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

LUCAS EMANUEL RAMOS FERNANDES

IMPLEMENTAÇÃO E VERIFICAÇÃO FORMAL DE PLANOS DE UM AGENTE RACIONAL MODELADO PARA CONDUÇÃO DE UM VEÍCULO AUTÔNOMO

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA
2017

LUCAS EMANUEL RAMOS FERNANDES

IMPLEMENTAÇÃO E VERIFICAÇÃO FORMAL DE PLANOS DE UM AGENTE RACIONAL MODELADO PARA CONDUÇÃO DE UM VEÍCULO AUTÔNOMO

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do título de Bacharel em Ciência da Computação, do Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Gleifer Vaz Alves

PONTA GROSSA
2017

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Ponta Grossa

Diretoria de Graduação e Educação Profissional
Departamento Acadêmico de Informática
Bacharelado em Ciência da Computação

TERMO DE APROVAÇÃO

IMPLEMENTAÇÃO E VERIFICAÇÃO FORMAL DE PLANOS DE UM AGENTE RACIONAL MODELADO PARA CONDUÇÃO DE UM VEÍCULO AUTÔNOMO

Por

LUCAS EMANUEL RAMOS FERNANDES

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 9 de junho de 2017 como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

____
Prof. Dr. Gleifer Vaz Alves
Orientador

____
Prof. Dr. André Pinz Borges
Membro titular

____
Prof. Dr. Augusto Foronda
Membro titular

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

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

- A Folha de Aprovação assinada encontra-se arquivada na Secretaria Acadêmica -

AGRADECIMENTOS

À minha mãe, Teresa Ramos, e ao meu pai, Nelio Fernandes, por apoiarem ao longo de minha jornada, sem eles essa conquista não seria possível.

Ao meu noivo, Michael Koontz, que mesmo longe, foi meu porto seguro, dando apoio moral e incentivos, além de me fazer companhia durante as diversas noites que passei em claro para concluir este trabalho.

Ao meu orientador, Professor Dr. Gleifer Vaz Alves, que me guiou durante esta etapa, pacientemente corrigiu o trabalho e fez o possível para me ajudar.

Ao meu irmão, Erik Fernandes, que se preocupou comigo e me ajudou quando eu mais precisava.

Ao meu amigo, Vinicius Custodio, pelas discussões produtivas e nosso apoio mútuo no decorrer de nossos trabalhos.

À todos que estiveram na minha vida nesses últimos meses e que direta ou indiretamente ajudaram no desenvolvimento deste trabalho.

"If you wish to make an apple pie from scratch,
you must first invent the universe."
Carl Sagan

RESUMO

Fernandes, Lucas Emanuel Ramos. **Implementação e Verificação Formal de Planos de um Agente Racional Modelado para Condução de um Veículo Autônomo.** 2017. 200 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) — Universidade Tecnológica Federal Do Paraná, 2017.

Em um futuro não tão distante, veículos autônomos serão uma realidade no tráfego urbano. Atualmente, pesquisas estão sendo realizadas para desenvolver essa nova tecnologia. Isto visa a criação de um sistema capaz de tomar suas próprias decisões sem intervenção humana no controle de um veículo. Neste trabalho é abordado o uso de um agente racional para modelar o comportamento de um veículo autônomo, considerando um mecanismo de condução básica. No cenário aqui considerado, este agente é capaz de atender corridas, transitar em um ambiente simulado, realizar desvio de obstáculos estáticos e conhecidos, e em caso de uma colisão inevitável, o agente tenta minimizar o dano (físico) causado ao veículo quando possível. A implementação do agente capaz de realizar autonomamente tais funções é feita por meio da linguagem Gwendolen. O ambiente onde o agente está inserido, desenvolvido em Java, é responsável por armazenar informações sobre todas as posições nas quais o veículo possa vir a se movimentar. Com o objetivo de obter uma representação gráfica das modificações ocorridas no ambiente, foi criado um simulador independente capaz de se comunicar com o ambiente Java pela rede. Como todo software é susceptível a erros, faz se necessário verificar as decisões tomadas pelo agente. Isto é evidenciado neste cenário, pois falhas no sistema comando um veículo autônomo pode causar fatalidades. Essa validação pode ser feita por meio de técnicas de verificação formal; onde é utilizado fórmulas lógica de linguagem temporal para especificar formalmente as propriedades que são esperadas do agente e, a técnica do *model checking program* para verifica-las. Com o auxílio da ferramenta AJPF (*Agent* Java *PathFinder*), é possível realizar a verificação durante o tempo de execução do agente, assim podendo se garantir a veracidade de uma propriedade durante seu funcionamento e, em casos onde há inconsistência em uma propriedade, dados relevantes para a investigação são fornecidos. Desta forma, aqui é proposto uma implementação de um agente racional e a verificação formal da tomada de decisão, onde tal agente representa um veículo autônomo operando em diferentes cenários simulados. Por meio deste trabalho, foi concluído que é possível realizar o desenvolvimento de um agente na linguagem Gwendolen modelado como um veículo autônomo. Além disso, as propriedades de tal agente podem ser verificadas formalmente através do *model checker*, utilizando a ferramenta AJPF.

Palavras-chaves: Veículos Autônomos. Agentes Racionais. Verificação formal.

ABSTRACT

Fernandes, Lucas Emanuel Ramos. **Implementation and Formal Verification of Plans of a Rational Agent Modeled to Control an Autonomous Vehicle**. 2017. 200 f. Work of Conclusion Course (Graduation in Computer Science) — Federal University of Technology - Paraná, 2017.

Autonomous vehicles, in the not too distant future, will be a reality in urban traffic. Currently, various research is being conducted to develop this new technology. The main goal of said research is to create and implement vehicle systems capable, and without human interaction, of making their own decisions. In this work, we will discuss the use of a rational agent to model the behavior of an autonomous vehicle, where a basic driving mechanism is considered. The scenario presented here is to define the following agent abilities: perform rides, move through a simulated environment, obstacle avoidance of static and known barriers, and given a situation when a collision is unavoidable; the agent will try to assure the least amount of (physical) damage to the vehicle possible. The approach taken to create this agent is with the Gwendolen programming language. Once formed, it will be implemented in a Java developed environment. The Java environment is then responsible for storing all information about locations were the agent may move during its performance. Once complete, the use of a UPD protocol will be implemented to receive messages through a local network from the Java environment. This is to create a graphical representation of when changes occur caused by the agent in the said environment. Since all software is prone to errors, it is necessary to verify the decisions taken by the agent. Evidence of such a need is easily understood through the basic knowledge of human impact, specifically human casualties. If a decisional flaw occurs in the system while controlling an autonomous vehicle it could directly impact human life. Such validation, shall be shown, can be performed with formal verification techniques. For this, linear temporal logic formulae is used to formally specify all the properties expected for the agent to hold. The Model Checking Program Technique with AJPF (Agent Java Pathfinder) then makes the verification possible. The latter is a tool which is used to verify agent properties during its run-time. Its purpose is to guarantee that a property hold and show true data to an investigation of an occurring error. Thus, this work proposes an implementation of a rational agent and the formal verification of its decision-making process, where such agent represent an autonomous vehicle in different simulated scenarios. With this work, it is was concluded that it is possible to develop an agent in the Gwendolen language modeled as an autonomous vehicle. In addition to this, the properties of such agent can be formally verified through the model checker using the AJPF tool.

Key-words: Autonomous Vehicles. Rational Agents. Formal Verification.

LISTA DE ILUSTRAÇÕES

Figura 1 – Sensores do veículo da Tesla ... 21
Figura 2 – *Chrysler Pacifica Hybrid* ... 22
Figura 3 – Waymo ... 23
Figura 4 – Representação do Dilema do Bonde ... 24
Figura 5 – Representação de um agente interagindo com o ambiente ... 27
Figura 6 – Agente e sua interação com o ambiente ... 27
Figura 7 – Arquitetura de agentes BDI ... 33
Figura 8 – Algoritmo básico para um agente racional ... 35
Figura 9 – Algoritmo básico para um agente BDI ... 36
Figura 10 – Algoritmo para um agente BDI ... 38
Figura 11 – Ciclo de Raciocínio de um agente em Gwendolen ... 39
Figura 12 – Esboço da verificação de um sistema ... 54
Figura 13 – Exemplo de um autômato de Büchi ... 56
Figura 14 – Esboço do *model checking* ... 57
Figura 15 – Autômato do sistema `Program`, $A_{program}$ ... 60
Figura 16 – Autômato da especificação $\Box(x \neq 1)$, A_{esp} ... 60
Figura 17 – Autômato produto de $A_{program}$ e A_{esp}, A_{PE} ... 61
Figura 18 – *Model Checking*: LTL e Autômato de Büchi ... 62
Figura 19 – Arquitetura do AJPF ... 64
Figura 20 – Matriz de Posições do Ambiente ... 69
Figura 21 – Representação da Movimentação e Posicionamento do Agente no Ambiente 70
Figura 22 – Diagrama de Componentes do Sistema ... 72
Figura 23 – Fluxograma do Conjunto de Planos para Atendimento de Passageiros ... 75
Figura 24 – Fluxograma do Plano para Completar um Trajeto ... 79
Figura 25 – Fluxograma do Conjunto de Planos para Limpar Informações sobre Trajetos Anteriores ... 80
Figura 26 – Fluxograma de Conjunto Planos para Deslocamento até o Destino do Trajeto 82
Figura 27 – Fluxograma do Plano para Localização das Direções do Destino do Trajeto 86
Figura 28 – Fluxograma do Plano para a Movimentação em uma Direção ... 87
Figura 29 – Fluxograma do Conjunto de Planos para o Desvio de Obstáculos ... 89
Figura 30 – Fluxograma do Plano para Adaptar de um Trajeto ... 95
Figura 31 – Fluxograma do Plano de Coordenadas Inválidas ... 95
Figura 32 – Fluxograma de Conjuntos de Plano de Escolha do Menor Dano Possível em Caso de Colisões ... 97
Figura 33 – Representação da Escolha por Danos Mínimos ... 99
Figura 34 – Fluxograma do Conjunto de Planos de Controle e Recuperação em Casos de Colisões ... 100
Figura 35 – Fragmento do Diagrama de Componentes do Sistema: Ambiente e suas relações ... 102
Figura 36 – Diagrama de Sequência do método `updateLocation()` ... 106
Figura 37 – Diagrama de Sequência do método `localize()` ... 108
Figura 38 – Diagrama de Sequência do método `drive()` ... 109
Figura 39 – Diagrama de Sequência do método `getRide()` ... 110
Figura 40 – Diagrama de Sequência do método `compass()` ... 111
Figura 41 – Diagrama de Sequência do método `park()` ... 112
Figura 42 – Diagrama de Sequência do método `refuseRide()` ... 113

Figura 43 – Diagrama de Sequência do método `noFurtherFrom()` 114
Figura 44 – Diagrama de Sequência do método `callEmergency()` 114
Figura 45 – Diagrama de Sequência do método `getAssistance()` 115
Figura 46 – Pré-condições para a Ocorrência de Colisões 116
Figura 47 – Classificação de Dano de Obstáculo 118
Figura 48 – Diagrama de Classes: Módulo UTIL 119
Figura 49 – Diagrama de Sequência: Comunicação Ambiente-Simulador 120
Figura 50 – Diagrama de Implantação: Comunicação Ambiente-Simulador 121
Figura 51 – Legenda das Representações do Ambiente Criadas pelo Simulador 123
Figura 52 – Representação da Matriz do Ambiente 124
Figura 53 – Representação da Coordenada do Depósito 125
Figura 54 – Representação das Crenças do Agente Referentes ao Ambiente 126
Figura 55 – Representação do Direcionamento do Veículo 127
Figura 56 – Representação de Obstáculos 128
Figura 57 – Representação de Obstáculos Conhecidos pelo Agente 129
Figura 58 – Representação dos Pontos de Partida e de Destino 130
Figura 59 – Representação dos Níveis de Dano de um Obstáculo 130
Figura 60 – Representação da Recusa de Corridas - Pontos de Partida e Destino Inacessíveis 131
Figura 61 – Representação: Recusar a Corrida - Veículo Indisponível 131
Figura 62 – Exemplo do Funcionamento do Simulador 134
Figura 63 – Autômato de Büchi: $\Box\neg A_{vehicle}honk$ 147
Figura 64 – Diagrama de Venn referentes as áreas de pesquisa 151

LISTA DE CÓDIGOS

Código 1 Exemplo do uso da Classe Predicate 44
Código 2 Exemplo do uso das Classes NumberTerm e NumberTermImpl 44
Código 3 Exemplo do uso da classe VarTerm 44
Código 4 Exemplo de um Ambiente em Gwendolen 46
Código 5 Exemplo Básico de um Agente em Gwendolen 48
Código 6 Exemplo de um agente robô pegando um cascalho 49
Código 7 Exemplo avançado de um agente robô pegando um cascalho 50
Código 8 Exemplo de um arquivo AIL 51
Código 9 Sistema Program 59
Código 10 Exemplo de um arquivo .jpf 66
Código 11 Exemplo de um arquivo .psl 66
Código 12 !finish_all_rides [achieve] : 1 76
Código 13 !finish_all_rides [achieve] : 2 78
Código 14 !complete_journey (X,Y) [perform] 79
Código 15 !clear 81
Código 16 !drive_to(X,Y) [achieve] 83
Código 17 !get_direction [perform] 85
Código 18 !drive_ direction (D) [perform] 86
Código 19 known_route (DIRECTION, KR_X, KR_Y) 88
Código 20 !adapt_route (D, X, Y) [achieve] 91
Código 21 !adapt_drive_direction(A_D, X, Y) [perform] 94
Código 22 !invalid_coordinate (TD) [perform] 96
Código 23 !choose_obstacle_collision [perform] 96
Código 24 !colide_obstacle(DIRECTION, DAMAGE_LEVEL) [perform] . . 98
Código 25 !control_emergency (AT_X, AT_Y) [achieve] 100
Código 26 Agente *Autonomous Car* 157
Código 27 Ambiente, Classe AutonomousCarEnv 166
Código 28 Classe Coordinate 183
Código 29 Classe GridCell 183
Código 30 Classe Passenger 185
Código 31 Classe Client 187
Código 32 Classe Simulator 188
Código 33 Código da Especificação Formal das Propriedades do Agente 199

LISTA DE ABREVIATURAS E SIGLAS

AIL	*Agent Infrastructure Layer*
AJPF	*Agent* Java *PathFinder*
AOP	Programação Orientada a Agentes
AVIA	*Autonomous Vehicles with Intelligent Agents*
BDI	*Beliefs*, *Desires and Intentions*
DOT	*Department of Transportation*
GPAS	Grupo de Pesquisa em Agentes de Software
GPS	*Global Positioning System*
JPF	Java *PathFinder*
LTL	Lógica de linguagem temporal
MCAPL	*Model Checking Agent Programming Languages*
NHTSA	*National Highway Traffic Safety Administration*
SMA	Sistema Multiagente
SUV	*Sport Utility Vehicle*
UDP	*User Datagram Protocol*
UML	*Unified Modeling Language*
VA	Veículo Autônomo

SUMÁRIO

1 INTRODUÇÃO **13**
1.1 OBJETIVOS 15
1.1.1 Objetivo Geral 15
1.1.2 Objetivos Específicos 16
1.2 JUSTIFICATIVA 16
1.3 ORGANIZAÇÃO DO TRABALHO 17
2 VEÍCULOS AUTÔNOMOS **18**
2.1 NÍVEIS DE AUTONOMIA 19
2.2 ADAPTAÇÃO E UTILIZAÇÃO DE VEÍCULOS AUTÔNOMOS EM CIDADES .. 19
2.3 PRINCIPAIS PROJETOS DA ÁREA 20
2.3.1 Tesla 21
2.3.2 Google 22
2.4 ÉTICA E VEÍCULOS AUTÔNOMOS 23
2.5 RESUMO DO CAPÍTULO 25
3 AGENTES RACIONAIS: ARQUITETURA E IMPLEMENTAÇÃO **26**
3.1 AMBIENTES 28
3.2 ARQUITETURA DE AGENTES RACIONAIS 30
3.3 IMPLEMENTAÇÃO DE UM AGENTE RACIONAL 34
3.4 IMPLEMENTAÇÃO DE UM AGENTE BDI 35
3.5 LINGUAGEM DE PROGRAMAÇÃO PARA AGENTES 38
3.6 GWENDOLEN 39
3.6.1 Propriedades e Sintaxe da Linguagem Gwendolen 40
3.6.2 Implementação de um Agente em Gwendolen 43
3.7 RESUMO DO CAPÍTULO 52
4 VERIFICAÇÃO FORMAL DE AGENTES INTELIGENTES **53**
4.1 LÓGICA TEMPORAL LINEAR 55
4.2 AUTÔMATO DE BÜCHI 55
4.3 *MODEL CHECKING* 56
4.4 ABORDAGEM *ON-THE-FLY* 60
4.5 *MODEL CHECKING PROGRAM* 62
4.6 AJPF 63
4.6.1 Sintaxe das Especificações Formais 64
4.6.2 Arquivo de Configuração 65
4.7 RESUMO DO CAPÍTULO 66
5 DESENVOLVIMENTO **68**
5.1 CONTEXTUALIZAÇÃO 68
5.1.1 Delimitação do Trabalho 71
5.2 AGENTE MODELADO COMO UM VEÍCULO AUTÔNOMO 72
5.2.1 Planos Básicos 74
5.2.2 Regras de Raciocínio para Rotas Conhecidas 87
5.2.3 Conjunto de Planos para o Desvio de Obstáculos 88
5.2.4 Conjunto de Planos de Escolha do Menor Dano Possível em Caso de Colisões 96
5.2.5 Conjunto de Planos de Controle e Recuperação em Casos de Colisões 98
5.3 AMBIENTE 101
5.3.1 Atualizar a Localização do Agente 104
5.3.2 Ações do Agente sobre o Ambiente 107

5.3.3 Colisões Inevitáveis ... 115
5.3.4 UTIL ... 118
5.3.5 Simulador ... 119
6 RESULTADOS ... 132
6.1 EXEMPLO DO FUNCIONAMENTO DO SIMULADOR ... 132
6.2 VERIFICAÇÃO FORMAL ... 135
6.2.1 Localizar Veículo no Ambiente ... 136
6.2.2 Buscar uma Nova Corrida ... 136
6.2.3 Embarque de Passageiros ... 136
6.2.4 Recusar Nova Corrida ... 138
6.2.5 Recusar Corrida Atual ... 138
6.2.6 Concluir Trajetos ... 139
6.2.7 Desvio de Obstáculos ... 141
6.2.8 Colisões Inevitáveis ... 145
6.2.9 Investigação de Erros da Verificação Formal ... 146
7 CONCLUSÃO ... 148
7.1 TRABALHOS FUTUROS ... 150
7.2 TRABALHOS RELACIONADOS ... 150
REFERÊNCIAS ... 156
APÊNDICE A - AGENTE ... 157
APÊNDICE B - AMBIENTE ... 166
APÊNDICE C - UTIL ... 183
APÊNDICE D - SIMULADOR ... 187
APÊNDICE E - VERIFICAÇÃO FORMAL ... 199

1 INTRODUÇÃO

O cotidiano em metrópoles no mundo inteiro foi alterado devido ao processo de urbanização presenciado nos últimos anos. Atualmente, mais da metade da população vive em centros urbanos, e a previsão para as próximas décadas é que o número de habitantes cresça ainda mais dentro das cidades (NATIONS, 2015). O aumento de automóveis em circulação é proporcional ao crescimento populacional nas zonas urbanas. Logo, problemas na área de transporte e mobilidade urbana serão intensificados, e portanto, novas estratégias para a administração do tráfego urbano devem ser produzidas. Com o auxílio de tecnologias de informação e comunicação, uma das soluções propostas é o desenvolvimento de veículos autônomos, considerado como o próximo grande avanço tecnológico da indústria automobilística (WALLACE; SILBERG, 2012).

