, o interpretador irá executar o objetivo remanescente da intenção I. Esta execução irá gerar um evento interno que será adicionado no conjunto de eventos E. Neste caso, o conjunto de eventos E possui apenas um único elemento, a tupla $\langle +!\text{localizacao(robo, b)}, i \rangle$. Pela Definição 12, o único plano relevante para este evento é o plano do Código 3.2, cujo unificador relevante é `{X/b}`. Este plano também é aplicável, possuindo `{X/b}` como unificador aplicável. Como o corpo do plano é verdadeiro (`true`), a intenção é satisfeita e o conjunto de eventos E torna-se vazio. Desta forma, a execução é finalizada até que um novo evento seja adicionado em E.

3.2 Ciclo de Raciocínio

As definições da semântica do AgentSpeak(L) descritas na Seção 3.1 permitem elaborar uma sequência de passos que representa o raciocínio de um agente a cada ciclo, conforme ilustrado

na Figura 3.2. No início de cada ciclo, o agente recebe as informações provenientes do ambiente e as confronta com o seu conjunto de crenças. Caso as percepções do ambiente sejam diferentes, o conjunto de crenças é atualizado para que elas reflitam o novo estado do ambiente. Cada crença modificada gera um novo evento que é adicionado no conjunto de eventos, conforme a Definição 3. A atualização dos conjuntos de crenças e eventos é feita através da Função de Revisão de Crenças, ou *Belief Revision Function* (BRF), e é identificada na imagem pelo módulo BRF.

Em seguida, a função de seleção S_E escolhe um único evento do conjunto de eventos E. A partir desse evento, é realizada a unificação dos eventos ativadores contidos no cabeçalho dos planos do conjunto de planos P para encontrar os planos relevantes, conforme a Definição 9. Esta etapa é identificada na figura pelos módulos S_E e *Unify Event*. Utilizando os planos relevantes, é realizada a substituição do contexto de cada plano com o conjunto de crenças B com o objetivo de encontrar os planos aplicáveis. Esta etapa é identificada na imagem pelo módulo *Unify Context* e refere-se a Definição 10. Então, a função de seleção S_O opta por um único plano pretendido que irá atualizar o conjunto de intenções I, identificado pelo módulo S_O na imagem, conforme as Definições 11 e 12.

Ao final, a função S_I seleciona a intenção contida no topo do conjunto de intenções I. O ciclo finaliza após a execução da fórmula do corpo da intenção, conforme as Definições 13, 14, 15 e 16. As etapas finais do ciclo de raciocínio do AgentSepak(L) são identificadas na imagem pelos módulos S_I e *Execute Intention*.

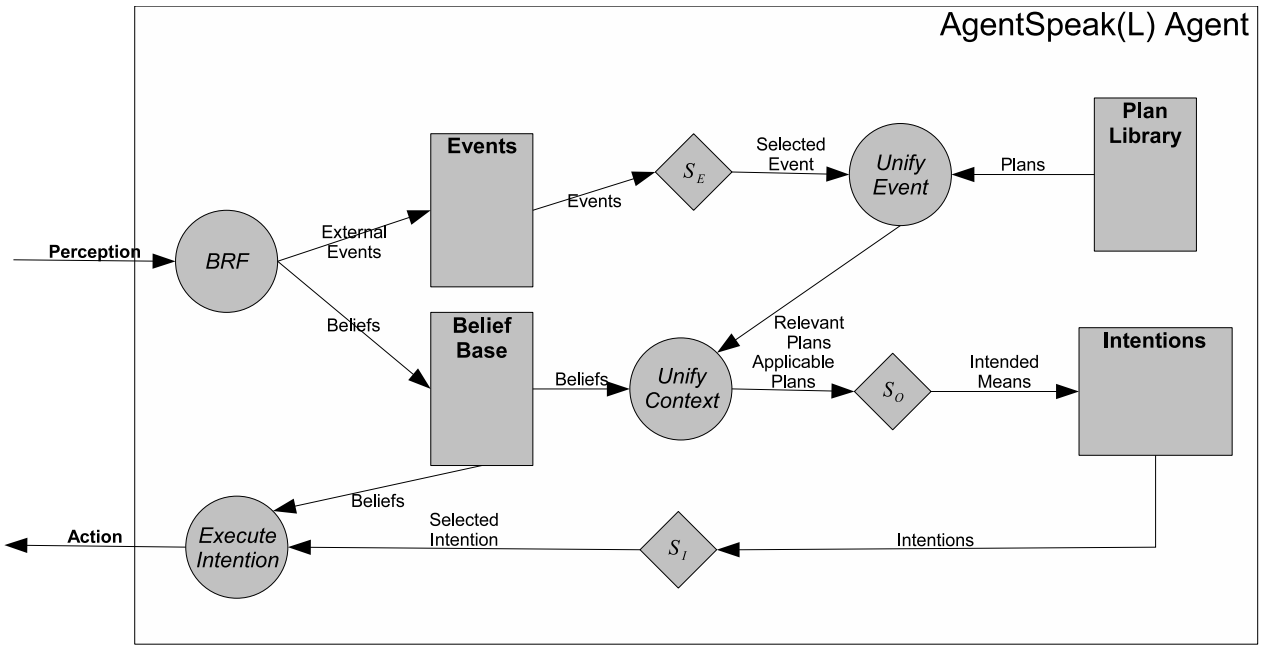

Figura 3.2 – Ciclo de raciocínio do AgentSpeak(L) [10].

3.3 Implementações do AgentSpeak(L)

Por ser uma linguagem abstrata de programação, o AgentSpeak(L) precisa de uma implementação de um compilador ou interpretador para ser utilizado. Uma das ferramentas que surgiram para a construção de agentes utilizando o AgentSpeak(L) foi o JASON.

3.3.1 JASON

O ***J**ava-based **A**gentSpeak interpreter used with **S**ACI for multi-agent distribution **O**ver the **N**et*[1] (JASON) é um interpretador de uma versão estendida do AgentSpeak(L). O JASON foi desenvolvido em Java e está disponível em código aberto sob a licença GNU LGPL[2]. Segundo Bordini e Hübner [4], as funcionalidades disponíveis no JASON, além daquelas definidas no AgentSpeak(L), são:

- Anotações nas crenças sobre as fontes de informação, como por exemplo a anotação `[source (self)]` proveniente de um evento interno.
- Anotações nos rótulos dos planos para serem utilizados em funções de seleção mais elaboradas, como por exemplo a anotação `[source(paranoico)]` do Código 3.8.
- Possibilita a customização, utilizando a linguagem de programação Java, das percepções, da função de atualização de crença e das funções de seleção de mensagem, evento, planos aplicáveis e intenções de um agente, conforme será apresentado na Seção 3.3.1.
- Permite a implementação do ambiente utilizando a linguagem de programação Java, conforme será apresentado na Seção 3.3.1.

