UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS TRINDADE
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

Matuzalém Muller dos Santos

Programação Orientada a Agentes BDI em Sistemas Embarcados

Florianópolis
2022

Matuzalém Muller dos Santos

Programação Orientada a Agentes BDI em Sistemas Embarcados

Dissertação submetida ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas da Universidade Federal de Santa Catarina para a obtenção do título de mestre em Engenharia de Automação e Sistemas.
Orientador: Prof. Jomi Fred Hübner, Dr.
Coorientador: Prof. Maiquel de Brito, Dr.

Florianópolis
2022

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Santos, Matuzalém
Programação Orientada a Agentes BDI em Sistemas Embarcados / Matuzalém Santos ; orientador, Jomi Hübner, coorientador, Maiquel Brito, 2022.
79 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Engenharia de Automação e Sistemas, Florianópolis, 2022.

Inclui referências.

1. Engenharia de Automação e Sistemas. 2. AgentSpeak. 3. Jason. 4. BDI. 5. Sistemas embarcados. I. Hübner, Jomi. II. Brito, Maiquel. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Automação e Sistemas. IV. Título.

Matuzalém Muller dos Santos

Programação Orientada a Agentes BDI em Sistemas Embarcados

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof. Carlos Roberto Moratelli, Dr.
Universidade Federal de Santa Catarina

Prof. Leandro Buss Becker, Dr.
Universidade Federal de Santa Catarina

Prof. Maicon Rafael Zatelli, Dr.
Universidade Federal de Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de mestre em Engenharia de Automação e Sistemas.

Documento assinado digitalmente
Werner Kraus Junior
Data: 07/02/2022 08:27:05-0300
CPF: 531.085.239-53
Verifique as assinaturas em https://v.ufsc.br

Prof. Werner Kraus Junior, Dr.
Coordenador do Programa

Documento assinado digitalmente
Jomi Fred Hubner
Data: 07/02/2022 08:57:07-0300
CPF: 596.334.769-68
Verifique as assinaturas em https://v.ufsc.br

Prof. Jomi Fred Hübner, Dr.
Orientador

Documento assinado digitalmente
Maiquel de Brito
Data: 07/02/2022 09:57:03-0300
CPF: 979.657.150-15
Verifique as assinaturas em https://v.ufsc.br

Prof. Maiquel de Brito, Dr.
Coorientador

Florianópolis, 07 de fevereiro de 2022.

Dedico este trabalho aos meus pais, Ani e Valmor,
e a meu irmão, Israel.

AGRADECIMENTOS

À Deus, pelo dom da vida.

Aos meus pais, Ani e Valmor, e ao meu irmão, Israel, pelos ensinamentos, apoio e carinho.

À Geovana, pelo incentivo, carinho e companheirismo.

Aos meus amigos e colegas de curso, pelos conselhos, amizades e suporte.

À Michael Fisher, pelo apoio e incentivo.

Aos meus orientadores, Jomi e Maiquel. Com certeza, este trabalho não teria sido realizado sem suas orientações, sua dedicação e seus ensinamentos.

À todo o corpo docente e administrativo da UFSC, em especial do DAS, que trabalha de forma árdua para fornecer educação pública de qualidade.

"Cuidado com gente que não tem dúvida.
Gente que não tem dúvida não é capaz de inovar,
de reinventar, não é capaz de fazer de outro modo.
Gente que não tem dúvida só é capaz de repetir."
(Mario Sergio Cortella)

RESUMO

As características de autonomia, reatividade e proatividade, comumente presentes em sistemas embarcados utilizados em sistemas ciberfísicos, são similares às características de agentes. Apesar da utilização de agentes nesses cenários ser potencialmente benéfica, a falta de ferramentas que implementam agentes em plataformas geralmente utilizadas em sistemas embarcados é um dos motivos que impede agentes embarcados de se tornarem uma realidade. Este trabalho discute os desafios de implementação de agentes BDI em sistemas embarcados e apresenta um *framework* para sua implementação. Múltiplas versões de uma aplicação são implementadas, usando o *framework* e uma abordagem tradicional de implementação para sistemas embarcados, possibilitando a comparação entre as diferentes versões do programa e a avaliação das vantagens e desvantagens da *engine* BDI presente nos agentes implementados.

Palavras-chave: AgentSpeak. Jason. BDI. Sistemas embarcados.

ABSTRACT

The characteristics of autonomy, reactivity, and proactivity, commonly present in embedded systems used in cyber-physical systems, are often similar to the ones of agents. Although the usage of agents in these scenarios would be beneficial, the lack of tools to implement agents in hardware commonly used in embedded systems is one of the reasons that prevent embedded agents from becoming a reality. This work discusses the challenges of implementing BDI agents in embedded systems and presents a framework for their implementation. Multiple versions of an application are implemented using the proposed framework and using a more usual approach, allowing the comparison between the different versions and evaluating the overhead and advantages of the BDI engine in the agent executable.

Keywords: AgentSpeak. Jason. BDI. Embedded systems.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

AOP	*Agent Oriented Programming*
BDI	*Belief-Desire-Intetion*
CI	*Continuous Integration*
CPS	*Cyber-Physical Systems*
dMARS	*Distributed Multi-Agent Reasoning System*
FIPA	Foundation for Intelligent Physical Agents
IoT	*Internet of Things*
JACK	*JACK Intelligent Agents*
JADE	*JAVA Agent DEvelopment Framework*
JVM	*Java Virtual Machine*
PRS	*Procedural Reasoning System*
RTOS	*Real Time Operating System*
SDK	*Software Development Kit*
UAV	*Unmanned Aerial Vehicle*
USB	*Universal Serial Bus*

SUMÁRIO

1 INTRODUÇÃO

Recentes avanços em tecnologias diversas têm possibilitado a criação de sistemas em que programas de computador embarcados nos mais diversos dispositivos, combinados com sensores e atuadores, interajam com o ambiente físico. Sistemas com esta característica são frequentemente referenciados na literatura como Sistemas Ciberfísicos (referenciados, neste trabalho, como CPS, do inglês, *Cyber-Physical Systems*). Estes sistemas podem ser considerados a intersecção de sistemas embarcados, sistemas de sensoriamento e sistemas de controle distribuído (RAJKUMAR *et al.*, 2010). De acordo com Park, Zheng e Liu (2012), a implementação de CPS é complexa e possui diversos desafios, tanto em termos de infraestrutura e integração quanto de implementação dos componentes individuais da borda do sistema, os quais podem ser complexos, apesar do *hardware* simples que muitas vezes é utilizado em redes de sensores e atuadores. Exemplos deste tipo de sistema são *smart grids*, *smart cities*, sistemas de transporte inteligente e veículos autônomos (STOJMENOVIC, 2014). No caso de veículos autônomos, por exemplo, é necessário que os sistemas embarcados sejam seguros, facilmente atualizáveis, altamente disponíveis e capazes de se adaptarem de acordo com as características do ambiente (FÜRST; BECHTER, 2016).

Considerando os cenários inteligentes nos quais CPS podem ser utilizados, é desejável que os dispositivos que formam estes sistemas apresentem características como (i) autonomia, para atuar sem a necessidade de um operador humano; (ii) habilidades sociais, para colaborar com outros dispositivos na solução de problemas; (iii) capacidade de adaptação e aprendizado, para aprimorar sua atuação à medida que participam do sistema; e (iv) proatividade, para atuar guiados por objetivos a serem atingidos e não por meras reações ao estado atual do sistema (ALIYUDA, 2016).

Agentes são uma metáfora adequada para o desenvolvimento de aplicações com as características anteriormente mencionadas (O'HARE *et al.*, 2012). Um agente pode ser definido como um sistema computacional situado em um ambiente e que é capaz de atuar de forma autônoma neste ambiente para atingir seus objetivos (WOOLDRIDGE, 2009). Além da autonomia, agentes possuem, por definição, as demais características citadas anteriormente como relevantes na construção de CPS: habilidades sociais, capacidade de aprender e proatividade (WOOLDRIDGE; JENNINGS, 1995b). Assim, tem-se indicativos de que a programação orientada a agentes (ou *Agent Oriented Programming* (AOP)) é um paradigma adequado à implementação de sistemas embarcados que constituam um CPS e que necessitem de autonomia, habilidades sociais, capacidade de aprender e proatividade.

Dentre as arquiteturas de agentes, a arquitetura BDI é uma das mais conhecidas

e estudadas (PEREIRA *et al.*, 2005). A arquitetura BDI permite – em comparação com paradigmas de programação tradicionais – o desenvolvimento de sistemas que permanecem em execução contínua, reagindo a eventos, revisando o curso de ações previstas em função de mudanças no ambiente, agindo de forma proativa para atingir objetivos de longo prazo, e incorporando aspectos sociais que permitem a interação e colaboração entre os agentes.

Ferramentas para o desenvolvimento de agentes BDI têm sido, tradicionalmente, concebidas considerando ambientes com abundância de recursos computacionais (capacidade de processamento, memória, etc), tais como computadores pessoais (O'HARE *et al.*, 2012). Entretanto, sistemas embarcados geralmente dispõem de recursos computacionais mais limitados (CALCE *et al.*, 2013), o que dificulta a implementação de agentes nestes sistemas através das ferramentas tradicionais. Assim, o problema tratado neste trabalho é a implementação de agentes BDI que possam ser embarcados em dispositivos com recursos limitados, típicos de CPS.

1.1 OBJETIVOS

Diversos trabalhos têm empregado diferentes abordagens para contornar as dificuldades encontradas para embarcar agentes em dispositivos com recursos computacionais limitados. Estas diferentes propostas, no entanto, são concebidas como soluções *ad hoc* aplicadas a problemas específicos, e não necessariamente replicáveis. Este trabalho, porém, propõe uma solução de propósito geral para que agentes BDI possam ser embarcados em dispositivos com recursos computacionais limitados, característicos de sistemas embarcados. Portanto, no decorrer deste trabalho serão abordadas as limitações das soluções existentes e as características do ciclo de raciocínio de agentes BDI que devem ser preservadas para que sistemas embarcados possam utilizar esta arquitetura. Para tal, será realizada a implementação de uma ferramenta que permita a programação de agentes BDI em sistemas embarcados, visando analisar e discutir as principais características da arquitetura e os desafios de implementação de agentes nestes sistemas. No contexto dissertado, seguem os objetivos centrais deste projeto de mestrado.

1.1.1 Objetivo Geral

O objetivo geral deste trabalho é conceber meios para o desenvolvimento e execução de agentes embarcados que utilizem a arquitetura BDI.

1.1.2 Objetivos Específicos

Para atingir o objetivo geral, define-se os seguintes objetivos específicos:

- Identificar as limitações de soluções existentes na implementação de agentes BDI embarcados;
- Identificar os desafios de implementação de uma ferramenta com multiplataforma, que permita o desenvolvimento de agentes BDI para sistemas embarcados;
- Propor soluções para os desafios de implementação da ferramenta;
- Implementar uma ferramenta que permita a implementação de agentes BDI em sistemas embarcados;
- Implementar uma aplicação embarcada utilizando a ferramenta concebida e uma linguagem de programação tradicional, geralmente utilizada em sistemas embarcados;
- Coletar dados da utilização de recursos computacionais nas implementações realizadas na plataforma embarcada;
- Avaliar a ferramenta desenvolvida e sua viabilidade de utilização, baseado nos dados de utilização coletados.

1.2 METODOLOGIA

Para atingir os objetivos deste trabalho, propõe-se o desenvolvimento de uma solução multiplataforma para a implementação de agentes BDI embarcados, visando suprir as limitações de soluções existentes. Entende-se que a solução a ser concebida deve levar em consideração os recursos computacionais disponíveis em plataformas geralmente utilizadas no desenvolvimento de aplicações embarcadas, como Arduino UNO, ESP8266, ou Raspberry Pi 4B.

Toma-se como base um interpretador de uma linguagem de programação orientada a agentes BDI. Neste trabalho, utiliza-se o interpretador de AgentSpeak Jason, devido à sua popularidade e funcionalidades suportadas. Considerando o interpretador selecionado, identifica-se desafios de implementação de agentes BDI em plataformas embarcadas com limitações de *hardware*, relacionados à execução do ciclo de raciocínio e às funcionalidades por ele suportadas. Tendo identificado os desafios, propõe-se um ciclo de raciocínio simplificado para os agentes, contando com estruturas de dados de tamanhos limitados e simplificação das funcionalidades suportadas.

Considerando a proposta do ciclo de raciocínio simplificado, é desenvolvido um *framework* que permite a programação de agentes BDI compatíveis com plataformas embarcadas. Este *framework* é programado em uma linguagem de programação comumente utilizada em sistemas embarcados. Neste caso, elege-se a linguagem de programação C++ para o desenvolvimento. Ademais, o *framework* permite a programação de agentes através de uma versão simplificada da linguagem de programação de agentes selecionada anteriormente, que, neste trabalho, é a linguagem AgentSpeak.

A avaliação dos resultados é feita a partir de três versões de uma aplicação. Cada versão da aplicação é implementada através de três abordagens, sendo que uma destas utiliza a linguagem de programação C para o desenvolvimento "tradicional" da solução e as outras duas implementações utilizam o *framework* para programação de agentes, explorando as características reativas e proativas dos mesmos. Estas implementações são realizadas em uma plataforma de desenvolvimento embarcado e possibilitam que seja feita uma análise quantitativa dos requisitos de *hardware* necessários para utilização da solução desenvolvida, em comparação com a implementação na linguagem de programação C.

1.3 ESTRUTURA DO DOCUMENTO

Na sequência deste documento, no Capítulo 2 é apresentada a fundamentação teórica, que apresenta conceitos pertinentes para a compreensão deste trabalho, sobretudo relacionados à agentes BDI e sistemas embarcados.

Posteriormente, no Capítulo 3, são apresentadas abordagens existentes para embarcar agentes BDI e suas limitações. Logo, explora-se os desafios na concepção de uma ferramenta que permita a implementação de agentes BDI embarcados, dado o funcionamento do ciclo de raciocínio do Jason e as funcionalidades da linguagem de programação de agentes AgentSpeak.

Após descrever os desafios, o Capítulo 4 descreve uma proposta que que possibilita a implementação de agentes BDI embarcados. Esta proposta consiste na concepção de um *framework* que permita a programação destes agentes.

Na sequência, no Capítulo 5 é apresentada a implementação do *framework*, na linguagem de programação C++, que consiste em dois módulos: a *engine* BDI, responsável por executar o ciclo de raciocínio do agente, e o módulo de tradução, que permite que a linguagem AgentSpeak possa ser utilizada para programação do agente, através de sua conversão para um código C++ equivalente.

O Capítulo 6 explora os resultados obtidos na utilização do *framework*, baseados na implementação de diferentes versões de uma aplicação.

Finalmente, são realizadas as considerações finais e sugeridos tópicos de pesquisa para trabalhos futuros no Capítulo 7.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, são apresentados conceitos necessários à compreensão deste trabalho. Inicialmente, é realizada uma apresentação de agentes, detalhando suas características comuns e suas classificações. Na sequência, a arquitetura BDI é apresentada, seguida da linguagem de programação de agentes *AgentSpeak* e o interpretador *Jason*, ferramentas populares na implementação de agentes. Ao final deste capítulo, são abordados conceitos relacionados a sistemas embarcados.

2.1 AGENTES

É possível encontrar variadas definições de agentes na literatura. De acordo com Wooldridge e Jennings (1995a), a utilização do termo "agente" pode variar de acordo com o domínio do sistema em que ele se encontra. Segundo a Foundation for Intelligent Physical Agents (FIPA) (FIPA, 2004), um agente é "um processo computacional que implementa uma funcionalidade autônoma e comunicativa de uma aplicação". Na perspectiva de Wooldridge (2009), um agente é "um sistema computacional que está situado em um ambiente e que pode agir de forma autônoma neste ambiente para atingir seus objetivos".

Apesar das diferentes definições de agentes disponíveis na literatura e da vasta quantidade de cenários e sistemas nos quais agentes podem ser empregados, entende-se que agentes são caracterizados por propriedades específicas. Segundo Wooldridge e Jennings (1995b), as propriedades a seguir caracterizam agentes:

- **Autonomia**: a autonomia refere-se à capacidade do agente de operar de forma independente de operador externo, tomando suas próprias decisões para alcançar seus objetivos. Suas decisões estão sob seu controle, e não são tomadas por outros;
- **Proatividade**: a proatividade refere-se à capacidade de o agente atuar guiado por objetivos e possuir iniciativa para alcançá-los;
- **Reatividade**: a reatividade se refere ao agente ser sensível a alterações no ambiente e possuir a habilidade de mudar seu curso de ação de acordo com estas alterações;
- **Habilidade social**: a habilidade social aborda a capacidade de agentes poderem interagir com outros agentes, através de uma linguagem de comunicação de agentes.

Além das propriedades citadas, outras propriedades como mobilidade e capacidade de aprendizado, por exemplo, também podem ser consideradas, a depender do ambiente e do sistema nos quais os agentes estão localizados (ANTHONY *et al.*, 2014).

Devido aos diversos sistemas e contextos nos quais agentes podem ser implementados, pode-se classificar agentes em três classes principais: reativos, cognitivos e híbridos. Agentes reativos reagem rapidamente à percepção de eventos e alterações no ambiente, sem executar um raciocínio complexo. Agentes cognitivos, por sua vez, possuem um modelo simbólico do ambiente e de outros agentes, desenvolvendo planos e tomando decisões baseadas nesse modelo. Por fim, os agentes híbridos combinam as vantagens das classes reativa e cognitiva, possibilitando que o agente se adapte à estímulos do ambiente e realize o processamento cognitivo de seus objetivos (BERGENTI; GLEIZES; ZAMBONELLI, 2004).

2.2 ARQUITETURA *BDI*

De acordo com Wooldridge (2009), arquiteturas de agentes são arquiteturas de *software* para sistemas de tomada de decisão que estão localizados em um ambiente.

A arquitetura BDI, baseada no modelo filosófico de racionabilidade de Bratman (BRATMAN, 1987), é uma das arquiteturas de agentes inteligentes mais conhecidas. Na arquitetura BDI, as ações realizadas por agentes são resultados de suas crenças (*beliefs)*, desejos (*desires*) e intenções (*intentions*) (BORDINI; DASTANI *et al.*, 2009). Estes conceitos podem ser definidos como:

- **Crenças**: são as informações que o agente possui sobre o mundo, o que envolve o ambiente, outros agentes e também a si mesmo. Estas informações são adquiridas através de percepções que o agente tem do ambiente, as quais podem ser imprecisas ou desatualizadas (BORDINI; HÜBNER; WOOLDRIDGE, 2007).

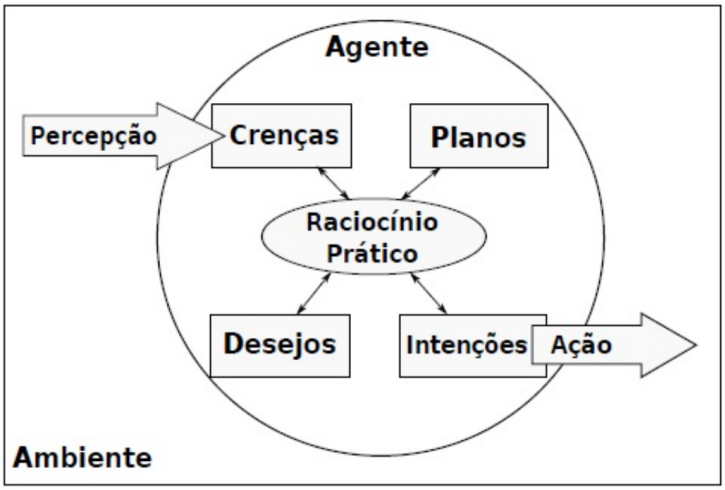
- **Desejos**: são estados de mundo que o agente gostaria que fossem atingidos. Desejos influenciam as ações de um agente, mas possuir um desejo não necessariamente significa que o agente irá agir para alcançá-lo (BORDINI; HÜBNER; WOOLDRIDGE, 2007).

- **Intenções**: são os estados de mundo que o agente está decidido e comprometido a atingir. Intenções podem ser delegadas ao agente, ou resultantes da escolha de seus desejos (MASCARDI; DEMERGASSO; ANCONA, 2005).

Crenças, desejos e intenções são utilizados no modelo de tomada de decisão denominado raciocínio prático, o qual consiste de duas atividades: deliberação e análise de meios-fins (BORDINI; HÜBNER; WOOLDRIDGE, 2007). Na atividade de deliberação, o agente decide qual de suas intenções ele irá tentar atingir, ou seja, qual o seu objetivo. A

seleção dos desejos nesta fase é baseada nas crenças, desejos e intenções atuais do agente. Na análise de meios-fins, o agente determina qual dos seus planos de ação deverá ser executado, tendo como objetivo alcançar a intenção escolhida (SANTOS, F. R., 2015). A Figura 1 é uma representação simplificada do ciclo de raciocínio BDI.

Figura 1 – Modelo de raciocínio prático utilizado pelos agentes BDI.



Fonte: Extraído de Fernando Rodrigues Santos (2015).