Um veículo autônomo (VA) é um automóvel capaz de transitar no perímetro urbano, ou em rodovias, por meio de decisões tomadas por um software. De acordo com NHTSA (2012) (*National Highway Traffic Safety Administration*), essa autonomia do sistema controlado sobre o veículo é classificada em um espectro de cinco níveis, variado dos níveis 0 (nenhuma autonomia) ao 4 (totalmente autônomo). Haverá uma grande mudança em toda a dinâmica da indústria automotiva e na forma como os carros são utilizados estimulada pela produção da tecnologia dos veículos autônomos. Atualmente, existem 33 empresas desenvolvendo projetos para carros autônomos (CBINSIGHTS, 2016); entre as companhias com maior destaque nos avanços da área estão o Google (2017) e o Tesla (2016).

Entre os benefícios da adoção do veículo autônomo, destaca-se: (i) o potencial de diminuir o congestionamento nas cidades através de predição de tráfego e análise de padrões no trânsito (FAGNANT; KOCKELMAN, 2013); (ii) jornadas mais rápidas e movimentação entre veículos, com o uso de comboios sincronizados de automóveis trocando dados entre si (CLIFFE, 2016); (iii) possível redução do número de acidentes de trânsito utilizando um software como motorista do veículo, uma vez que o fator humano é a causa de 90% dos acidentes no trânsito (PEDEN *et al.*, 2004); (iv) diminuição de veículos em circulação e aumento da disponibilidade de vagas em estacionamentos com a adoção de um sistema de partilha de automóveis; durante sua vida útil, um carro só está ativo 5% do tempo. Por meio do compartilhamento de carros autônomos este índice pode subir para 75%. Assim, menos veículos serão necessários e é estimado que haja uma redução de 90% nessa demanda (CLIFFE, 2016). Consequentemente, a utilização de menos carros acarretará na diminuição de emissão de dióxido de carbono (CO_2) e eliminação do índice de construção de novos estacionamentos.

A habilidade de se locomover sozinho, sem intervenção humana, é uma das capacidades esperadas em um veículo autônomo. Desta forma, o sistema controlador gerencia as funções básicas e cotidianas do automóvel, como: aceleração, manter a velocidade, frear, controlar a direção e detectar a sinalização do trânsito. Além destas habilidades elementares, o veículo deve estar preparado para lidar com situações críticas do tráfego urbano, como o desvio de obstáculos,

onde o trajeto atual deve ser modificado para evitar colisões como: pessoas atravessando a rua fora da faixa de pedestres, animais na rodovia, sinalização de veículos de emergência e acidentes no trânsito. Então, é esperado que o automóvel esteja preparado para garantir a segurança de seus passageiros nestes possíveis cenários.

Lin *et al.* (2015) propõem a seguinte situação: "Um veículo autônomo está trafegando em uma rodovia. Movendo-se a sua frente está um caminhão carregado de madeiras, e nas faixas laterais ao veículo encontram-se uma motocicleta e uma SUV (carro utilitário esportivo). Uma das toras se desprende do carregamento do caminhão e irá colidir com o veículo. No entanto, não há tempo para frear antes da batida. Agora o automóvel tem três opções: desviar em direção a SUV ou colidir com a moto, ou enfrentar o impacto ocasionando em ferimento de seus passageiros". O cenário descrito anteriormente é uma versão moderna e real do famoso Dilema do Bonde (*Trolley Problem*) proposto por Phillipa Foot (1967). Independente da escolha feita em cenários como o anterior, deve-se garantir que o veículo autônomo irá se comportar conforme a sua implementação. Como o veículo irá atuar no trânsito e poderá causar danos à terceiros e aos seus próprios passageiros, é imprescindível que o sistema autônomo funcione corretamente. Portanto, deve ser garantido que não haja falhas no software de controle, que poderá ocasionar que o veículo se comporte de maneira inesperada perante situações do trânsito. Logo, o automóvel não deve executar uma ação própria de um cenário crítico (*e.g.*, eminência de colisão com outro veículo) em um cenário comum (*e.g.*, desacelerar e fazer uma leve curva à direita), ou vice-versa.

Para entender como verificar as decisões de um veículo autônomo, é necessário compreender a tecnologia utilizada para o desenvolvimento do seu software controlador. Como este é considerado um sistema autônomo, uma das abordagens para o seu desenvolvimento é através do uso de agentes. Um agente é uma entidade computacional inserida em um ambiente capaz de tomar decisões por si próprio (WOOLDRIDGE, 2009). Para verificar as decisões desempenhadas por um agente, é necessário saber os motivos que o levou a tomar uma decisão e o que o mesmo pretendia realizar (BORDINI *et al.*, 2008). Os agentes racionais, uma das possíveis arquiteturas para a construção de agentes, realizam seu processo de tomada de decisão com razões explícitas e são capazes de explica-las (DENNIS; FISHER, 2016). A abordagem de agentes racionais mais utilizada é o modelo BDI (*beliefs, desires and intentions* ou crenças, desejos e intenções), onde os estados mentais internos do agente são responsáveis pelo seu comportamento. Uma forma de implementar um agente BDI é por meio da utilização da linguagem de programação Gwendolen (DENNIS; FARWER, 2008), e esta tem sua implementação baseada em Java.

Por meio da verificação formal, será possível validar as escolhas feitas pelo agente controlador do veículo autônomo e suas especificações. A verificação formal é utilizada para garantir a corretude do algoritmo utilizado em um sistema com relação as suas propriedades. Esta metodologia é utilizada em sistemas de segurança crítica, onde é imprescindível o desempenho correto do software (CLARKE JR.; GRUMBERG; PELED, 1999). O *model checking* é uma das técnicas de verificação formal. Que, por meio da análise do conjunto finito de estados do modelo

de um sistema, valida seus requisitos e garante seu desempenho correto. No entanto, este método verifica um modelo do sistema, e não sua implementação ou seu protótipo. Assim, falhas de sistema indetectáveis na modelagem e presentes no código podem ocorrer ao longo da execução do software mesmo após a sua verificação formal. Além disso, não é possível determinar com exatidão quais decisões serão tomadas por um agente antes de sua execução, desta forma, a utilização de um modelo não seja a mais apropriada para verificar suas propriedades. Visser *et al.* (2003) desenvolveram o método *model checking program*, uma extensão do *model checking*, que é utilizado por meio da ferramenta JPF (Java *PathFinder*), permitindo a realização da verificação formal do sistema durante sua execução. Tendo como base a ferramenta JPF, Bordini *et al.* (2008) criaram o framework AJPF (*Agent* Java *PathFinder*), uma versão específica para a verificação de agentes racionais.

Neste trabalho, é apresentando um agente BDI que modela o comportamento de um veículo autônomo desenvolvido na linguagem de programação Gwendolen. O agente possui um acervo de planos capazes de tomar decisões relacionadas com os cenários impostos, como: realização de trajetos pelo ambiente, desvio de obstáculos, e controle de emergências. Com o intuito de obter uma representação gráfica da interação entre agente pelo ambiente, é proposto um simulador capaz de receber tais informações e ilustra-las. São apresentadas as especificações formais das propriedades do agente por meio de fórmulas LTL (lógica de linguagem temporal), onde uma propriedade se refere a um aspecto do processo de tomada de decisão pelo agente. Essa validação é realizada por meio do uso do *model checking program* com o AJPF feita após a conclusão do desenvolvimento do agente. Cenários são impostos por meio da configuração do ambiente, onde é esperado que o veículo transite entre posições e escolha pelos planos corretos para desempenhar tais funções. Garantindo assim que o agente está de acordo com suas especificações. Desta forma, a proposta deste trabalho é realizar a implementação de um agente racional modelado para condução de um veículo autônomo e a verificação formal do seu comportamento utilizando o *model checking program*.

1.1 OBJETIVOS

As seções a seguir descrevem o objetivo geral e os objetivos específicos a serem realizados com este trabalho.

1.1.1 Objetivo Geral

O objetivo geral deste trabalho consiste em implementar um agente BDI modelado como veículo autônomo em um ambiente simulado e realizar a verificação formal de seu processo de tomada de decisão.

1.1.2 Objetivos Específicos

Os objetivos específicos deste trabalho são:

1. Pesquisar o estado atual da arte referente aos veículos autônomos;
2. Consultar referências bibliográficas das áreas de estudo apresentadas no trabalho, agentes e verificação formal;
3. Implementar um agente BDI na linguagem Gwendolen que se comporta como um veículo autônomo, e criar planos para determinar suas habilidades e comportamentos, como:
   a) Realizar trajetos sobre o ambiente onde está inserido;
   b) Desviar de obstáculos ao longo de um percurso;
   c) Aptidão para tomar decisões que minimizem os danos causados ao veículo em situações onde o desvio de obstáculos não é possível.
4. Definir um ambiente computacional na linguagem Java capaz de simular as situações para as quais o agente foi programado;
5. Desenvolver um simulador gráfico com auxílio da biblioteca *2D Graphics* da linguagem Java para ilustrar a interação entre ambiente e agente;
6. Analisar as propriedades do agente e especifica-las formalmente;
7. Realizar a verificação formal do agente utilizando o *model checking program*, por meio da:
   a) Especificação formal das propriedades do agente desenvolvido utilizando a notação da lógica de linguagem temporal;
   b) Imposição de cenários no ambiente onde estas propriedades devem ser respeitadas;
   c) Execução da ferramenta AJPF para verificar a veracidade de tais propriedades durante o funcionamento do agente.

1.2 JUSTIFICATIVA

Carros autônomos serão uma realidade do cotidiano das grandes cidades em um futuro não tão distante (WALLACE; SILBERG, 2012). No trabalho de Fisher, Dennis e Webster (2013) salienta-se a necessidade da verificação formal de sistemas autônomos. Devido ao fato do VA atuar em um ambiente crítico, é imprescindível que o software por trás dessa tecnologia funcione corretamente. O veículo autônomo deve proteger seus passageiros, especialmente em

situações críticas. Qualquer decisão errada tomada pelo sistema autônomo controlando o automóvel pode ter consequências fatais. No entanto, tais erros do sistema podem ser verificados através da análise de sua implementação.

Este trabalho tem o intuito de aplicar a verificação formal em um agente implementado que representa um veículo autônomo, para que se possa assegurar a sua corretude ao longo de seu desempenho. Sendo possível utilizar as informações providas pela realização do *model checking program* sobre o software como garantia de seu funcionamento correto. Portanto, será possível contribuir com o projeto AVIA (*Autonomous Vehicles with Intelligent Agents*, ou Veículos Autônomos com Agentes Inteligentes) do Grupo de Pesquisa em Agentes de Software (GPAS), no campus Ponta Grossa da UTFPR. Como essa é uma área de pesquisa recente e em virtude deste trabalho ser um dos primeiros dentro deste projeto AVIA, busca-se apresentar resultados iniciais que evidenciem a possibilidade de usar o AJPF na verificação de planos de um agente racional modelado como um veículo autônomo. Dessa forma, futuros trabalhos poderão expandir o cenário de atuação do agente apresentado por este trabalho criando novos planos para seu comportamento; e também, utilizar o estudo realizado sobre o AJPF para verificar novas propriedades.

1.3 ORGANIZAÇÃO DO TRABALHO

A estrutura deste trabalho consiste em seis capítulos principais. No Capítulo 2 é abordado em mais detalhes o estado da arte atual para veículos autônomos e as principais empresas da área. Definições sobre agentes, possíveis arquiteturas e linguagens de programação voltada para agentes são assuntos discutidos no Capítulo 3. Na sequência, o Capítulo 4 foca nos detalhes sobre a verificação formal, *modal checking*, JPF e AJPF. O desenvolvimento realizado neste trabalho pode ser encontrado no Capítulo 5, e os resultados obtidos são apresentados no Capítulo 6. E por fim, o Capítulo 7 é destinado para as conclusão das conquistas do estudo apresentado aqui e quais futuros trabalhos podem ser derivados.

2 VEÍCULOS AUTÔNOMOS

Um veículo autônomo (ou VA) é um automóvel que possui um software embarcado e o hardware necessário que o permite a trafegar de maneira autônoma. Para Wallace e Silberg (2012) essa é uma das possíveis soluções propostas para problemas enfrentados no cotidiano do trânsito dentro de cidades, tais como:

- Acidentes no tráfego: que acarretam em fatalidades e em despesas extras para o conserto e manutenção de infraestruturas públicas. O uso de um software como motorista do veículo pode acarretar na redução do número de acidentes de trânsito, uma vez que pesquisas apontam que o fator humano é a causa de 90% dos acidentes no trânsito (PEDEN *et al.*, 2004);

- Congestinamentos: durante sua vida útil, um carro só está ativo 5% do tempo, e não está sendo utilizado no restante do tempo. Uma das possíveis aplicações para o veículo autônomo é um sistema de partilha de automóveis (i.e., táxi), e com isso será possível aumentar o índice de utilização de veículos em até 75%. Assim a demanda por novos veículos será menor, e é estimado que haja essa redução seja de 90% (CLIFFE, 2016). Além disso, Fagnant e Kockelman (2013) afirmam que uma quantidade menor de automóveis em circulação pode acarretar na redução de congestionamento dentro das cidades e as jornadas serão mais rápidas (CLIFFE, 2016). Em rodovias, é estimado que a necessidade por automóveis irá cair em 80% (LUBELL, 2016);

- Espaço necessário para a construção de estacionamentos: como uma produtividade dos e utilização veículos maior, já que podem ficar trafegando por um tempo muito grande, o espaço necessário para estacionamentos será menor (MARSHALL; DAVIES, 2016). As áreas reservadas para estacionamentos são uma das preocupações de muitas cidades americanas, que ocupam até mais de um terço do território urbano (WALLACE; SILBERG, 2012).

Sridhar Lakshmanan (PULLEN, 2015) afirma que para dar autonomia a um carro comum são necessários os seguintes itens: (i) Localizador geográfico, i.e., GPS; (ii) Sistema de reconhecimento de ambiente, obtido através do uso de radar e *Lidar* (Detector que utiliza emissões de luzes para identificar o espaço ao redor); (iii) E por fim, um software capaz de interpretar e conectar as informações atuais do ambiente com o espaço geográfico já conhecido para guiar o veículo.

Atualmente, há imposições legais quanto a ausência de um humano no assento do motorista e é necessário criar leis e regulamentos a respeito de possíveis acidentes envolvendo VAs. Portanto, é necessário que os governos e as cidades se adaptem para receber esta tecnologia.

As seções a seguir irão abordar em mais detalhes os seguintes tópicos: níveis de autonomia dos veículos autônomos, efeito dos veículos autônomos em cidades, os principais projetos da área que estão em desenvolvimento, os problemas éticos envolvendo o veículo autônomo e as perspectivas da área.

2.1 NÍVEIS DE AUTONOMIA

O nível da autonomia de um VA determina quais funções que este é capaz de realizar sem interferência externa, i.e., sem motorista humano. A *National Highway Traffic Safety Administration* (NHTSA), órgão nacional norte-americano responsável pela segurança em rodovias, classifica os níveis de autonomia nas seguintes cinco categorias (NHTSA, 2012):

- Nível 0 - Nenhuma Autonomia: O motorista tem controle total sobre as funcionalidades do veículo;
- Nível 1 - Autonomia específica em funções: Determinadas funcionalidades do veículo são controladas pelo software, porém de forma independente entre si. Nessa fase, cabe ao motorista operar sobre situações críticas;
- Nível 2 - Autonomia em funções combinadas: O sistema controla ao menos duas funções principais do veículo e compartilha o comando do automóvel com o motorista;
- Nível 3 - Direção autônoma parcial: O software embarcado no veículo é capaz de administrar totalmente sobre as funcionalidades, mas o motorista pode optar por tomar o controle da direção em situações de segurança-crítica;
- Nível 4 - Direção autônoma total: O sistema controla todo o funcionamento do veículo e não há a presença de um motorista humano.