Ciclo de Raciocínio

O ciclo de raciocínio de um agente no JASON segue, em grande parte, aquele descrito para o AgentSpeak(L) na Seção 3.2. A semelhança entre o fluxo ilustrado nas Figuras 3.2 e 3.3 permite identificar, de forma visual, as etapas propostas por Rao [12] que foram implementadas no interpretador. As diferenças encontram-se, principalmente, nas funcionalidades *checkMail*, S_M, *SocAcc* e *sendMsg*, criadas para a troca de mensagens entre os agentes.

As mensagens enviadas por outros agentes são tratadas, de acordo com Bordini *et al.* [3, p. 71], pela função *checkMail*, identificada na Figura 3.3 pelo rótulo 3. Em seguida, a função de seleção S_M escolhe uma única mensagem para ser processada no ciclo atual, enquanto que a função *SocAcc* verifica se a mensagem pode ser aceita pelo agente. Segundo o autor, esta função

[1] http://jason.sourceforge.net/faq/
[2] https://www.gnu.org/licenses/lgpl-3.0.en.html

é identificada pelo rótulo 4 e, normalmente, é customizada para cada um dos agentes. Por fim, quando a intenção selecionada na função S_I for uma função `.send`, a função *sendMsg* envia uma mensagem para outros agentes.

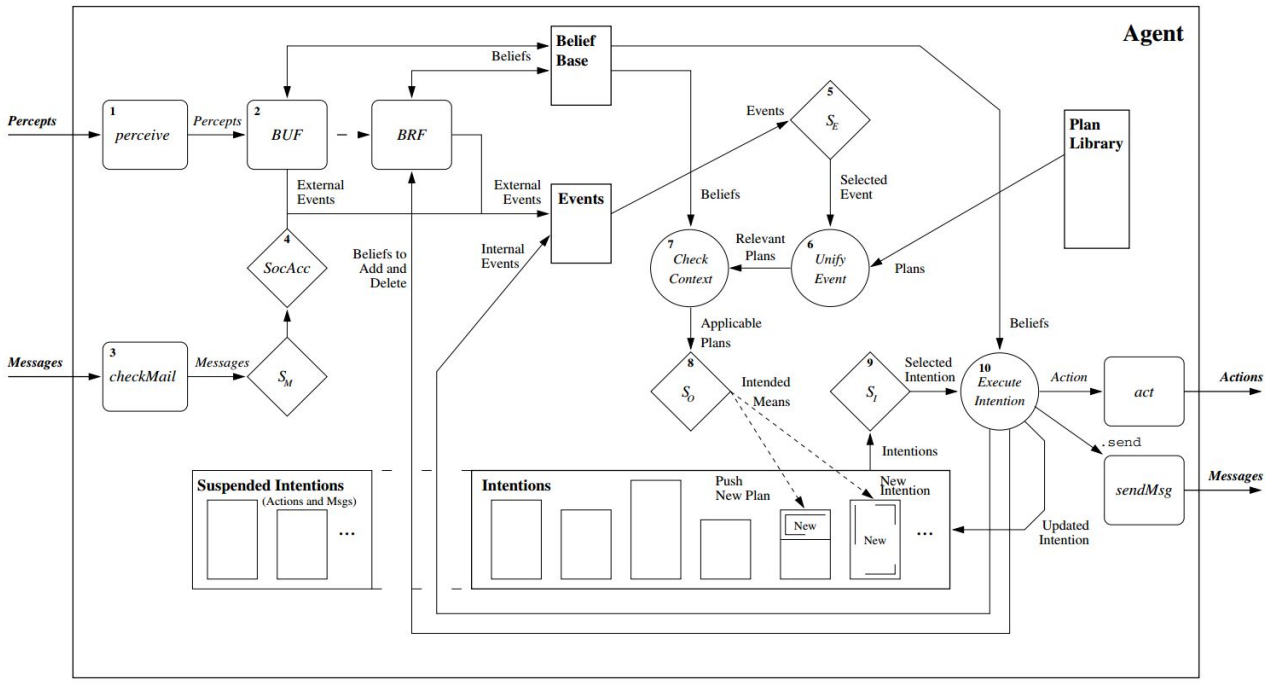

Figura 3.3 – Ciclo de raciocínio do JASON [3].

Como exemplo [3, p. 115], o autor apresenta um ambiente representado por uma sala que possui apenas uma porta. A sala contém 3 agentes: um claustrofóbico que insiste em manter a porta destrancada, um paranoico que quer a porta trancada e o porteiro, que realiza as ações no ambiente de acordo com as solicitações enviadas pelos agentes. O agente paranoico, ao descobrir que a porta não está trancada, através do evento `+~trancada(porta)`, deve solicitar que ela seja trancada ao enviar para o porteiro um novo objetivo de realização, conforme Código 3.6.

```
1   +~trancada(porta) : true
2                     <- .send(porteiro, achieve, trancada(porta)).
```

Código 3.6 – Código-fonte em AgentSpeak(XL) do agente paranoico.

Por sua vez, o agente claustrofóbico, ao descobrir que a porta está trancada, através do evento `+trancada(porta)`, deve solicitar que ela seja destrancada ao enviar para o porteiro um novo objetivo de realização, de acordo Código 3.6.

```
1   +trancada(porta) : true
2                    <- .send(porteiro, achieve, ~trancada(porta)).
```

Código 3.7 – Código-fonte em AgentSpeak(XL) do agente claustrofóbico.

O agente porteiro, ao receber do paranoico o objetivo de trancar a porta com ela destrancada, realiza a ação de trancá-la. Se o objetivo proveniente do claustrofóbico for o de não trancar a porta e ela estiver trancada, realiza a ação de destrancá-la, conforme Código 3.8.

```
1  +!trancada(porta)[source(paranoico)]
2                   :  ~trancada(porta)
3                   <- trancar.
4
5
6  +!~trancada(porta)[source(claustrofobico)]
7                    :  trancada(porta)
8                    <- destrancar.
```

Código 3.8 – Código-fonte em AgentSpeak(XL) do agente porteiro.

Descrição do Ambiente

O conceito de agentes apresentado na Seção 2 envolve a percepção e a realização de ações em um ambiente por parte do agente. Apesar de trazer diversas definições de como descrever o comportamento de um agente, o AgentSpeak(L) deixa a cargo do interpretador como serão feitas estas interações com o ambiente.

De acordo com Bordini *et al.* [3, p. 101], o JASON permite a implementação do ambiente que irá interagir com os agentes utilizando a linguagem de programação Java. A partir da extensão da classe `Environment` provida pelo interpretador, é possível definir as percepções iniciais dos agentes, a alteração nos estados do ambiente geradas a partir da realização de uma ação por parte de um agente e a atualização das percepções dos agentes.