Como representado na Figura 1, o agente percebe o ambiente e atualiza suas crenças. Em seguida, ele realiza o raciocínio prático, que envolve a deliberação e análise de meios-fins. Durante a deliberação o agente decidirá qual seu objetivo, levando em consideração os desejos, crenças, planos e intenções atuais do agente. Por fim, um dos planos de ação é escolhido (análise de meios-fins) e posto em prática. O Algoritmo 1, de Wooldridge (2009), apresenta o ciclo de raciocínio BDI em pseudocódigo.

Algoritmo 1 *Ciclo de Controle de Raciocínio Prático*

```
1: B ← B_0
2: I ← I_0
3: while true do
4:    get next percept ρ through see(...) function
5:    B ← brf(B, ρ)
6:    D ← options(B,I)
7:    I ← filter(B,D,I)
8:    π ← plan(B,I,Ac)
9:    while not (empty(π) or succeeded(I,B) or impossible(I,B)) do
10:      α ← head(π)
11:      execute(α)
12:      π ← tail(π)
13:      get next percept ρ through see(...) function
14:      B ← brf(B, ρ)
15:      if reconsider(I,D) then
16:         D ← options(B,I)
17:         I ← filter(B,D,I)
18:      end if
19:      if not sound(π,I,B) then
20:         π ← plan(B,I,Ac)
21:      end if
22:   end while
23: end while
```

Fonte: Extraído de Wooldridge (2009).

De forma simplificada, o algoritmo funciona como explicado a seguir:

- Linhas 1 e 2: inicialização de crenças e intenções;
- Linhas 3 à 8: percepção e atualização de crenças a partir da função `see`(...) (3-5); revisão de desejos, considerando as crenças e intenções atuais (6); escolha de intenção (7); geração de um plano com curso de ações `Ac`, para atingir os objetivos do agente (8);
- Linha 9: verificação de condições para execução de planos:
  - `not empty(`π`)`: a fila de planos não está vazia, ou seja, há planos a serem executados;
  - `not succeeded(I,B)`: as intenções atuais do agente ainda não foram atingidas;
  - `not impossible(I,B)`: não é impossível alcançar as intenções considerando as crenças atuais do agente;

- Linhas 10 à 12: execução de um dos planos da lista de planos;
- Linhas 13 e 14: percepção e atualização de crenças a partir das função `see`(...);
- Linhas 15 à 18: reconsideração de intenções, onde o agente reavalia se suas intenções devem ser alteradas;
- Linhas 19 à 21: reavaliação de planos. Se os planos atuais não satisfazem mais as intenções do agente, novos planos devem ser gerados e adotados pelo agente para que os objetivos sejam alcançados.

É importante ressaltar que o Algoritmo 1, além de realizar a percepção do ambiente, atualizar as crenças, decidir qual objetivo deve ser atingido e determinar as ações a serem executadas, também avalia se suas intenções devem ser reconsideradas, a depender do estado atual do ambiente. A reavaliação constante de intenções é importante, pois torna o agente sensível a mudanças no meio e permite que objetivos possam ser adotados ou abandonados dinamicamente.

De acordo com Mascardi, Demergasso e Ancona (2005), exemplos de linguagens de programação de agentes BDI são: *Procedural Reasoning System* (PRS), *Distributed Multi-Agent Reasoning System* (dMARS), *JACK Intelligent Agents* (JACK), *Jadex* (compatível com a plataforma *JAVA Agent DEvelopment Framework* (JADE)) e AgentSpeak, sendo a última uma das linguagens mais populares na implementação de agentes BDI.

2.3 AGENTSPEAK

AgentSpeak é uma linguagem de programação lógica de agentes BDI desenvolvida por Rao (1996). A sintaxe da linguagem, inspirada em lógica de predicados (FITTING, 1996), é uma extensão do paradigma da programação em lógica (BORDINI; VIEIRA, 2003), sendo uma das principais características da linguagem a de que agentes são programados através da especificação de como atingir seus objetivos (BORDINI; HÜBNER; WOOLDRIDGE, 2007).

Os principais construtores da linguagem são as crenças (***beliefs***), os objetivos (***goals***) (também chamados de metas) e os planos (***plans***). Assim como na arquitetura BDI, as crenças representam o conhecimento do agente sobre o ambiente e outros agentes. Os objetivos, por sua vez, representam estados que o agente quer atingir (desejos). Finalmente, os planos são instruções de como o agente pode agir (SANTOS, F. R., 2015). Assim como na arquitetura BDI, o comprometimento na execução de um plano gera uma intenção.

Além dos construtores da linguagem, o AgentSpeak também possui componentes e estruturas de dados que auxiliam na programação e execução dos agentes. Estes componentes são: a base de crenças, a biblioteca de planos e o conjunto de eventos e intenções (SANTOS, F. R., 2015).

A base de crenças armazena as crenças do agente, as quais são atualizadas através da percepção do ambiente. A atualização de crenças gera eventos, os quais compõem o conjunto de eventos e podem levar a execução de um plano disponível na biblioteca de planos.

Quanto aos planos, estes possuem um contexto, que define se o plano é aplicável no momento em que é considerado, e uma lista de ações básicas, que são operações atômicas que o agente pode realizar para alterar o ambiente (BORDINI; HÜBNER; VIEIRA, 2005). Ao decidir que um plano é aplicável e será executado, uma intenção é criada e o agente compromete-se a executá-la.

O Código 1 é um exemplo de código AgentSpeak que ilustra os principais construtores e componentes da linguagem.

Código 1 – Exemplo de código AgentSpeak

```
1 !iniciar.
2
3 +!iniciar <- +bom_trabalho;
4              !aprovar.
5
6 +!aprovar : bom_trabalho <- +dissertacao(aprovada);
7                             bater_palmas.
```

O Código 1 é explicado a seguir:

- Linha 1: Adoção do objetivo `iniciar`.
- Linha 3: Declaração do plano referente ao objetivo `iniciar`; adição da crença `bom_trabalho`, em formato de proposição, à base de crenças.
- Linha 4: Adoção do sub-objetivo `aprovar`.
- Linha 6: Declaração do plano referente ao (sub-)objetivo `aprovar`; verificação de contexto, onde é verificado se a crença `bom_trabalho` existe na base de crenças, caso contrário, o plano não é executado; adição da crença `dissertacao(aprovada)`, em formato de predicado, à base de crenças.
- Linha 7: execução da ação externa `bater_palmas`.

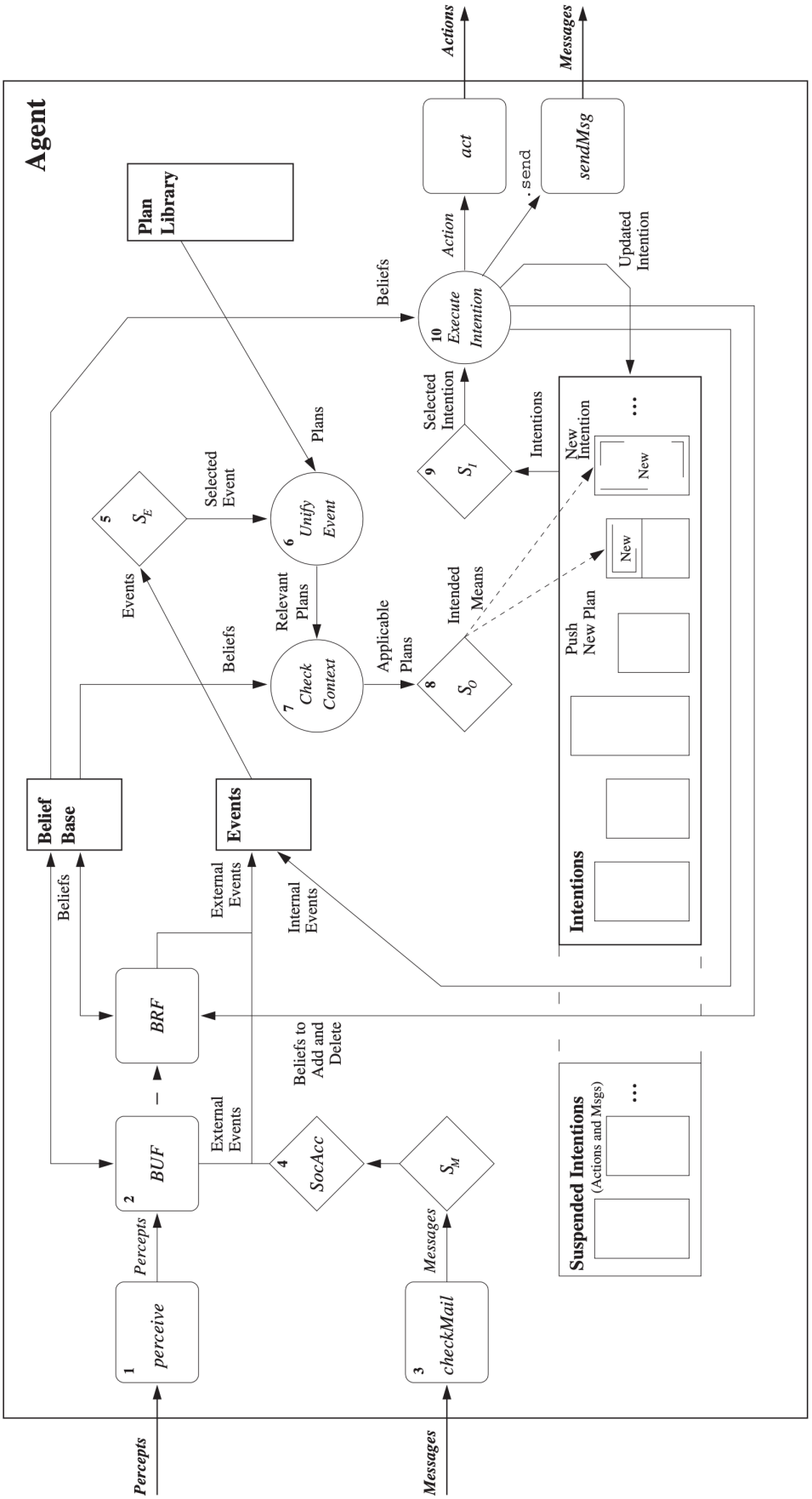
Ao inicializar a execução do agente o objetivo `iniciar` é adotado, resultando na execução do plano `iniciar` e consequentemente do plano `aprovar`. Ao completar a execução inicial do plano `aprovar`, a base de crenças do agente consiste das crenças `bom_trabalho` e `dissertacao_aprovada`. Também é importante ressaltar que o agente continua em execução ao terminar de executar o plano `aprovar`, atualizando suas crenças de acordo com sua percepção do ambiente e buscando atingir novos objetivos que possam ser adotados.

Um método comum na implementação de linguagens de programação é a criação de um interpretador para a linguagem alvo, geralmente escrito em outra linguagem (REYNOLDS, 1972). Dentre os interpretadores disponíveis para AgentSpeak destacam-se o SIM_Speak, AgentTalk e Jason, sendo o último o interpretador mais popular de AgentSpeak, implementado em Java e de código aberto (MASCARDI; DEMERGASSO; ANCONA, 2005; BORDINI; HÜBNER, 2007).

2.4 JASON

O Jason foi projetado tendo em mente a cooperação de agentes, permitindo que eles possam se coordenar e comunicar utilizando uma interface de alto nível (BORDINI; HÜBNER; WOOLDRIDGE, 2007). Assim, o Jason não apenas interpreta o código AgentSpeak, que realiza o raciocínio do agente, mas também oferece funcionalidades adicionais. A Figura 2 mostra o ciclo de raciocínio do Jason. Uma versão ampliada da figura está disponível no Anexo A.

Figura 2 – Modelo de raciocínio prático do Jason.



Fonte: Extraído de Bordini, Hübner e Wooldridge (2007).

De acordo com Bordini, Hübner e Wooldridge (2007), o ciclo de raciocínio do Jason é dividido em dez passos, explicados a seguir.

1. *(perceive)* Percepção do ambiente: o agente coleta informações do ambiente através da percepção;
2. *(BUF)* Atualização da base de crenças: o agente atualiza sua base de crenças, a partir das informações coletadas através da percepção. A atualização de crenças gera eventos, os quais são processados posteriormente;
3. *(checkMail)* Recepção de mensagens: o interpretador verifica se mensagens de outros agentes foram recebidas na "caixa de entrada" do agente;
4. *(SocAcc)* Seleção de mensagem "socialmente aceitável": as mensagens recebidas são filtradas para que apenas mensagens que podem ser aceitas pelo agente sejam processadas;
5. *(Se)* Seleção de evento: um dos eventos pendentes é selecionado;
6. *(Unify Event)* Recuperação dos planos relevantes: após selecionar um evento, o agente encontra todos os planos que permitam que o evento seja processado;
7. *(Check Context)* Determinação de planos aplicáveis: os planos relevantes são analisados e os que possuem uma chance de sucesso, dadas as crenças atuais, são selecionados;
8. *(So)* Seleção de um plano aplicável: um dos planos aplicáveis é escolhido para execução e se torna uma intenção;
9. *(Si)* Seleção de uma intenção para execução: escolhe uma das intenções ativas para execução;
10. *(Execute Intention)* Execução de etapa da intenção: executa uma instrução da intenção escolhida, que pode ser uma ação no ambiente ou atualização de crença, por exemplo.

O ciclo de raciocínio do Jason apresenta as características essenciais de agentes. Os passos 1 e 2 permitem que o agente seja sensível ao ambiente através da atualização constante de crenças, ou seja, a sensibilidade ao ambiente e constante reavaliação de objetivos - os quais podem ser decorrência de novas crenças adquiridas pela percepção do ambiente - tornam o agente **reativo**. As etapas 3 e 4 são funcionalidades do Jason

não inclusas no AgentSpeak, as quais permitem que o agente possa trocar mensagens com outros agentes, possuindo **habilidades sociais**. A **autonomia** do agente é caracterizada nos passos de 5 à 8, onde ele decide quais são seus objetivos e como alcançá-los, através da seleção de planos. Finalmente, a **proatividade** do agente dá-se pela sua orientação a objetivos, onde o mesmo está constantemente analisando suas crenças e sociabilizando com outros agentes para que novos objetivos possam ser adotados e alcançados.

Além de ser um interpretador de AgentSpeak, o Jason também apresenta funcionalidades adicionais que facilitam a implementação de sistemas multiagentes robustos. Duas das funcionalidades adicionais do Jason são anotações e ações internas. Anotações são termos associados à crenças que podem expressar características sobre estas crenças. A crença com anotação `acreditar(A)[source(agente_B)]`, por exemplo, descreve que a crença `acreditar(A)` tem como origem (*source*) o `agente_B` (anotação `[source(agente_B)]`. As ações internas, por sua vez, são ações já disponíveis no Jason e que não precisam ser implementadas pelo usuário da linguagem. A ação interna `.print("Palavra")`, por exemplo, exibe o texto "Palavra" na interface de saída padrão do sistema, enquanto a ação interna `.wait(1000)` faz com que o agente aguarde 1 segundo para continuar a execução do programa.

Finalmente, o Jason ainda apresenta outra grande vantagem: a possibilidade de configuração dos passos de seu ciclo de raciocínio. Na fase de seleção de mensagem, por exemplo, um agente pode ser configurado para aceitar mensagens apenas de remetentes conhecidos. Outra possibilidade é a de configurar a etapa de seleção de intenção para execução, onde intenções poderiam ser selecionadas de acordo com prioridades, por exemplo.

2.5 SISTEMAS EMBARCADOS

Para Berger (2002), sistemas embarcados são sistemas computacionais projetados para tarefas específicas e que utilizam um *hardware* específico, enquanto computadores pessoais são plataformas computacionais genéricas. De acordo com Cunha (2007), sistemas embarcados podem ser classificados de acordo com as aplicações nas quais eles são utilizados:

- Sistemas de propósito geral: são aplicações parecidas com computadores comuns, mas que possuem um propósito específico. Geralmente apresentam uma interface de interação com o usuário. Exemplos são TVs digitais e caixas eletrônicos de bancos.
- Sistemas de controle: são aplicações mais robustas, com circuitos dedicados e múltiplos sensores de entrada e saída. Possuem pouca interação com o usuário. Exemplos

são sistemas utilizados em motores de automóveis e em controle de voo de aeronaves.

- Processamento de sinais: este tipo de aplicação envolve um grande volume de informação a ser processada em um curto espaço de tempo. Exemplos são filtros digitais, modens e radares.

- Comunicações e redes: realizam a distribuição e chaveamento de informações. Sistemas de telefonia são um exemplo de aplicação.

De acordo com Marwedel (2010), devido à complexidade de sistemas embarcados e seus variados casos de uso, principalmente quando utilizados em *Cyber-Physical Systems* (CPS) e aplicações de Internet das Coisas (*Internet of Things* (IoT)), alguns desafios emergem em sua implementação, como segurança, confidencialidade, confiabilidade, manutenibilidade e disponibilidade. Marwedel (2010) define a **segurança** como um desafio relacionado à preservar a integridade das informações do sistema de ataques externos. A **confidencialidade**, relacionada à segurança, se refere à disponibilizar as informações do sistema apenas a entidades autorizadas. A **confiabilidade** é relacionada ao mau funcionamento de componentes do sistema, que não se comportam como esperado. Por sua vez, a **manutenibilidade** é a probabilidade de que uma falha no sistema possa ser corrigida em um intervalo de tempo. Finalmente, a **disponibilidade** é a probabilidade do sistema estar disponível. Para ter uma alta disponibilidade, a confiabilidade e manutenibilidade do sistema devem ser altas.

Ademais, Marwedel (2010) também cita a utilização de recursos computacionais como um dos maiores desafios no desenvolvimento de sistemas embarcados, tendo em vista que estes sistemas devem buscar minimizar a quantidade de recursos utilizados em sua implementação. Dentre os desafios na minimização dos recursos necessários, destacam-se:

- Consumo de energia: muitas das implementações de sistemas embarcados utilizam fontes limitadas de energia, como baterias, sendo este uma das maiores limitações destes sistemas;

- Tempo de execução: sistemas embarcados devem explorar o *hardware* disponível ao máximo. O uso ineficiente de tempo de execução deve ser evitado e tanto os algoritmos quanto o *hardware* devem ser otimizados, tendo em vista que os microprocessadores utilizados podem demorar a realizar instruções complexas que não sejam otimizadas;

- Tamanho de código: como podem haver limitações rígidas de memória e armazenamento, torna-se necessário que o código seja otimizado para que não se exceda os recursos disponíveis;
- Peso: sistemas portáveis devem ser leves;
- Custo: como muitos sistemas embarcados são utilizados em dispositivos manufaturados em massa, principalmente no mercado de eletrônicos, o custo destes dispositivos deve ser baixo.

As limitações de recursos usualmente disponíveis em sistemas embarcados tornam sua utilização em sistemas complexos, como os de inteligência artificial, um grande desafio de implementação (MEHALAINE *et al.*, 2018).

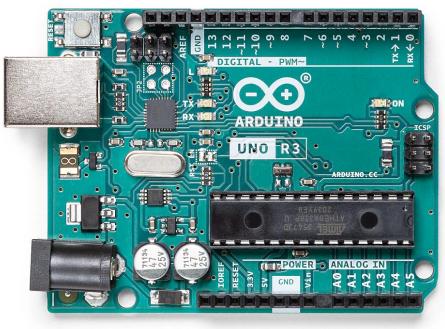
Plataformas comuns para desenvolvimento de sistemas embarcados incluem o Arduino UNO, LoLin NodeMCU v3 e Raspberry Pi 4B. A seguir, são apresentadas algumas características destas plataformas.

2.5.1 Arduino UNO Rev3

A placa Arduino UNO Rev3 é a terceira revisão da plataforma Arduino UNO. Esta placa pode ser utilizada na coleta de dados a partir de sensores acoplados, como de temperatura e umidade, e em cenários de automação residencial, por exemplo. Além disso, módulos de comunicação sem fio podem ser acoplados à placa, permitindo a comunicação através de rede sem fio, como Wi-Fi e *Bluetooth*, por exemplo. A linguagem de programação C++ é a mais utilizada nesta plataforma.

A Figura 3 exibe a placa Arduino UNO Rev3 e o Quadro 1 mostra algumas das principais especificações desta plataforma.

Figura 3 – Placa Arduino UNO R3



Fonte: Arduino Store (2021).

Quadro 1 – Especificação Arduino UNO Rev3

Especificação	**Descrição**
Microcontrolador	ATmega328P
Tensão de operação	5 V
Pinos I/O digitais	14
Memória Flash	32 KB
SRAM	2 KB
EEPROM	1 KB
Frequência de operação (Clock)	16 MHz

Fonte: Arduino Store (2021).