2.2 ADAPTAÇÃO E UTILIZAÇÃO DE VEÍCULOS AUTÔNOMOS EM CIDADES

Um dos maiores impactos do aumento da população em cidades é na área de transporte e locomoção urbana. Esse problema é uma das preocupações de diversas órgãos públicos e instituições ao redor do mundo, entre estas estão o órgão *Department of Transportation*[1] (DOT) e a *Bloomberg Philanthropies*.

Em 2015, foi lançado pelo DOT o programa *Smart City Challenge*, buscando propostas de cidades americanas sobre como melhorar a mobilidade urbana. Todos os sete projetos finalistas apresentaram soluções envolvendo veículos autônomos para o desafio, que teve como

[1] Agência Norte-Americana.

vencedor a cidade de Columbus. Em seu projeto, Columbus propôs uma frota de ônibus elétricos para deslocar pessoas em centros comerciais lotados e capaz de atender a requisições de moradores de áreas remotas em casos de emergência (MARSHALL, 2016).

A *Bloomberg Philanthropies* começou uma iniciativa para auxiliar cidades se adaptarem aos veículos autônomos e a melhor forma de aplicar sua utilização em problemas reais. As cidades dentro do programa atualmente são: Los Angeles (EUA), Nashville (EUA), Austin (EUA), Paris (França) e Buenos Aires (Argentina). Estas metrópoles irão compartilhar dados sobre seus projetos e receberão treinamento de urbanistas e cientistas de área da tecnologia (MCGEE, 2016).

O teste de veículos autônomos em espaços públicos parece ser uma ideia arriscada. Contudo, prefeitos ao redor do mundo, como os das cidades de Pittsburgh e Columbus nos Estados Unidos, têm aderido a ideia e facilitado regulamentos para que empresas e pesquisadores possam ter seus automóveis transitando dentro das cidades (MARSHALL; DAVIES, 2016).

2.3 PRINCIPAIS PROJETOS DA ÁREA

Hoje, 33 companhias estão trabalhando em busca da tecnologia que seja capaz de criar o veículo autônomo (CBINSIGHTS, 2016). Entre estas empresas, estão grandes multinacionais do setor automotivo e gigantes da tecnologia. Existem duas possíveis abordagens no desenvolvimento da tecnologia adotados por empresas, são estas:

- Autonomia fragmentada: Veículos, com a estrutura dos automóveis atuais, implementados desta forma possuem um controle de navegação com funções básicas, tais como manter na faixa atual e mudar de faixa em rodovias (MARSHALL; DAVIES, 2016). No entanto, tais automóveis necessitam de um motorista capaz de lidar com as situações para as quais o veículo não foi programado. Desta forma, o software funciona como um piloto automático, similar aos softwares utilizados para comandar aeronaves. Atualmente, as grandes fabricantes de carros são mais favoráveis a esta abordagem. A Tesla já disponibiliza a aquisição de veículos capazes de navegar de forma autônoma em rodovias (TESLA, 2016);
- Autonomia total: Esta é a implementação que o Google tem utilizado em seu projeto chamado Waymo (GOOGLE, 2017). Seu propósito é construir um veículo completamente autônomo, de tal forma que será possível remover o volante e os pedais de sua estrutura (MARSHALL; DAVIES, 2016). Carros utilizando esta abordagem utilizam *geofencing*, uma ferramenta utilizada para modelar barreiras físicas virtualmente (ROUSE, 2015). Contudo, para este modelo entrar em circulação, as leis referentes ao tráfego terão que ser alteradas em diversos países, onde é indispensável a presença de uma pessoa como motorista do veículo.

O consenso atual entres as fabricantes é que automóveis semiautônomos, ou nível 3 de autonomia, estarão prontos para transitar no início da próxima década. Em contra partida, um estudo realizado pela *National League of Cities* mostra que somente 6% das cidades americanas estão se preparando para receber carros autônomos em suas ruas (MORRIS, 2015). Considerando que as pesquisas mais relevantes da área são realizadas nos Estados Unidos, esse atraso pode indicar uma demora para que o público geral tenha acesso a esta tecnologia. Entre esses projetos, estão os conduzidos pela Tesla e Google, que serão abordados nas subseções a seguir.

2.3.1 Tesla

Sob a liderança do CEO Elon Musk, a Tesla é uma das fabricantes de carros mais tecnologicamente desenvolvida na atualidade (CROTHERS, 2015). Parte desse reconhecimento deve-se aos avanços alcançados pela empresa nas áreas de assistência avançada ao motorista e a tecnologia de auto direção (CBINSIGHTS, 2016). A empresa já comercializa o Model S e o Model X, seus modelos de veículos autônomos, que possuem autonomia no nível 3.

Figura 1 – Sensores do veículo da Tesla

Rearward Looking Side Cameras
Max distance 100m
Wide Forward Camera
Max distance 60m
Main Forward Camera
Max distance 150m
Narrow Forward Camera
Max distance 250m
Rear View Camera
Max distance 50m
Ultrasonics
Max distance 8m
Forward Looking Side Cameras
Max distance 80m
Radar
Max distance 160m

Fonte: Tesla (2016)

O pacote de hardware do *Autopilot* foi introduzido pela Tesla em seus veículos em setembro de 2014 (LAMBERT, 2016). Este conjunto, exibido na Figura 1 compõe-se de: seis diferentes câmeras (onde realizam as funções de visão traseira, visão lateral traseira, visão ampla frontal, visão principal frontal, visão estreita frontal e visão lateral frontal), sensor ultrassônico ao redor do veículo e um radar frontal. Pode-se observar na figura tais componentes sensoriais dos veículos da Tesla atuais, onde estes são o hardware necessário para a direção autônoma do

veículo. Uma vez que todos veículos já possuem o equipamento necessário para serem autônomos, é esperado que o software seja atualizado para aumentar o nível de autonomia de um veículo com tais componentes.

Além disso, os veículos da Tesla possuem um software chamado *Traffic-Aware Cruise Control* (TACC), responsável por funções como monitorar velocidades máximas na rodovia e manter distâncias seguras entre os veículos. No final de 2015, foi lançado a primeira grande expansão do *Autopilot*, a versão 7.0, que conta com a funcionalidades *Autosteer* e *Autopark* (LAMBERT, 2016). A primeira é responsável por realizar a direção do veículo em rodovias, permitindo que o mesmo se mantenha dentro de sua faixa. Já o segundo, permite o carro procure vagas e estacione autonomamente. Desde então, outras funcionalidades foram lançadas como: detectar e obedecer às regras de velocidade máxima permitida, manter-se na mesma faixa, mudança de faixa (e.g., o veículo é capaz de detectar faixas lentas e trocar por uma mais rápida) e sair e entrar na garagem sob comando. As medidas de segurança desses veículos incluem freio de emergência automático, capaz de detectar objetos que possam atingir o automóvel e avisos de colisões frontais e laterais e habilidade de desvio de rota.

2.3.2 Google

O Google possui um dos projetos mais notáveis para a criação de veículos autônomos, o Waymo, que conta atualmente com dois automóveis operando sob o controle um software.

Um desses modelos é a mini van *Chrysler Pacifica Hybrid*, mostrado na Figura 2, que possui a estrutura de um carro comum porém está adaptado para o controle autônomo. Já o outro modelo, exibido na Figura 3, é um automóvel totalmente voltado para a direção autônoma, e não possui volante e pedais.

Figura 2 – *Chrysler Pacifica Hybrid*

Fonte: Google (2017)

O software por trás desse veículo possui o seguinte ciclo de processamento para sua

Figura 3 – Waymo

Fonte: Google (2017)

tomada de decisões:

1. Localização: o automóvel é capaz de determinar onde está localizado por meio do GPS e sensores;

2. Ambientação: sensores e câmeras auxiliam o veículo a observar objetos em sua volta;

3. Predição: o sistema embarcado do veículo faz uma predição das próximas ações dos objetos em sua volta;

4. Deliberação: com base nas informações adquiridas nas etapas 1, 2 e 3, o software é capaz de decidir uma trajetória segura para o carro.

John Krafcik, chefe executivo do projeto, afirma em Naughton (2016) que "... é preciso tirar o fator humano da jogada", e este tem sido o foco da empresa ao implementar seus projetos.

O Waymo teve seu início em 2009 sob o nome Google *Self-Driving Car Project*, e atingiu a marca de 3 milhões de milhas percorridas (aproximadamente 4,8 milhões quilômetros) de forma autônoma em maio de 2017 (GOOGLE, 2017). Primeiramente, os automóveis foram testados em rodovias por ser considerado um cenário menos complexo. Mas já é possível encontrar estes carros trafegando em centros urbanos. Os testes deste projeto estão presentes em diferentes cidades americanas, Kirkland, Mountain View, Phoenix e Austin, com o propósito de avaliar os veículos em diferentes condições espaciais e climáticas (GOOGLE, 2017).

2.4 ÉTICA E VEÍCULOS AUTÔNOMOS

Cientistas, engenheiros e CEOs estão otimistas e preveem a presença de veículos autônomos transitando até o início da próxima década. Contudo, a disponibilização desses auto-

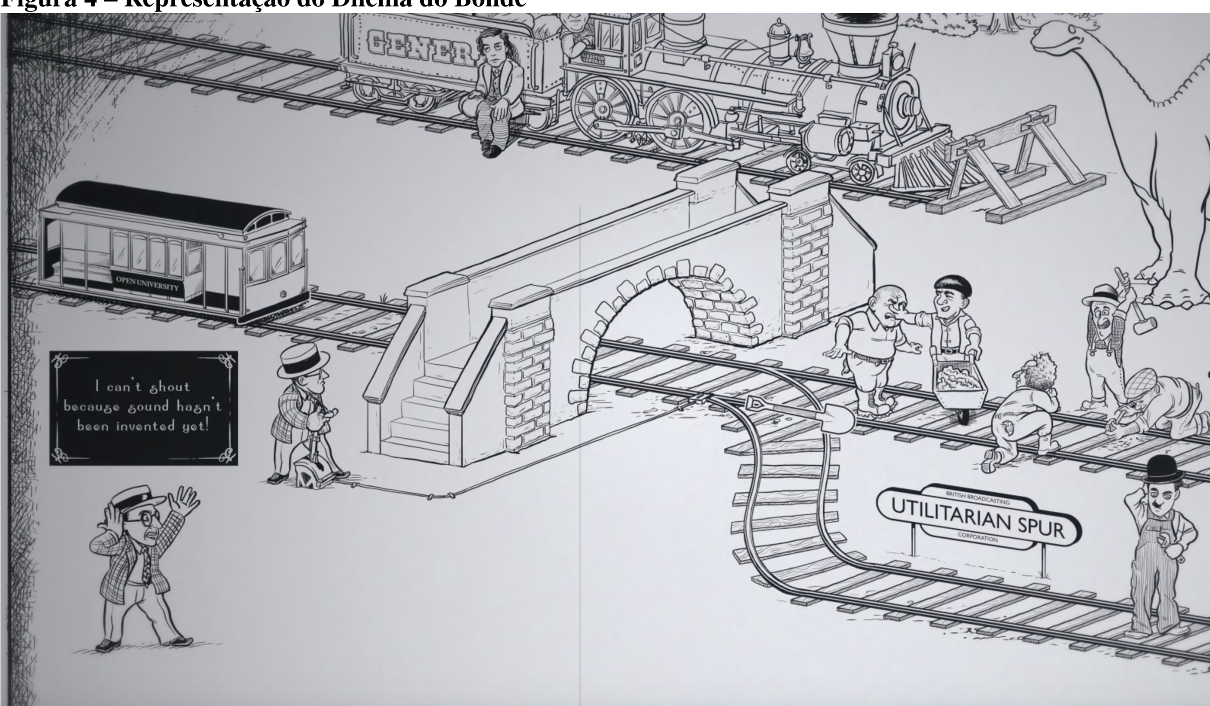
móveis para o público geral deve passar deste prazo. Além das dificuldades tecnológicas para desenvolve-los, os veículos autônomos dão origem a questões éticas e morais que levarão algum tempo para serem respondidas (BATTELLE, 2016).

Um clássico problema moral é o Dilema do Bonde (*Trolley Problem*), apresentado por Phillipa Foot em 1967. Tal problema propõe o cenário exibido na Figura 4 e descrito a seguir:

> Cinco pessoas inocentes estão amaradas em um trilho e um trem desgovernado irá passar por cima delas. Para salva-las, pode-se puxar uma alavanca para mudar a trajetória do trem para outro trilho. No entanto, existe uma pessoa, também inocente, desatenta parada sob o trilho. Nesta situação, não há tempo para desamarram ninguém ou avisar o sujeito distraído. Logo, deve-se decidir se a alavancada deve ser puxada ou não (FOOT, 1967).

Entre as diversas variações desse dilema propostas através dos anos, há uma envolvendo veículos autônomos, como o discutido anteriormente na introdução deste trabalho.

Figura 4 – Representação do Dilema do Bonde



Fonte: BBC Radio (2014)

Os veículos autônomos devem ser capazes de transitar sozinhos e decidir quais as melhores ações devem ser executadas. Dentre os possíveis cenários, há situações onde software deverá decidir entre por a vida dos passageiros em risco ou causar danos à terceiros. Doctorow (2015) apresenta o caso em que o veículo poderá desviar-se para não atingir um ônibus lotado com crianças, e consequentemente levar seus passageiros à óbito. A outra opção seria o automóvel continuar na mesma trajetória, porém as crianças no ônibus não irão sobreviver. Diante dessas circunstâncias, o dilema é: Qual ação o software deve ser programado para realizar? Ao contrário de um ser humano, que quando imposto a esta situação irá ter uma reação instintiva, a decisão tomada pelo software é proposital, uma vez que está descrita no design de seu código

(LIN *et al.*, 2015). Logo, em teoria, qualquer fatalidade causada pelo veículo pode ser interpretada como um crime premeditado. Segundo as leis atuais, a responsabilidade por qualquer dano causado por um automóvel é do motorista dirigindo o veículo. Mas se não há um humano guiando o automóvel, quem deve ser responsabilizado são os passageiros, a fabricante ou o governo, por permitir a circulação desses robôs?

Embora o debate sobre o comportamento ético do veículo ainda esteja em aberto, é imprescindível que o sistema controlador do funcione corretamente para obter a confiança do público geral. Além disso, sua corretude é necessária para possuir certificados de qualidade, que podem vir a auxiliar na legalização do uso destes veículos em espaços públicos. Na sequência deste trabalho, os conceitos do software controlador do veículo autônomo e a verificação da corretude do sistema serão explorados em mais detalhes.

2.5 RESUMO DO CAPÍTULO

A perspectiva da adoção de veículos autônomos é bem promissora, já que as vantagens trazidas por essa tecnologia são enormes. Várias empresas estão interessadas em desenvolver um automóvel desta categoria.

As principais barreiras para essa implantação são os aspectos éticos e a regulamentação necessária para o seu uso em espaços públicos. Atualmente há debates sobre as escolhas realizadas pelo sistema autônomo controlando o veículo, suas consequências, quem deve decidir o que deve ser feito e quem deve ser tido como responsável pelas ações do veículo. Além disso, é preciso regulamentar o uso destes veículos em centros urbanos, pois legislações atuais não permitem a ausência de um motorista no controle do veículo.

Logo, é necessário que no mínimo este software seja certificado à prova de erros referentes ao seu design, ou seja, seu código e sua modelagem devem estar corretos; onde tais erros são consequências da falta de uma verificação completa do software. Logo, a verificação e garantia do funcionamento correto deste sistema é indispensável para garantir a qualidade de seu funcionamento. No capítulo 5, é abordado a verificação das propriedades de um sistema modelado como um veículo autônomo.

3 AGENTES RACIONAIS: ARQUITETURA E IMPLEMENTAÇÃO

Para que um veículo seja autônomo é necessário a existência de um sistema embarcado capaz de controlar suas funcionalidades. Uma das possíveis abstrações para o desenvolvimento de um sistema autônomo é por meio do uso de agentes (WOOLDRIDGE, 2009).

No primeiro capítulo deste trabalho foi apresentada uma breve definição sobre o que é um agente, contudo, não há uma definição exata sobre este termo dentro da Ciência da Computação. Diferentes autores na área utilizam suas próprias interpretações sobre o conceito; Wooldridge (2009) relata que um dos motivos da dificuldade em definir o termo deve-se ao fato de que a relevância de determinadas características do agente estão diretamente relacionadas com seu domínio de aplicação. No entanto, o consenso geral é que um agente deve possuir autonomia (WOOLDRIDGE, 2009). O termo autonomia se refere a capacidade de o agente ter controle sobre seu processo de tomada de decisão, sem a intervenção externa.