Conforme o exemplo descrito acima, é possível criar o ambiente "sala" estendendo a classe pai, `Environment`, conforme a linha 1 do Código 3.9. A porta da sala pode ter 2 estados possíveis: trancada e destrancada; que são representados no ambiente pela variável *booleana* `portaTrancada`, de acordo com a linha 4. Os agentes percebem esses estados através do literal `trancada(porta)` e sua negação, respectivamente. As percepções iniciais dos agentes, linhas 6, 7, 8 e 9, são implementadas sobrescrevendo o método `init()` da classe pai. Neste caso, o programa inicia inserindo, para todos os agentes, a percepção `trancada(porta)` através da função `addPercept()`.

Por sua vez, as ações realizadas pelos agentes que modificam os estados do ambiente são implementadas sobrescrevendo o método `executeAction()` da classe pai. Nele, é utilizada a função `act.getFunctor().equals()` para identificar a ação e atualizar o estado desejado do ambiente . Em seguida, as percepções dos agentes são atualizadas de acordo com os novos estados, conforme demonstrado entre as linhas 11 e 30.

```
1 public class AmbienteSala extends Environment {
2     Literal literalTrancada  = Literal.parseLiteral("trancada(porta)");
3     Literal literalDestrancada = Literal.parseLiteral("~trancada(porta)");
4     boolean portaTrancada = true;
5
6     @Override
7     public void init(String[] args) {
8       addPercept(literalTrancada);
9     }
10
11    @Override
12    public boolean executeAction(String ag, Structure act) {
13        // Limpa as percepções dos agentes
14        clearPercepts();
15        // Ação de trancar a porta pelo agente paranoico
16        if (act.getFunctor().equals("trancar")) {
17            portaTrancada = true;
18        }
19        // Ação de destrancar a porta pelo agente claustrofóbico
20        if (act.getFunctor().equals("destrancar")) {
21            portaTrancada = false;
22        }
23        // Atualiza as percepções de acordo com os estados do ambiente
24        if (portaTrancada) {
25            addPercept(literalTrancada);
26        } else {
27            addPercept(literalDestrancada);
28        }
29        return true;
30    }
31 }
```

Código 3.9 – Código-fonte em Java para a descrição do ambiente em JASON.

4. INTERPRETADORES

Um interpretador é definido por Aho *et al.* [1, p. 2] como um programa que possui como entrada um outro programa escrito em uma linguagem-fonte e executa suas operações de forma direta, instrução por instrução. De acordo com os autores, os interpretadores possuem uma sequência de passos que analisam e criam uma representação intermediária da linguagem-fonte antes de iniciar a execução de suas operações. As 3 primeiras etapas correspondem a fase de análise e contêm o analisador léxico, sintático e semântico, conforme ilustrado pela Figura 4.1. Cada um desses passos recebe como entrada a saída do passo anterior, com exceção do primeiro, que recebe o código-fonte que será interpretado.

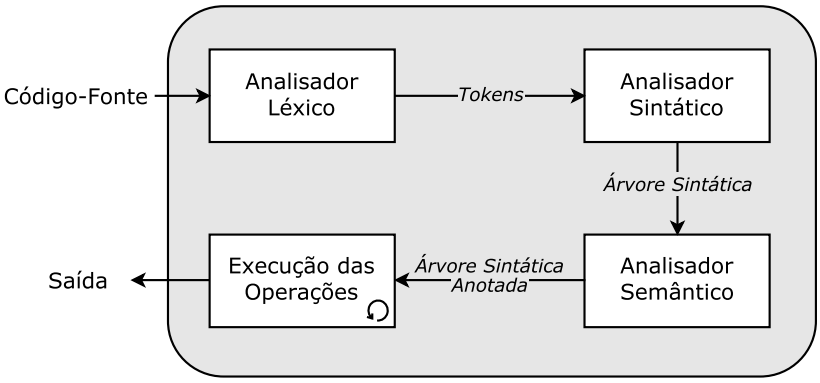

Figura 4.1 – Fases de um interpretador [1].

O analisador léxico realiza a leitura das linhas de caracteres que compõem o código-fonte e os organiza em unidades lógicas chamadas de *tokens*. Em seguida, o analisador sintático utiliza os *tokens* para criar uma representação intermediária que determina os elementos estruturais do programa e seus relacionamentos. Essa representação corresponde a uma árvore sintática em que cada nó representa uma operação e os filhos representam os argumentos da operação. Por fim, o analisador semântico verifica se cada um dos operadores da árvore sintática possui um operando compatível e, se todos forem compatíveis, o interpretador inicia a execução das operações da árvore sintática.

O desenvolvimento do interpretador para este trabalho não abordará, em um primeiro momento, a etapa de análise semântica. Os passos anteriores, de análise léxica e sintática, serão tratados com mais profundidade neste capítulo, assim como a unificação e substituição de variáveis mencionada nas definições do AgentSpeak(L).

4.1 Analisador Léxico

O analisador léxico, ou *scanner*, é a primeira etapa de análise do programa-fonte que será interpretado. Sua tarefa é ler as linhas de caracteres do código-fonte para organizá-las em uma

sequência de símbolos que representam uma unidade de informação elementar denominada lexema. Por sua vez, os lexemas encontrados criam unidades lógicas chamadas de *tokens*, que contêm a classificação do lexema e informações de linha e posição no código-fonte [1, p. 109]. Os *tokens* podem representar palavras reservadas próprias da linguagem, identificadores definidos pelos usuários, literais numéricos e de texto, operadores e pontuações.

```
1 +!localizacao(robo, X) : localizacao(robo, X) <- true.
```

Código 4.1 – Exemplo de plano em AgentSpeak(L).

Por exemplo, o plano descrito no Código 4.1 contém 54 caracteres, mas apenas 18 *tokens*: "**+**", operador de adição de evento; "**!**", símbolo de objetivo de realização; "**localizacao**", identificador; "**(**", símbolo parêntese esquerdo; "**robo**", identificador; "**,**", símbolo de separação de argumentos; "**X**", identificador; "**)**", símbolo de parêntese direito; "**:**", operador de condição; "**localizacao**", identificador; "**(**", símbolo de parêntese esquerdo; "**robo**", identificador; "**,**", símbolo de separação de argumentos; "**X**", identificador; "**)**", símbolo parêntese direito; "<-", operador de implicação; "**true**", palavra reservada e "**.**", símbolo de finalização.

De acordo com Aho *et al.* [1, p. 111], um lexema pode ser reconhecido pelo analisador léxico a partir de um determinado padrão de caracteres utilizando técnicas como por expressões regulares ou autômatos finitos.