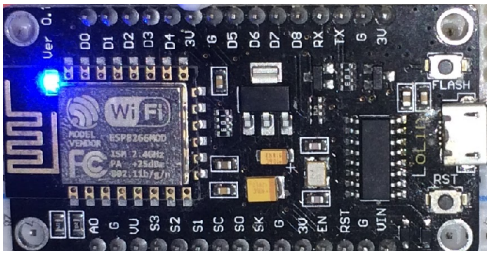
2.5.2 LoLin NodeMCU v3

O NodeMCU é um *firmware* de código aberto utilizado no desenvolvimento de aplicações de IoT. Este *firmware* é utilizado em plataformas de desenvolvimento que também são de código e design aberto, manufaturadas por diversos fabricantes, como LoLin, Wemos e Amica. As plataformas de desenvolvimento que utilizam este *firmware* também são comumente chamadas de NodeMCU, como a plataforma LoLin NodeMCU v3 (AZWAR *et al.*, 2019).

A plataforma LoLin NodeMCU v3 possui o módulo WiFi ESP12-E, desenvolvido pela Ai-Thinker e que possui o microcontrolador ESP8266 (AI-THINKER, 2015). Este módulo consiste no microcontrolador e nos componentes necessários para realizar comunicação sem fio, como a antena. Além disso, a LoLin NodeMCU v3 também contém uma interface de conexão *Universal Serial Bus* (USB), facilitando assim sua programação e alimentação, e outros componentes que complementam as funcionalidades oferecidas pela plataforma, como uma memória *flash* para armazenamento do programa. Dentre as linguagens de programação suportadas por esta plataforma, destacam-se C, C++ e Python, além do suporte a *Real Time Operating System* (RTOS).

A Figura 4 exibe a placa LoLin NodeMCU v3 e o Quadro 2 resume algumas das principais especificações desta plataforma.

Figura 4 – Placa LoLin NodeMCU v3



Quadro 2 – Especificação LoLin NodeMCU v3

Especificação	**Descrição**
Pinos de entrada/saída analógicos	1
Pinos de entrada/saída digitais	16
Microcontrolador	Tensilica 32-bit RISC CPU Xtensa LX106
Memória flash	4 MB
SRAM	64 KB
Frequência de operação (Clock)	80 MHz
Conectividade	802.11 b/g/n Wireless LAN

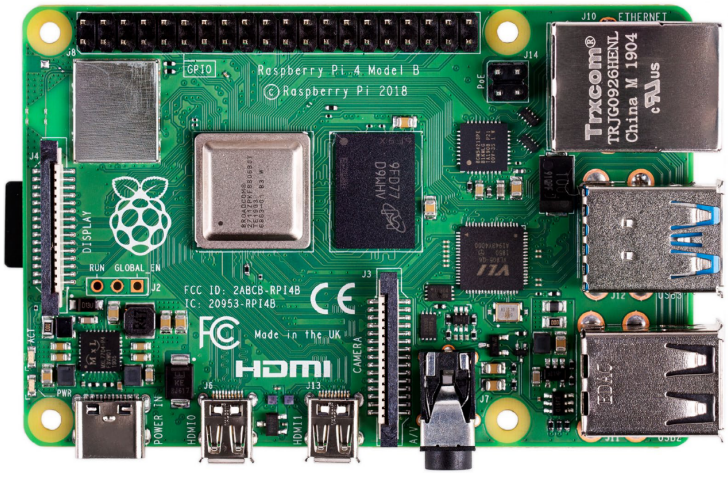
2.5.3 Raspberry Pi 4B

A Raspberry Pi 4B é a quarta geração de computadores da linha Raspberry. De acordo com a Raspberry Pi Foundation (2021), as Raspberry Pis são computadores miniaturizados, do tamanho de um cartão de crédito, que podem ser utilizados para tarefas de desenvolvimento ou como computadores pessoais.

Esta plataforma possui mais recursos computacionais que as anteriores, suportando a execução de sistemas operacionais e podendo ser utilizada em aplicações que exijam mais processamento, como na implementação de robôs de vigilância de vídeo (BOKADE; RATNAPARKHE, 2016) ou *clusters* Kubernetes (LESKINEN, 2020).

A seguir, uma foto da placa é apresentada na Figura 5, e o Quadro 3 detalha algumas das especificações desta plataforma.

Figura 5 – Placa Raspberry Pi 4B



Fonte: Raspberry Pi Trading Ltd. (2021).

Quadro 3 – Especificação Raspberry Pi 4B

Especificação	**Descrição**
Processador	BCM2711 Quad core 64-bit ARM-Cortex A72
Frequência operação (Clock)	1,5 GHz
Memória RAM	2 GB, 4 GB, ou 8 GB (A depender do modelo)
Conectividade	802.11 b/g/n/ac Wireless LAN Bluetooth 5.0 com BLE Porta Gigabit Ethernet
Conexões HDMI	2
Portas USB	2x USB2 2x USB3
GPIO	6x UART 6x I2C 5x SPI 1x SDIO 1x DPIO 1x PCM 2x PWM 3x GPCLK

Fonte: Raspberry Pi Trading Ltd. (2021).

3 AGENTES BDI EMBARCADOS

De acordo com Aliyuda (2016), as características de autonomia, proatividade, reatividade e habilidades sociais de agentes são, muitas vezes, similares às características de sistemas embarcados utilizados em CPS. Torna-se, portanto, válido explorar a utilização de agentes nos sistemas embarcados que apresentam estas características.

Diversos autores exploram a implementação de agentes embarcados. A seguir, são apresentadas algumas das abordagens utilizadas para embarcar agentes BDI e suas limitações.

3.1 ABORDAGENS EXISTENTES E LIMITAÇÕES EM AGENTES BDI EMBARCADOS

Em Aliyuda (2016), é discutida a utilização de sistemas multiagentes em CPS. O autor afirma que o paradigma de agentes é uma boa abordagem para a implementação de sistemas embarcados em CPS e sugere que a implementação de tais agentes pode ser realizada utilizando o *framework* JADE; entretanto, esta implementação não é realizada no trabalho. Apesar do trabalho descrever como a utilização de sistemas multiagentes pode ser benéfica no cenário apresentado, o *framework* JADE, proposto para implementação dos agentes, é baseado em Java, não sendo compatível com plataformas de desenvolvimento embarcado mais simples, que não tenham recursos computacionais suficientes para executar a *Java Virtual Machine* (JVM).

Barros *et al.* (2014), por sua vez, exploram a utilização do paradigma de agentes BDI na implementação de um veículo autônomo. A implementação realizada neste trabalho utiliza dois componentes principais, sendo o primeiro um computador pessoal, onde o raciocínio dos agentes é realizado através do Jason, e um microcontrolador, que se comunica com o computador através do *middleware* de comunicação Javino (PANTOJA; LAZARIN, 2015), e é responsável para interação com os sensores e atuadores do veículo, possibilitando a atuação no ambiente. Entretanto, o custo do sistema é alto, tendo em vista que dois componentes são necessários para a implementação da solução. Neste caso, o ideal seria que o raciocínio e as ações realizadas pelo agente ocorressem no mesmo componente, diminuindo o custo da solução. De forma análoga, em Manoel *et al.* (2018) se utilizam de dois sistemas computacionais na implementação de agentes no contexto de automação residencial, onde um microcontrolador é utilizado para sensoriamento do ambiente e o raciocínio do agente é executado em um computador pessoal. Assim como no trabalho anterior, também se utilizam o Jason e o Javino na implementação da solução, além de outras tecnologias na implementação dos agentes. Logo, o agente não é de fato embarcado,

e depende de outro sistema computacional para sua execução.

Em Bucheli *et al.* (2015), é realizada a implementação de um tradutor da linguagem de programação de agentes AgentSpeak para a linguagem C, possibilitando assim que a metodologia de programação orientada a agentes possa ser utilizada em um *Unmanned Aerial Vehicle* (UAV). A proposta deste trabalho é um avanço em relação aos demais, ao propor uma ferramenta que permite a utilização de uma linguagem de programação orientada a agentes para implementar uma solução diretamente em um sistema embarcado. Entretanto, este trabalho caracteriza-se como um tradutor de AgentSpeak para C concebido para uma aplicação específica, que possui como especificações a não utilização de instruções recursivas e de memória dinâmica. Portanto, esta não é uma solução genérica para implementação de agentes embarcados.

Torna-se evidente que a falta de ferramentas para implementação de agentes em plataformas computacionais comumente utilizadas em sistemas embarcados é um obstáculo que dificulta a utilização de agentes nestes sistemas. A seguir, serão descritos alguns dos desafios na concepção de uma ferramenta que permita a implementação de agentes BDI em plataformas embarcadas, tendo como base o interpretador de AgentSpeak Jason.

3.2 DESAFIOS NA CONCEPÇÃO DE UMA FERRAMENTA PARA IMPLEMENTAÇÃO DE AGENTES BDI EMBARCADOS

Considerando as restrições de recursos computacionais disponíveis em sistemas embarcados e o modelo de raciocínio prático do Jason, torna-se necessário realizar uma avaliação do funcionamento dos passos do modelo de raciocínio do interpretador. Com isto, é possível determinar quais os desafios na implementação destes passos para de agentes embarcados.

3.2.1 Desafios na implementação do ciclo de raciocínio do Jason

A seguir, discute-se a relevância e viabilidade de implementação dos passos de raciocínio do Jason, apresentados na Figura 2, na implementação de agentes BDI embarcados.

1. *(perceive)* Percepção do ambiente: a percepção é a característica que permite que agentes sejam sensíveis a mudanças no ambiente. A implementação deste passo no Jason conta com funções internas para realizar a percepção do ambiente. Entretanto, considerando as múltiplas padronizações de interfaces de entrada e saída de diferentes plataformas embarcadas, torna-se um desafio a implementação de mecanismos

de atualização da percepção do ambiente em um *framework* que forneça suporte multiplataforma;

2. *(BUF)* Atualização da base de crenças: no Jason, a atualização da base de crenças é realizada de forma dinâmica, em que crenças, no formato de predicados, são adicionadas, atualizadas e removidas da base de crenças no decorrer da execução do agente. No contexto embarcado, dois desafios emergem neste passo. O primeiro é a utilização de memória da base de crenças, que por não possui um tamanho definido, resulta em um desafio no gerenciamento de memória do agente, caso o mesmo possua muitas crenças. Portanto, é desejável que o agente possua uma base de crenças de tamanho fixo ou limitado, tendo em vista que estruturas de dados de tamanhos variáveis podem ocasionar problemas de funcionamento, como em situações em que o agente esgota a quantidade de memória disponível para utilização. Outro desafio é o custo computacional associado ao processamento das crenças em diversos algoritmos, como por exemplo no algoritmo de unificação (JUNIOR, 2015; MILLER; ESFANDIARI, 2021);

3. *(checkMail)* Recepção de mensagens: a recepção de mensagens facilita e viabiliza a interação entre agentes em plataformas multiagentes. Entretanto, a implementação deste passo em sistemas embarcados também traz seus desafios, pois diferentes plataformas possuem diferentes interfaces e métodos de acesso a redes de comunicação, o que também dificulta a implementação de uma solução genérica para a comunicação entre agentes;

4. *(SocAcc)* Seleção de mensagem "socialmente aceitável": antes de processar as mensagens recebidas, é necessário verificar se elas podem ser aceitas pelo agente. Esta verificação considera quem é o emissor e qual o propósito e conteúdo da mensagem, a qual pode, por exemplo, delegar um objetivo ao agente ou informar sobre uma nova crença. O processamento do propósito e conteúdo da mensagem pode ser um desafio, considerando que a base de crenças pode ter um tamanho limitado, por exemplo;

5. *(Se)* Seleção de evento: assim como a base de crenças, a não definição de um tamanho limitado para a fila de eventos pode ocasionar problemas na gestão da utilização de memória do agente. Logo, é desejado que o tamanho da fila de eventos seja fixo ou limitado. Esta limitação no tamanho da fila levanta uma questão interessante, relacionada ao comportamento do agente em casos nos quais a fila de eventos esteja cheia e novos eventos precisem ser adicionados;

6. *(Unify Event)* Recuperação dos planos relevantes: sendo o algoritmo de unificação do Jason um dos passos que utiliza maior tempo de processamento do ciclo de raciocínio do agente (JUNIOR, 2015; MILLER; ESFANDIARI, 2021), é necessário avaliar se o algoritmo pode ser simplificado ou otimizado para o contexto embarcado, pois as plataformas embarcadas geralmente possuem menos recursos computacionais de processamento quando, comparadas com computadores pessoais;

7. *(Check Context)* Determinação de planos aplicáveis: assim como no passo anterior, é necessário avaliar se o algoritmo de determinação de planos aplicáveis pode ser simplificado ou otimizado;

8. *(So)* Seleção de um plano aplicável: a seleção de um plano aplicável pode resultar na inserção de uma nova intenção na fila de intenções. Por sua vez, a fila de intenções deve ter tamanho fixo ou limitado e os casos em que esta fila está cheia e novas intenções precisem ser adicionadas devem ser considerados;

9. *(Si)* Seleção de uma intenção para execução: no Jason, as intenções são organizadas em uma fila, e a cada ciclo de raciocínio do agente, uma instrução da intenção à frente da fila é executada. Considerando que o *hardware* comumente utilizado em sistemas embarcados possui recursos computacionais limitados, a complexidade das instruções pode influenciar no tempo necessário para o agente completar a execução das mesmas e, consequentemente, reavaliar seus objetivos no ciclo de raciocínio posterior. A reavaliação constante dos objetivos do agente e percepção do ambiente são características importantes de agentes BDI, sendo assim desejável que, caso necessário, as instruções de planos sejam simplificadas para garantir que as limitações de *hardware* não afetem o comportamento do agente BDI no que diz respeito à reavaliação de objetivos e percepção do ambiente;

10. *(Execute Intention)* Execução de etapa da intenção: após executar uma instrução da intenção, é necessário determinar o que fazer com a intenção. Por exemplo, se a intenção for finalizada, não é necessário adicioná-la novamente à fila de intenções; caso contrário, ela deve ser adicionada ao final da fila de intenções. Além disso, uma intenção pode gerar um evento e, consequentemente, deve ser suspensa até que o evento seja processado; nestes casos, é necessário avaliar como tratar cenários em que a fila de eventos está cheia, por exemplo. Finalmente, a execução de uma intenção pode falhar, o que também pode gerar um evento de falha da execução da intenção.

3.2.2 Desafios relacionados à linguagem *AgentSpeak*

A sintaxe e as funcionalidades da linguagem *AgentSpeak* também trazem desafios na implementação do *framework*.

3.2.2.1 Ações internas

No Jason, é possível utilizar ações internas. Um agente pode, por exemplo, utilizar a ação `.wait(1000)` para "esperar" durante 1000 milisegundos (1 segundo), como mostrado no Código 2.

Código 2 – Exemplo de código com ações internas

```
1 +!esperar <- .wait(1000).
2 +!ola_mundo <- .print("Ola Mundo!).
3 +!dizer_oi <- .send(bob,tell,oi).
```

Entretanto, diferentes plataformas embarcadas podem ter diferentes formas de implementar um período de espera internamente. De forma análoga, a ação interna `.print("Ola Mundo!")` exige acesso à uma interface de saída para exibição da mensagem, o que também não ocorre de forma homogênea em plataformas embarcadas. O mesmo ocorre para a ação `.send(bob,tell,oi)`, a qual exige funcionalidades de rede para que a mensagem seja enviada entre agentes.

Logo, a heterogeneidade das interfaces de entrada, saída e comunicação é um desafio na implementação de algumas das ações internas suportadas pelo Jason.

3.2.2.2 Predicados

O interpretador Jason permite que crenças possam ser representadas através de predicados. Como mencionado anteriormente, a heterogeneidade dos termos associados às crenças pode ser um desafio nos algoritmos internos do agente, além de aumentar a complexidade do algoritmo de unificação. O Código 3 exemplifica crenças válidas para um agente.

Código 3 – Exemplo de código com predicados e proposições

```
1 calor.
2 temperatura(30).
3 estudante("joao").
```

No exemplo acima, a crença `calor` está em formato de proposição, enquanto as crenças `temperatura` e `estudante` são proposições que possuem termos dos tipos inteiro e *string*, respectivamente.

3.2.2.3 Avaliação das expressões lógicas

Múltiplos operadores podem ser utilizados no contexto de um plano. Alguns destes operadores são mostrados no Código 4.

Código 4 – Exemplo de código Jason com operadores de contexto

```
1 +!plano1 : temperatura(X) & X > 24 <- resfriar.
2 +!plano2 : temperatura(X) & X <= 23 <- parar_resfriar.
3 +!plano3 : not frio <- resfriar.
4 +!plano4 : fumaca & fogo <- soar_alarme.
```

No código acima, os seguintes operadores são utilizados:

- "`not`": negação lógica;
- "`>`": maior que;
- "`>`": menor ou igual que;
- "`&`" "e" lógico.

A utilização destes e de outros operadores, aliados às crenças em formatos de predicados, pode resultar em algoritmos de avaliação de expressões lógicas e unificação complexos e que possuem um alto custo computacional no raciocínio do agente.

3.2.2.4 Recursão

Técnicas de recursão são comuns em linguagens de programação e podem ser facilmente implementadas no Jason. O exemplo do Código 5 exemplifica um plano recursivo no Jason, onde a instrução `!plano` ocasiona a execução do plano recursivamente, até que a crença `continuar` deixe de existir na base de crenças do agente.

Código 5 – Exemplo de código recursivo

```
1 +!plano : continuar <- contador++
2                        !plano.
```

A implementação de planos recursivos em plataformas embarcadas requer cautela, tendo em vista que a pilha de planos das intenções, onde planos são empilhados para execução, deve ter tamanho fixo ou limitado. A limitação no tamanho da pilha de planos das intenções pode ocasionar a falha na execução de uma intenção, devido à ausência de espaço na pilha para adicionar novos planos. Com isto, é necessário que o programador do agente esteja ciente do funcionamento de seu programa e busque alcançar o balanço ideal entre o tamanho da pilha de planos e a utilização de memória do agente.

3.2.2.5 Anotações

Anotações são uma extensão do Jason que permitem que informações adicionais sejam adicionadas à crenças ou planos (BORDINI; HÜBNER; WOOLDRIDGE, 2007). O Código 6 mostra a sintaxe de uma anotação.

Código 6 – Exemplo de código com anotações

```
1 quente[source(bob)].
```

No Código 6, o agente adota a crença `quente` com a anotação `source(bob)`, ou seja, ele passa a acreditar em `quente`, tendo registrado que `bob` é a fonte (`source`) desta crença. Como mencionado em Bordini, Hübner e Wooldridge (2007), anotações permitem que o código AgentSpeak seja mais "elegante" e que a base de crenças possa ser gerenciada mais facilmente. No caso de agentes embarcados, o principal questionamento é relacionado ao custo de possuir esta funcionalidade na linguagem, considerando que ela não é essencial para o funcionamento do agente e os recursos computacionais são limitados nestas plataformas.

3.3 RESUMO DOS DESAFIOS

Finalmente, o Quadro 4 resume os desafios apresentados neste capítulo. Dada a inviabilidade de utilização do Jason na implementação de agentes BDI em plataformas com recursos computacionais limitados, seja devido ao uso de alocação dinâmica de memória, pela dependência do java, ou em decorrência dos algoritmos complexos, como o de unificação, por exemplo, torna-se necessário estudar alternativas que viabilizem a implementação destes agentes no contexto embarcado. No próximo capítulo, será apresentada uma proposta de desenvolvimento de um *framework* que possibilite a implementação de agentes BDI embarcados.

Quadro 4 – Resumo dos desafios identificados na implementação de agentes BDI embarcados

Seção	Descrição	Desafio
Ciclo de raciocínio do Jason	1. Percepção do ambiente	Heterogeneidade das interfaces de entrada de diferentes plataformas
	2. Atualização da base de crenças	Base de crenças necessita de tamanho fixo ou limitado Alto custo computacional para algoritmos que utilizem predicados
	3. Recepção de mensagens	Heterogeneidade das interfaces de comunicação de diferentes plataformas
	4. Seleção de mensagem "socialmente aceitável"	Processamento da mensagem
	5. Seleção de evento	Fila de eventos necessita de tamanho fixo ou limitado
	6. Recuperação dos planos relevantes	Complexidade do algoritmo de unificação
	7. Determinação de planos aplicáveis	Complexidade do algoritmo de unificação
	8. Seleção de um plano aplicável	Fila de intenções necessita de tamanho fixo ou limitado
	9. Seleção de uma intenção para execução	Instruções complexas podem exigir muitos recursos computacionais
	10. Execução de etapa da intenção	Tratamento de falhas devido a estruturas de dados cheias
Linguagem AgentSpeak	Ações internas	Heterogeneidade das interfaces de entrada, saída e comunicação de diferentes plataformas Custo computacional de implementação
	Predicados	Custo computacional de algoritmos de unificação
	Avaliação das expressões lógicas	Custo computacional de algoritmos de avaliação de expressões lógicas e unificação
	Recursão	Pilha de planos instanciados por intenção necessita de tamanho fixo ou limitado
	Anotações	Relevância Custo computacional de implementação

4 PROPOSTA PARA AGENTES BDI EMBARCADOS

Neste capítulo, é apresentada a proposta de desenvolvimento de um *framework* que permita a implementação de agentes BDI embarcados. Esta proposta leva em consideração as limitações e desafios expostos no capítulo anterior, bem como métodos convencionais de implementação de agentes BDI e de sistemas embarcados.