Segundo Ferber (1999), um agente é uma entidade virtual ou física tal que,

a) é capaz de agir em um ambiente;

b) pode se comunicar diretamente com outros agentes;

c) é motivado por um conjunto de tendências (na forma de metas individuais);

d) possui recursos próprios;

e) consegue perceber o ambiente onde está situado de forma limitada;

f) dispõe de uma representação parcial (ou nenhuma) do seu ambiente;

g) tem habilidades e pode oferecer serviços;

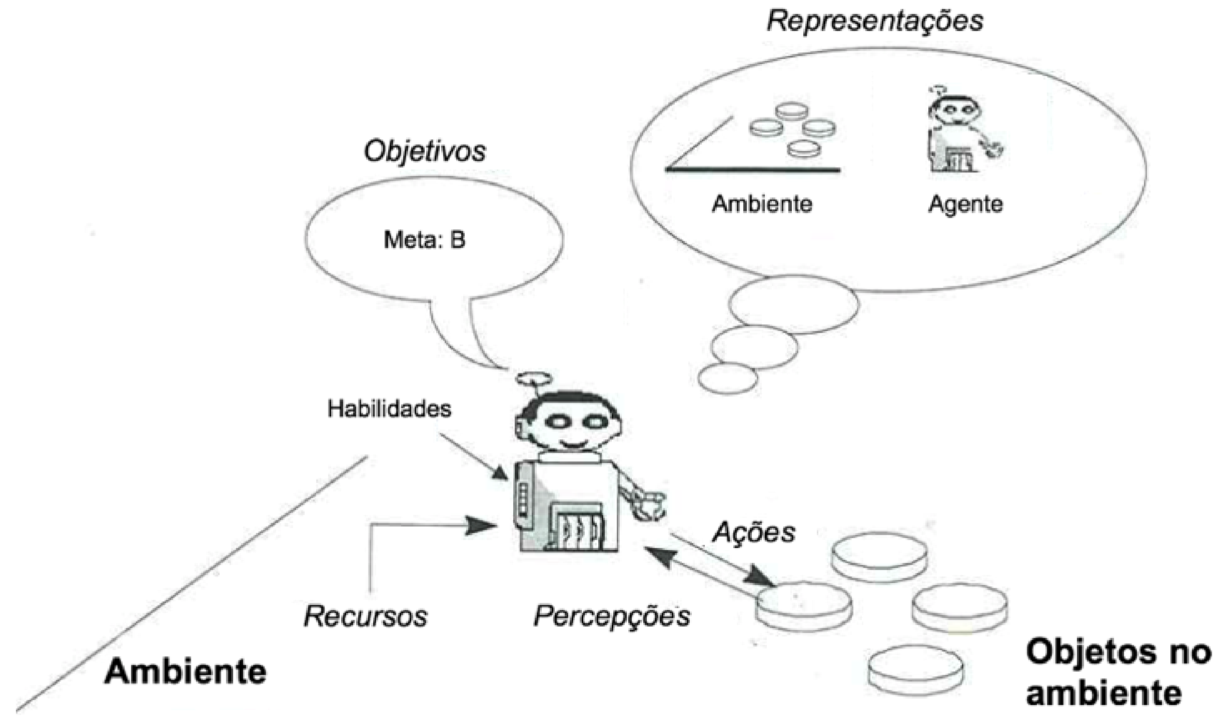
h) pode ser capaz de se reproduzir;

i) cujo o comportamento é atingir seus objetivos, considerando os recursos e as habilidades disponíveis e, dependendo de sua percepção, representações sobre o ambiente e a comunicação recebida.

Agentes executam ações para modificar seu ambiente, e são motivados por um conjunto de tendências, onde este conjunto pode conter metas individuais a serem atingidas (FERBER, 1999). Na Figura 5 é ilustrada a definição de agente descrita por Ferber (1999), onde é possível observar um agente inserido em um ambiente. Tal agente pode perceber modificações por meio de percepções e executar ações no ambiente para alterá-lo, além de possuir habilidades próprias, objetivos individuais e representações de si próprio e do meio onde está inserido.

No entanto, dentro do escopo deste trabalho, será utilizada a definição apresentada por Wooldridge (2009), que define um agente como "um sistema computacional que está situado em

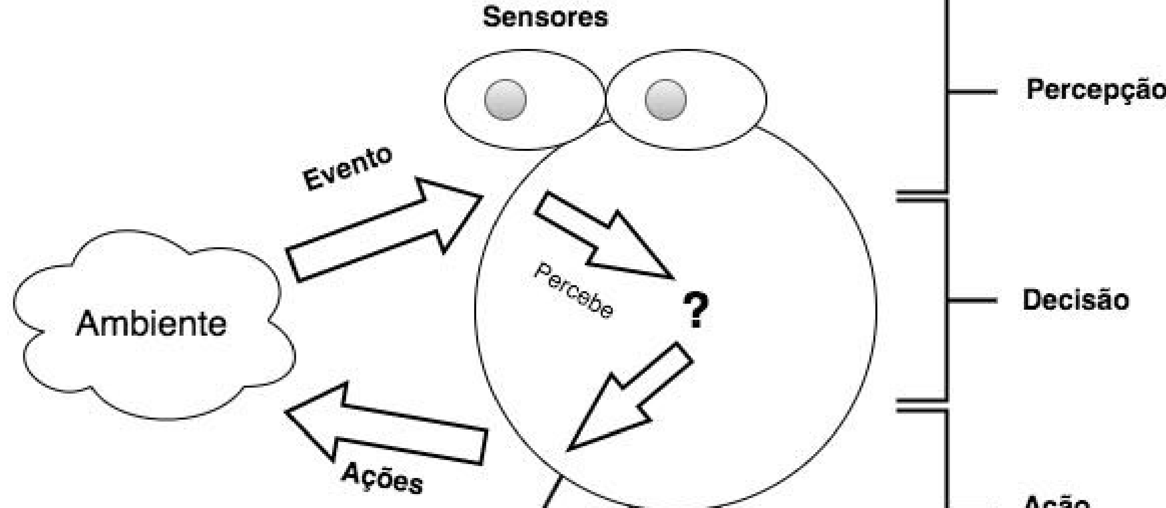
Figura 5 – Representação de um agente interagindo com o ambiente



Fonte: Traduzido de Ferber (1999)

algum ambiente, e é capaz de realizar ações autônomas neste ambiente, visando alcançar seus objetivos propostos". A abstração desse conceito é demonstrado na Figura 6; onde o agente é capaz de observar e perceber eventos no ambiente por meio de sensores, processar qual decisão deve ser tomada e está apto a realizar ações com o auxílio de seus atuadores para modificar o ambiente.

Figura 6 – Agente e sua interação com o ambiente



Fonte: Traduzido de Wooldridge (2009)

Wooldridge (2009) define que um agente inteligente possui as seguintes três características:

- Reatividade: Define a capacidade de o agente perceber mudanças no ambiente e responder as mesmas em tempo hábil para que possa satisfazer seus objetivos;
- Pró atividade: O agente deve ser capaz de tomar iniciativa e executar ações para concluir seu objetivo quando houver uma oportunidade; e
- Habilidade Social: Os agentes presentes em um sistema multiagente deve estar apto a se comunicar com outros agentes presentes no mesmo ambiente onde está inserido para concluir seu objetivo.

Em suma, o agente deve reagir as modificações do ambiente e explorar as oportunidades que surgirem e que favoreçam a conclusão de seus objetivos.

O termo Sistema MultiAgente (SMA) é utilizado para caracterizar um sistema que contém dois ou mais agentes comunicando entre si para executar um conjunto de tarefas (WOOLDRIDGE, 2009). Agentes que estão contidos em um mesmo sistema podem ter objetivos e motivações diferentes para representar os interesses de seus proprietários. No entanto, um SMA pode ser construído com o intuito de resolver um problema complexo, nesses casos além de seus objetivos individuais, os agentes trabalham para alcançar o objetivo global do sistema. Este não é o tipo de sistema abordado por este trabalho, no entanto é necessário para entender o potencial da utilização de agentes.

Nas próximas seções desse capítulo, será abordado em detalhes os ambientes de agentes, a arquitetura de agentes racional, e formas de implementação. É apresentado também a linguagem Gwendolen para a programação de agentes. E por fim, será discutido a utilização de agentes no contexto deste trabalho.

3.1 AMBIENTES

Um ambiente pode ser interpretado como um problema para o qual um agente é a solução (RUSSELL; NORVIG, 2010). As propriedades de um ambiente determinam qual tipo de arquitetura de agentes é a mais apropriada a ser utilizada; e os componentes descrevem o relacionamento entre um agente e o ambiente onde está inserido. As subseções a seguir irão descrever em detalhes as características de um ambiente de agentes.

Propriedades de um ambiente

Em Wooldridge (2009) é sugerido que as características de um ambiente são: nível de observação, determinístico ou estocástico, estático ou dinâmico, discreto ou contínuo, e episódico ou sequêncial.

O nível de observação determina o grau de acesso que um agente tem sobre seu ambiente. Quando os sensores do agente podem obter informações completas, precisas e atualizadas, a qualquer dado momento, sobre o estado do ambiente, é dito que o ambiente é totalmente observável (RUSSELL; NORVIG, 2010). Caso contrário, quando há imprecisão nos dados recebidos ou o agente não possui acesso a determinadas partes do ambiente, o ambiente é caracterizado como parcialmente observável. Um jogo de xadrez possui um ambiente totalmente observável, uma vez que ambos jogadores, agentes, têm conhecimento sobre a localização de todas as peças do tabuleiro. Já em um jogo de cartas como pôquer, o ambiente é parcialmente observável, pois o apostador somente tem ciência sobre suas cartas e as cartas da mesa.

Caracteriza-se um ambiente determinístico se qualquer ação de um agente tem a garantia de resultar em um efeito único sobre o ambiente. Caso contrário, o ambiente é estocástico. No exemplo de uma partida de xadrez, todos os movimentos possíveis dentro do ambiente podem ser determinados a qualquer momento do jogo. E em um ambiente de um jogo de pôquer, pode-se apenas gerar uma probabilidade das cartas de cada jogador, não sendo possível determinar com exatidão.

Em um ambiente estático, nenhuma modificação ocorre enquanto o agente está decidindo qual decisão tomar. Dessa forma, o agente é o único responsável pela alteração do ambiente. Em contrapartida, em um ambiente dinâmico, outros processos estão operando e realizando mudanças além do controle do agente. Ambos os jogos de xadrez e pôquer possuem um ambiente estático, uma vez que nenhuma mudança irá ocorrer enquanto o agente estiver deliberando suas ações. No cenário de veículos autônomos, o ambiente é dinâmico, uma vez que enquanto o agente raciocina sobre seus movimentos, pode haver uma mudança no estado do ambiente.

Ambientes discretos possuem um número finito de estados, dessa forma, limitando o número de percepções e ações do agente. Por outro lado, em um ambiente contínuo não restringe o número de interações do agente. Um jogo de xadrez e pôquer são considerados discretos, e no caso de veículos autônomos, o ambiente é contínuo.

Em um ambiente episódico, cada nova situação em que o agente se depara é única, e não depende dos eventos anteriores. No entanto, em um ambiente sequencial, as ações do agente pode vir a influenciar seu desempenho no futuro (RUSSELL; NORVIG, 2010). Um jogo de xadrez pode ser considerado com sequencial, pois a movimentação das peças podem afetar jogadas futuras.

Um ambiente parcialmente observável, não-determinístico, dinâmico, contínuo e sequencial é considerado por Wooldridge (2009) o tipo mais complexo para ser desenvolvido. No

entanto, dentro do contexto deste trabalho, o ambiente implementado é caracterizado como parcialmente observável, determinístico, estático (porém randômico), discreto e sequêncial.

Componentes de um ambiente

Em seu trabalho, Russell e Norvig (2010) descrevem quatro componentes da relação entre um agente e seu ambiente, sendo estes:

- Função de Desempenho: utilizada para definir a qualidade de uma possível decisão, baseado nas especificações do agente e quais são suas prioridades;
- Elementos do ambiente: são objetos, físicos ou virtuais, presentes no ambiente;
- Atuadores: definem os meios com os quais um agente pode influenciar o ambiente;
- Sensores: definem os meios com os quais um agente pode perceber modificações e eventos ocorridos no ambiente.

No Quadro 1 é demonstrado um exemplo dos componentes de um (possível) ambiente onde um veículo autônomo, modelado por meio de um agente pode vir a atuar.

Quadro 1 – Componentes de um ambiente contendo um veículo autônomo.

Função de Desempenho	**Elementos do ambiente**	**Atuadores**	**Sensores**
Segurança, rapidez, respeito as leis do trânsito, conforto dos passageiros	Estradas, ruas, condições do tráfego, pedestres	Direção, acelerador, freios, setas, faróis	Câmeras, sonar, velocímetro, GPS, odômetro, sensores de condições do veículo

Fonte: Traduzido de (RUSSELL; NORVIG, 2010)

3.2 ARQUITETURA DE AGENTES RACIONAIS

A principal preocupação ao criar um agente é assegurar que o mesmo seja capaz de decidir qual ação executar para melhor satisfazer seus objetivos (WOOLDRIDGE, 2009). Todas as ações que um agente pode realizar devem ser definidas durante sua criação, disponibilizadas em um repertório de ações e requerer uma condição do estado interno do agente para serem executadas (WOOLDRIDGE, 2009). Logo, seu design representa o ponto de vista do projetista sobre a forma que as tarefas para as quais o agente foi planejado serão executadas (FERBER,

1999). Portanto, ao desenvolver um agente é necessário adotar uma metodologia capaz de especificar a interação entre o agente e seu ambiente e como suas ações são executadas. Este conceito é denominado *arquitetura de agentes*.

Assim como a definição do termo agente, não há um comum acordo sobre as arquiteturas de agentes. No entanto, aqui será utilizada a arquitetura de agentes racionais apresentada em Wooldridge (2000) e (2009).

Um agente é considerado racional quando seu processo de tomada de decisão sempre escolhe o melhor plano de ações para atender seus interesses internos, baseado nas informações conhecidas sobre o ambiente onde está inserido (WOOLDRIDGE, 2009). Logo, o agente racional possui razões explícitas para tomar suas decisões, sendo assim capaz de explicá-las se necessário (DENNIS; FISHER, 2016).

O raciocínio prático consiste em duas tarefas distintas a deliberação e o raciocínio meio-e-fim. A deliberação têm como objetivo definir **quais** as intenções[1] que o agente compromete-se em alcançar, após considerar diferentes opções e crenças[2], (WOOLDRIDGE, 2000). Já o raciocínio meio-e-fim preocupa-se em **como** o agente irá alcançar tais intenções.

Para entender estas atividades, considere o seguinte exemplo adaptado de Wooldridge (2009) : "No final de sua graduação, um aluno tem que decidir se vai seguir a carreira acadêmica ou trabalhar na indústria. Após avaliar suas opções, este aluno decide que quer se tornar um pesquisador. Para isso, o aluno deverá cursar um mestrado (e futuramente um doutorado). Porém, antes de cursar o mestrado, o aluno deve enviar sua proposta de pesquisa, cartas de recomendações, entre outros, para a universidade de sua escolha". O processo de decidir a carreira é a *deliberação* sobre as opções e a definição de qual intenção o aluno compromete-se a alcançar. Já a escolha de qual curso de mestrado matricular-se e o processo de inscrição é o *raciocínio meio-e-fim*, que tem como objetivo criar um planejamento de como alcançar a intenção que o aluno deseja.

Para Wooldridge e Rao (1999), outras características importantes do agente racional são: (i) equilíbrio entre comportamento reativo e proativo; (ii) equilíbrio dos recursos utilizadas para percepção do ambiente, deliberação e ação; (iii) equilíbrio entre interesses internos do agente e interesses de outros agentes presentes no mesmo ambiente.

O agente desenvolvido por este trabalho, que será apresentado no Capítulo 5 considera os itens citados anteriormente da seguinte forma:

(i) O agente está ativando buscando atingir seu objetivo geral, no entanto, quando determinados eventos ocorrem no ambiente, o mesmo suspende suas atividades para atender aquela situação;

(ii) Não será compreendido dentro do escopo deste trabalho;

[1] Objetivo(s) que o agente quer atingir.
[2] Base de conhecimento que o agente possui sobre si próprio e seu ambiente.

(iii) Assim como o item (ii), este não será abordado por este trabalho.

Agentes BDI

O modelo BDI é baseado no raciocínio prático, onde o agente irá tomar sua decisão de acordo com os seus estados mentais (WOOLDRIDGE; RAO, 1999). Sua nomenclatura é uma referência aos seus estados mentais aqui utilizados: crenças, desejos e intenções (do inglês *beliefs*, *desires*, *intentions*).

Tais estados mentais são utilizados para prever o comportamento de uma entidade e se relacionam com as condições do ambiente onde esta entidade está inserida (WOOLDRIDGE, 2000). Ao utilizar essa abstração em agentes computacionais permite um melhor entendimento do seu funcionamento interno e como seu processo de raciocínio ocorre.

As crenças de um agente constituem seu conhecimento, representando informações sobre si próprio, outros agentes e o ambiente onde está inserido (DENNIS; FISHER, 2016). Tais crenças são atualizadas de acordo com novas percepções recebidas pelos sensores. Vale ressaltar que, as crenças podem ser incompletas e incorretas (WOOLDRIDGE, 2000). Isto pode ser causado por problemas nos sensores do agente ou uma consequência de um ambiente dinâmico onde mais de uma entidade pode modificá-lo.

Desejos de um agente representam seus objetivos a longo prazo, onde estes objetivos são os estados do ambiente que o agente tem interesse em alcançar (DENNIS; FISHER, 2016). É importante que os desejos de um agente sejam mutuamente consistentes, para que não ocorram conflitos entre as ações que o agente irá realizar (WOOLDRIDGE, 2000).

As intenções representam desejos que o agente está comprometido a alcançar (WOOLDRIDGE, 2000). Dessa forma, representam objetivos imediatos para os quais o agente está dedicando seus recursos para alcançar (DENNIS; FISHER, 2016). Quando um agente adquire uma intenção o mesmo realiza um planejamento de como alcança-la por meio do raciocínio meio-e-fim e, suas ações serão influenciadas pelos planos elaborados. Quando um planejamento falhar em alcançar uma intenção, o agente tentará um novo plano de ações (WOOLDRIDGE, 2000). Portanto, as intenções são o fator principal que levam um agente a agir. Além disso, as intenções devem ser *persistentes* e o agente deve ao menos tentar alcançá-las. Isto significa que, o agente irá manter-se comprometido com uma intenção até o momento em que atingi-la ou concluir que a mesma é inviável. O ato de adotar uma intenção implica que o agente acredita que a mesma é possível de ser atingida, caso contrário, é dito que o agente está sendo irracional, assim caracterizando a inconsistência de intenção-crença (WOOLDRIDGE, 2009).