4.2 Analisador Sintático

O analisador sintático, ou *parser*, percorre o conjunto de *tokens* proveniente do analisador léxico e identifica elementos estruturais como expressões aritméticas e comandos de atribuição. A partir destes elementos e seus relacionamentos, é possível criar uma representação intermediária, denominada de árvore sintática, com uma estrutura onde os nós representam as operações e os filhos representam os argumentos da operação correspondente. A construção da árvore sintática pode ser feita utilizando técnicas como os método *top-down* ou *bottom-up*. No primeiro método, a árvore sintática é construída da raiz para as folhas, enquanto que, no segundo método, a construção da árvore é iniciada nas folhas em direção à raiz [1, p. 192].

Considerando o exemplo acima, a árvore sintática produzida para a sequência de *tokens* é ilustrada na Figura 4.2. Os *tokens* relativos ao predicado `localizacao(robo, X)` são ilustrados, para uma melhor visualização, na Figura 4.3. Na primeira figura, os operadores "**:**" e "<-" ramificam a árvore e identificam, o evento ativador, o contexto e o corpo do plano. Em seguida, o operador "**+**" e o símbolo "**!**" ramificam o evento ativador até encontrarem a expressão da segunda imagem. Esta expressão, assim como a do contexto do plano, são ramificadas a partir dos símbolos "**(**", "**,**" e "**)**". Ao final, a árvore sintática deve conter, em seus nós folhas, todos os *tokens* identificados no analisador léxico.

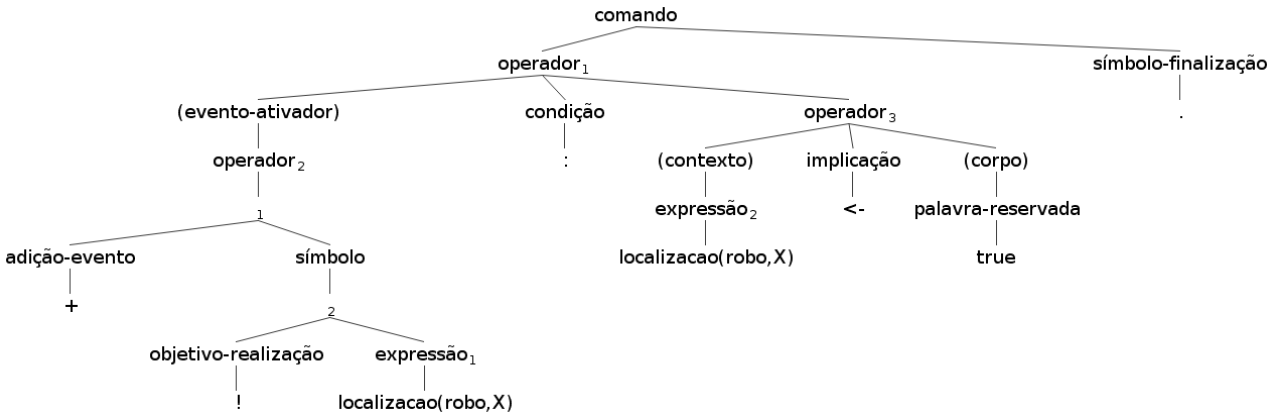

Figura 4.2 – Árvore sintática do Código 4.1

.

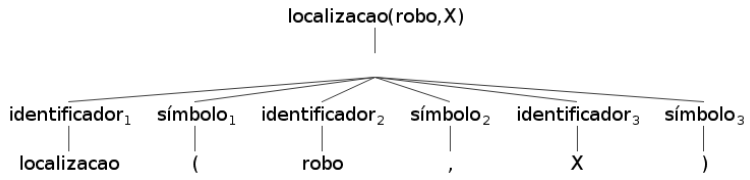

Figura 4.3 – Continuação da árvore sintática da Figura 4.2

.

Esta etapa, assim como a anterior, também identifica erros. Enquanto que o analisador léxico verifica caracteres inválidos que não deveriam fazer parte de um *token*, o analisador sintático valida se esses *tokens* seguem as regras gramaticais especificadas nas definições do Capítulo 3.

4.3 Unificação e Substituição de Variáveis

A unificação é o processo de encontrar uma substituição capaz de tornar idênticas, 2 expressões lógicas diferentes. Por sua vez, o unificador mais geral é o conjunto mínimo de substituições possíveis para o par de expressões que deve ser unificado [15, p. 326].

De acordo com Sterling *et al.* [17, p. 88], ocorre a unificação e substituição de 2 termos, T_1 e T_2, nos casos descritos abaixo e implementados no Código 4.2[1].

- Se T_1 e T_2 são constantes ou variáveis idênticas, conforme as linhas 4 e 5.
- Se T_1 é uma variável e T_2 é um termo que não contém T_1, é feito uma busca por todas as ocorrências de T_1 para serem substituídas por T_2. Estes casos são contemplados pelas linhas 6 e 7.
- Se T_2 é uma variável e T_1 é um termo que não contém T_2, é feito uma busca por todas as ocorrências de T_2 para serem substituídas por T_1. Estes casos são contemplados pelas linhas 8 e 9.

- Se T_1 e T_2 são estruturas com o mesmo nome de função, ou *functor*, e a mesma quantidade de argumentos, conforme as linhas 10 a 18.

```
1 def unify(t1, t2, theta):
2   if theta == None:
3     return None
4   elif t1 == t2:
5     return theta
6   elif is_variable(t1):
7     return unify_variable(t1, t2, theta)
8   elif is_variable(t2):
9     return unify_variable(t2, t1, theta)
10  elif is_structure(t1) and is_structure(t2) and (t1.head == t2.head):
11    # Unifica recursivamente quando o argumento for uma estrutura
12    new_theta = theta
13    if len(t1.arguments) == len(t2.arguments):
14      for i in range(len(t1.arguments)):
15        new_theta = unify(t1.arguments[i], t2.arguments[i], new_theta)
16      return new_theta
17    else:
18      return None
19  else:
20    return None
```

Código 4.2 – Código-fonte em Python para unificação de variáveis.

```
1 def occur_check(variable, term):
2     if variable == term:
3       return True
4     elif is_structure(term):
5       return term.head == variable.head or occur_check(variable, term.
      arguments)
6     return False
7
8 def extend(theta, variable, term):
9     new_theta = copy.copy(theta)
10    new_theta[variable] = term
11    return new_theta
12
13 def unify_variable(variable, term, theta):
14   if variable in theta:
15     return unify(theta[variable], term, theta)
16   elif occur_check(variable, term):
17     return None
18   else:
19     return extend(theta, variable, term)
```

Código 4.3 – Código-fonte em Python para substituição de variáveis.

[1] http://aima.cs.berkeley.edu/python/logic.html#unify

5. OBJETIVOS

A utilização de uma abstração para descrição do comportamento de agentes, independente de uma linguagem de programação, é importante para que seja possível a migração para tecnologias mais performáticas ou atuais. Conforme exposto neste trabalho, a utilização do AgentSpeak(L) permite manter uma certa independência, no entanto, são poucas as opções de implementação da linguagem.