Este capítulo é dividido em quatro seções. A seção 4.1 descreve o método adotado para desenvolver agentes BDI. A seção 4.2 descreve uma série de adaptações no processo de tradução da especificação do agente para código executável, bem como em algumas representações dos elementos de BDI, para que os agentes possam manter as características de BDI e, ainda assim, serem executados em dispositivos com hardware limitado, típicos de sistemas embarcados. O ciclo de raciocínio do agente também requer adaptações para adequar-se às limitações do hardware, conforme descrito na seção 4.3. Por fim, a seção 4.4 descreve como as limitações do hardware e configuração das estruturas de dados devem ser tratadas na programação dos agentes.

4.1 METODOLOGIA DE DESENVOLVIMENTO

Conforme discutido no capítulo anterior, a variedade de plataformas utilizadas na implementação de sistemas embarcados e os recursos computacionais limitados das mesmas, aliada à heterogeneidade destes sistemas, torna a implementação de agentes BDI embarcados um desafio. Portanto, propõe-se que a implementação do *framework* para desenvolvimento dos agentes BDI embarcados ocorra de forma incremental, onde inicialmente é realizada a implementação e validação de funcionalidades mais simples, para posterior adição de funcionalidades mais complexas, caso os recursos computacionais das plataformas-alvo sejam suficientes.

É desejável que os agentes implementados através do *framework* proposto possam ser executados em plataformas que possuam requisitos mínimos de recursos computacionais equivalentes aos de uma plataforma situada em uma faixa entre um Arduino UNO e uma Raspberry Pi 4B, em termos de recursos computacionais. Apesar dos recursos computacionais disponíveis e as características destas plataformas serem distintas (o Arduino UNO possui um microcontrolador mais simples, enquanto a Raspberry Pi 4B suporta a execução de um sistema operacional), ambas são utilizadas em pesquisas de sistemas embarcados e no desenvolvimento de agentes BDI; considera-se, portanto, que uma plataforma com recursos computacionais situada nesta "faixa" possa ser utilizada para implementação do agente embarcado.

Finalmente, a programação do *framework* deve ser realizada em uma linguagem de programação compatível com as plataformas-alvo e que permita a gestão de recursos computacionais, como alocação de memória. Além disso, deve-se evitar a utilização de bibliotecas proprietárias e específicas de alguma plataforma. Portanto, indica-se a linguagem de programação C++, que além de atender aos requisitos apresentados, também oferece o paradigma de programação orientada a objetos, facilitando a modularização do código e apresentando características úteis, como herança e polimorfismo. Ademais, é desejável que os agentes possam ser programados através de uma linguagem de programação de agentes. Devido à sua popularidade, a linguagem AgentSpeak/Jason é indicada para a programação dos agentes. Entretanto, é importante ressaltar que a sintaxe, assim como as funcionalidades da linguagem e do interpretador, devem ser simplificadas, tendo em vista a complexidade de sua implementação.

O Quadro 5 resume os requisitos do *framework*, descritos nesta seção.

Quadro 5 – Resumo das características do *framework*

Característica	**Descrição**
Metodologia de desenvolvimento	Incremental
Plataforma-alvo	Multiplataforma, com requisitos mínimos compatíveis com um *harware* com recursos computacionais situados em uma faixa entre Arduino UNO e Raspberry Pi 4B
Linguagem de programação do *framework*	C++
Linguagem de programação dos agentes	AgentSpeak/Jason

4.2 TRADUÇÃO DO CÓDIGO AGENTSPEAK PARA C++

Os agentes devem apresentar as características do ciclo de raciocínio BDI, onde objetivos são continuamente reavaliados e o agente é sensível a mudanças no ambiente. Outro ponto importante é de que o *framework* não deve funcionar como um interpretador de uma linguagem de agentes. Para apresentar melhor desempenho, deseja-se que o agente seja compilado em um binário que contenha tanto as instruções necessárias para sua execução, quanto a *engine* de execução do ciclo de raciocínio BDI. A Figura 6 é um diagrama que exemplifica como o código do agente e o *framework* podem ser integrados no processo de compilação.

Para realizar a compilação do código do agente, em AgentSpeak, juntamente com o ciclo de raciocínio implementado no *framework*, propõe-se que o código do agente seja traduzido para um código equivalente em C++, onde as crenças, planos e objetivos são traduzidos para estruturas de dados, possibilitando que o código do agente, agora em C++, possa ser compilado juntamente com a *engine* BDI do *framework*. A Figura 7 é um

Figura 6 – Diagrama inicial do processo de compilação

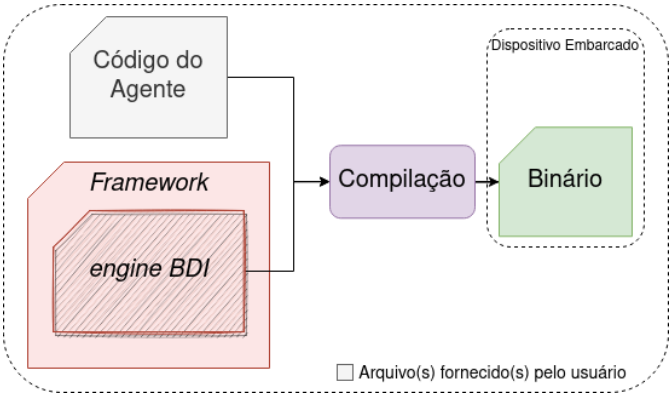
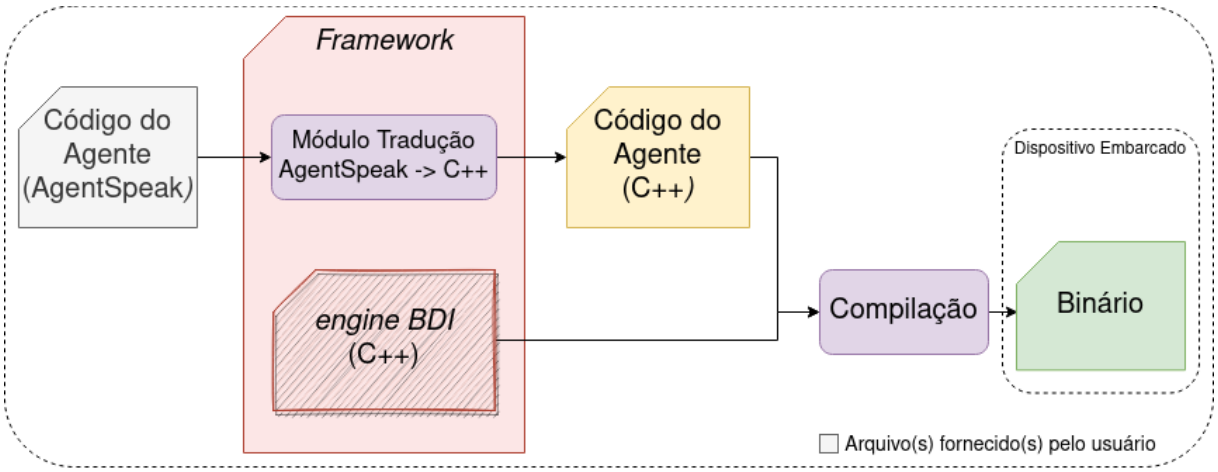


diagrama que exemplifica como o código do agente, traduzido para C++, e a *engine* BDI do *framework* podem ser compiladas em um mesmo binário.

Figura 7 – Diagrama do processo de compilação com código do agente traduzido para C++



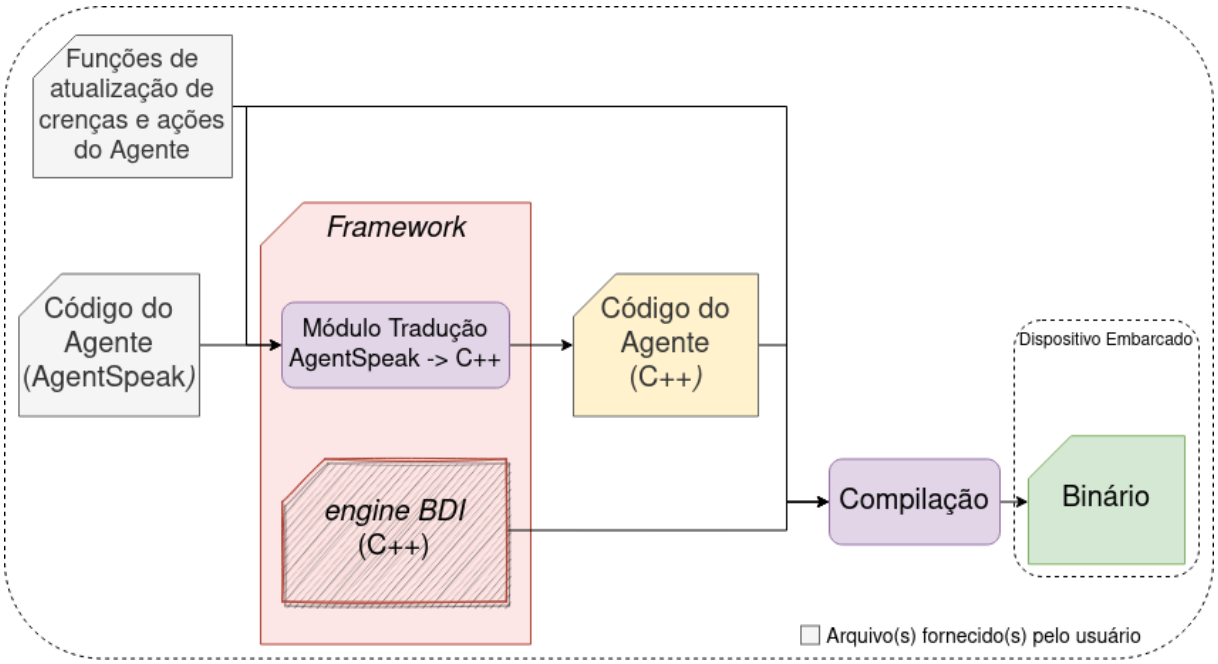
A tradução do código do agente de AgentSpeak para C++ possibilita determinar quais estruturas de dados devem ser utilizadas para o armazenamento das crenças, planos, intenções e eventos do agente. Considerando a quantidade de memória geralmente limitada em sistemas embarcados, é desejável que as estruturas de dados utilizadas pelo agente possuam tamanho limitado.

No caso da base de crenças, pode-se definir o tamanho máximo da estrutura de dados ao analisar o código do agente: ao mapear todas as crenças presentes no código do agente, é possível realizar uma estimativa de quantas crenças podem existir na base de crenças. De forma análoga, a biblioteca de planos é criada a partir do mapeamento de planos do código AgentSpeak, tendo seu tamanho definido no momento de sua criação. É importante ressaltar que as crenças e planos obtidos através da comunicação com outros agentes não são contabilizados neste cálculo, pois o *framework* não fornece suporte à

comunicação e coordenação entre agentes. Apesar do suporte a sistemas multiagentes ser uma característica importante do Jason, esta não é uma funcionalidade que foi considerada na primeira versão do *framework*, dados os desafios adicionais de comunicação entre agentes embarcados em diferentes plataformas computacionais.

Também devido ao desafio de fornecer suporte multiplataforma, o *framework* não fornece ações internas já implementadas, como no Jason. Portanto, é necessário que o programador do agente (usuário do *framework*) forneça as funções de ações e atualização das crenças. A Figura 8 é um diagrama que exemplifica o processo de tradução do código do agente e compilação do binário final, levando em consideração a *engine* BDI do *framework* e as funções para atualização das crenças e implementação das ações do agente.

Figura 8 – Diagrama do processo de compilação com funções de atualização de crenças e ações do agente



Finalmente, o Quadro 6 resume as características discutidas nesta seção.

Quadro 6 – Resumo das características de tradução do código AgentSpeak

Característica	**Descrição**
Método de execução do código do agente	Compilação, juntamente com a *engine* BDI do framework
Tamanho da base de crenças	De tamanho definido, limitado a partir do mapeamento do código do agente
Tamanho da biblioteca de planos	De tamanho definido, limitado a partir do mapeamento do código do agente
Funções de atualização de crenças e ações do agente	Fornecidas pelo usuário do *framework*
Ações internas	Não suportados
Comunicação entre agentes	Não suportada

4.3 FUNCIONAMENTO DOS AGENTES

Nesta seção, são apresentadas algumas características do funcionamento interno dos agentes programados através do *framework* proposto, assim como as decisões tomadas para superar alguns dos desafios expostos no capítulo anterior. É importante ressaltar que a metodologia de desenvolvimento incremental, aliada ao desafio de conceber um *framework* multiplataforma, resulta na simplificação e não implementação de certas funcionalidades no *framework*. Na sequência, são apresentadas as estruturas de dados presentes nos agentes implementados, seguida da explicação do ciclo de raciocínio BDI.

4.3.1 Estruturas de dados dos agentes implementados

Para descrever a proposta de ciclo de raciocínio, é necessário primeiro apresentar as principais estruturas de dados do agente: a base de crenças, a biblioteca de planos, a fila de eventos, a fila de intenções e a pilha de planos instanciados em uma intenção. A seguir, é realizada uma explicação breve de como cada estrutura funciona no *framework*.

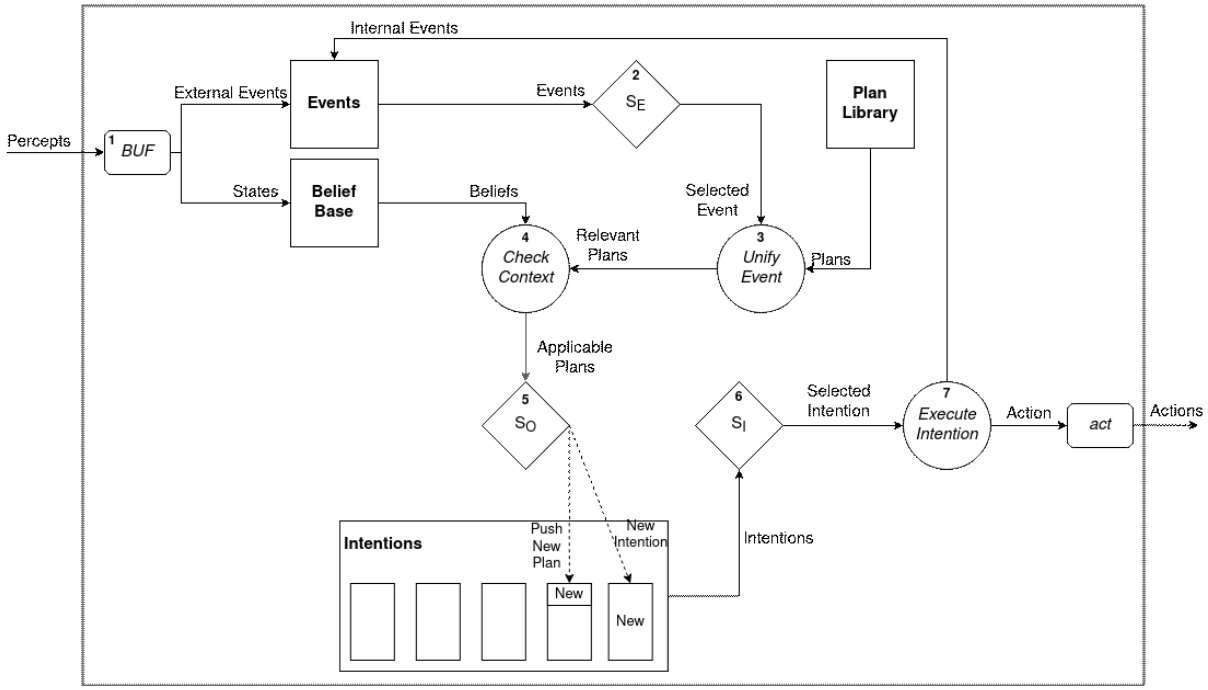
- Base de crenças: possui as crenças do agente e é completamente populada ao traduzir o código do agente de AgentSpeak para C++;
- Biblioteca de planos: possui os planos do agente e é criada ao traduzir o código do agente de AgentSpeak para C++;
- Fila de eventos: possui os eventos a serem processados pelo agente. Assim como no Jason, um evento da fila é processado a cada ciclo de raciocínio;

- Fila de intenções: possui todas as intenções do agente. A cada ciclo de raciocínio, uma etapa/instrução de uma intenção é executada. A escolha da intenção a ser executada geralmente ocorre através de um algoritmo de *round robin*, onde após uma intenção ser selecionada do início da fila, a mesma é executada e então adicionada ao final da fila, caso ainda hajam instruções a serem executadas;
- Pilha de planos instanciados em uma intenção: consiste de todos os planos a serem executados em uma intenção. O empilhamento de planos é resultado do processamento de eventos originados a partir da execução de instruções da própria intenção.

4.3.2 Ciclo de raciocínio

O ciclo de raciocínio do agente deve ser similar ao do Jason, com a exceção dos passos 3 e 4 do ciclo de raciocínio, referentes a recepção de mensagens e seleção de mensagem "socialmente aceitável", respectivamente. Como mencionado anteriormente, o suporte às funcionalidades de comunicação e coordenação de agentes não é implementado na primeira versão do *framework*. A Figura 9 mostra o ciclo de raciocínio prático do *framework* proposto. Uma versão ampliada deste diagrama está disponível no Apêndice A.

Figura 9 – Ciclo de raciocínio prático do *framework*



A seguir, são listados os passos da Figura 9.

1. Atualização da base de crenças a partir de novas percepções;
2. Seleção de evento;
3. Recuperação dos planos relevantes;
4. Determinação de planos aplicáveis;
5. Seleção de um plano aplicável;
6. Seleção de uma intenção para execução;
7. Execução de etapa de uma intenção.

Cada um desses passos é detalhado na sequência.

4.3.2.1 Atualização da base de crenças a partir de novas percepções

Este passo é referente a percepção do ambiente e atualização da base de crenças, que ocorrem em um único passo. Isto é possível devido ao formato da base de crenças do agente. Diferentemente do Jason, onde o agente armazena as crenças em sua base à medida que ele as adquire, o *framework* realiza o armazenamento de todas as crenças do agente desde o início de sua execução. Ou seja, ao realizar a "tradução" e o mapeamento das crenças do código original, em AgentSpeak, para o código C++ correspondente, o *framework* cria a base de crenças e armazena todas as crenças possíveis associadas ou a valores verdadeiros - que representam que o agente possui a crença - ou a valores falsos - que significam que o agente "não possui" esta crença. Portanto, a função de atualização de crenças do agente (`BUF`, na Figura 9), apenas atualiza o valor das crenças já existentes na base de crenças do agente, não sendo necessário criar novas crenças em tempo de execução. É importante lembrar que a funcionalidade de comunicação entre agentes não é implementada, e, por este motivo, não há crenças adicionais a serem adotadas pelo agente durante sua execução.

O Código 7 exemplifica um programa AgentSpeak com crenças em formato de proposição. Ele faz referência a três proposições: `segunda_feira`, `feriado` e `feliz`. Na linha `1`, o agente adota a crença `feriado`; na linha `2`, o plano indica que, a partir do momento que o agente acreditar na crença `segunda_feira`, ele deve adotar a crença `feliz`, caso ele já acredite na crença `feriado`. O Quadro 7 exemplifica como as crenças do Código 7 seriam armazenadas na base de crenças do agente, após o processo de "tradução" e mapeamento do código AgentSpeak para C++.

Código 7 – Exemplo de código AgentSpeak com crenças em formato de proposição

```
1 feriado.
2 +segunda_feira : feriado <- +feliz.
```

Quadro 7 – Base de crenças do agente a partir da execução do Código 7

Crença	**Valor**
`feriado`	Verdadeiro
`segunda_feira`	Falso
`feliz`	Falso

Apesar das limitações deste método, a criação e população da base de crenças com tamanho definido permite um melhor controle da estrutura de dados correspondente, evitando casos em que o agente necessita de mais recursos computacionais do que os disponíveis, além de resultar em algoritmos de unificação mais simples. O Algoritmo 2 é um pseudocódigo que descreve o funcionamento deste passo. Na linha `1`, para cada `crença` do agente na base de crenças (`base_de_crenças`), o agente executa a função de atualização de crença fornecida pelo programador (`funcao_atualizacao()`) e atualiza o valor da crença (`crença.estado`).

Algoritmo 2 *Pseudocódigo da função de atualização da base de crenças*

```
1: for crença in base_de_crenças do
2:     estado ← crença.funcao_atualizacao(crenca)
3:     crença.estado ← estado
4: end for
```

4.3.2.2 Seleção de evento

Assim como no Jason, a cada ciclo de raciocínio, o agente implementado através do *framework* proposto seleciona um evento da fila de eventos para realizar seu processamento.

4.3.2.3 Recuperação de planos relevantes

Devido à ausência de variáveis, a recuperação de planos relevantes pode ser realizada através de um algoritmo de unificação mais simples do que o do Jason, pois apenas a proposição e o tipo de evento são levados em consideração.

Além disso, anotações também não são incorporadas ao *framework*, em favor de um algoritmo de unificação mais simples.