Considerando o exemplo dado anteriormente na seção 3.2, sobre o aluno no final de sua graduação, suas as crenças são: "Estou prestes a me formar e posso ser pesquisador". Seus desejos são: "Seguir na área acadêmica, mudar-se para o exterior e ter sucesso profissionalmente".

E, sua intenção é: "conseguir um mestrado fora do país".

Como o agente BDI baseia-se no raciocínio prático, os processos de deliberação e raciocínio meio-e-fim estão presentes neste modelo. Em Wooldridge (2000), o processo da deliberação em agentes BDI é caracterizado por duas etapas: (i) geração de opções: responsável por produzir um conjunto de desejos *possíveis*, com base nas crenças e intenções atuais do agente; (ii) filtragem: processo responsável por definir e selecionar os melhores desejos para os quais o agente deve-se comprometer em dado momento.

O ciclo do processo de raciocínio exibido na Figura 7 ocorre da seguinte forma: Os perceptores do agentes recebem informações do ambiente, e atualizam a base de crenças por meio da função de revisão de crenças continuamente. Para cada modificação ocorrida na base de crenças, a função de geração de opções determina novos desejos para o agente, considerando as intenções atuais. A seguir, o filtro irá determinar quando desejos o agente irá se comprometer em alcançar. E por fim, a função de seleção de ação determina quais ações o agente irá executar para alcançar suas intenções atuais.

Figura 7 – Arquitetura de agentes BDI

Entrada
(Sensor)
Função de Revisão
de Crenças
Crenças
Função de Geração
de Opções
Desejos
Filtro
Intenções
Função de Seleção
de Ação
Saída
(Atuador)

Fonte: Traduzido de Nunes *et al.* (2016)

Estratégias de Comprometimento

O comprometimento de um agente com uma intenção não deve desaparecer após esta ser selecionada por meio do processo deliberativo; onde tal propriedade é denominada persis-

tência temporal (WOOLDRIDGE, 2000). Além disso, deve ser definido quão comprometido com tal intenção o agente está e quando esta pode ser abandonada. Através de estratégias de comprometimento, onde é possível determinar quando e como um agente pode desistir de um comprometimento. A estratégia de comprometimento considerada neste trabalho é abordado no Capítulo 5.

Reconsideração de Intenções

De modo geral, um agente deve reconsiderar suas intenções perante as seguintes situações (WOOLDRIDGE, 2000): (i) final da execução de uma lista de ações referentes a intenção atual; (ii) realização das intenções atuais; ou (iii) quando as intenções atuais tornam-se inviáveis de serem alcançadas.

Além disso, diferentes ambientes requerem formas distintas de reconsideração de intenções. Uma mudança no ambiente pode afetar a intenção de um agente, tornando-a impossível de ser atingida ou fazendo com que o agente necessite mudar seus planos para alcança-la.

Em ambientes estáticos, onde o estado do ambiente tende a permanecer o mesmo, não é necessário que o agente reconsidere suas intenções frequentemente, e talvez nenhuma reconsideração se faz necessária. Em ambiente dinâmicos, onde outras entidades estão manipulando o ambiente, é mais indicado que o agente repense suas intenções com periodicamente, de forma que o mesmo seja capaz de reconhecer oportunidades que possam surgir e quando deve abandonar intenções atuais (WOOLDRIDGE, 2009). A forma como o agente implementado por este trabalho reconsidera suas intenções é abordado no Capítulo 5.

3.3 IMPLEMENTAÇÃO DE UM AGENTE RACIONAL

Baseado nas definições na Seção 3.2, é demonstrado no Figura 8 uma implementação básica para um agente racional. É possível observar que em seu funcionamento, o algoritmo:

linha 1: executa repetidamente o seguinte ciclo ordenado de comandos;

linha 2: percebe o ambiente onde o agente está inserido e armazenar a informação recebida em per;

linha 3: atualiza as informações do modelo interno, $modelo_interno$, sobre o ambiente com base em per, utilizando a função $atualizar()$;

linha 4: delibera sobre qual deve ser a próxima intenção I com a função $deliberar()$, baseado nas informações atuais do agente $modelo_interno$;

linha 5: define um planejamento p para alcançar a intenção atual I através de $raciocinio_meio_fim$; e,

linha 6: realiza a função $executar$ para executar as ações definidas no plano per.

Planos, neste contexto, são definidos em Wooldridge (2000) como "... receitas para atingir intenções". . A composição de plano é definida por: pré-condição, que define em quais condições o plano deve ser aplicado; pós-condição, define quais intenções são atingidas com a execução de um plano; e um corpo, constituído por uma sequência de ações que o agente é capaz de realizar (WOOLDRIDGE, 2000).

Figura 8 – Algoritmo básico para um agente racional

1 **enquanto** *verdadeiro* **faça**
2 obtêm próxima percepção per;
3 $modelo_interno \leftarrow atualizar(per)$;
4 $I \leftarrow deliberar(modelo_interno)$;
5 $plano \leftarrow raciocinio_meio_fim(I)$;
6 $executar(plano)$;
7 **fim**

Fonte: Traduzido de Wooldridge (2000)

3.4 IMPLEMENTAÇÃO DE UM AGENTE BDI

A implementação de um agente BDI é similar ao de um agente racional. Entretanto, para construir tal agente é necessário representar explicitamente seus estados mentais. Na Figura 9 é demonstrado uma adaptação do algoritmo apresentado na Figura 8, modelado no contexto do modelo BDI, tal qual em seu funcionamento:

linha 1: o agente atribui o conjunto inicial de crenças B_0 no conjunto de crenças atuais B;

linha 2: na sequência, executa repetidamente o seguinte ciclo ordenado de comandos a seguir;

linha 3: obtem uma nova percepção per do ambiente;

linha 4: atualiza as crenças atuais B, tendo como base B e per atuais por meio da função brf. Esta é a função de revisão de crenças, responsável por atualizar as crenças do agente com base no conjunto de crenças atuais e a percepção que o agente possui sobre o ambiente;

linha 5: delibera, considerando o conjunto de crenças atuais B, por meio de $deliberar(B)$ qual deve ser a intenção atual I;

linha 6: define um planejamento em π com $planejar(B, I)$ para alcançar a intenção atual I com base em B;

linha 7: e, executa a função $executar()$ o plano π para atingir a intenção atual;

Logo, a linha 5 caracteriza o processo da deliberação por meio da função $deliberar()$ e a linha 6, a realização do raciocínio meio-e-fim com a função $planejar()$ presentes no agente racional.

Figura 9 – Algoritmo básico para um agente BDI

```
1 B ← B_0;
2 enquanto verdadeiro faça
3     obtêm próxima percepção per;
4     B ← brf(B, per);
5     I ← deliberar(B);
6     π ← planejar(B, I);
7     executar(π);
8 fim
```

Fonte: Traduzido de Wooldridge (2000)

Contudo, o agente BDI possui algumas especializações não presentes na implementação do agente racional comum, tais como: (i) as informações conhecidas pelo agente BDI são especificadas como crenças. (ii) os agentes BDI possuem desejos; (iii) o agente deve verificar que um planejamento é capaz de alcançar uma intenção; (iv) o processo de deliberação é composto por duas etapas diferentes, a geração de opções e filtragem; (v) e o agente deve ser capaz detectar quando deve abandonar e reconsiderar suas intenções.

O algoritmo da Figura 10 esboça uma possível implementação para um agente BDI, levando em consideração todas as suas propriedades, onde em:

linha 1: o conjunto inicial de crenças B_0 no conjunto de crenças atuais B, e

linha 2: o conjunto atual de intenções I é definido pelo conjunto inicial de intenções I_0.

linha 3: Na sequência executa repetidamente um ciclo ordenado de comandos:

linha 4: obtêm a percepção do ambiente ρ através da função $perceber()$;

linha 5: atualiza as crenças atuais B, tendo como base B e ρ atuais por meio da função $brf()$;

linha 6: cria um novo conjuntos de desejos D através da função $opcoes(B, I)$, que é a função de geração de opções, responsável por determinar novos desejos para o agente, com base nas crenças e intenções atuais;

linha 7: executa a função $filtrar(B, D, I)$ para selecionar uma nova intenção atual I para o agente. É responsável por realizar o processo de deliberação do agente, baseado em seu conjunto de crenças, desejos e intenções atuais;

linha 8: realiza um planejamento π para alcançar a intenção atual I por meio de $planejar(B, I)$.

linha 9: Em seguida, ainda dentro do ciclo definido em na linha 3, o algoritmo inicia um novo ciclo de comandos que tem o propósito de reconsiderar as intenções do agente e verificar se um plano ainda é válido para a situação atual. Tal reconsideração verifica se: há ações no corpo do plano π por meio da função $vazio()$, a intenção atual I foi atingida, $bem_sucedido(I, B)$, ou se a intenção atual I não é mais atingível, $inviavel(I, B)$. Neste novo ciclo, as operações executadas são:

linha 10: seleciona a primeira ação disponível no corpo do plano atual π na função $cabeca(\pi)$ e a atribuir em α;

linha 11: executar por meio da função $executar()$ a ação α selecionada em na linha 10;

linha 12: remove a ação α executada na linha 10 do corpo do plano atual pi, dessa forma, seleciona todas as ações no corpo de π, exceto a primeira com o auxílio da função $extremidade()$;

linha 13: obtêm a percepção do ambiente ρ através da função $perceber()$;

linha 14: atualiza as crenças atuais B, tendo como base B e ρ atuais por meio da função $brf()$;

linha 15: reconsidera a intenção atual com $reconsiderar(I, B)$ para verificar se esta permanece é válida, e

linha 16: caso não seja, é criado um novo conjuntos de desejos D, e

linha 17: é definida uma nova intenção atual;

linha 19: verifica com $sensato(\pi, I, B)$ se plano atual pode atingir a intenção atual, e

linha 20: caso não seja, um novo planejamento é realizado.

linha 22: Por fim, os ciclos iniciados na linha 9, e

linha 23: linha 3 são encerrados.

As funções $opcoes()$ e $filtrar()$ da Figura 10 representam o processo de deliberação do agente BDI, por meio da função de geração de opções e o filtro, respectivamente. Já a função $planejar()$ demonstra o processo de raciocínio meio-e-fim presente no agente BDI.

Figura 10 – Algoritmo para um agente BDI

1 $B \leftarrow B_0$;
2 $I \leftarrow I_0$;
3 **enquanto** $verdadeiro$ **faça**
4 $\quad \rho \leftarrow perceber()$;
5 $\quad B \leftarrow brf(B, \rho)$;
6 $\quad D \leftarrow opcoes(B, I)$;
7 $\quad I \leftarrow filtrar(B, D, I)$;
8 $\quad \pi \leftarrow planejar(B, I)$;
9 $\quad$ **enquanto** ***não*** *(* $vazio(\pi)$ *ou* $bem_sucedido(I, B)$ *ou* $inviavel(I, B)$ *)* **faça**
10 $\quad\quad \alpha \leftarrow cabeca(\pi)$;
11 $\quad\quad executar(\alpha)$;
12 $\quad\quad \pi \leftarrow extremidade(\pi)$;
13 $\quad\quad \rho \leftarrow perceber()$;
14 $\quad\quad B \leftarrow brf(B, \rho)$;
15 $\quad\quad$ **se** $reconsiderar(I, B)$ **então**
16 $\quad\quad\quad D \leftarrow opcoes(B, I)$;
17 $\quad\quad\quad I \leftarrow filtrar(B, D, I)$;
18 $\quad\quad$ **fim**
19 $\quad\quad$ **se** ***não*** $sensato(\pi, I, B)$ **então**
20 $\quad\quad\quad \pi \leftarrow planejar(B, I)$;
21 $\quad\quad$ **fim**
22 $\quad$ **fim**
23 **fim**

Fonte: Traduzido de Wooldridge (2000)

3.5 LINGUAGEM DE PROGRAMAÇÃO PARA AGENTES

O paradigma de linguagens de programação conhecido como Programação Orientada a Agentes (AOP) é utilizado para implementação de agentes, oferecendo uma abstração para a construção e modelagem das propriedades de tais entidades computacionais (SHOHAM, 1993). Atualmente existem diversas linguagens para o desenvolvimento de agentes, onde muitas destas abordam arquiteturas específicas de agentes. Entre as linguagens para o desenvolvimento de agentes racionais utilizando o modelo BDI tem-se: *AgentSpeak* (RAO, 1996), Jason (BORDINI; HUBNER; WOOLDRIDGE, 2007) e Gwendolen (DENNIS; FARWER, 2008). Na sequência será detalhada a linguagem Gwendolen, utilizada no desenvolvimento deste trabalho.

3.6 GWENDOLEN

A linguagem Gwendolen, desenvolvida por Dennis e Farwer (2008), oferece suporte para o desenvolvimento de agentes racionais, especificamente utilizando o modelo BDI, e assim como diversas outras linguagens para agentes BDI, a Gwendolen é derivada do Java (DENNIS; FISHER, 2016). Esta é a primeira linguagem a ter sua implementação baseada no *AIL* (*Agent Infrastructure Layer*, ou Camada de Infraestrutura de Agentes), uma coleção de classes Java voltada para a utilização de *model checking*[3] em agentes (DENNIS; FISHER, 2016). Dessa forma, um dos principais propósitos da Gwendolen é permitir a utilização do *AJPF* [4] para a verificação formal dos agentes.

O funcionamento de um agente Gwendolen é demonstrado por um *ciclo de raciocínio* (DENNIS; FISHER, 2016). Este processo demonstrado na Figura 11 é composto por diferentes estágios, onde cada estágio é responsável por um aspecto das propriedades desse agente. As fases do funcionamento do agente Gwendolen são discutidas a seguir:

Figura 11 – Ciclo de Raciocínio de um agente em Gwendolen

Fonte: Traduzido de (DENNIS; FISHER, 2016)

Fase A: Esta é a fase inicial do agente em Gwendolen, responsável por selecionar uma das intenções do agente para ser a intenção atual e descartar intenções vazias. Essa intenção selecionada não pode estar suspensa. Caso não seja possível escolher uma intenção,

[3] Ver mais na Seção 4.3 *Model Checking*.
[4] Ver mais na seção 4.6 AJPF.

o agente irá hibernar (i.e., interromper suas atividades) e ficará aguardando alguma intenção ser ativada novamente.

Fase B: Na fase B é o momento onde o agente irá procurar todos os planos possíveis para atingir a intenção atual. Caso não haja um plano disponível, a intenção é descartada.

Fase C: Nesta fase, um dos planos encontrados na fase B é selecionado e vinculado com a intenção.

Fase D: Nesse momento, a primeira tarefa listada pelo plano vinculado a intenção atual é executada.

Fase E: Na fase E, o agente recebe novas percepções sobre o ambiente e novas mensagens[5].

Fase F: Durante esta etapa, o agente processa novas percepções e mensagens. Para cada nova percepção do agente, uma intenção é criada para adicionar uma crença correspondente. Da mesma forma, para cada percepção que está presente na base de crenças e não foi detectada novamente, uma intenção é criada para remover sua crença corresponde. E por fim, cada mensagem é convertida para uma nova intenção.

Em um comparativo com a definição do modelo BDI apresentado na subseção 3.2, a fase A representa o processo de deliberação do agente, logo, a função de geração de opções e o processo de filtragem de intenções. As fases B e C representam o raciocino meio-e-fim do agente. A função de revisão de crenças ocorre durante as fases E e F.

3.6.1 Propriedades e Sintaxe da Linguagem Gwendolen

A seguir, serão definidas as propriedades da linguagem Gwendolen e suas respectivas sintaxes.

- Ações: a sintaxe em Gwendolen define uma ação como um predicado α. Uma ação pode ser somente executada pelo agente e não pode ser adicionada nem removida.
- Crenças: uma crença pode ser adquirida por percepções sobre o ambiente e por meio da deliberação interna do agente. A sintaxe representa uma crença como b e as operações em uma crença são: $+b$, para adicionar b, e $-b$, para remover b.
- Eventos: são fenômenos ocorridos no sistema do agente para os quais o mesmo poderá reagir (DENNIS; FISHER, 2016). Entre os exemplos dessas ocorrências estão mudanças em crenças (*e.g.*, adição ou remoção de uma crença) e comprometimento a novos objetivos (*e.g.*, um plano do agente adiciona um novo objetivo).

[5] Mensagens são utilizadas para a comunicação entre agentes em um SMA. Como SMAs não estão no escopo deste trabalho, a comunicação em Gwendolen não será abordada.

- Unificador (*unifier*): são variáveis do sistema do agente utilizadas para abstrair grupos de objetos semelhantes.
- Tarefas (*deeds*): uma tarefa pode ser uma ação a ser realizada pelo agente, comprometimento ou descarte de um objetivo, adição ou remoção de uma crença, ou até mesmo a suspensão de uma intenção (DENNIS; FISHER, 2016).
- Intenções: esta é a estrutura de dados mais complexa para se representar em Gwendolen, utilizada para armazenar a forma como se pretende atingir um objetivo (DENNIS; FISHER, 2016). Uma intenção pode ser classificada como:
  - Atual: intenção que está sendo processada atualmente pelo agente;
  - Suspensa: é uma intenção que está aguardando um determinado evento para que o agente possa retomar seu processamento (esperando que um *guard* seja verdadeiro); e,
  - Não vazia: Não há mais nenhuma linha para ser executada na matriz da intenção.

É possível organizar a estrutura de uma intenção como uma matriz com quatro colunas: eventos, *guards*, tarefas e unificadores. Onde cada linha associa uma tarefa ao evento que causou a inserção dessa tarefa na intenção, uma condição para que a tarefa seja executada (*guard*) e um unificador (DENNIS; FISHER, 2016). O processamento de uma intenção é semelhante ao funcionamento da estrutura de dados pilhas, a linha no topo é executada primeiro, e novas linhas são adicionadas no topo. Uma linha é adicionada por tarefas que envolvem o comprometimento com objetivos.