5.1 Objetivo Geral

O objetivo geral deste trabalho é o desenvolvimento de um interpretador da sintaxe do AgentSpeak(L) para Python. Após concluído, também queremos disponibilizar o projeto livremente à comunidade para que o mesmo possa ter novas funcionalidades desenvolvidas, ou ser utilizado em outros projetos de IA.

5.2 Objetivos Específicos

Os objetivos específicos listados abaixo são essenciais para alcançar o objetivo geral proposto neste trabalho.

- Implementar um interpretador para um único agente que suporte o AgentSpeak(L).
- Implementar um *framework* do ambiente que suporte múltiplas instâncias do interpretador para um único agente.
- Construir cenários de simulação e garantir a extensibilidade da implementação para que a comunidade possa contribuir para o projeto.

5.3 Objetivos Desejáveis

Os demais objetivos são considerados desejáveis e visam complementar o trabalho proposto.

- Avaliar o desempenho do interpretador através de *benchmarks* com outras implementações do AgentSpeak(L).
- Submeter um artigo para um *workshop* do *International Conference on Autonomous Agents & Multi-Agent Systems* (AAMAS).

6. MODELAGEM E PROJETO

O objetivo deste capítulo é o de apresentar a metodologia, as funcionalidades e os artefatos necessários para o desenvolvimento do interpretador proposto para este trabalho. Durante a elaboração do referencial teórico para a criação da proposta deste trabalho, foi compreendido, de forma incorreta, que seria possível implementar apenas um subconjunto da sintaxe e semântica do AgentSpeak(L) proposta por Rao [13]. No entanto, sem uma implementação de todas as definições, não seria possível que um agente realizasse o seu ciclo de raciocínio de forma completa. Portanto, a modelagem elaborada contempla todas as definições apresentadas no Capítulo 3.

6.1 Metodologia de Desenvolvimento

O gerenciamento do desenvolvimento deste trabalho utilizará algumas práticas da metodologia *Scrum*. O *Scrum* é um *framework* que possui um conjunto de papéis, cerimônias e artefatos para a organização e gerenciamento de trabalho para o desenvolvimento de produtos e serviços inovadores [14, p. 13]. As práticas que serão utilizadas correspondem aos papéis de *product owner*, *scrum master* e equipe; as cerimônias de *sprints* e *sprint review*; e os artefatos *product backlog* e *sprint backlog*.

Um projeto que utiliza a metodologia *Scrum* inicia criando o *product backlog*. Este artefato é uma lista que contém as funcionalidades que devem ser implementadas no projeto, descritas no formato de *user stories*. Esta lista é priorizada pelo responsável do projeto, ou *product owner*, e dividida para ser executada em um ciclo que pode durar de uma a quatro semanas. Este ciclo corresponde a uma *sprint* e, ao seu final, é realizada uma entrega funcional, que pode ser testada e aprovada na *Scrum Review* pelo *Product Owner*. O conjunto de tarefas que será trabalhado durante a *Sprint* é denominado de *Sprint Backlog*. Durante estes ciclos iterativos, o *Scrum Master* garante o bom andamento do projeto e assegura que os ritos da metodologia sejam cumpridos pelos membros da equipe que realizam construção das tarefas [14, p. 15].

O *Scrum* foi escolhido devido às características da metodologia, que permite um desenvolvimento iterativo e incremental. Outra característica importante é o fato do *Scrum* proporcionar *feedback* antecipados de problemas decorrentes do desenvolvimento do protótipo. Desta forma, é possível identificar e corrigir, de forma precoce, erros que possam impactar nos objetivos do trabalho, além de proporcionar uma resposta mais rápida para qualquer mudança de escopo. A escolha dos papéis, cerimônias e artefatos do *framework* devem-se pelo fato do trabalho estar sendo conduzido por apenas um integrante. Neste caso, o autor irá acumular os papeis de *Scrum Master* e equipe de desenvolvimento, enquanto que o orientador desempenhará o papel de *Product Ower*. Como as reuniões de acompanhamento deste trabalho são realizadas de forma semanal, optou-se por uma *Sprint* com duração de uma semana. Desta forma é possível realizar as *Sprint Reviews* com o acompanhamento do *Product Owner*.

6.2 *Product Backlog*

O *Product Backlog*, conforme mencionado anteriormente, é uma lista que contém todas as funcionalidades que devem ser implementadas no produto. O conteúdo desta lista é definido, mantido e priorizado pelo *Product Owner*. Além disso, seus itens não precisam, necessariamente, estar completos no início de um projeto. Com o tempo, o *Protuct Backlog* pode crescer a medida que ocorre um entendimento maior sobre o produto e seus usuários [14, p. 18].

Por sua vez, as *User Stories* (US) descrevem, de forma simples e leve, as funcionalidades do *Product Backlog* sob o ponto de vista do usuário. Esta abordagem permite um entendimento maior entre todos os envolvidos no projeto, desde os responsáveis pelo negócio até os membros da equipe de desenvolvimento [14, p. 83]. O protótipo que será desenvolvido para este trabalho contém 8 *User Stories* que serão descritas a seguir, conforme a prioridade de cada uma.

6.2.1 US01 - Executar os Ciclos de Interpretação

"Como um usuário, eu quero executar os ciclos de interpretação para permitir as interações de um ou mais agentes em um ambiente."

Esta *User Story* permite que o usuário execute os ciclos de interpretação que servição como base para as interações de um ou mais agentes em um ambiente, unificando as demais funcionalidades descritas abaixo. A estimativa para o desenvolvimento desta funcionalidade é de 14 dias, contemplando 2 *sprints*.

6.2.2 US02 - Descrever o Comportamento do Agente

"Como um usuário, eu quero realizar a descrição do comportamento de um agente individualmente utilizando o AgentSpeak(L)."

Esta *User Story* permite que o usuário realize a descrição do comportamento de um agente de forma individual em um arquivo de texto externo. Este arquivo, que contém as crenças iniciais e a biblioteca de planos do agente, será importado utilizando um *parser*. Em seguida, um agente será instanciado para executar o seu ciclo de raciocínio. A estimativa para o desenvolvimento desta funcionalidade é de 28 dias, contemplando 4 *sprints*.

6.2.3 US03 - Imprimir Crenças do Agente

"Como um usuário, eu quero poder imprimir o conjunto de crenças de um agente no terminal."

Esta *User Story* permite que o usuário crie uma ação de impressão (`.print()`) quando desejar visualizar o conjunto de crenças de um agente no terminal de execução do interpretador. A estimativa para o desenvolvimento desta funcionalidade é de 7 dias, contemplando 1 *sprints*.

6.2.4 US04 - Descrever o Comportamento do Ambiente

"Como um usuário, eu quero realizar a descrição dos estados do ambiente e a mudança destes estados a partir de ações provenientes dos agentes utilizando a linguagem Python."