4.3.2.4 Determinação de planos aplicáveis

O contexto dos planos é essencial no processo de determinação de planos aplicáveis. O Jason fornece suporte a diversos operadores lógicos e possui um algoritmo robusto

de avaliação de contexto. Entretanto, a utilização de proposições, que assumem apenas valores booleanos, torna alguns dos operadores suportados pelo Jason desnecessários, como o operador "`>`" (maior que).

A simplificação dos operadores de contexto torna o processo de determinação de planos aplicáveis computacionalmente mais simples, o que é desejável neste trabalho. Consequentemente, torna-se necessário avaliar quais operadores são necessários para a implementação de contextos, baseados em lógica booleana. Neste caso, indica-se a adoção de apenas o operador "`&`" ("e" lógico), que permite que o algoritmo de avaliação de contexto seja mais simples. É importante ressaltar que o operador "ou" lógico também é importante, mas este pode ser implementado de forma indireta, como mostrado no Código 8. Neste código, o agente adota a crença `feliz`, caso passe a acreditar em `segunda_feira` e possua as crenças `feriado` **ou** `dia_de_pagamento`.

Código 8 – Exemplo de código AgentSpeak sem operador "ou" lógico

```
1 +!segunda_feira : feriado <- +feliz.
2 +!segunda_feira : dia_de_pagamento <- +feliz.
```

4.3.2.5 Seleção de um plano aplicável

Para a seleção de um plano aplicável, o *framework* pode seguir o comportamento padrão do Jason e executar o primeiro plano aplicável disponível no código do agente. Por exemplo, no Código 9, a adição da crença `atrasado` à base de crenças resulta na adição de um evento correspondente à fila de eventos. Ao processar este evento, o agente recupera os planos relevantes e determina que dois planos são aplicáveis, como exemplificado pelos planos `+atrasado` nas linhas `1` e `2` do Código 9. Neste caso, a seleção do plano ocorre pela ordem dos planos do programa fonte, sendo o plano da linha `1` o selecionado pelo agente.

Código 9 – Exemplo de seleção de plano aplicável

```
1 +atrasado <- correr.
2 +atrasado <- andar.
```

4.3.2.6 Seleção de uma intenção para execução

No *framework* proposto, ao executar uma instrução onde o agente adota um objetivo ou atualiza o valor de uma crença, um evento correspondente à execução desta instrução é gerado e adicionado à fila de eventos. A intenção, por sua vez, é suspensa, até que o evento seja processado. Logo, a seleção de uma intenção para execução considera apenas intenções ativas, isto é, intenções que não estejam suspensas, aguardando pelo processamento do evento.

4.3.2.7 Execução de etapa de uma intenção

Assim como no Jason, a cada execução do ciclo de raciocínio, o agente seleciona uma intenção e executa apenas uma instrução do plano correspondente. As instruções suportadas pelo *framework* podem ser conferidas no Código 10.

Código 10 – Instruções suportadas pelo *framework*

```
1 +!exemplo_plano <- +adotar_crenca;
2                    -adotar_crenca;
3                    !sub_objetivo;
4                    !!novo_objetivo.
5 +!sub_objetivo  <- agir.
6 +!novo_objetivo <- agir.
7 -!exemplo_plano <- comunicar_falha.
```

Na linha 1, ao executar a instrução `+adotar_crenca`, o valor correspondente a esta crença, que já existe na base de crenças, é alterado para "verdadeiro". Além disso, a intenção em execução é suspensa e um evento de "adição" de crença é adicionado a fila de eventos, caso o valor anterior da crença seja "falso". Ao processar o evento, o agente irá avaliar se há planos aplicáveis, ou seja, planos relacionados à "adição" da crença na base de crenças. Caso hajam, um destes planos será adicionado à pilha de execução da intenção original e a mesma será tornada "ativa". Caso não hajam planos aplicáveis, como no Código 10, a intenção será tornada "ativa". De forma similar, a linha 2 possui a instrução `-adotar_crenca`. Esta instrução é similar a anterior, mas o evento gerado é para a alteração do valor da crença de "verdadeiro" para "falso".

Na linha 3, a instrução `!sub_objetivo` resulta na criação de um evento de adição de objetivo, o qual é adicionado à fila de eventos do agente e a intenção em execução é suspensa. Após processar o evento, a intenção é tornada "ativa" novamente. Se houverem planos aplicáveis para alcançar este objetivo, um destes planos é adicionado à pilha de planos da intenção original. Caso contrário, a intenção original falha.

Na linha 4, `!!novo_objetivo` resulta em um evento de adição de objetivo, o qual é adicionado à fila de eventos. Diferentemente das instruções anteriores, a intenção atual não é suspensa ao executar esta instrução. O operador `!!` não suspende a execução da intenção: ele adiciona um novo objetivo para o agente, que não é condicionado à intenção em execução. Além disso, o plano resultante do processamento da instrução `!!sub_objetivo` não é adicionado à pilha de planos da intenção original.

Finalmente, nas linhas 5, 6 e 7, as instruções `agir` e `comunicar_falha` são ações a serem tomadas pelo agente, as quais devem ser fornecidas pelo usuário e podem tanto resultar em um sucesso ou falha.

Considerando o processamento das instruções descrito acima e a estratégia de limitar o tamanho das estruturas de dados do agente, torna-se necessário avaliar como o agente deve se comportar nos casos em que as estruturas de dados estão cheias e novos valores precisam ser adicionados às mesmas. Na sequência, são detalhadas as decisões tomadas para garantir que o agente minimize o número de falhas durante a execução de intenções.

4.3.3 Limitação das estruturas de dados do agente

Dadas as estruturas de dados do agente e o funcionamento descrito, casos críticos de estruturas de dados cheias podem ocorrer. O tratamento destes casos é descritos a seguir.

4.3.3.1 Fila de eventos

Eventos podem ser originados a partir da atualização de crenças ou da execução de planos. Nesta proposta, é sugerido que caso a fila de eventos esteja cheia e um novo evento precise ser adicionado à fila, este evento seja descartado.

Para os eventos gerados a partir da atualização de crenças, o descarte destes eventos pode ocasionar a não execução de planos relacionados a estas operações. Adicionalmente, o descarte de eventos gerados a partir da execução de intenções resulta na falha da execução do plano correspondente, ocasionando a falha da intenção. Consequentemente, enquanto o primeiro caso impede que uma possível intenção seja iniciada, o segundo caso resulta na falha da intenção em execução.

A vantagem desta abordagem está não apenas em sua simplicidade, mas também na garantia de que eventos que estão na fila de eventos serão processados, garantindo assim que intenções que estejam suspensas voltem a estar ativas e continuem sua execução. Este método “conservador“ busca garantir que os eventos e intenções já criados permaneçam em suas respectivas filas, garantindo assim uma maior “consistência” no comportamento do agente.

4.3.3.2 Fila de intenções

De forma similar à fila de eventos, novas intenções são descartadas caso a fila de intenções esteja cheia, buscando garantir que as intenções que já estejam em execução não sejam interrompidas.

4.3.3.3 Pilha de planos instanciados em uma intenção

Caso a pilha de planos de uma intenção esteja cheia e um novo plano precise ser adicionado, a intenção falha.

O Quadro 8 resume as considerações realizadas sobre o funcionamento dos agentes.

Quadro 8 – Resumo do funcionamento do agente

Característica	**Descrição**
Passos do ciclo de raciocínio do agente	1. Percepção do ambiente e atualização da base de crenças 2. Seleção de evento 3. Recuperação dos planos relevantes 4. Determinação de planos aplicáveis 5. Seleção de um plano aplicável 6. Seleção de uma intenção para execução 7. Execução de etapa da intenção
Fila de eventos cheia	Novos eventos são descartados
Fila de intenções cheia	Novas intenções são descartadas
Pilha de planos instanciados em uma intenção cheia	Novos planos instanciados resultam em falha da intenção

4.4 PROGRAMAÇÃO DOS AGENTES

Finalmente, torna-se necessário realizar a configuração do tamanho da fila de eventos, fila de intenções e pilha de planos instanciados por intenção. Para tal, sugere-se que um arquivo contendo estas configurações seja fornecido no processo de tradução do código AgentSpeak do agente, como ilustrado na Figura 10.

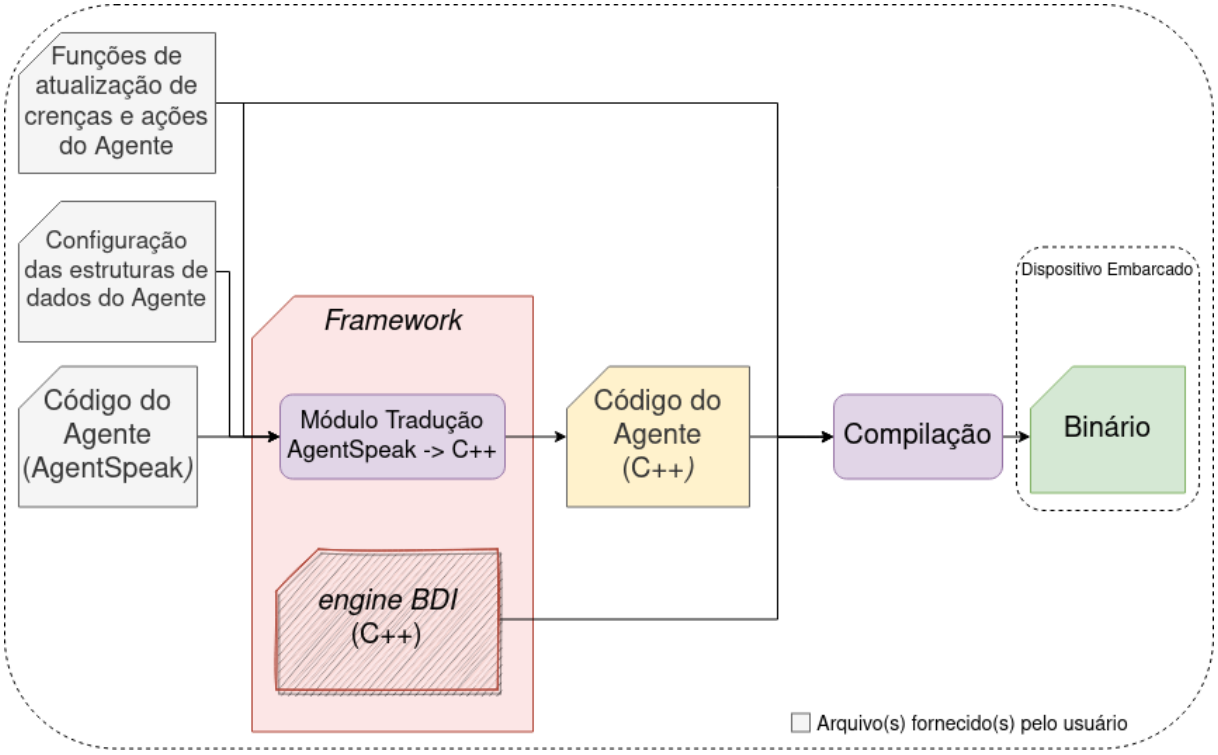
Portanto, no diagrama da Figura 10, é possível ver que três arquivos são esperados para utilização do *framework*: o código do agente, em AgentSpeak; as funções de atualização de crenças e ações do agente, em C++; e o arquivo de configuração que determina o tamanho das estruturas de dados do agente.

O Quadro 9 resume como a proposta deste capítulo busca abordar os desafios discutidos no Capítulo 3. No próximo capítulo, será descrita a implementação do *framework* proposto, com enfoque no ciclo de raciocínio BDI e na tradução do código AgentSpeak para C++.

Quadro 9 – Comparativo entre a proposta e os desafios descritos no Capítulo 3

Seção	**Desafio**	**Proposta**
Ciclo de raciocínio do Jason	1. Percepção do ambiente	Realizada através de funções fornecidas pelo programador do agente
	2. Atualização da base de crenças	Base de crenças é criada com tamanho fixo a partir da tradução do código AgentSpeak para C++
	3. Recepção de mensagens	Este passo não faz parte do ciclo proposto
	4. Seleção de mensagem "socialmente aceitável"	Este passo não faz parte do ciclo proposto
	5. Seleção de evento	Fila de eventos possui tamanho fixo Caso cheia, novos eventos são descartados
	6. Recuperação dos planos relevantes	Base de planos é criada com tamanho fixo a partir da tradução do código AgentSpeak para C++ Algoritmo simplificado
	7. Determinação de planos aplicáveis	Algoritmo simplicado
	8. Seleção de um plano aplicável	Fila de intenções possui tamanho fixo Caso cheia, novas intenções são descartados
	9. Seleção de uma intenção para execução	A seleção de uma intenção para execução considera apenas intenções não suspensas e a complexidade da instrução a ser executada é determinada pela implementação das ações pelo programador do agente
	10. Execução de etapa da intenção	Tratamento de falhas em casos de fila de eventos cheia e pilha de planos instanciados cheia
Linguagem AgentSpeak	Ações internas	Não suportadas
	Predicados	Não suportados
	Avaliação das expressões lógicas	Simplificada, com suporte ao operador ("e") lógico no contexto
	Recursão	Suportada, permitindo que o programador do agente configure o tamanho da pilha de planos instanciados por intenção
	Anotações	Não suportadas

Figura 10 – Diagrama do processo de compilação completo do agente



5 IMPLEMENTAÇÃO DO FRAMEWORK

Este capítulo descreve a implementação do *framework* seguindo as características descritas no capítulo anterior. A implementação do *framework* é dividida em dois módulos. O primeiro módulo implementa a *engine* BDI, com a utilização da linguagem de programação C++ para implementação do ciclo de raciocínio do agente. No segundo módulo, é realizada a implementação da tradução do código AgentSpeak para C++.

5.1 IMPLEMENTAÇÃO DA *ENGINE* BDI

A implementação da *engine* BDI tem como base o ciclo de raciocínio descrito na seção 4.3.2 do Capítulo 4. Inicialmente, foi realizada a implementação da classe do agente, que realiza a execução do ciclo de raciocínio através do método `run()`. Ou seja, a cada execução do método `run()`, uma iteração do ciclo de raciocínio é executada. O Código 11 mostra a implementação do método `run()`.

Código 11 – Implementação do método `run()`

```
1  void Agent::run()
2  {
3    beliefs->update(events);
4
5    if (!events->is_empty())
6    {
7      event_to_process = events->get_event();
8      plan_to_act = plans->revise(&event_to_process, beliefs);
9      if (plan_to_act) {
10       intentions->add_intention(plan_to_act, &event_to_process);
11     }
12   }
13
14   if (!intentions->is_empty())
15   {
16     intentions->run_intention_base(beliefs, events, plans);
17   }
18 }
```

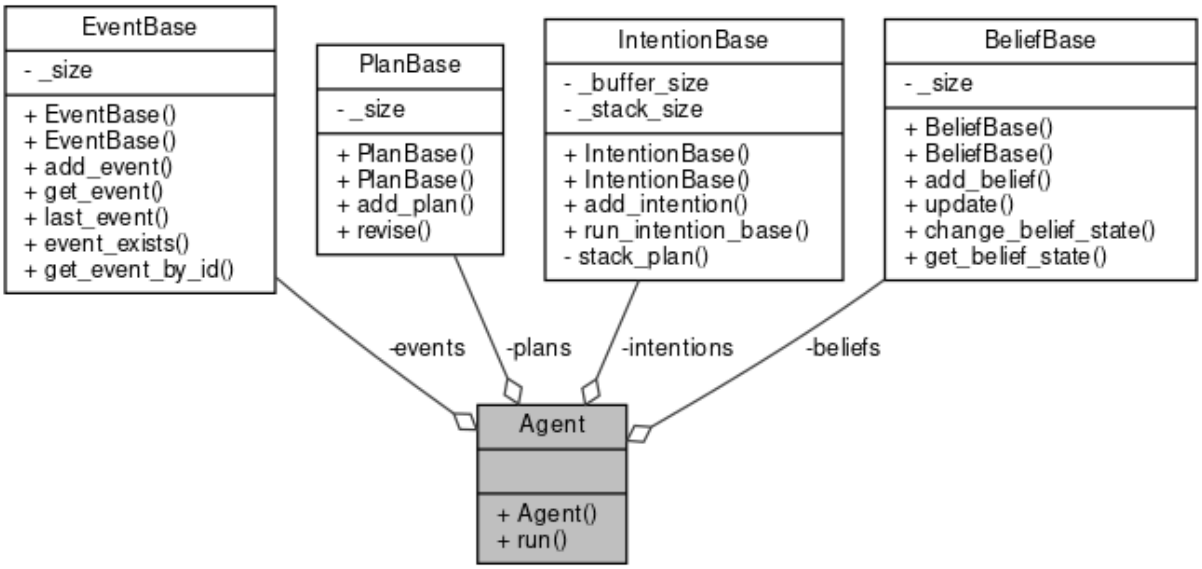
A seguir, são identificadas as etapas do ciclo de raciocínio do agente no Código 11.

- Linha 3: percepção do ambiente e atualização da base de crenças;
- Linhas 5-7: seleção de evento;
- Linha 8: recuperação dos planos relevantes;
- Linha 8: determinação de planos aplicáveis;
- Linhas 8-11: seleção de um plano aplicável;
- Linhas 14-16: seleção de uma intenção para execução;
- Linha 16: execução de etapa de uma intenção.

No Código 11 a recuperação de planos relevantes, determinação de planos aplicáveis e seleção de um plano aplicável ocorrem através do método `revise(&event_to_process, beliefs)` da classe `PlanBase`, referente à biblioteca de planos. De forma análoga, a seleção de uma intenção para execução e a execução de etapa da intenção ocorrem através do método `run_intention_base(beliefs, events, plans)`, da classe `IntentionBase`, referente à fila de intenções.

A Figura 11 é o diagrama de classe simplificado da classe `Agent`, gerado através da documentação criada pelas ferramentas Doxygen (HEESCH, 2021) e Graphviz (GRAPHVIZ, 2021). O diagrama completo está disponível no Apêndice B e na documentação do *framework* (SANTOS, M. M. dos, 2021b).

Figura 11 – Diagrama de classe simplificado da classe `Agent`



Como mostrado na Figura 11, as classes `EventBase`, `PlanBase`, `IntentionBase` e `BefiefBase`, referentes à fila de eventos, biblioteca de planos, fila de intenções e base de crenças, respectivamente, possuem tamanhos limitados pelos atributos `_size`. As classes do *framework* foram implementadas de forma modularizada, permitindo que métodos possam ser alterados para adição de funcionalidades ou alteração do comportamento do agente. Por exemplo, o método `add_event(Event event)`, que adiciona eventos à fila de eventos, pode ser alterado de sua implementação original, mostrada no Código 12, para adicionar eventos à fila através de uma instrução diferente.

Código 12 – Implementação do método `add_event(Event event)`

```
1  bool EventBase::add_event(Event event)
2  {
3    if (this->is_full())
4    {
5      return false;
6    }
7
8    _pending_events.push_front(event);
9    return true;
10 }
```

Na linha 3, antes de adicionar um evento à fila de eventos, o agente verifica se a mesma está cheia. Caso a fila não esteja cheia, o agente adiciona o evento à fila, na linha 8. De forma similar, o tipo de estrutura de dados utilizada para armazenar os eventos também pode ser alterada, desde que os métodos correspondentes para adição, consulta, e remoção de item da fila também sejam modificados.

Com a implementação da *engine* BDI, torna-se possível executar o ciclo de raciocínio do agente. Para realizar a programação do agente, torna-se necessário realizar a tradução do código AgentSpeak para C++.

5.2 IMPLEMENTAÇÃO DO MÓDULO DE TRADUÇÃO AGENTSPEAK PARA C++

A tradução do código AgentSpeak para C++ ocorre em duas etapas. Na primeira etapa, o código AgentSpeak é mapeado para identificar quais instruções estão presentes no código do agente. Na segunda etapa, o código C++ correspondente é escrito, levando em consideração o mapeamento do código do agente, as funções de atualização de crenças e ação no ambiente e as configurações das estruturas de dados internas do agente.

Para realizar o mapeamento do código AgentSpeak, utiliza-se o pacote `jason.asSyntax` do interpretador Jason. Com este pacote, é possível identificar a estrutura de cada crença, plano e instrução do código AgentSpeak, utilizando a linguagem de programação Java. O

Código 13 é um exemplo de código AgentSpeak suportado pelo módulo de tradução do *framework*.

Código 13 – Código AgentSpeak a ser traduzido

```
1 happy.
2 +!hello : happy <- say_hello.
```

No Código 13, pode-se identificar a crença `happy`, a qual inicia com o valor "verdadeiro", e o plano `+!hello`. O plano `+!hello` possui um contexto de tamanho unitário e um corpo de instruções também de tamanho unitário, definido pela ação `say_hello`. As funções de atualização da crença `happy` e da ação `say_hello` são fornecidas separadamente, como mostrado no Código 14. A identificação das funções de atualização de crenças e ação no ambiente é realizada pelos prefixos `update_` e `action_`, respectivamente.