No Quadro 2 é demonstrado a estrutura de uma intenção em Gwendolen. Tal intenção ativada pelo *desejo* de limpar, que é o comprometimento com o objetivo $!limpar()$. Esse comprometimento é o evento de motivo para as duas linhas da intenção. Ao iniciar a execução, o agente irá verificar se o *guard* é verdadeiro para a linha no topo, se sim, a tarefa relacionada a tal linha será executada. Nessa caso, a tarefa é o comprometimento com o objetivo $!irAte(Comodo)$. Isso irá fazer com que novas tarefas relacionadas com esse objetivo sejam adicionados no topo da intenção atual, de forma que essa se comportará como uma pilha. Logo as novas tarefas serão executadas antes do agente executar a tarefa $+!aspirar(Comodo)$, e portanto neste exemplo, serão adicionados antes da primeira linha exibida no quadro 2.

Quadro 2 – Exemplo de estrutura de uma intenção em Gwendolen

evento	***guard***	**tarefa**	**unificador**
$+!limpar()$	$sujo(Comodo)$	$+irAte(Comodo)$	$Comodo = comodo1$
$+!limpar()$	$\top$	$+!aspirar(Comodo)$	$Comodo = comodo1$

Fonte: Traduzido de Dennis e Fisher (2016)

Vale ressaltar que as intenções em Gwendolen não são programadas e fazem parte do processamento e interpretação da linguagem sobre o código do agente.

- Objetivos: são uma forma de representação dos desejos de um agente BDI. Para cada objetivo que um agente adicionar na sua lista de execução, o agente irá procurar um plano que possibilite atingi-los. Em seguida, o plano selecionado será executado.

  Os agentes programados em Gwendolen possuem dois tipos de objetivos. Um deles é o objetivo de execução (*perform goal*), que é descartado pelo agente após o seu plano relacionado ser executado. Logo, o agente não verifica se o plano utilizado foi bem sucedido em alcançar o objetivo (DENNIS; FARWER, 2008).

  O outro tipo de objetivo é o de conquista (*achievement goal*). Quando um plano relacionado a esta categoria de objetivo é executado, o agente verifica se existe uma crença correspondente ao objetivo (ou seja, o objetivo foi atingido) (DENNIS; FISHER, 2016). Se tal crença existir, o agente irá descartar o objetivo. Caso contrário, o agente irá procurar novamente por um plano.

  A sintaxe Gwendolen representa um objetivo como $!g$ e as operações em um objetivo são: $+!g$, para adicionar (comprometer-se a) g, e $-!g$, para descartar g. Para indicar o tipo do objetivo é utilizado um rótulo de identificação, onde $!g[achieve]$, indica que g é um objetivo de conquista, e $!g[perform]$, indica que g é um objetivo de execução.

- Planos: a função de um plano é fornecer uma sequência de tarefas para um agente atingir um objetivo. Existem duas classificações possíveis para um plano, disparado (*triggered*) e não disparado (*untriggered*). Planos disparados são ativados por eventos, onde tais eventos podem ser mudanças em crenças ou quando o agente se compromete a um determinado objetivo. Por outro lado, os planos não disparados dependem somente do estado interno do agente para serem ativados. Em Dennis e Fisher (2016), os componentes de um plano são representados por: *trigger*, *guard*, e corpo. Um *trigger* representa a adição de um objetivo ou de uma crença. O *guard* define sobre quais condições das crenças e objetivos um plano pode ser acionado. No corpo de um plano está contido um lista de tarefas que devem ser executadas pelo agente quando o plano é requisitado. Dessa forma, um plano é acionado por sua *trigger* e irá executar as tarefas do seu corpo se a condição da *guard* for verdadeira. A sintaxe Gwendolen representa um plano da seguinte forma: *trigger* : *guard* $\leftarrow$ corpo.

- Pré-condições (*Guards*): são utilizados para representar pré-condições de planos, as quais devem devem ser verdadeiras para que tal plano seja executado. Onde tais condições podem ser a existência ou não de crenças e objetivos. Dentro de uma pré-condição de um plano em um plano em Gwendolen, é utilizado a letra `B` para identificar uma crença e a letra `G` para identificar um objetivo. *Guards* especificados como $\top$[6] representam uma condição que é sempre verdadeira[7].

- Regras de Raciocínio: são um conjunto de crenças associadas, geralmente estão associadas a pré-condição de de planos (*guards*). São baseadas na lógica da linguagem Prolog

[6] Esse símbolo em lógica representa **Verdadeiro**, também chamado de *top*.
[7] Considerado um *guard* trivial.

e utilizadas em Gwendolen para combinar crenças a serem utilizadas em pré-condições de planos. Por exemplo, na regra `reach(X,Y) :- try_to_reach (X, Y), at(X, Y)`, uma crença `reach(X, Y)` é satisfeita pela existência das crenças `try_to_reach (X, Y)` e `at(X, Y)`, não sendo necessário a adição ou remoção da crença `reach(X, Y)` para que esta seja verdadeira. Nota que, os unificadores utilizadas, `X` e `Y` devem unificar os mesmo valores nas crenças e na regra de raciocínio.

3.6.2 Implementação de um Agente em Gwendolen

Este subseção tem como objetivo definir os componentes necessários para o desenvolvimento de um agente simples em Gwendolen. Para tal, serão discutidos como codificar um agente e a criação de um ambiente onde tal agente estará inserido.

Ambiente

Em Gwendolen, um ambiente é codificado em Java, que é uma característica herdada do AIL. A interação entre agente e ambiente dá-se por meio da execução de ações e percepção de modificações. O ambiente padrão disponibilizado pelo AIL é a classe `DefaultEnvironment` definida em Java e tem como sua extensão a classe `DefaultEnvironmentwRandomness`, que permite a geração de eventos randômicos pelo ambiente.

A classe `DefaultEnvironment` é utilizada pelo Gwendolen e provê um conjunto de funcionalidades tais como o controle de percepções que o agente recebe e a alteração do ambiente por meio de ações do agente. Essas percepções são informações referentes ao estado do ambiente que após inseridas ao agente são interpretadas como crenças. O pacote AIL implementa as seguintes classes para auxiliar na criação de ambientes, `Predicate`, `NumberTerm`, `NumberTermImpl` e `VarTerm`.

A classe `Predicate` é a principal para lidar com fórmulas lógicas no ambiente, utilizada para a especificação de predicados (DENNIS; FISHER, 2016). Para criar um objeto `Predicate` deve-se invocar o método construtor e definir uma variável do tipo `String`, que especifica o 'nome' do predicado, como argumento da função. Como demonstrado na linha 1 do Código 1, o objeto `personagem` especifica o predicado character. É possível adicionar argumentos[8] ao final um predicado por meio da função `addTerm`. É possível observar na linha 2 do código 1 que o `personagem` foi alterado de `character` para `character(man)`, e na linha 3, o predicado passa a ser character(man,tall). Por meio do uso da função setTerm é possível especificar a ordem de um argumento de um predicado. Após a modificação na linha 4, `personagem` passa a ser `character(man,short)`. Para recuperar argumentos de um predicado com a função

[8] Indexados a um predicado a partir de zero.

getTerm(int i) onde i é um índice para um argumento. Logo, o retorno da função executada na linha 6 será man. E por fim, a função getFunctor() retorna o nome de um predicado em uma variável String. Portanto, o retorno da linha 7 será character.

As classes NumberTerm e NumberTermImpl tornam possível o uso de números por um ambiente e seus agentes. NumberTermImpl é utilizado para criar números a partir do ambiente. Para tal, deve-se invocar seu método construtor e definir uma variável do tipo double, que especifica o valor do número que NumberTermImpl representa como argumento da função, como é demonstrado na linha 1 do Código 2. A classe NumberTerm permite a interpretação de um número contido em um predicado por meio da função solve(). É demonstrado nas linhas 3-5 do Código 2 a conversão de um número contido em um predicado em uma variável do tipo double.

Código 1 – Exemplo do uso da Classe Predicate

```
1 Predicate personagem = new Predicate("character");
2 personagem.addTerm(new Predicate("man"));
3 personagem.addTerm(new Predicate("tall"));
4 personagem.setTerm(1, new Predicate("short"));
5
6 personagem.getTerm(0);
7 personagem.getFunctor();
```

Fonte: Autoria Própria

Código 2 – Exemplo do uso das Classes NumberTerm **e** NumberTermImpl

```
1 NumberTermImpl x = new NumberTermImpl(1.5);
2
3 Predicate coordenada = new Predicate("coordinate");
4 coordenada.addTerm(x);
5 NumberTerm varX = (NumberTerm) coordenada.getTerm(0);
6 double realX = varX.solve();
```

Fonte: Autoria Própria

VarTerm é utilizado para a criação de variáveis (também denominado unificadores) que podem ser interpretados por um agente. Ao criar um objeto VarTerm deve-se invocar seu método construtor e definir uma string para especificar o ’nome’ do objeto como argumento da função. É demonstrado na linha 1 do Código 3 a criação do objeto x que especifica a variável X. É possível também criar um predicado que possua uma variável em um de seus argumentos, como demostrado nas linhas 3 e 4 do Código 3.

Código 3 – Exemplo do uso da classe VarTerm

```
1 VarTerm x = new VarTerm("X");
2 Predicate coordenada = new Predicate("coordinate");
```

```
3 coordenada(x);
4 coordenada(new VarTerm("Y"));
```

Fonte: Autoria Própria

Em Dennis e Fisher (2016), são definidas as seguintes funções disponibilizadas pela classe `DefaultEnvironment`:

- **public void addPercept(Predicate per)**: adiciona uma percepção[9] `per` à todos os agentes inseridos no ambiente.
- **public void addPercept(String agName, Predicate per)**: adiciona uma percepção `per` ao agente `agName`.
- **public boolean removePercept(Predicate per)**: remove uma percepção `per` de todos os agentes inseridos no ambiente.
- **public void removePercept(String agName, Predicate per)**: remove uma percepção `per` de um agente `agName`.
- **public boolean removeUnifiesPercept(Predicate per)**: remove quaisquer percepções que possam ser *unificadas* a `per` de todos os agentes inseridos no ambiente. Por exemplo, sejam `localizado_em(1,1)` e `localizado_em(5,2)` percepções de um agente qualquer inserido no ambiente, a percepção `localizado_em(X,Y)` pode ser unificadas a ambas, pois as variáveis `X` e `Y` funcionam com unificadores.
- **public void removeUnifiesPercept(String agName, Predicate per)**: remove quaisquer percepções que possam ser *unificadas* a `per` de um agente `agName`.
- **public Unifier executeAction(String agName, Action act)**: recebe uma variável `agName` com o nome de um agente e a ação `act` que o mesmo deseja realizar sobre o ambiente. Este método é invocado toda fez que um agente executa uma ação. É comum a utilização de declarações condicionais para verificar qual ação foi acionada por um agente. Vale ressaltar que a classe `Action` é uma extensão da classe `Predicate`.

No Código 4 é demonstrado um ambiente simples codificado em Java, que possibilita a movimentação do agente bem como a perceber objetos no na sua posição atual e interagir com os mesmos. Neste exemplo o ambiente `SearchAndRescueEnv`, que estende a classe `DefaultEnvironment`[10] está controlando as percepções do agente. O ambiente configura que um objeto `rubble` está contido nas coordenadas (5,5) (ver linhas 3-4), cria variáveis para manter internamente a posição do agente `robot` (ver linhas 6-7) e que o `robot` inserido no ambiente não está carregando segurando `rubble` (ver linha 9). Entre 11 até 55 é definido a maneira como o

[9] Uma percepção dada pelo ambiente é interpretada como uma crença pelo agente.
[10] Por padrão do AIL, qualquer ambiente criado deve estender o ambiente padrão.

ambiente interpreta as ações executas pelo agente por meio da função executeAction(String agName, Action act). Como este é o exemplo de um ambiente simples que possui somente um agente, portanto, o termo robot sempre irá se referir ao agente agName. No entanto, em casos mais complexos, o ambiente pode vir a manter informações sobre mais de um agente.

Caso a ação realizada pelo agente seja move_to (ver linha 13), o código espera que sejam enviados dois argumentos x e y para identificar a posição que o agente deseja se mover (ver linhas 14-15). O objeto old_pos especifica um predicado at com as variáveis X e Y como unificadores, e em seguida, é utilizado a função removePercept(agName, old_pos) para remover qualquer percepção/crença at do agente agName que possa ser unificada com o objeto old_pos (ver linhas 17-20). A seguir, é criado um objeto at que indica um predicado at, com os argumentos x e y retirados de move_to, que é interpretado pelo agente como sua localização. E na sequência é adicionado como percepção do agente agName por meio da função addPercept(agName, at) (ver linhas 22-25). Nas linhas 27 e 28, o ambiente atualiza sua base de conhecimento com as novas posições do agente. Caso rubble esteja localizada na mesma posição em que o agente se encontra (ver linha 30), uma percepção sobre a localização de rubble na posição atual do agente agName (ver linha 31-35).

Se o agente decidir executar a ação lift_rubble (ver linha 38), o ambiente confere se o agente robot se encontra na mesma posição que rubble e não está o segurando (ver linhas 39-40). Caso verdadeiro, o ambiente remove a percepção rubble, nas coordenadas onde acredita que este se encontra (rubble_x , rubble_y), do agente agName (ver linhas 41-44). A seguir, o ambiente adiciona a percepção holding com um argumento rubble ao agente agName, para que o agente acredite que esteja segurando o rubble encontrado (ver linhas 46-48). E por fim, o ambiente atualiza sua variável interna robot_rubble para que este saiba que o agente robot está segurando rubble.

A partir dessa implementação é possível perceber que o ambiente sempre deve manter informações armazenadas em si referentes aos agentes inseridos nele.

Código 4 – Exemplo de um Ambiente em Gwendolen

```
1  public class SearchAndRescueEnv extends DefaultEnvironment {
2
3     double rubble_x = 5;
4     double rubble_y = 5;
5
6     double robot_x;
7     double robot_y;
8
9     boolean robot_rubble = false;
10
11    public Unifier executeAction(String agName, Action act) throws AILexception {
12
```

```
13    if (act.getFunctor().equals("move_to")) {
14        double x = ((NumberTerm) act.getTerm(0)).solve();
15        double y = ((NumberTerm) act.getTerm(1)).solve();
16
17        Predicate old_pos = new Predicate("at");
18        old_pos.addTerm(new VarTerm("X"));
19        old_pos.addTerm(new VarTerm("Y"));
20        removePercept(agName, old_pos);
21
22        Predicate at = new Predicate("at");
23        at.addTerm(new NumberTermImpl(x));
24        at.addTerm(new NumberTermImpl(y));
25        addPercept(agName, at);
26
27        robot_x = x;
28        robot_y = y;
29
30        if (robot_y == rubble_y && robot_x == rubble_x && !robot_rubble) {
31            Predicate rubble = new Predicate("rubble");
32            rubble.addTerm(new NumberTermImpl(rubble_x));
33            rubble.addTerm(new NumberTermImpl(rubble_y));
34
35            addPercept(agName, rubble);
36        }
37    }
38    else if (act.getFunctor().equals("lift_rubble")) {
39        if (robot_x == rubble_x) {
40            if (robot_y == rubble_y && !robot_rubble) {
41                Predicate rubble = new Predicate("rubble");
42                rubble.addTerm(new NumberTermImpl(rubble_x));
43                rubble.addTerm(new NumberTermImpl(rubble_y));
44                removePercept(agName, rubble);
45
46                Predicate holding = new Predicate("holding");
47                holding.addTerm(new Predicate("rubble"));
48                addPercept(agName, holding);
49                robot_rubble = true;
50            }
51        }
52    }
53    super.executeAction(agName, act);
54 }
```

```
55 }
```

Fonte: Dennis *et al.* (2017)

Agente

A codificação básica de um agente em Gwendolen é exemplificada no código 5, onde tal agente está situado no ambiente `DefaultEnvironment`. Sendo assim, nenhuma ação do agente irá ser interpretada pelo ambiente[11].

Como a Gwendolen é derivada do AIL, é necessário identificar a linguagem que está sendo utilizada (ver linha 1). Deve-se também nomear o agente, neste caso o agente chama-se `hello` (ver linha 3). O conjunto inicial de crenças do agente é definida na seção `Initial Beliefs` (ver linha 5), mas neste exemplo nenhuma crença é definida. A seção `Initial Goals` define os objetivos iniciais do agente (ver linha 7). No exemplo, o agente possui o objetivo de execução (`perform`) `say_hello`[12] (ver linha 8). Os planos do agente são definidos na seção `Plans` (ver linha 10). Neste exemplo o plano do agente `hello` (ver linha 11), é executar a ação `print(hello)` quando o agente se comprometer em realizar o objetivo `say_hello`. Como a condição apresentada *guard* é sempre verdadeira ($\top$), o agente pode executar o plano em na linha 11 a qualquer momento.

Código 5 – Exemplo Básico de um Agente em Gwendolen

```
1  GWENDOLEN
2
3  :name: hello
4
5  : Initial  Beliefs :
6
7  : Initial  Goals:
8  say_hello[perform]
9
10 :Plans:
11 +!say_hello [perform] : { ⊤ } ← print(hello);
```

Fonte: Dennis e Fisher (2016)

No Código 6, o agente está situado em um ambiente apresentado no Código 4, onde este é baseado em coordenadas cartesianas e cada espaço é representado por `(X, Y)`[13]. Existe um `rubble` nas coordenadas `(5, 5)` do ambiente e o agente só poderá perceber isso se estiver

[11] A ação `print` é uma das únicas definidas por `DefaultEnvironment`

[12] Vale lembrar que, como este é um objetivo de execução o agente irá descarta-lo após a conclusão do plano.