Esta *User Story* permite que o usuário realize a descrição dos estados do ambiente e a mudança destes estados a partir de ações provenientes dos agentes. Esta descrição deverá ser realizada utilizando instruções em Python que serão tratadas por um *framework*, que será responsável pelas interações dos agentes com o ambiente. A estimativa para o desenvolvimento desta funcionalidade é de 21 dias, contemplando 3 *sprints*.

6.2.5 US05 - Executar ações no Ambiente

"Como um usuário, eu quero descrever o comportamento de um agente para executar ações no ambiente."

Esta *User Story* permite que o usuário descreva o comportamento de um agente para executar ações no ambiente utilizando o *framework* descrito na Seção 6.2.4. A estimativa para o desenvolvimento desta funcionalidade é de 14 dias, contemplando 2 *sprints*.

6.2.6 US06 - Visualizar os Estados Mentais e a Caixa de Mensagem dos Agentes

"Como um usuário, eu quero visualizar os estados mentais e a caixa de mensagens de um agente."

Esta *User Story* permite que o usuário visualize os estados mentais e a caixa de mensagens dos agentes, em um determinado ciclo de interpretação, de forma individual. O interpretador deverá

armazenar, em um arquivo de texto externo, todos os dados necessários para a recuperação destas informações para visualização, a cada ciclo de interpretação. A estimativa para o desenvolvimento desta funcionalidade é de 7 dias, contemplando uma *sprint*.

6.2.7 US07 - Visualizar os Estados do Ambiente

"Como um usuário, eu quero visualizar os estados do ambiente a cada ciclo de interpretação."

Esta *User Story* permite que o usuário visualize os estados do ambiente em um determinado ciclo de interpretação. O interpretador deverá armazenar, em um arquivo de texto externo, todos os dados necessários para a recuperação destas informações para visualização, a cada ciclo de interpretação. A estimativa para o desenvolvimento desta funcionalidade é de 7 dias, contemplando uma *sprint*.

6.2.8 US08 - Trocar Mensagens entre Agente

"Como um usuário, eu quero poder enviar uma mensagem de um agente para outro."

Esta *User Story* permite que o usuário crie uma ação de mensagem (`.send()`) quando desejar que um agente envie uma mensagem para um outro agente. A estimativa para o desenvolvimento desta funcionalidade é de 7 dias, contemplando uma *sprint*.

6.3 Cronograma

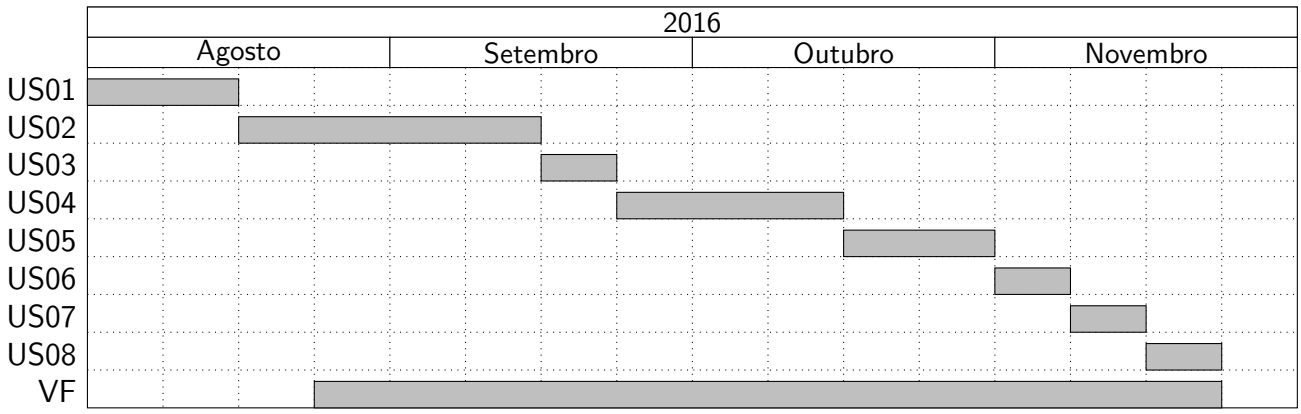

Figura 6.1 – Cronograma das atividades para o TC II.

A Figura 6.1 ilustra o cronograma para o desenvolvimento das *user stories* descritas na Seção 6.2. O item "VF" corresponde a tarefa de escrever o volume final do trabalho de conclusão II. O cronograma contém as semanas em que cada uma das atividades será executada.

6.4 Modelagem

A modelagem é uma das principais atividades para o planejamento e desenvolvimento de um *software*. Nela, são construídos modelos que representam uma simplificação da realidade e que nos auxiliam a compreender melhor o sistema que está sendo desenvolvido. Estes modelos são elaborados utilizando uma linguagem-padrão, denominada *Unified Modeling Language* (UML), que permite a visualização, especificação, construção e documentação de artefatos que façam uso do sistema em questão [2, p. 14].

6.4.1 Modelo de Caso de Uso de Negócio

O modelo de caso de uso de negócio permite descrever as funções pretendidas com o negócio, e descrevem uma sequência de ações que produz um resultado desejado a um determinado ator do negócio. A partir das *User Stories* descritas na Seção 6.2, foi elaborado o modelo de casos de uso de negócio ilustrado na Figura 6.2.