Código 14 – Funções de atualização de crenças e ação no ambiente

```
1  #include <iostream>
2
3  bool update_happy(bool var)
4  {
5    return true;
6  }
7
8  bool action_say_hello()
9  {
10   std::cout << "Hello world!" << std::endl;
11   std::cout << "I am an agent and I will keep running until " <<
12                "I am terminated" << std::endl;
13   return true;
14 }
```

Tendo o código em AgentSpeak e as funções de atualização de crenças e ações do agente, é necessário ainda determinar o tamanho das estruturas de dados do agente. Esta configuração é realizada separadamente, como mostrado no Código 15.

Código 15 – Arquivo de configuração das estruturas de dados do agente

```
1 EVENT_BASE_SIZE=6
2 INTENTION_BASE_SIZE=10
3 INTENTION_STACK_SIZE=4
```

No Código 15, as estruturas de dados do agente possuem seu tamanho determinado: a fila de eventos possui tamanho 6, a fila de intenções possui tamanho 10, e a pilha de planos instanciados por intenção possui tamanho 4. Como mencionado no Capítulo 4, o tamanho da base de crenças e da biblioteca de planos é determinado ao mapear o código

do agente. Consequentemente, para o Código 15, o tamanho da base de crenças e da biblioteca de planos é unitário.

Logo, tendo como base o código do agente, suas funções e a configuração de suas estruturas de dados, pode-se realizar o processo de escrita do código C++. O Código 16 é o construtor da classe `AgentSettings`, gerado a partir da tradução do código AgentSpeak. O código completo em C++, gerado a partir da tradução do código AgentSpeak, está disponível no Apêndice C.

Código 16 – Construtor da classe `AgentSettings` no código C++

```
1  AgentSettings()
2  {
3    belief_base = BeliefBase(1);
4    event_base = EventBase(6);
5    plan_base = PlanBase(1);
6    intention_base = IntentionBase(10, 4);
7
8    Belief belief_happy(0, update_happy, true);
9    belief_base.add_belief(belief_happy);
10
11   Proposition prop_0(1);
12   context_0 = Context(1);
13   body_0 = Body(1);
14
15   Proposition prop_0_happy(0);
16   ContextCondition cond_0_0(prop_0_happy);
17   context_0.add_context(cond_0_0);
18
19   Proposition prop_0_body_0(2);
20   BodyInstruction inst_0_0(BodyType::ACTION, prop_0_body_0,
       action_say_hello);
21   body_0.add_instruction(inst_0_0);
22
23   Plan plan_0(EventOperator::GOAL_ADDITION, prop_0, &context_0, &body_0)
       ;
24   plan_base.add_plan(plan_0);
25 }
```

No Código 16, as linhas 3, 4, 5 e 6 se referem à criação da base de crenças, fila de eventos, biblioteca de planos e fila de intenções, respectivamente. O tamanho da base de crenças (1) e da biblioteca de planos (1) é definido a partir do mapeamento do código do agente, enquanto o tamanhos das filas de eventos (6) e de intenções (10) e da pilha de planos instanciados por intenção (4) é definido a partir do arquivo de configuração, definido no Código 15.

A tradução das instruções do Código 13, em AgentSpeak, é exibida nas linhas 8 à 24 do Código 16. As linhas 8 e 9 se referem à criação da crença `happy` na base de crenças, com o valor "verdadeiro". As linhas 11 à 21, por sua vez, se referem à criação do contexto e do corpo do plano `+!hello`. Nas linhas 23 e 24, o plano é criado e adicionado à biblioteca de planos. Das variáveis criadas, vale destacar que as proposições são representadas pela classe `Proposition`, a qual utiliza variáveis do tipo `uint8_t`, com o intuito de utilizar menos memória.

5.3 IMPLEMENTAÇÕES ADICIONAIS

As instruções para compilação do agente constam em um arquivo `Makefile` (GNU, 2020) (SANTOS, M. M. dos, 2020). Os alvos (*targets*) existentes neste arquivo `Makefile` permitem que seja realizada a compilação do agente e de um executável com 71 testes unitários das classes da *engine* BDI, além da geração da documentação do código, realizada através das ferramentas Doxygen e Graphviz.

Adicionalmente, também foi realizada a implementação de um fluxo de *Continuous Integration* (CI) (SANTOS, M. M. dos, 2021c) em que, a cada ação de *push* no repositório do projeto, realiza a compilação e execução dos testes unitários, verificação de *memory leaks* de um agente-exemplo utilizando a ferramenta `valgrind` (VALGRIND DEVELOPERS, 2021) e publicação da documentação de código gerada em uma página *web* (SANTOS, M. M. dos, 2021a).

Adicionalmente, foi criado um *wesbite* (`https://embedded-bdi.github.io/`) para documentação do *framework* e publicação de seu manual de utilização (SANTOS, M. M. dos, 2021b).

6 RESULTADOS

Para validar a implementação do *framework*, iremos avaliar a possibilidade de embarcar agentes desenvolvidos pelo *framework* em uma plataforma de desenvolvimento embarcado. Para tal, foi realizada a implementação de várias versões de agentes para uma aplicação-exemplo na plataforma NodeMCU v3 da fabricante Lolin, apresentada na subseção 2.5.2. A escolha desta plataforma foi baseada em seus recursos computacionais, os quais devem ser suficientes para execução do agente, além de seu baixo custo e popularidade. Para o experimento de avaliação, foi realizada a implementação de uma aplicação cujos requisitos são apropriadamente atendidos por um agente BDI. Para isso, utilizou-se um cenário também empregado para ilustrar o desenvolvimento de agentes BDI com Jason (HÜBNER, 2021). Este cenário é descrito na seção 6.1, onde é realizada a descrição da aplicação e das versões implementadas através do *framework* desenvolvido. Além disso, também é apresentada a configuração dos experimentos na plataforma embarcada. Posteriormente, na seção 6.2, é feita a análise de resultados das implementações realizadas, baseada nas métricas de tamanho de imagem gerado para os programas e tamanho do código.

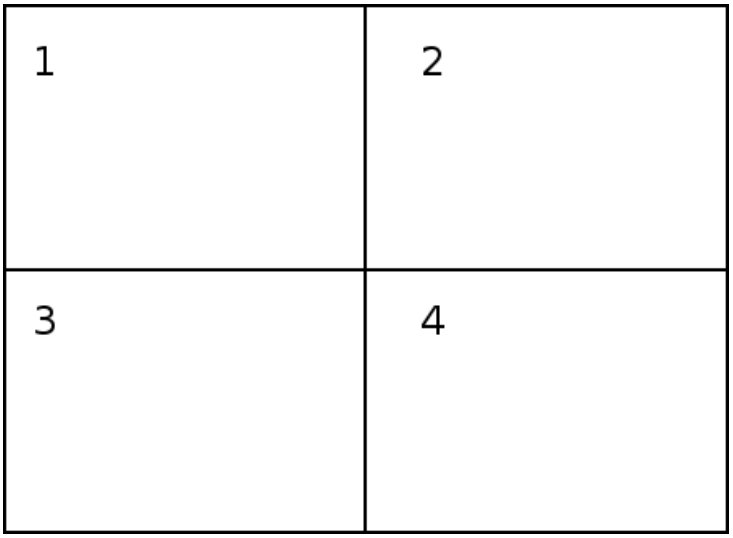
6.1 ASPIRADOR

A aplicação consiste em um aspirador situado em um ambiente com quatro espaços, como mostrado na Figura 12. Neste ambiente, o aspirador deve se movimentar continuamente, seguindo a direção das posições 1-2-4-3, e, ao detectar que uma posição está suja, realizar a sucção da sujeira. Esta aplicação é baseada em um exemplo existente na documentação do Jason (HÜBNER, 2021).

Neste trabalho, é realizada a implementação de três versões do aspirador:

- Versão 1: como descrito anteriormente, a movimentação e limpeza são realizadas continuamente;
- Versão 2: adição da funcionalidade de "iniciar" o aspirador à versão 1. O aspirador sempre inicia seu funcionamento na posição 1 e segue seu caminho original, se movimentando para as posições 2, 4, 3 e novamente para a posição 1, onde encerra sua movimentação e permanece parado até que lhe seja indicado para iniciar novamente;
- Versão 3: adição da funcionalidade de "esvaziar o depósito de sujeira" à versão 2. Após iniciar seu funcionamento, o aspirador segue sua estratégia de movimentação e limpeza e, ao detectar que seu depósito de sujeira está cheio, se dirige a posição

Figura 12 – Representação do ambiente-exemplo

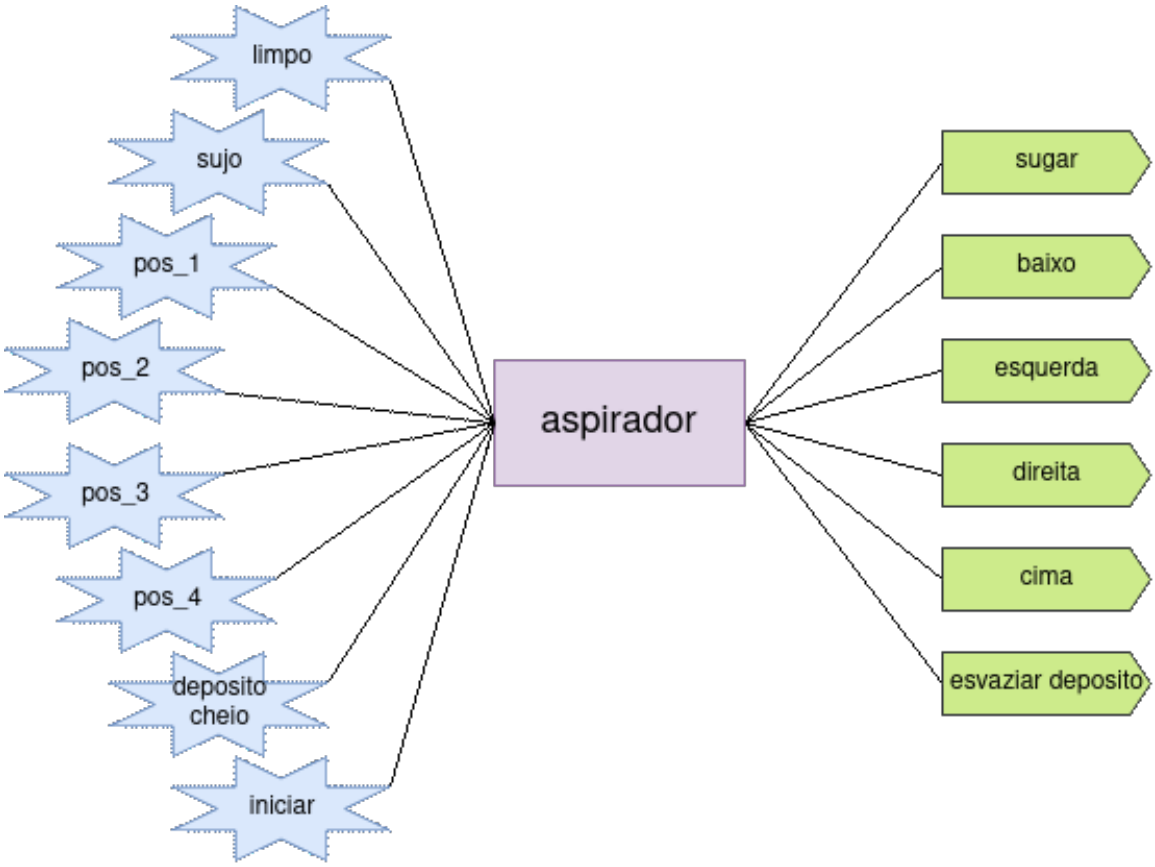


Fonte: Hübner (2021).

1 para esvaziá-lo. Posteriormente, o aspirador retorna a posição original e continua seu processo de limpeza.

A Figura 13 mostra as percepções do agente, em azul, e as ações que o mesmo pode realizar, em verde.

Figura 13 – Diagrama de crenças e ações do aspirador

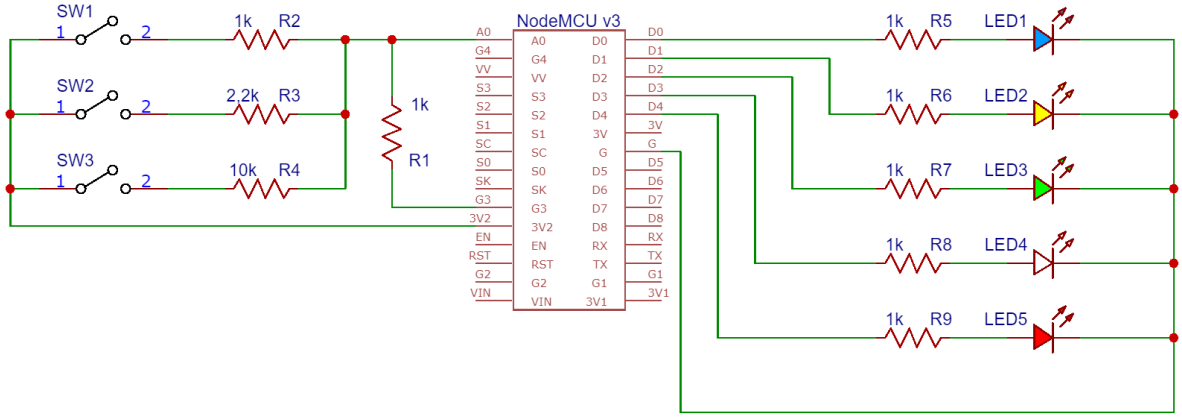


A seguir, é detalhada a configuração do circuito utilizado para implementação das versões descritas.

6.1.1 Configuração dos experimentos

Como o objetivo principal das implementações do aspirador é avaliar a possibilidade de embarcar os agentes implementados através do *framework*, não é necessário que as implementações sejam realizadas em um *hardware* que desempenhe as funções de movimentação e sucção. Portanto, pode-se realizar a implementação dos sensores referentes as crenças de posição e sujeira através de botões, enquanto as ações do agente podem ser representadas por indicativos visuais, como *leds*. Portanto, um circuito eletrônico equivalente, com *switches* e *leds*, pode ser montado juntamente à plataforma Lolin NodeMCU v3, permitindo a simulação das crenças e ações do agente. A Figura 14 é um esquemático do circuito implementado utilizando resistores, *switches*, *leds* e a plataforma Lolin NodeMCU v3. Na Figura 14, os *leds* `LED1`, `LED2`, `LED3` e `LED4` equivalem aos espaços 1, 2, 3 e 4, respectivamente. Estes leds são acendidos conforme a localização do agente. O `LED5` indica que o agente realizou as ações de sugar ou esvaziar o depósito de sujeira. Por sua vez, os *switches* `SW1`, `SW2` e `SW3` são utilizados para alterar as crenças de iniciar o ciclo de movimentação do aspirador, presença de sujeira (na posição atual do aspirador) e depósito de sujeira cheio. Enquanto os *leds* são conectados à saídas digitais, os *switches* são conectados a uma entrada analógica.

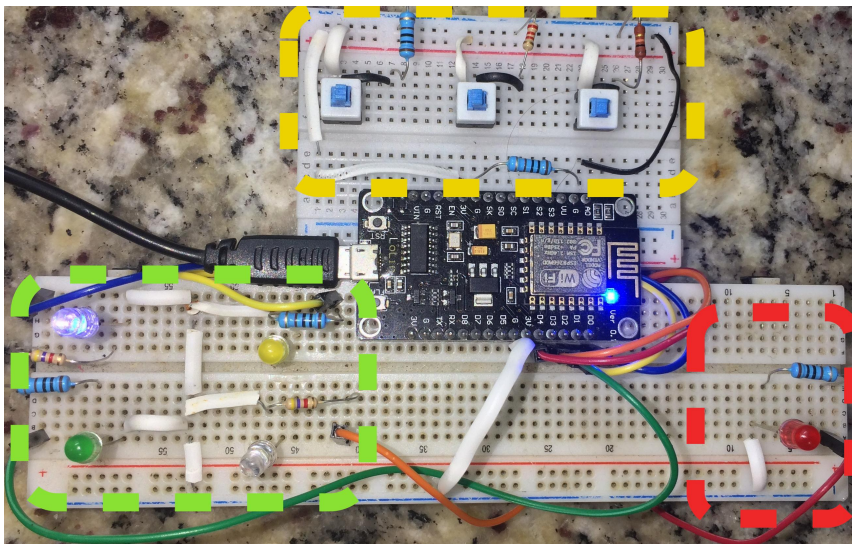
Figura 14 – Esquemático do Circuito Implementado com a Lolin NodeMCU v3



A Figura 15 é uma foto do circuito real implementado. Nesta figura, o campo em amarelo mostra os três *switches*, da esquerda para a direita: `SW1`, `SW2` e `SW3`. O retângulo em verde mostra os quatro *leds* que representam a posição do aspirador (`LED1-4`), enquanto o *led* vermelho, no canto inferior direito, indica a ação de sucção ou esvaziar o depósito

de sujeira (`LED5`). A alimentação do circuito é realizada através da entrada microUSB da placa.

Figura 15 – Circuito Implementado com a Lolin NodeMCU v3



6.1.2 Implementações do aspirador

Tendo sido realizada a montagem do circuito, torna-se possível realizar a implementação das versões propostas do aspirador. Para tal, é utilizado o *Software Development Kit* (SDK) `ESP8266_RTOS_SDK`, mantido pela Espressif, fabricante do microprocessador ESP8266 (ESPRESSIF, 2021), o qual possibilita a utilização de código C e C++ na plataforma Lolin NodeMCU v3.

Cada versão do aspirador é implementada de três formas distintas. A primeira implementação utiliza a linguagem de programação C e visa conceber uma solução "tradicional", ou seja, nesta implementação é utilizada uma linguagem de programação e estratégias de implementação comumente utilizadas em sistemas embarcados. As outras duas implementações são realizadas através do *framework* desenvolvido, sendo que uma implementação explora as características reativas do agente, enquanto a outra busca explorar características de proatividade. Busca-se, através da implementação dos agentes com características reativas e proativas, identificar se há diferença significativa na utilização de recursos computacionais na implementação de cada abordagem, tendo em vista que certas aplicações podem requerer que sistemas embarcados tenham reatividade e proatividade. Consequentemente, o aspirador é implementado nove vezes. A seguir, são apresentados detalhes sobre as implementações de cada versão do aspirador.

6.1.2.1 Versão 1

A implementação do aspirador através da linguagem de programação C ocorre através de uma instrução de *loop*, onde o programa verifica se o espaço onde ele se encontra está sujo e, caso esteja, ele realiza a sucção da sujeira e se move para o espaço seguinte. Caso o espaço esteja limpo, o agente simplesmente se movimenta para o próximo espaço. O código desta implementação está disponível no Código 18 do Apêndice D.

Por sua vez, o agente reativo possui planos baseados nas crenças refentes à sua posição atual. Ou seja, de acordo com a posição do agente, o mesmo irá executar um plano correspondente, seja para limpar o espaço em que se encontra ou para se movimentar para o próximo espaço. O código desta implementação está disponível no Código 19 do Apêndice D.

Finalmente, o agente proativo possui planos orientados a objetivos, onde o mesmo tem como metas realizar a limpeza do espaço em que se encontra e se movimentar para o espaço seguinte. O código desta implementação está disponível no Código 20 do Apêndice D.

6.1.2.2 Versão 2

No código em C, para adição da funcionalidade de "iniciar", é adicionada uma verificação para confirmar se o botão correspondente está pressionado quando o aspirador está na posição 1. Caso esteja, o aspirador inicia seu ciclo de limpeza, já implementado através da instrução de *loop*. O código desta implementação está disponível no Código 21 do Apêndice E.

Para o agente reativo, o contexto dos planos referentes à posição 1 é alterado para verificar se o botão de iniciar está pressionado. Caso esteja, o agente inicia seu ciclo de limpeza. O código desta implementação está disponível no Código 22 do Apêndice E.

Para o agente proativo, o plano referente ao início do ciclo de limpeza é alterado para verificar se o botão de iniciar está pressionado. Caso esteja, o agente inicia seu ciclo de limpeza. O código desta implementação está disponível no Código 23 do Apêndice E.

6.1.2.3 Versão 3

Na terceira versão do aspirador, o código em C é alterado para adicionar a verificação do depósito de sujeira durante o ciclo de limpeza do aspirador. O código desta implementação está disponível no Código 24 do Apêndice F.

No agente reativo, é adicionado um plano para tratar a crença referente ao depósito cheio. O código desta implementação está disponível no Código 25 do Apêndice F.

Finalmente, no agente proativo, a verificação do depósito de sujeira também é realizada através de um novo plano. Neste caso, o contexto do novo plano de limpeza verifica se a crença referente ao depósito cheio é verdadeira. O código desta implementação está disponível no Código 26 do Apêndice F.

6.2 ANÁLISE DE RESULTADOS

A seguir, são apresentadas as métricas utilizadas para avaliar cada versão do aspirador. Para cada programa implementado, foram coletadas informações sobre o tamanho da imagem gerada para programação da placa e tamanho de código escrito para programação. Para facilitar a apresentação dos dados e sua análise, estabelece-se que cada versão programada através da linguagem de programação C é referenciada como "Tradicional", enquanto as versões programadas através do *framework* são denominadas "Reativa" e "Proativa".