[13] `X` e `Y` são unificadores.

nessas mesmas coordenadas. O conjunto de ações do agente é composto por `move_to(X, Y)` e `lift_rubble`. A primeira ação é para mover o agente até as coordenadas `(X, Y)` e atualizar a crença sobre a posição atual do agente no ambiente. Ao realizar a ação `lift_rubble` o agente tentará pegar um `rubble` localizado na sua posição atual, e caso haver algum, irá adicionar uma crença que está segurando algo.

Neste cenário, o agente como é identificado por `robot` (ver linha 3) e nenhuma crença inicial é definida (ver linha 5). O agente possui o objetivo de execução `go_to55`, que indica seu desejo de se mover para as coordenadas `(5, 5)`. Por fim, os planos de `robot` são: (i) quando o agente se comprometer com o objetivo `go_to55`, o agente irá mover-se paras as coordenadas `(5, 5)` do ambiente (ver linha 11); (ii) se a crença sobre a existência de um `rubble` em `(5, 5)` for adicionada, o agente irá executar a ação `lift_rubble` para segurar o cascalho (ver linha 12); e, (iii) quando a crença `holding(rubble)` for adicionada, o agente irá executar a ação `print(done)` (ver linha 13).

Código 6 – Exemplo de um agente robô pegando um cascalho

```
1  GWENDOLEN
2
3  :name: robot
4
5  : Initial  Beliefs :
6
7  : Initial  Goals:
8  go_to55 [perform]
9
10 :Plans:
11 +!go_to55 [perform] : { ⊤ } ← move_to(5,5);
12 +rubble(5,5): { ⊤ } ← lift_rubble;
13 +holding(rubble):  { ⊤ } ← print(done);
```

Fonte: Dennis e Fisher (2016)

O ambiente abordado no cenário do Código 7 é o mesmo discutido no Código 6. Portanto, há um `rubble` nas coordenadas `(5, 5)`. Neste exemplo é incluído a seção `Reasoning Rules` (ver linha 11).

Aqui, o agente possui crenças sobre possíveis localizações do `rubble` nas coordenadas `(1,1)`, `(3,3)` e `(5,5)` (ver linhas 7-9). Na linha 13, a regra `square_to_check(X,Y)` define o seguinte conjunto de crenças: O agente deve crer que possivelmente há um `rubble` em `(X,Y)`, `possible_rubble(X,Y)`, e não deve haver uma crença afirmando que `(X,Y)` não possui nenhum `rubble`, `~no_rubble(X,Y)`[14] . O objetivo `holding(rubble)` em (ver linha 17) é do tipo conquista, indicando que o agente irá executar o plano associado a este objetivo até haver

[14] Na linguagem Gwendolen, o símbolo ∼ é utilizado para indicar *não*, ou seja, uma negação.

uma crença associada a sua conclusão.

Na seção `Plans` é definido que: (i) o plano é associado ao objetivo `holding(rubble)` possui a regra `square_to_check(X,Y)` como *guard* (ver linha 21). Isso significa que o agente irá mover-se (`move_to(X,Y)`) para todas as coordenadas onde acredita-se que possivelmente o `rubble` está (`possible_rubble`) e não está constado que o cascalho não está lá (`~no_rubble`); (ii) se o agente estiver em uma coordenada (`at(X, Y)`) e constatar que o `rubble` não se encontra naquela posição (`~B rubble(X, Y)`)[15], uma crença será adicionada sobre essa inexistência (`no_rubble(X, Y)`) (ver linha 22). Logo, este plano irá atualizar a base de crenças e influenciar o plano na linha 21; (iii) quando o agente estiver em `(X,Y)` (`at(X, Y)`) e perceber um `rubble` na mesma posição (`rubble(X, Y)`), o agente irá executar a ação `lift_rubble` para segura-lo (23); e, (iv) o plano na linha 24 é semelhante ao plano apresentando na linha 13 do Código 6 .

Código 7 – Exemplo avançado de um agente robô pegando um cascalho

```
1  GWENDOLEN
2
3  :name: robot
4
5  : Initial  Beliefs :
6
7  possible_rubble(1, 1)
8  possible_rubble(3, 3)
9  possible_rubble(5, 5)
10
11 :Reasoning Rules:
12
13 square_to_check(X,Y) := possible_rubble(X,Y), ~no_rubble(X,Y);
14
15 : Initial  Goals:
16
17 holding(rubble)[achieve]
18
19 :Plans:
20
21 +!holding(rubble) [achieve] : {B square_to_check(X,Y) } ← move_to(X,Y);
22 +at(X, Y) : {~B rubble(X, Y)} ← +no_rubble(X, Y);
23 +rubble(X, Y): {B at(X, Y)} ← lift_rubble ;
24 +holding(rubble): { ⊤ } ← print(done);
```

Fonte: Dennis e Fisher (2016)

Embora não seja abordado nos Códigos 5, 6 e 7, o Gwendolen implementa uma função

[15] O caracter `B` indica que uma crença é o *guard* de um plano.

para a suspensão temporária de objetivos. Um objetivo é suspenso até que um evento ocorra, geralmente sendo a adição de uma nova crença. Nesses casos, é utilizado o símbolo `*` antes de um predicado para indicar que o resto do objetivo não será concluído até que tal evento ocorra. Por exemplo, ao modificar a linha 21 do código 7 para: `+!holding(rubble) [achieve]: {B square_to_check(X,Y)} ← move_to(X,Y), *at(X,Y)`, o símbolo `*` implica que o objetivo espera que `at(X,Y)` seja verdadeiro para prosseguir com sua execução.

Arquivos de Configuração

Para a execução de um agente Gwendolen são necessários três arquivos: o arquivo `.gwen` que especifica o agente em si, o ambiente em Java (quando não se utiliza o ambiente padrão do Gwendolen (`DefaultEnvironment`) e um arquivo `.ail` que define as configurações básicas para sua execução. O conteúdo de um arquivo `.gwen` foi explorado nos códigos 5, 6 e 7, enquanto um exemplo de ambiente em Java é demonstrado no código 4. O arquivo AIL, mostrado no código 8, é composto por quatro elementos essenciais (DENNIS; FISHER, 2016):

- **mas.file**: especifica o caminho para o arquivo `.gwen`, contendo a programação do agente;
- **mas.builder**: identifica a classe Java que será responsável por compilar e executar os agentes e o ambiente. Esta linha está presente em qualquer arquivo de configuração de um agente programado em Gwendolen;
- **env**: especifica o caminho para o ambiente em qual o agente está inserido, podendo ser um ambiente Java específico ou o ambiente padrão.

Código 8 – Exemplo de um arquivo AIL

```
1 mas.file = "Caminho para o arquivo do agente"
2 mas.builder = gwendolen.GwendolenMASBuilder
3 env = "Caminho para o arquivo do ambiente"
```

Fonte: Adaptado de Dennis e Fisher (2016)

É possível obter informações sobre a execução do agente e saídas do ambiente por meio da adição do seguintes comando no arquivo `.ail`:

- **log.warning = ail.mas.DefaultEnvironment**, exibe somente saídas (*e.g.*, textos) vindos do ambiente; e,
- **log.info = ail.mas.DefaultEnvironment**, exibe saídas do ambiente e demonstra cada ação executada pelo agente.

3.7 RESUMO DO CAPÍTULO

Agentes é uma área relativamente recente dentro da Ciência da Computação, tendo seu primórdio na década de 80 (WOOLDRIDGE, 2009). Logo, ainda há divergências entre autores desse ramo sobre uma definição do termo agente. Aqui neste trabalho, opta-se por utilizar o trabalho de Wooldridge (2009) para definir um agente. A escolha da arquitetura de agentes racionais, particularmente BDI, deve-se ao fato de que esta possui um grande acervo de materiais de consulta, além de contar com diversas linguagens de programação para realizar sua implementação. Vale lembrar que o autor Michael Wooldridge, com auxílio de outros autores, deu origem aos primeiros estudos sobre o agentes racionais em Wooldridge e Rao (1999) e sobre o modelo BDI em (WOOLDRIDGE, 2000). Por fim, a escolha da linguagem Gwendolen deu-se principalmente por esta ser compatível com a ferramenta AJPF, utilizada para verificar formalmente agentes, cujo é o foco deste trabalho. Na sequência deste trabalho, será abordado a verificação formal de agentes racionais com o AJPF, e no Capítulo 5 é apresentado a implementação de um agente na linguagem Gwendolen.

4 VERIFICAÇÃO FORMAL DE AGENTES INTELIGENTES

Assegurar o funcionamento correto de sistemas críticos de segurança e de negócios é imprescindível. Nesses casos, qualquer defeito encontrado pode comprometer a segurança, causar fatalidades, comprometer investimentos de empresas e provocar perdas financeiras (CLARKE JR.; GRUMBERG; PELED, 1999). A seguir, são exemplificadas tais situações e suas graves consequências:

- Em junho de 1996, devido a uma conversão equívoca de variáveis do software, o foguete Ariane-5 explodiu no ar após 36 segundos de seu lançamento (CLARKE JR.; GRUMBERG; PELED, 1999).
- Um erro na divisão de variáveis do tipo *float* causou uma despesa calculada em $475 milhões de dólares no começo dos anos 90 para que a empresa Intel pudesse repor processadores Pentium II defeituosos. (BAIER; KATOEN, 2008).

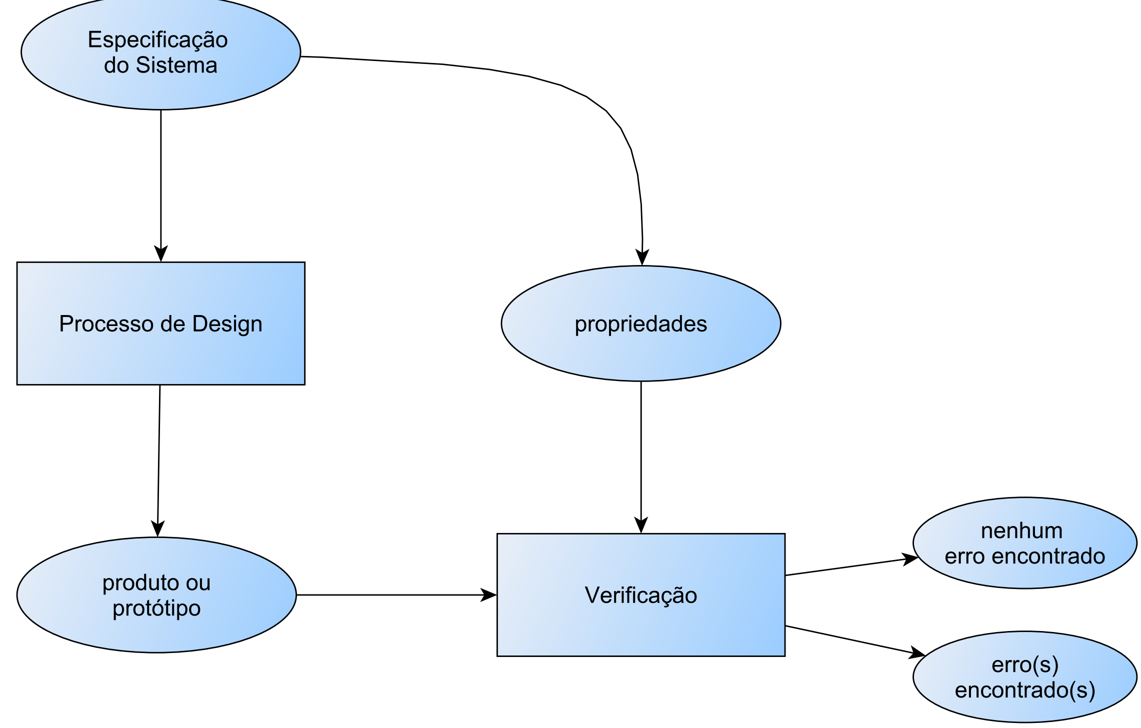
Além desses cenários críticos de atuação, falhas em sistemas utilizados em situações comuns também podem ter consequências graves como má publicidade para fabricante e *recall* do produto (DENNIS; FISHER, 2016). A detecção e conserto de erros ainda no processo de desenvolvimento do produto é até 500 vezes menos custoso que a reparação de falhas encontradas na manutenção (BAIER; KATOEN, 2008). Barendregt (1996) descreve que "é justo dizer que nesta era digital, um sistema de informação funcionando corretamente é mais valioso que ouro". Portanto, é necessário averiguar e assegurar a inexistência de erros no sistema.

A verificação formal é um método aplicado para averiguar se um sistema atende todos os seus requerimentos e especificações por meio de técnicas matemáticas, especificamente provas lógicas (BAIER; KATOEN, 2008). Nesse processo ambos, o sistema e seus requerimentos, são descritos por meio de uma especificação formal para verificar se o sistema atende todos os requerimentos necessários e nenhum é violado. Aqui, todos os cenários são considerados na avaliação do sistema para assegurar que este apresenta um comportamento totalmente correto. Dessa forma, possíveis erros podem vir a ser detectados ainda na fase de desenvolvimento do sistema, evitando problemas na sua implantação. Assim, a corretude do sistema é garantida por meio do rigor matemático (BAIER; KATOEN, 2008).

O esboço da verificação de sistema apresentado por Baier e Katoen (2008) é demonstrado na Figura 12. As especificações do sistemas são descritas para esclarecem quais propriedades são desejadas e quais situações são indesejáveis e não devem ocorrer. Tais especificações servem de base para a verificação do produto (ou seu protótipo) após seu desenvolvimento. É dito que o sistema está *correto* se este possui todas as características previstas e nenhum cenário indevido ocorre. Caso contrário, há um erro no sistema que deve ser consertado.

O software controlador do veículo autônomo é considerado um sistema autônomo, como dito no Capítulo 1. Os principais problemas para a implantação de sistemas autônomos

Figura 12 – Esboço da verificação de um sistema



Fonte: Traduzido de Baier e Katoen (2008)

envolvem a legalização de seu uso, fatores éticos a serem considerados e falta da credibilidade do público (DENNIS; FISHER, 2016). Portanto, entre os motivos para a aplicação da verificação formal em sistemas autônomos estão: esta técnica pode comprovar o comportamento correto e seguro do software, onde esta prova pode ser utilizada como evidência em certificados de qualidade e em regulamentações; e também, aumentar a confiança do público geral sobre a utilização desses sistemas (DENNIS; FISHER, 2016). Dessa forma, a verificação pode demonstrar indicativos de qualidade, bom desempenho e confiabilidade de sistemas.

Como o veículo autônomo irá transportar passageiros, a ocorrência de um erro pode ter consequências fatais. Sendo um dos fatores principais para assegurar a corretude de seu sistema controlador. Durante testes ocorridos nos veículos autônomos do Google entre setembro de 2014 a novembro de 2015, houveram 272 falhas do sistema e outros 13 casos onde o automóvel teria colidido se o motorista não tivesse assumido o controle (HARRIS, 2016). Ressaltando que de acordo com a legislação do governo onde os testes ocorreram, é necessário a presença de um motorista para testes em vias públicas. Outro motivo parte da seguinte premissa: sistemas de informação representam aproximadamente 20% do custo de desenvolvimento de produto em meios de transporte modernos, tais como carros, aviões e trens-bala (BAIER; KATOEN, 2008). Logo, supõe-se que o valor investido por montadores na pesquisa de veículos autônomos seja ainda maior, uma vez que o software controla todo o sistema do automóvel, aumentando ainda mais a gravidade de possíveis erros encontrados.

Vale ressaltar que neste trabalho, o sistema autônomo do veículo é implementado por meio de agentes racionais. No caso da verificação formal de agentes racionais, a investigação de

erros além de se preocupar com o que o agente faz, também se interessa em saber as crenças que motivaram sua tomada de decisão do agente e quais eram suas intenções.

Uma das diferentes categorias de verificação formal é a baseada em modelos. Aqui, utiliza-se um modelo matemático preciso e inequívoco para descrever o comportamento do sistema (BAIER; KATOEN, 2008). Por meio da modelagem do sistema realizada antes da verificação em si, é possível encontrar especificações informais[1] do sistema incompletas, ambíguas e inconsistentes (CLARKE JR.; GRUMBERG; PELED, 1999).

Neste trabalho, para realizar a verificação formal de agentes será utilizado o AJPF, uma extensão do JPF, que tem sua implementação baseada no *model checking program*. Por sua vez, esta técnica é derivada do *model checking*, um método de verificação formal baseada em modelos. Tais conceitos serão abordados nas próximas seções deste capítulo.

4.1 LÓGICA TEMPORAL LINEAR

A Lógica Temporal Linear (ou LTL, *linear temporal logic*) permite a especificação de propriedades temporais em fórmulas matemáticas. Com isso, é possível estabelecer uma sucessão temporal de eventos, tais como frequência de ocorrência e sequências entre ocorrências destes. A sintaxe da LTL é composta pelos operadores da lógica clássica $\neg$ (negação) e $\wedge$ (e) e dois operadores temporais, próximo (*next*, $\bigcirc$) e até que (*until*, $\cup$). Com base no operador $\cup$, são derivados outros dois operadores: o agora, ou em algum momento no futuro será verdadeiro (*eventually*, $\Diamond$) e o sempre será verdadeiro (*always*, $\Box$). Fórmulas de lógica temporal linear são utilizados para especificar propriedades de um sistema no *model checking*.

4.2 AUTÔMATO DE BÜCHI

Autômatos de Büchi não-determinísticos são utilizados em *model checking* para representar fórmulas de lógica linear temporal que especificam propriedades de um sistema (BAIER; KATOEN, 2008). Este se diferencia dos autômatos finitos por aceitar como entrada uma palavra infinita e, consequentemente, seu critério de aceitação de palavras também é distinto (BÜCHI, 1990). Uma palavra é dita aceita se existe um caminho no autômato percorrido por esta sequência que atinja ao menos um dos estados finais infinitamente frequentes.