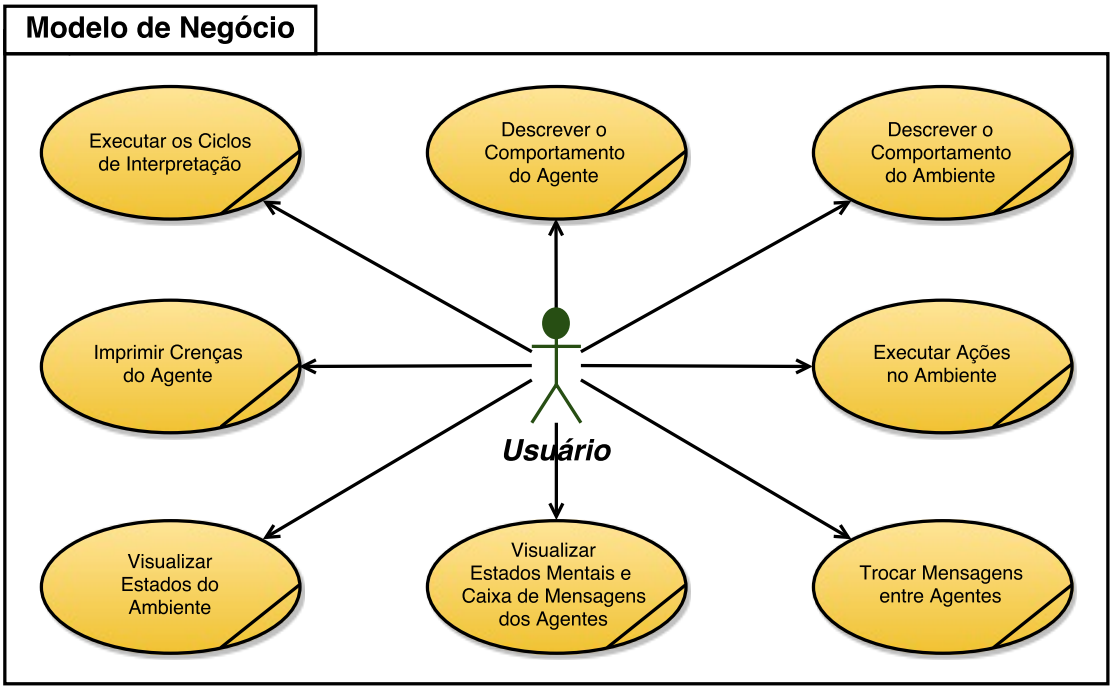

Figura 6.2 – Modelo de negócio das funcionalidades.

6.4.2 Diagrama de Atividades

O diagrama de atividades permite descrever o fluxo das atividades de um caso de uso de negócio. As atividades resultam em ação antes de prosseguir para a próxima atividade. Desta forma, é possível organizar, dentro de um processo de negócio, o fluxo com a ordem e circunstâncias em que cada ação ocorre.

O ciclo de raciocínio de um agente, será desenvolvido seguindo o fluxo do diagrama de atividades ilustrado na Figura 6.3. De forma geral, o agente recebe, no início de cada ciclo, as percepções do ambiente e as mensagens destinadas a ele. Em seguida, as percepções são confrontadas com suas crenças atuais para, caso seja necessário, atualizar o conjunto de crenças. As mensagens recebidas e as divergências de crenças criam novos eventos no conjunto de eventos. Na próxima etapa, é selecionado um evento deste conjunto para ser unificado com a biblioteca de planos e encontrar a lista de planos possíveis. Então, é feita a seleção do plano aplicável e, em seguida, da intenção que será executada.

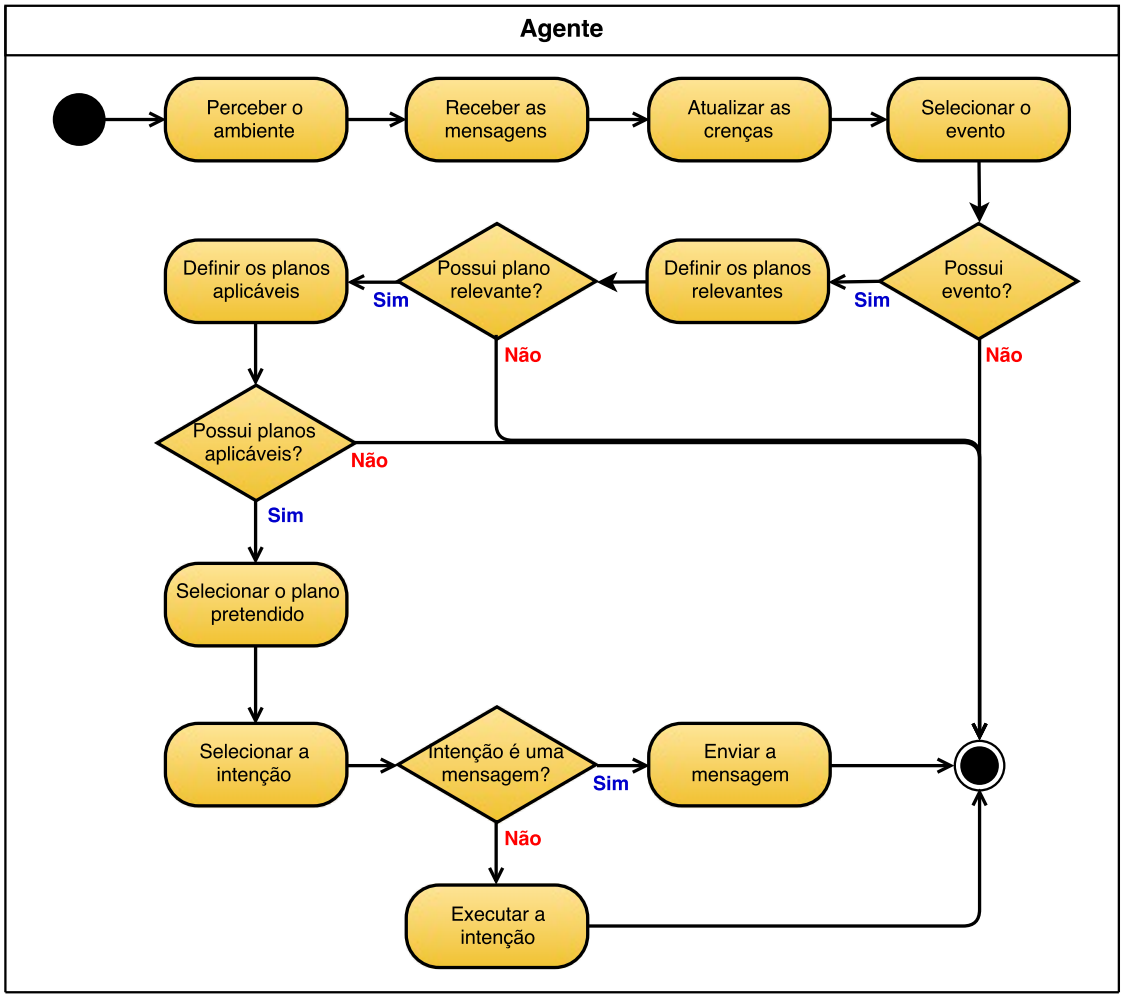

Figura 6.3 – Diagrama de atividades do ciclo de raciocínio de um agente.

O ciclo de execução do *framework*, que será responsável pelas interações dos agentes com o ambiente, será desenvolvido seguindo o fluxo do diagrama de atividades ilustrado na Figura 6.4. De forma geral, o interpretador processa o comportamento individual de cada agente descrito nos arquivos texto externos e cria as instâncias dos agentes. Com os agentes criados e o ambiente definido, o ciclo de interpretação com as interações entre os agentes e o ambiente é iniciado. Estes ciclos permanecem sendo executados até que o usuário solicite o encerramento das interações.