6.2.1 Tamanho de imagem gerado

Informações sobre o tamanho das imagens geradas para programação da placa foram obtidas através do SDK `ESP8266_RTOS_SDK`, o qual usa o compilador `xtensa-lx106-elf` (versão 8.4.0) para geração de binários. A Tabela 1 mostra o tamanho total de cada imagem gerada para programação da placa Lolin NodeMCU v3, com nível de otimização definido para tamanho de binário (`-Os`).

Tabela 1 – Tamanho total das imagens geradas

Implementação	**Tamanho total (*bytes*)**		
	Versão 1	Versão 2	Versão 3
Tradicional	122494	122582	123066
Reativa	145490	145850	146518
Proativa	145438	145982	146734

De acordo com a Tabela 1, é possível perceber que o tamanho das imagens geradas para os agentes implementados através do *framework* é maior do que as geradas pelas implementações "tradicionais". Isto é esperado, pois as bibliotecas e estruturas de dados da *engine* BDI contribuem para o tamanho da imagem. Além disso, a diferença de tamanho entre as implementações das versões apresenta pouca variação, conforme a Tabela 2.

Além da *engine* BDI, a utilização da linguagem de programação C++ e do contêiner `std::vector` também colaboram para o tamanho da imagem, como mostrado na Tabela 3, que detalha o tamanho das bibliotecas estáticas utilizadas no processo de compilação

Tabela 2 – Diferença de tamanho entre as imagens geradas

Versão	**Diferença para versão tradicional (em %)**	
	Reativa	Proativa
1	< 18,77	< 18,73
2	< 18,98	< 19,08
3	< 19,05	< 19,23

da imagem final. Na Tabela 3, as colunas representadas pela letra "T" se referem à abordagem tradicional e as colunas "R" e "P" correspondem às abordagens Reativa e Proativa, respectivamente.

Tabela 3 – Tamanho das bibliotecas estáticas

Biblioteca estática	**Tamanho das bibliotecas estáticas (*bytes*)**								
	Versão 1			Versão 2			Versão 3		
	T	R	P	T	R	P	T	R	P
`libphy.a`	22889	22889	22889	22889	22889	22889	22889	22889	22889
`libgcc.a`	14049	14049	14049	14049	14049	14049	14049	14049	14049
`libpp.a`	13670	13670	13670	13670	13670	13670	13670	13670	13670
`libfreertos.a`	13344	13344	13344	13344	13344	13344	13344	13344	13344
`libc_nano.a`	11102	11102	11102	11102	11102	11102	11102	11102	11102
`libnvs_flash.a`	10917	10913	10913	10917	10913	10913	10917	10913	10913
`libesp8266.a`	10496	10496	10496	10496	10496	10496	10599	10599	10599
`libstdc++.a`	9753	23357	23357	9753	23357	23357	9753	23357	23357
`libvfs.a`	5530	5530	5530	5530	5530	5530	5526	5526	5526
`libspi_flash.a`	4068	4068	4068	4068	4068	4068	4068	4068	4068
`libpthread.a`	2108	2108	2108	2108	2108	2108	2108	2108	2108
`libesp_common.a`	1646	1646	1646	1646	1646	1646	1646	1646	1646
`libnewlib.a`	1411	1411	1411	1411	1411	1411	1470	1411	1411
`libmain.a`	999	10202	10150	1087	10559	10691	1411	11126	11342
`libheap.a`	888	888	888	888	888	888	888	888	888
`libesp_ringbuf.a`	590	590	590	590	590	590	590	590	590
`libnet80211.a`	585	585	585	585	585	585	585	585	585
`liblog.a`	519	519	519	519	519	519	519	519	519
`libcore.a`	432	432	432	432	432	432	432	432	432
`librtc.a`	269	269	269	269	269	269	269	269	269
`libapp_update.a`	256	256	256	256	256	256	256	256	256
`libhal.a`	37	37	37	37	37	37	37	37	37

Na Tabela 3, é possível perceber que a biblioteca `libstdc++`, referente à implementação da biblioteca padrão da linguagem de programação C++, possui um tamanho 139,48% maior nas versões programadas com o *framework* do que a mesma biblioteca na versão tradicional do aspirador. Isto mostra que a contribuição das bibliotecas da linguagem de programação C++ nos tamanhos das imagens geradas é significativa na comparação das versões implementadas. Além disso, os valores referentes a biblioteca

`libmain.a`, correspondentes as implementações de códigos das versões do aspirador, mostram que a contribuição dos códigos dos agentes programados através do *framework* são maiores do que as dos aspiradores programados através da abordagem tradicional. Como mencionado anteriormente, isto também é esperado, considerando a contribuição das bibliotecas da *engine* BDI. Entretanto, é possível perceber que, conforme as versões do aspirador ficam mais complexas, a diferença de tamanho da biblioteca estática `libmain.a` em cada versão passa a ser menor entre as implementações tradicionais e de agentes. A Tabela 4 mostra a diferença de tamanho da biblioteca `libmain.a` entre as implementações tradicionais e de agentes de cada versão do aspirador.

Tabela 4 – Diferença de tamanho entre versões da biblioteca estática `libmain.a`

Versão	**Diferença para versão tradicional (em %)**	
	Reativa	Proativa
1	< 921,22	< 916,10
2	< 871,38	< 883,53
3	< 688,51	< 703,82

Estes resultados são conflitantes, pois enquanto a Tabela 2 mostra que a diferença entre os tamanhos das implementações tradicionais e de agentes aumenta conforme novas funcionalidades são adicionadas, a Tabela 4 mostra que a diferença de tamanho entre as bibliotecas estáticas referentes ao código principal de cada implementação diminui. Portanto, não é possível identificar qual a relação exata entre os tamanhos das imagens geradas e os códigos implementados.

6.2.2 Tamanho do código

Finalmente, foi analisado o tamanho dos códigos fonte escritos para programação das aplicações. Como a quantidade de linhas escritas pode variar com o estilo de programação do desenvolvedor, foi utilizada a ferramenta `gzip` para compactar os códigos de cada versão e verificar o seu tamanho em bytes (SANTOS, F. R., 2015). A compactação do código diminui a influência de espaços em branco e número de linhas no tamanho de código considerado (MENEGOL; HÜBNER; BECKER, 2018). É importante ressaltar que comentários foram removidos dos códigos fonte antes de sua compactação e que os resultados analisados não incluem as modificações necessárias para utilizar o SDK, como alterações no arquivo `CMakeLists.txt`, por exemplo. A Tabela 5 exibe o tamanho total dos códigos compactados para cada implementação e Tabela 6 detalha o tamanho dos arquivos de cada implementação.

Tabela 5 – Tamanho dos códigos compactados

Implementação	**Tamanho dos códigos compactados (bytes)**		
	Versão 1	Versão 2	Versão 3
Tradicional	1139	1198	1373
Reativa	1109	1133	1336
Proativa	1108	1166	1368

Os resultados obtidos mostram que a cada versão implementada, o tamanho do código escrito pelo programador em cada abordagem possui tamanhos próximos, sendo que as implementações tradicionais são marginalmente maiores. Apesar dos códigos AgentSpeak possuírem um tamanho menor, os arquivos de funções de atualização de crenças e atuação no ambiente dos agentes possuem um tamanho maior dos que os das implementações tradicionais. Isto ocorre pois as funções de atualização das crenças de posição precisam ser individualizadas para os agentes implementados através do *framework*, enquanto a abordagem tradicional permite que uma única função possa ser utilizada para atualização da posição do aspirador. A Tabela 6 detalha o tamanho dos arquivos compactados. Percebe-se, nesta tabela, que o tamanho do arquivo referente ao código que descreve o "comportamento" dos agentes BDI implementados através do *framework* (`agentspeak.asl.gz`) é inferior ao código da programação tradicional (`agent_loop.c.gz`).

Tabela 6 – Tamanho dos arquivos compactados

Implementação	**Arquivo**	**Tamanho do arquivo (bytes)**		
		Versão 1	Versão 2	Versão 3
Tradicional	`agent_loop.c.gz`	252	295	310
	`functions.c.gz`	623	635	782
	`functions.h.gz`	264	268	281
Reativa	`agent.config.gz`	70	71	70
	`agentspeak.asl.gz`	118	141	159
	`functions.cpp.gz`	626	642	806
	`functions.h.gz`	270	279	301
Proativa	`agent.config.gz`	71	71	71
	`agentspeak.asl.gz`	141	174	190
	`functions.cpp.gz`	626	642	806
	`functions.h.gz`	270	279	301

7 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Nesta dissertação, investigou-se o problema de embarcar agentes BDI em *hardware* limitado, procurando determinar características para uma plataforma que viabilize tal aplicação. Para isso, foi desenvolvido um *framework* para a implementação destes agentes no contexto de sistemas embarcados. Este *framework* foi desenvolvido tendo como base o interpretador Jason e, devido aos desafios de implementação de agentes BDI em sistemas embarcados, foram propostas e avaliadas várias simplificações no ciclo de raciocínio e funcionalidades dos agentes.

A partir dos experimentos realizados, foi possível comprovar que o *framework* permite o desenvolvimento de agentes BDI que podem ser embarcados na plataforma LoLin NodeMCU v3 e analisar a utilização de recursos computacionais dos agentes programados. Foi possível confirmar que a plataforma escolhida possui recursos computacionais suficientes para implementar a aplicação de exemplo e o ciclo de raciocínio BDI. Portanto, seguindo a metodologia de desenvolvimento incremental proposta, é possível realizar a implementação de novas funcionalidades no *framework*, considerando que a plataforma escolhida possui recursos computacionais suficientes para suportar novas funcionalidades.

Constatou-se que o requisito de suporte multiplataforma teve grande impacto nas escolhas de implementação do *framework*. Por exemplo, a heterogeneidade das interfaces de entrada e saída de plataformas comumente utilizadas no desenvolvimento de sistemas embarcados fez com que se optasse pela não implementação de funções de atualização de crenças e ações no ambiente, as quais devem ser fornecidas pelo programador do agente.

Adicionalmente, a funcionalidade de comunicação no ciclo de raciocínio do agente não foi implementada nesta versão do *framework*, pois as interfaces de comunicação de plataformas embarcadas também são heterogêneas e, muitas vezes, necessitam da utilização de APIs e SDKs proprietários para sua utilização. Entretanto, apesar da falta de suporte à funcionalidade de comunicação no ciclo de raciocínio de agentes, é possível realizar a comunicação entre agentes com o envio de mensagens através de ações fornecidas pelo programador. Por sua vez, a recepção das mensagens pode ser realizada tanto através de funções de atualização de crenças quanto por ações que verificam se mensagens foram recebidas de outros agentes.

Outra funcionalidade não implementada foi o suporte a predicados, os quais permitem a utilização de variáveis, valores inteiros, *strings* e outros no código do agente, além de algoritmos de unificação mais complexos. Neste caso, a não implementação desta funcionalidade permitiu o desenvolvimento de um *framework* mais simples, que necessita de menos recursos computacionais para execução.

Também é importante ressaltar que a limitação de tamanho e o tratamento de *overflow* das estruturas de dados do agente podem impactar seu funcionamento. Por exemplo, o descarte de intenções e eventos em decorrência da falta de espaço nas filas correspondentes pode resultar em cenários em que o agente não responde a mudanças no ambiente. Portanto, torna-se necessário que, ao configurar o tamanho das estruturas de dados do agente, o programador conheça o funcionamento do ciclo de raciocínio, buscando minimizar o tamanho das estruturas de dados, enquanto mantendo o funcionamento esperado do agente.

Finalmente, ao analisar os resultados obtidos no Capítulo 6, observa-se que a utilização de recursos computacionais dos agentes programados através do *framework* é superior aos de uma implementação "tradicional". Isto é esperado, levando em consideração as estruturas de dados utilizadas no ciclo de raciocínio BDI e a *engine* de execução BDI. Ademais, os tamanhos dos códigos implementados para a programação dos agentes e das aplicações tradicionais foram semelhantes em cada versão do *framework*, mostrando que, apesar de uma maior utilização de recursos computacionais, a utilização do *framework* não resulta em códigos mais longos. Entretanto, uma vez aprendida a linguagem AgentSpeak, subjetivamente, podemos considerar que o programa na linguagem de agentes é mais simples de ler e dar manutenção. Esse aspecto não foi quantitativamente avaliado no trabalho e é alvo de trabalhos futuros.

7.1 TRABALHOS FUTUROS

Por fim, seguem sugestões para trabalhos futuros:

- Do ponto de vista da implementação do *framework*, observou-se que a contribuição da biblioteca estática `libstdc++`, referente à implementação da biblioteca padrão da linguagem de programação C++, possui uma grande contribuição no tamanho do programa final. Portanto, sugere-se que sejam realizadas otimizações na implementação da versão atual do *framework*, sejam estas relacionadas as estruturas de dados utilizadas, ou à implementação do *framework* na linguagem de programação C.

- No que se refere à usabilidade do *framework*, sugere-se que seja realizada a implementação de agentes em aplicações mais complexas, com o intuito de explorar cenários de aplicabilidade dos agentes implementados e a utilização de recursos computacionais. *E*spera-se que as simplificações realizadas na implementação do *framework* viabilizem sua utilização em plataformas com menos recursos computacionais. Portanto,

é desejável que sejam exploradas também aplicações em plataformas mais simples, possibilitando uma análise quantitativa dos recursos computacionais mínimos para execução dos agentes.

- Do ponto de vista de desempenho, sugere-se que seja realizada uma análise quantitativa dos recursos utilizados pelos agentes implementados em termo de execução, bem como sua responsividade à percepção de mudanças no ambiente.

- Finalmente, sugere-se que sejam adicionadas outras funcionalidades ao *framework*, como suporte a comunicação e predicados. Para tal, pode ser que seja necessário abandonar o caráter multiplataforma do *framework*, buscando concentrar o desenvolvimento da(s) funcionalidade(s) para uma plataforma específica.

REFERÊNCIAS

ALIYUDA, Ali. Towards the Design of Cyber-Physical System via Multi-Agent System Technology. v. 7, out. 2016.

ANTHONY, Patricia *et al.* Agent Architecture: An Overview. **TRANSACTIONS ON SCIENCE AND TECHNOLOGY**, p. 18–35, jan. 2014.

ARDUINO STORE. Arduino Store: Arduino Uno Rev3, out. 2021. Disponível em: https://store.arduino.cc/products/arduino-uno-rev3.

AZWAR, Ade Geovania *et al.* Smart Trash Monitoring System Design Using NodeMCU-based IoT. *In*: 2019 IEEE 13th International Conference on Telecommunication Systems, Services, and Applications (TSSA). [*S.l.*: *s.n.*], 2019. P. 67–71. DOI: 10.1109/TSSA48701.2019.8985517.

BARROS, Reydson *et al.* An Agent-oriented Ground Vehicle's Automation using Jason Framework. *In*: p. 261–266. DOI: 10.5220/0004917102610266.

BERGENTI, Federico; GLEIZES, Marie-Pierre; ZAMBONELLI, Franco. **Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook**. [*S.l.*: *s.n.*], jan. 2004. P. 506. DOI: 10.1007/b116049.

BERGER, Arnold S. **Embedded Systems Design: An Introduction to Processes, Tools, and Techniques**. [*S.l.*]: Taylor & Francis, 2002. (CMP Books). ISBN 9781578200733. Disponível em: https://books.google.com.br/books?id=3vY35UkvXrAC.

BOKADE, Ashish U.; RATNAPARKHE, V. R. Video surveillance robot control using smartphone and Raspberry pi. *In*: 2016 International Conference on Communication and Signal Processing (ICCSP). [*S.l.*: *s.n.*], 2016. P. 2094–2097. DOI: 10.1109/ICCSP.2016.7754547.

BORDINI, Rafael H.; DASTANI, Mehdi *et al.* **Multi-Agent Programming: Languages, Tools and Applications**. 1st. [*S.l.*]: Springer Publishing Company, Incorporated, 2009. ISBN 0387892982.

BORDINI, Rafael H.; HÜBNER, Jomi F. Jason: A Java-based interpreter for anextended version of AgentSpeak. *In*. Disponível em: http://jason.sourceforge.net/Jason.pdf. Acesso em: 2 mai. 2020.

BORDINI, Rafael H.; HÜBNER, Jomi F.; VIEIRA, Renata. Jason and the Golden Fleece of Agent-Oriented Programming. *In*: **Multi-Agent Programming: Languages, Platforms and Applications**. Edição: Rafael H. Bordini. Boston, MA: Springer US, 2005. P. 3–37. ISBN 978-0-387-26350-2. DOI: 10.1007/0-387-26350-0_1. Disponível em: https://doi.org/10.1007/0-387-26350-0_1.

BORDINI, Rafael H.; HÜBNER, Jomi F.; WOOLDRIDGE, Michael. **Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)**. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2007. ISBN 0470029005.

BORDINI, Rafael H.; VIEIRA, Renata. Linguagens de Programação Orientadas a Agentes: Uma Introdução Baseada em AgentSpeak(L). **RITA**, v. 10, p. 7–38, jan. 2003.

BRATMAN, M. **Intention, plans, and practical reason**. Cambridge, MA: Harvard University Press, 1987. ISBN 9780674458185. Disponível em: http://books.google.de/books?id=I0nuAAAAMAAJ.

BUCHELI, Samuel *et al.* From AgentSpeak to C for Safety Considerations in Unmanned Aerial Vehicles, p. 69–81, set. 2015. DOI: 10.1007/978-3-319-22416-9_9.

CALCE, A. *et al.* Autonomous Aquatic Agents. *In*: INSTICC. PROCEEDINGS of the 5th International Conference on Agents and Artificial Intelligence - Volume 1: ICAART, [*S.l.*]: SciTePress, 2013. P. 372–375. ISBN 978-989-8565-38-9. DOI: 10.5220/0004220003720375.

CUNHA, Alessandro F. O que são sistemas embarcados? **Saber Eletrônica**, v. 43, p. 1–6, jul. 2007.

ESPRESSIF. **ESP8266 RTOS Software Development Kit**. [*S.l.*: *s.n.*], 2021. Disponível em: https://github.com/espressif/ESP8266_RTOS_SDK/tree/0f200b46406642ffa590673e244bad06fc736016. Acesso em: 11 nov. 2021.

FIPA. **FIPA Agent Management Specification**. Geneva, Switzerland, mar. 2004. Disponível em: http://www.fipa.org/specs/fipa00023/SC00023K.pdf.

FITTING, Melvin. **First-Order Logic and Automated Theorem Proving**. 2. ed. [*S.l.*]: Springer, 1996.

FÜRST, S.; BECHTER, M. AUTOSAR for Connected and Autonomous Vehicles: The AUTOSAR Adaptive Platform. *In*: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W). [*S.l.*: *s.n.*], 2016. P. 215–217.

GNU. **GNU Make**. [*S.l.*: *s.n.*], 19 jan. 2020. Disponível em: https://www.gnu.org/software/make/. Acesso em: 20 nov. 2021.

GRAPHVIZ. **Graphviz**. [*S.l.*: *s.n.*], ago. 2021. Disponível em: https://graphviz.org/. Acesso em: 15 nov. 2021.

HEESCH, Dimitri van. **Doxygen**. [*S.l.*: *s.n.*], out. 2021. Disponível em: https://www.doxygen.nl/manual/index.html. Acesso em: 15 nov. 2021.

HÜBNER, Jomi Fred. **Getting Started with Jason**. [*S.l.*: *s.n.*], ago. 2021. Disponível em: https://github.com/jason-lang/jason/blob/e2aa0f20e29039c0605e7ca795e3ea7e5e58492c/doc/tutorials/getting-started/readme.adoc. Acesso em: 7 nov. 2021.

JUNIOR, Márcio Fernando Stabile. Melhorando o desempenho de agentes BDI Jason através de filtros de percepção, 2015. DOI: 10.11606/D.45.2016.tde-05022016-160602. Disponível em: https://www.teses.usp.br/teses/disponiveis/45/45134/tde-05022016-160602/pt-br.php. Acesso em: 24 abr. 2021.

LESKINEN, Arseni. **Applicability of Kubernetes to Industrial IoT Edge Computing System**. 2020. F. 55+4. Master's thesis – Aalto University. School of Electrical Engineering. Disponível em: http://urn.fi/URN:NBN:fi:aalto-202010256055. Acesso em: 1 nov. 2021.

MANOEL, Fabian *et al.* A Heterogeneous Architecture for Integrating Multi-Agent Systems in AmI Systems. *In*. DOI: 10.18293/SEKE2018-211.

MARWEDEL, Peter. **Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems**. 2nd. [*S.l.*]: Springer Publishing Company, Incorporated, 2010. ISBN 9789400702561.

MASCARDI, Viviana; DEMERGASSO, Daniela; ANCONA, Davide. Languages for Programming BDI-style Agents: an Overview., p. 9–15, jan. 2005.

MEHALAINE, Ridha *et al.* Intelligent Embedded Software: New Perspectives and Challenges. *In*: [*S.l.*: *s.n.*], ago. 2018. ISBN 978-1-78923-606-4. DOI: 10.5772/intechopen.72417.