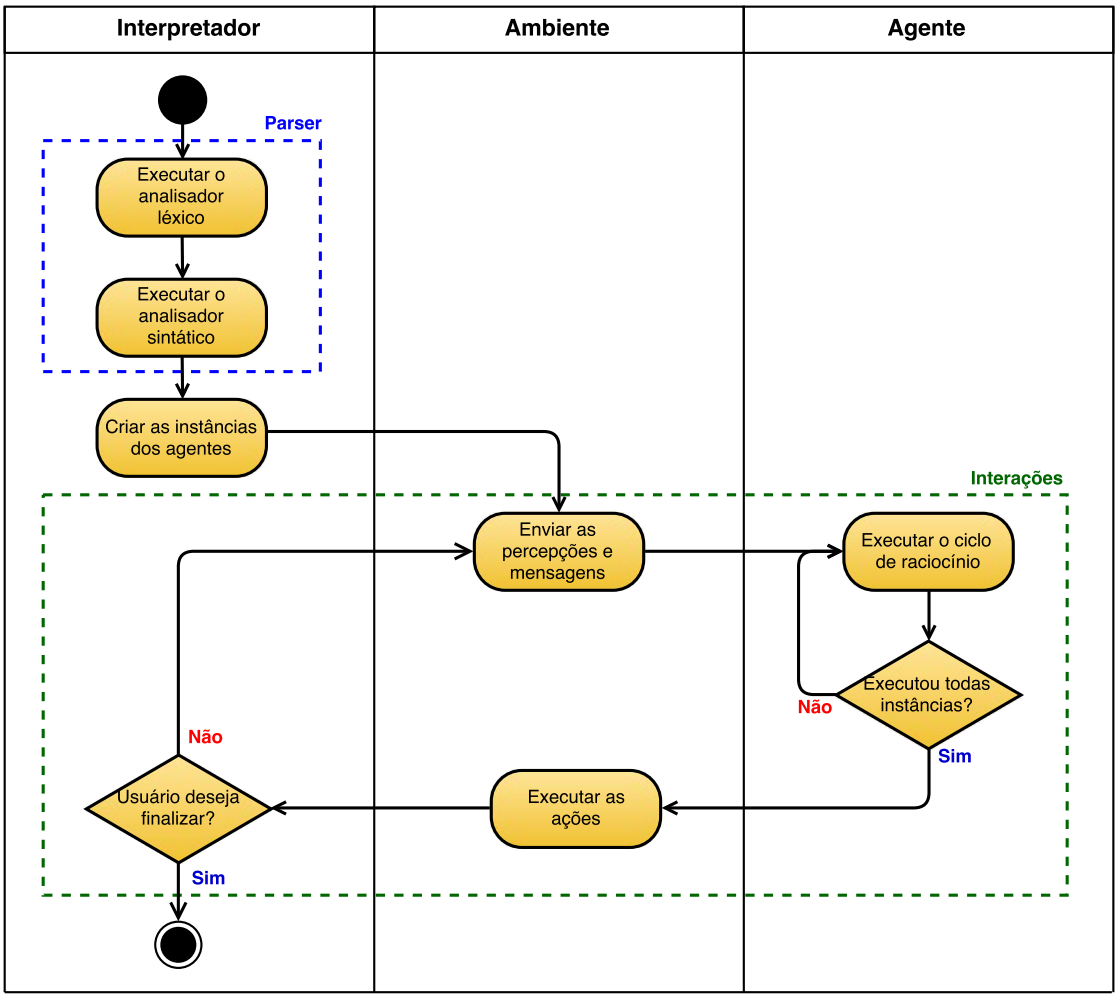

Figura 6.4 – Diagrama de atividades do ciclo de interpretação.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Aho, A. V.; Lam, M. S.; Sethi, R.; Ullman, J. D. "Compilers : Principles, Techniques, & Tools." Addison-Wesley, 2007, second ed..

[2] Booch, G.; Rumbaugh, J.; Jacobson, I. "The Unified Modeling Language User Guide". Addison-Wesley, 2005.

[3] Bordini, R. H.; Hübner, J. F.; Wooldridge, M. "Programming Multi-Agent Systems in AgentSpeak Using Jason". Wiley, 2007.

[4] Bordini, R. H.; Hübner, J. F. "Bdi agent programming in agentspeak using jason." In: CLIMA, Toni, F.; Torroni, P. (Editores), 2005, pp. 143–164.

[5] d'Inverno, M.; Kinny, D.; Luck, M.; Wooldridge, M. "A formal specification of dmars." In: ATAL, Singh, M. P.; Rao, A. S.; Wooldridge, M. (Editores), 1997, pp. 155–176.

[6] Georgeff, M.; Lansky, A. L. "Reactive reasoning and planning". In: Proceedings of AAAI-87, 1987, pp. 677–682.

[7] Maes, P. "The agent network architecture", *SIGART Bulletin*, vol. 2–4, 1991, pp. 115–120.

[8] Mascardi, V.; Demergasso, D.; Ancona, D. "Languages for programming bdi-style agents: an overview." In: WOA, Corradini, F.; Paoli, F. D.; Merelli, E.; Omicini, A. (Editores), 2005, pp. 9–15.

[9] McKinney, W. "Python for data analysis." O'Reilly, 2013.

[10] Meneguzzi, F. "Extending agent languages for multiagent domains", Tese de Doutorado, King's College London, 2009.

[11] Pokahr, A.; Braubach, L.; Lamersdorf, W. "Jadex: A bdi reasoning engine." In: *Multi-Agent Programming*, Bordini, R. H.; Dastani, M.; Dix, J.; Fallah-Seghrouchni, A. E. (Editores), Springer, 2005, *Multiagent Systems, Artificial Societies, and Simulated Organizations*, vol. 15, pp. 149–174.

[12] Rao, A. S. "Agentspeak(l): Bdi agents speak out in a logical computable language". In: Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, van Hoe, R. (Editor), 1996.

[13] Rao, A. S.; Georgeff, M. "Bdi agents: From theory to practice". In: First International Conference on Multi-Agent Systems, 1995, pp. 312–319.

[14] Rubin, K. "Essential Scrum: A Practical Guide to the Most Popular Agile Process". Pearson Education, 2012.

[15] Russel, S.; Norvig, P. "Artificial Intelligence: A Modern Approach". Pearson Education Inc., 2010.

[16] Santos, F. R. "Avaliação do uso de agentes no desenvolvimento de aplicações com veículos aéreos não-tripulados", Dissertação de Mestrado, Universidade Federal de Santa Catarina, 2015.

[17] Sterling, L.; Shapiro, E. Y. "The Art of Prolog - Advanced Programming Techniques, 2nd Ed." MIT Press, 1994.

[18] Wooldridge, M. "Intelligent agents". In: *Multiagent Systems*, Weiss, G. (Editor), MIT Press, 1999, cap. 1, pp. 27–77.

[19] Wooldridge, M. "Reasoning About Rational Agents". MIT Press, 2000.

[20] Wooldridge, M. "An Introduction to MultiAgent Systems". John wiley & Sons Ltd., 2002.

[21] Wooldridge, M.; Jennings, N. "Intelligent agents: Theory and practice", *Knowledge Engineering Review*, vol. 2–10, 1995.