MENEGOL, Marcelo S; HÜBNER, Jomi F; BECKER, Leandro B. Evaluation of multi-agent coordination on embedded systems. *In*: SPRINGER. INTERNATIONAL Conference on Practical Applications of Agents and Multi-Agent Systems. [*S.l.*: *s.n.*], 2018. P. 212–223.

MILLER, Jason; ESFANDIARI, Babak. Analyzing the Execution Time of the Jason BDI Reasoning Cycle. 9th International Workshop on Engineering Multi-Agent Systems, mai. 2021. Disponível em: `https://emas2021.in.tu-clausthal.de/index.php/schedule`.

O'HARE, Gregory *et al.* Embedding Agents within Ambient Intelligent Applications. *In*: [*S.l.*: *s.n.*], jan. 2012.

PANTOJA, Carlos; LAZARIN, Nilson Mori. A Robotic-agent Platform for Embedding Software Agents Using Raspberry Pi and Arduino Boards. *In*.

PARK, Kyung-Joon; ZHENG, Rong; LIU, Xue. Cyber-physical systems: Milestones and research challenges. **Comput. Commun.**, v. 36, p. 1–7, 2012.

PEREIRA, D. *et al.* Towards an Architecture for Emotional BDI Agents. *In*: 2005 portuguese conference on artificial intelligence. [*S.l.*: *s.n.*], dez. 2005. P. 40–46. DOI: `10.1109/EPIA.2005.341262`.

RAJKUMAR, R. *et al.* Cyber-physical systems: The next computing revolution. *In*: DESIGN Automation Conference. [*S.l.*: *s.n.*], jun. 2010. P. 731–736. DOI: `10.1145/1837274.1837461`.

RAO, Anand S. AgentSpeak(L): BDI agents speak out in a logical computable language. *In*: VAN DE VELDE, Walter; PERRAM, John W. (Ed.). **Agents Breaking Away**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996. P. 42–55.

RASPBERRY PI FOUNDATION. What is a Raspberry Pi? Disponível em: `%7Bhttps://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/%7D`. Acesso em: 1 nov. 2021.

RASPBERRY PI TRADING LTD. Raspberry Pi 4 Computer Model B: Product Brief, jan. 2021. Disponível em: `https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf`.

REYNOLDS, John C. Definitional interpreters for higher-order programming languages. *In*: REPRINTED from the proceedings of the 25th ACM National Conference. [*S.l.*]: ACM, 1972. P. 717–740.

SANTOS, Fernando Rodrigues. Avaliação do uso de agentes no desenvolvimento de aplicações com veículos aéreos não-tripulados, 2015. Disponível em: `https://repositorio.ufsca.br/xmlui/handle/123456789/158820`. Acesso em: 11 abr. 2020.

SANTOS, Matuzalem Muller dos. **Embedded BDI Documentation**. [*S.l.*: *s.n.*], jun. 2021. Disponível em: `https://embedded-bdi.github.io/embedded-bdi/index.html`. Acesso em: 15 nov. 2021.

SANTOS, Matuzalem Muller dos. **Embedded BDI: Agent**. [*S.l.*: *s.n.*], jun. 2021. Disponível em: `https://embedded-bdi.github.io/embedded-bdi/d9/d3a/class_agent.html`. Acesso em: 15 nov. 2021.

SANTOS, Matuzalem Muller dos. **Embedded-BDI: Makefile**. [*S.l.*: *s.n.*], 25 set. 2020. Disponível em: `https://github.com/Embedded-BDI/embedded-bdi/blob/6373f3a7316e1dac9116e6bcfff78e16d87df05e/Makefile`. Acesso em: 20 nov. 2021.

SANTOS, Matuzalem Muller dos. **Embedded-BDI: workflows**. [*S.l.*: *s.n.*], 7 jun. 2021. Disponível em: `https://github.com/Embedded-BDI/embedded-bdi/tree/6373f3a7316e1dac9116e6bcfff78e16d87df05e/.github/workflows`. Acesso em: 20 nov. 2021.

STOJMENOVIC, I. Machine-to-Machine Communications With In-Network Data Aggregation, Processing, and Actuation for Large-Scale Cyber-Physical Systems. **IEEE Internet of Things Journal**, v. 1, n. 2, p. 122–128, abr. 2014. ISSN 2372-2541. DOI: `10.1109/JIOT.2014.2311693`.

AI-THINKER. **ESP-12E WiFi Module**. 1.0. ed. [*S.l.*], 2015. Disponível em: `https://docs.ai-thinker.com/_media/esp8266/docs/esp12e_datasheet.pdf`. Acesso em: 20 nov. 2021.

VALGRIND DEVELOPERS. **Valgrind**. [*S.l.*: *s.n.*]. Disponível em: `https://valgrind.org/`. Acesso em: 20 nov. 2021.

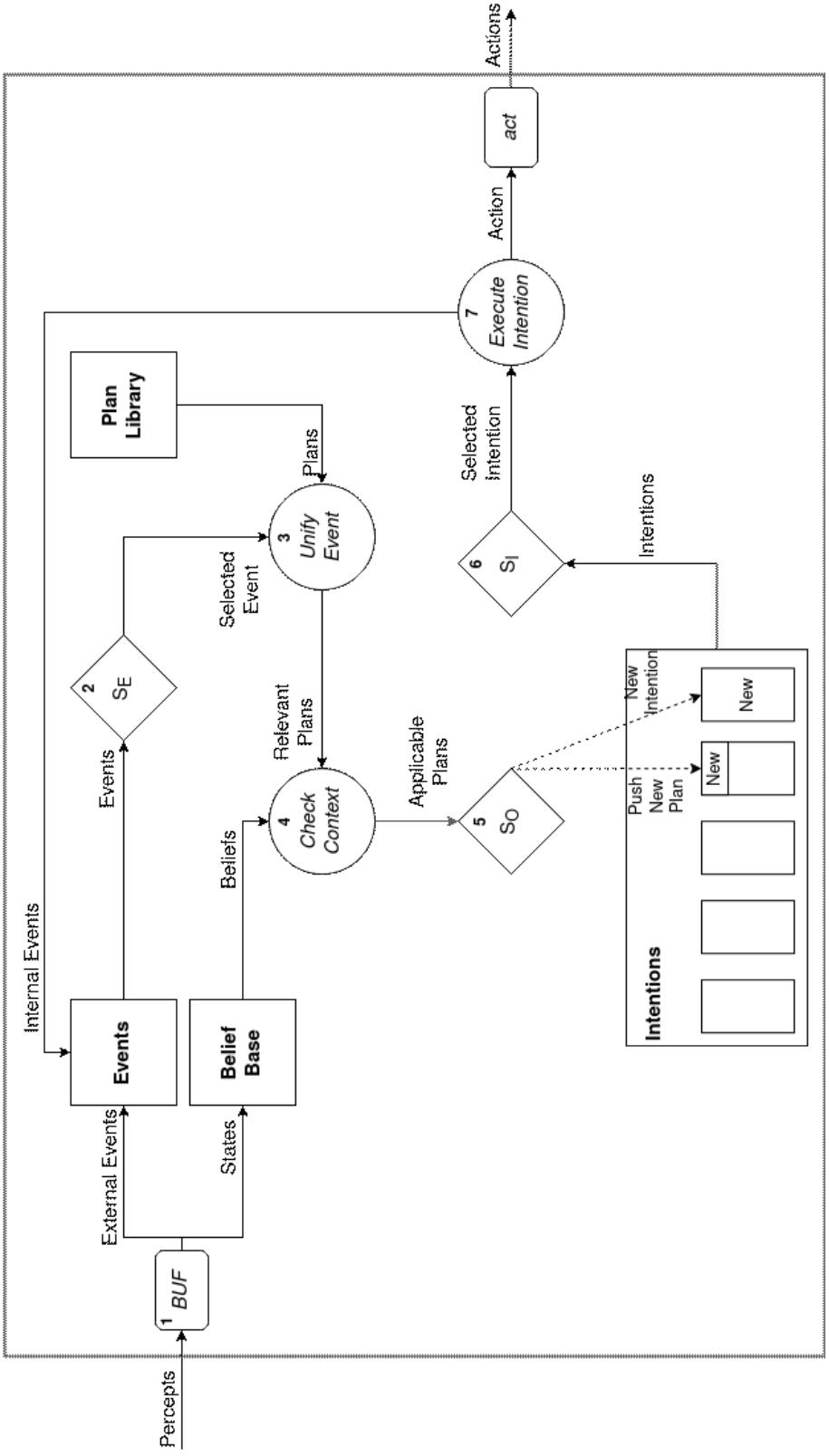
WOOLDRIDGE, Michael. **An Introduction to MultiAgent Systems**. 2nd. [*S.l.*]: Wiley Publishing, 2009. ISBN 0470519460.

WOOLDRIDGE, Michael; JENNINGS, Nicholas R. Agent Theories, Architectures, and Languages: A Survey. *In*: PROCEEDINGS of the Workshop on Agent Theories, Architectures, and Languages on Intelligent Agents. Amsterdam, The Netherlands: Springer-Verlag, 1995. (ECAI-94), p. 1–39.

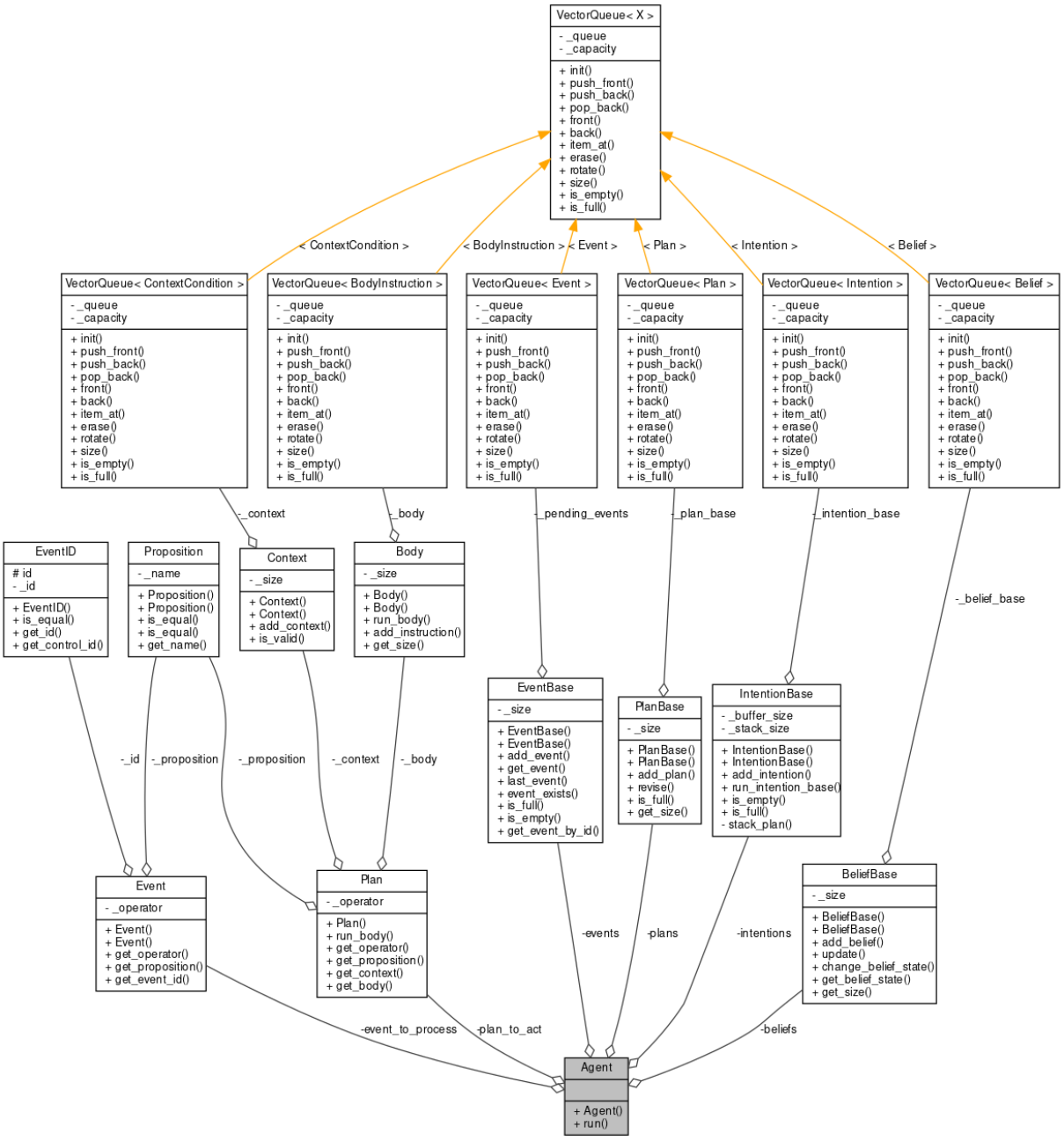
WOOLDRIDGE, Michael; JENNINGS, Nicholas R. Intelligent agents: theory and practice. **The Knowledge Engineering Review**, Cambridge University Press, v. 10, n. 2, p. 115–152, 1995. DOI: `10.1017/S0269888900008122`.

Apêndices

APÊNDICE A – CICLO DE RACIOCÍNIO PRÁTICO DO *FRAMEWORK*



APÊNDICE B – DIAGRAMA DE CLASSE COMPLETO DA CLASSE AGENT



APÊNDICE C – CÓDIGO C++ GERADO A PARTIR DA TRADUÇÃO DE CÓDIGO AGENTSPEAK

Código 17 – Código C++ completo

```
/*
 * AgentSpeak code:
 *
 * happy.
 *
 * +!hello : happy <- say_hello.
 */

#ifndef CONFIGURATION_H_
#define CONFIGURATION_H_

#include "bdi/belief_base.h"
#include "bdi/event_base.h"
#include "bdi/plan_base.h"
#include "bdi/intention_base.h"
#include "../../data/functions.h"

class AgentSettings
{
private:
  Body body_0;
  Context context_0;
  BeliefBase belief_base;
  EventBase event_base;
  PlanBase plan_base;
  IntentionBase intention_base;

public:
  AgentSettings()
  {
    belief_base = BeliefBase(1);
    event_base = EventBase(6);
    plan_base = PlanBase(1);
    intention_base = IntentionBase(10, 4);


   //------------------------------------------------------------------------



    Belief belief_happy(0, update_happy, true);
    belief_base.add_belief(belief_happy);

```

```
41
    //------------------------------------------------------------------------


42
     Proposition prop_0(1);
43
     context_0 = Context(1);
44
     body_0 = Body(1);
45
46
     Proposition prop_0_happy(0);
47
     ContextCondition cond_0_0(prop_0_happy);
48
     context_0.add_context(cond_0_0);
49
50
     Proposition prop_0_body_0(2);
51
     BodyInstruction inst_0_0(BodyType::ACTION, prop_0_body_0,
52
    action_say_hello);
     body_0.add_instruction(inst_0_0);
53
54
     Plan plan_0(EventOperator::GOAL_ADDITION, prop_0, &context_0, &
55
    body_0);
     plan_base.add_plan(plan_0);
56
   }
57
58
   BeliefBase * get_belief_base()
59
   {
60
     return &belief_base;
61
   }
62
63
   EventBase * get_event_base()
64
   {
65
     return &event_base;
66
   }
67
68
   PlanBase * get_plan_base()
69
   {
70
     return &plan_base;
71
   }
72
73
   IntentionBase * get_intention_base()
74
   {
75
     return &intention_base;
76
   }
77
};
78
79
#endif /*CONFIGURATION_H_ */
80
```

APÊNDICE D – CÓDIGOS DA VERSÃO 1 DO ASPIRADOR

Código 18 – Implementação na linguagem de programação C da versão 1 do aspirador

```
/*
 *  Created on: Feb 28, 2021
 *      Author: Matuzalem Muller
 */

#include <stdint.h>
#include "../data/functions.h"

uint8_t pos;

void app_main()
{
  setup();

  while (1)
  {
    if (update_dirty())
    {
      action_suck();
    }
    else
    {
      switch (pos)
      {
        case 1:
          action_right();
          break;
        case 2:
          action_down();
          break;
        case 3:
          action_up();
          break;
        case 4:
          action_left();
          break;
        default:
          break;
      }
      update_pos(pos);
    }
  }
}
```

Código 19 – Implementação do agente reativo da versão 1 do aspirador

```
+pos_1 : dirty <- suck; right.
+pos_2 : dirty <- suck; down.
+pos_3 : dirty <- suck; up.
+pos_4 : dirty <- suck; left.

+pos_1 : clean <- right.
+pos_2 : clean <- down.
+pos_3 : clean <- up.
+pos_4 : clean <- left.
```

Código 20 – Implementação do agente proativo da versão 1 do aspirador

```
!init.

+!init <- !!clean.

+!clean : clean <- !move; !!clean.
+!clean : dirty <- suck;  !!clean.
-!clean         <- !!clean.

+!move : pos_1 <- right.
+!move : pos_2 <- down.
+!move : pos_3 <- up.
+!move : pos_4 <- left.
```

APÊNDICE E – CÓDIGOS DA VERSÃO 2 DO ASPIRADOR

Código 21 – Implementação na linguagem de programação C da versão 2 do aspirador

```
/*
 * agent_loop.c
 *
 *  Created on: Feb 28, 2021
 *      Author: Matuzalem Muller
 */

#include <stdint.h>
#include "../data/functions.h"

uint8_t pos;
uint8_t stop;

void app_main()
{
  setup();

  while (1)
  {
    if (update_start())
    {
      stop = 0;
      while (!stop)
      {
        if (update_dirty())
        {
          action_suck();
        }
        else
        {
          switch (pos)
          {
            case 1:
              action_right();
              break;
            case 2:
              action_down();
              break;
            case 3:
              action_up();
              stop = 1;
              break;
            case 4:
```

```
44                action_left();
45                break;
46              default:
47                break;
48            }
49            update_pos(pos);
50          }
51        }
52      }
53    }
54  }
```

Código 22 – Implementação do agente reativo da versão 2 do aspirador

```
1  +button_start <- +pos_1.
2
3  +pos_1 : dirty & button_start <- suck; right.
4  +pos_2 : dirty <- suck; down.
5  +pos_3 : dirty <- suck; up.
6  +pos_4 : dirty <- suck; left.
7
8  +pos_1 : clean & button_start <- right.
9  +pos_2 : clean <- down.
10 +pos_3 : clean <- up.
11 +pos_4 : clean <- left.
```

Código 23 – Implementação do agente proativo da versão 2 do aspirador

```
1  !init.
2
3  +!init : button_start <- !!clean.
4  +!init <- !!init.
5
6  +!clean : stop  <- -stop; !!init.
7  +!clean : clean <- !move; !!clean.
8  +!clean : dirty <- suck;  !!clean.
9  -!clean         <- !!clean.
10
11 +!move : pos_1 <- right.
12 +!move : pos_2 <- down.
13 +!move : pos_3 <- +stop; up.
14 +!move : pos_4 <- left.
```

APÊNDICE F – CÓDIGOS DA VERSÃO 3 DO ASPIRADOR

Código 24 – Implementação na linguagem de programação C da versão 3 do aspirador

```
/*
 * agent_loop.c
 *
 *  Created on: Feb 28, 2021
 *      Author: Matuzalem Muller
 */

#include <stdint.h>
#include "../data/functions.h"

uint8_t pos;
uint8_t stop;

void app_main()
{
  setup();

  while (1)
  {
    if (update_start())
    {
      stop = 0;
      while (!stop)
      {
        if (update_full_deposit())
        {
          action_empty();
        }
        if (update_dirty())
        {
          action_suck();
        }
        else
        {
          switch (pos)
          {
            case 1:
              action_right();
              break;
            case 2:
              action_down();
              break;
            case 3:
```

```
44                action_up();
45                stop = 1;
46                break;
47              case 4:
48                action_left();
49                break;
50              default:
51                break;
52            }
53            update_pos(pos);
54          }
55        }
56      }
57    }
58  }
```

Código 25 – Implementação do agente reativo da versão 3 do aspirador

```
1  +button_start <- +pos_1.
2
3  +pos_1 : dirty & button_start <- suck; right.
4  +pos_2 : dirty <- suck; down.
5  +pos_3 : dirty <- suck; up.
6  +pos_4 : dirty <- suck; left.
7
8  +pos_1 : clean & button_start <- right.
9  +pos_2 : clean <- down.
10 +pos_3 : clean <- up.
11 +pos_4 : clean <- left.
12
13 +full_deposit <- empty.
```

Código 26 – Implementação do agente proativo da versão 3 do aspirador

```
1  !init.
2
3  +!init : button_start <- !!clean.
4  +!init <- !!init.
5
6  +!clean : full_deposit <- empty; !!clean.
7  +!clean : stop  <- -stop; !!init.
8  +!clean : clean <- !move; !!clean.
9  +!clean : dirty <- suck;  !!clean.
10 -!clean          <- !!clean.
11
12 +!move : pos_1 <- right.
13 +!move : pos_2 <- down.
14 +!move : pos_3 <- +stop; up.
```

```
15 +!move : pos_4 <- left.
```

Anexos

ANEXO A – MODELO DE RACIOCÍNIO PRÁTICO DO JASON