Os componentes de um autômato de Büchi A são definidos pela tupla A = (Q, $\sum$, Δ, I, F), onde:

- Q representa um conjunto finito de estados de A;
- $\sum$ representa um conjunto finito de elementos do alfabeto de A;

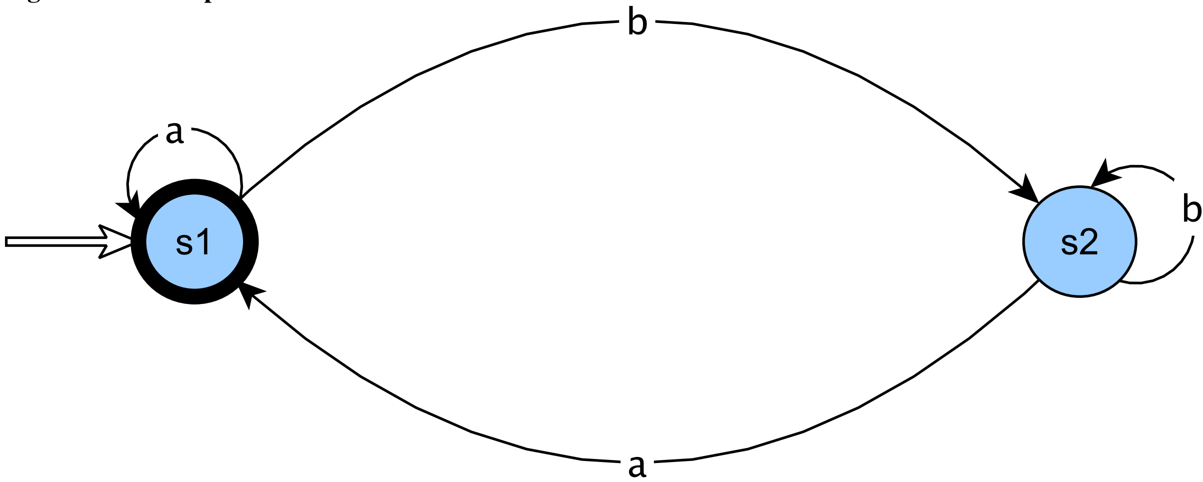
[1] Especificações Informais são os requerimentos do sistemas que não estão descritos na forma lógica.

- Δ representa a função de transição de A, $Q \times \sum \to Q$;
- I representa um conjunto de estados iniciais de A, e $I \subseteq Q$;
- F representa um conjunto de estados finais de A, e $F \subseteq Q$; uma palavra é aceita se seu caminho no autômato percorre ao menos um estado em F infinitamente.

Um autômato de Büchi generalizado é uma variação do autômato de Büchi. Tais autômatos se divergem por seus critérios de aceitação de uma palavra; o primeiro aceita palavras que gerem um caminho que passe por um estado final de cada conjunto de estados finais infinitamente frequente (BAIER; KATOEN, 2008).

Na Figura 13 é descrito um autômato de Büchi B, tal qual aceita palavras que possuam infinitas ocorrências de a. Este autômato é definido por meio da seguinte tupla, B = ($\{s1, s2\}$, $\{a, b\}$, $\{(s1 \times a \to s1), (s1 \times b \to s2), (s2 \times a \to s1), (s2 \times b \to s2)\}$, $\{s1\}$, $\{s1\}$).

Figura 13 – Exemplo de um autômato de Büchi



Fonte: Mukund (1996)

4.3 *MODEL CHECKING*

O *model checking* explora exaustivamente todos os possíveis estados do sistema para realizar sua verificação formal (CLARKE JR.; GRUMBERG; PELED, 1999).

> É uma técnica automatizada, que por meio do modelo de estados finitos de um sistema e uma de suas propriedades formais, é capaz de verificar se uma propriedade é mantida em qualquer estado daquele modelo (BAIER; KATOEN, 2008).

Em Baier e Katoen (2008), são definidas as seguintes três diferentes fases do processamento do *model checking*:

- Modelagem: Aqui, é criado o modelo do sistema com a linguagem de descrição formal utilizada pelo *model checker* (ferramenta que realiza o *model checking*) escolhido. Também

são formalizadas as propriedades por meio de uma linguagem de especificação de propriedades, de forma precisa e a prova de ambiguidade. Consequentemente, esta fase colabora para a descoberta de ambiguidades e inconsistências da documentação informal[2]. É importante que o modelo e a especificação das propriedades sejam validadas. Validação e verificação são termos relacionados, porém, distintos. O processo da validação investiga se a verificação está analisando o sistema correto. Enquanto a verificação averigua a corretude de sistema sendo desenvolvido (BAIER; KATOEN, 2008).

- Execução: Nesta fase, o *model checker* é executado e cada propriedade do sistema é verificada.

- Análise: Esta fase funciona intercalada com a fase de execução. Se uma propriedade é satisfeita, a próxima é verificada. Se não, um contraexemplo é demonstrado, sendo necessário refinar o modelo do sistema ou a propriedade em si. Caso o modelo esteja errado, todas as propriedades devem ser verificadas novamente após sua correção. Se for um erro na especificação de uma propriedade, somente a propriedade deve ser refinada e executada novamente.

Figura 14 – Esboço do *model checking*

Fonte: Traduzido de Baier e Katoen (2008)

[2] Temos como exemplo: a formalização de todas as propriedades de sistema de um subconjunto do protocolo de usuário ISDN relevou que 55% das especificações informais do sistema eram inconsistentes (BAIER; KATOEN, 2008).

Na Figura 14 é exemplificado o processo realizado pelo *model checking*. A primeira tarefa é elaborar os requerimentos do sistema, que descrevem os comportamentos esperados e os estados indesejáveis. Por meio de sua formalização, é possível criar uma representação de suas propriedades para ser verificada com o *model checking*. Em paralelo, deve-se modelar o sistema a ser verificado. A partir do modelo do sistema e da especificações de suas propriedades é possível examiná-lo com o *model checking*. Se nenhuma violação de propriedade for encontrada após a verificação, o *model checker* diz que o sistema está correto. Caso contrário, as propriedades violadas são exibidas com seu respectivo contra-exemplo pelo *model checker*; e com o auxílio de um simulador, é possível rastrear e reproduzir a falha. Baier e Katoen (2008) definem em seu trabalho as seguintes vantagens e desvantagens do *model checking*:

Vantagens:

- Descoberta de todos erros do modelo do sistema;
- Produz contra-exemplos para os erros encontrados;
- Possibilidade de ser utilizada em diversas aplicações.

Desvantagens:

- Não realiza a verificação diretamente no sistema, mas sim em seu modelo. As técnicas baseada em modelos só produzem bons resultados se o modelo estiver correto (BAIER; KATOEN, 2008).;
- Possui o problema da explosão de estados, onde o número de estados produzidos pelo modelo pode exceder as capacidades de hardware disponíveis[3].

Implementação do *Model Checking*

Ao implementar o *model checking* é possível utilizar uma estrutura de estados finitos como um autômato finito para modelar o sistema e, especificar suas propriedades formalmente por meio de fórmulas lógicas (DENNIS; FISHER, 2016). Aqui, cada transição do autômato finito representa uma modificação no estado do sistema. As definições, explicações e exemplos utilizados a seguir são adaptadas de Dennis e Fisher (2016).

Seja S o sistema a ser verificado e o requisito formal R uma propriedade do mesmo. Então, A_S é o autômato que representa o conjunto de todas as execuções possíveis de S, e similarmente, A_R é o conjunto de todas as execuções necessárias para validar R. Como cada autômato produz uma linguagem[4], a função $aceita_por(Automato)$ define a linguagem aceita por um

[3] Existe atualmente diversas técnicas aplicadas por *model checker* para lidar com esse problema. No entanto, esse assunto não será abordado nesse trabalho.

[4] Nesse contexto significa um conjunto de *strings*.

autônomo identificado por $Automato$. Assim para verificar se R é uma propriedade mantida por todo sistema, a proposição $aceita_por(A_S) \subseteq aceita_por(A_R)$ deve ser verdadeira.

Seja N uma especificação formal de uma propriedade do sistema S que **não** deve ocorrer e A_N o autômato finito representando todas as execuções onde N é verdadeira. Para o sistema S estar correto, N deve ser falso. Então, qualquer *string* produzida por A_N não pode ser produzida por A_S. Logo, $aceita_por(A_S) \cap aceita_por(A_N) = \emptyset$. No entanto, não se faz necessário verificar a intersecção das linguagens produzidas por A_S e A_N, basta somente criar um autômato produto $A_S \times A_N$ e checar se nenhuma linguagem é produzida por este. Consequentemente, seja A_{PSN} o autômato produto formado por A_S e A_N, então $A_{PSN} = A_S \times A_N$. Portanto, $aceita_por(A_{PSN}) = aceita_por(A_S) \cap aceita_por(A_N)$. Essa é a abordagem utilizada pela maioria dos *model checkers* (DENNIS; FISHER, 2016). Portanto, um autômato produto A_P de dois autômatos diferentes, A_1 e A_2 irá aceitar linguagens que são simultaneamente válidas em ambos os autômatos, onde $A_P = A_1 \times A_2$. Logo, $aceita_por(A_P) = aceita_por(A_1 \times A_2) = aceita_por(A_1) \cap aceita_por(A_2)$.

Considere o trecho de Código 9 como um sistema denominado `Program`:

Código 9 – Sistema `Program`

```
1  int x = random(0,3); /* Escolhe 0, 1, 2 ou 3 randomicamente */
2  
3  while (x != 1)
4  {
5     if (x > 1) x -= 1;
6     if (x < 1) x += 1;
7  }
```

Fonte: Dennis e Fisher (2016)

O sistema `Program` possui a seguinte propriedade, descrita informalmente: "*Em algum momento no futuro, a variável x terá o valor 1*". Representando formalmente esta propriedade com o uso da lógica temporal, temos: $\Diamond(x = 1)$. Como dito anteriormente nesta seção, a abordagem utilizada por muitos *model checkers* é a criação do autômato produto do modelo do sistema e da representação da propriedade a ser verificada. No entanto, isso só é possível a partir de comportamentos que **não** devem estar presentes no sistema. No caso de propriedades válidas, é necessário *negar* a proposição lógica que as representam. Dessa forma, a negação da propriedade apresentada acima será "*A variável x nunca terá o valor 1*", ou $\Box(x \neq 1)$. Portanto, para verificar se a propriedade $\Diamond(x = 1)$ é obedecida pelo sistema, basta verificar se a sua negação ($\Box(x \neq 1)$) não ocorre.

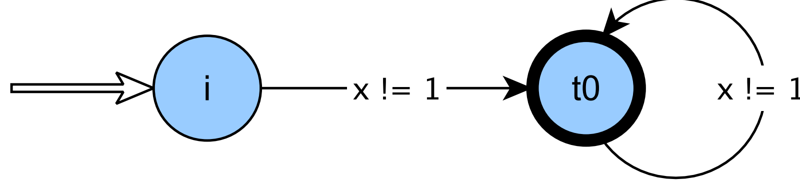
As Figuras 15 e 16 apresentam autônomos finitos representando o modelo do sistema `Program`, chamado $A_{program}$, e a especificação formal $\Box(x \neq 1)$, denominado A_{esp}, respectivamente. O objetivo é garantir que nenhuma sequência de execuções de A_{esp} aconteça em $A_{program}$. Seja $sequencia_de(Automato)$ uma função para encontrar todas as sequências produzidas por

$Automato$. Então, espera-se que $sequencia_de(A_{program}) \cap sequencia_de(A_{esp}) = \emptyset$. O autômato produto A_{PE} formado pela combinação de $A_{program}$ e A_{esp} é demonstrado na Figura 17, onde $A_{PE} = A_{program} \times A_{esp}$. Consequentemente, $sequencia_de(A_{PE}) = sequencia_de(A_{program} \times A_{esp}) = sequencia_de(A_{program}) \cap sequencia_de(A_{esp})$. Como pode ser observado, nenhuma sequência será aceita por esse autônomo A_{PE}, pois não há como atingir um estado final válido. Logo, a especificação formal $\Diamond(x = 1)$ é válida no modelo do sistema `Program`.

Figura 15 – Autômato do sistema `Program`, $A_{program}$

Fonte: Dennis e Fisher (2016)

Figura 16 – Autômato da especificação $\Box(x \neq 1)$, A_{esp}

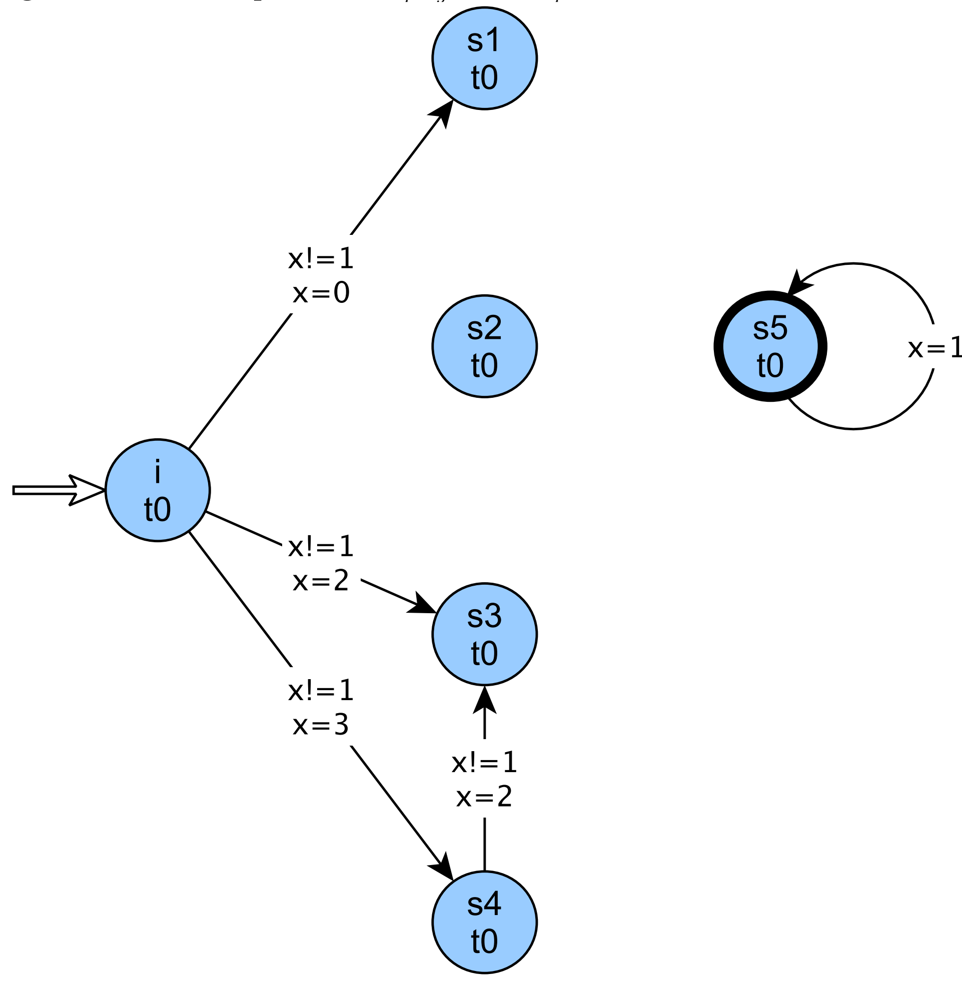


Fonte: Dennis e Fisher (2016)

4.4 ABORDAGEM *ON-THE-FLY*

A construção de autômatos produtos é *custosa* no que diz a respeito dos recursos de espaço e tempo que utiliza (DENNIS; FISHER, 2016). Uma forma de lidar com esse problema

Figura 17 – Autômato produto de $A_{program}$ e A_{esp}, A_{PE}



Fonte: Dennis e Fisher (2016)

é por meio da abordagem *on-the-fly* (BAIER; KATOEN, 2008). Nessa técnica, o sistema de transição do modelo de um sistema S é representado por um autônomo A_M, onde seus estados são explorados simultaneamente com os estados do autônomo A_E, que representa uma especificação formal E, para verificar se estes compartilham alguma sequência que satisfaça E. O processo funciona da seguinte forma: o sistema de transição A_M é executado, onde cada passo é transmitido para A_E. Se, ao final da execução, um caminho válido for realizado em A_E, uma falha foi encontrada e o próprio caminho é o contra-exemplo. Caso nenhuma rota for encontrada, um novo caminho de A_M é percorrido. Se após executar todas as rotas possíveis de A_M e nenhum caminho for encontrado em A_E, é dito que a propriedade E é válida. Outra alternativa é gerar o autômato produto de A_M e A_E e verificar se algum caminho é satisfeito.

Dentro do processo do *model checking*, uma propriedade do sistema especificada através de um fórmula de lógica linear temporal será negada, e a nova formula obtida será traduzida em um autômato de Büchi . O *model checker* irá realizar a verificação formal daquela propriedade a partir do modelo do sistema e do autômato gerado (BAIER; KATOEN, 2008). As linguagens aceitas por autômatos de Büchi são equivalentes as reconhecidas por especificações feitas na lógica temporal linear.

Na Figura 18 é apresentado um diagrama do funcionamento do *model checking* ao utilizar a abordagem *on-the-fly* para verificar uma propriedade de um sistema. Primeiramente é necessário um sistema de transição TS representando o modelo de um sistema e uma especificação em lógica linear temporal φ, tal que φ seja a negação de uma propriedade do sistema. Na sequência, o *model checker* irá traduzir φ em termos de um autômato de Büchi generalizado, e depois irá transforma-lo em um autômato de Büchi $A_{\neg\varphi}$. A partir disso, um autômato produto do sistema de transição TS com $A_{\neg\varphi}$ é criado. Caso haja ao menos um caminho π no autômato produto $TS \otimes A_{\neg\varphi}$, que também satisfaça o autômato A, então a propriedade P é inválida no sistema e um contra-exemplo é demonstrado. Se não, é verificado que a propriedade é verdadeira no sistema, logo, $TS \otimes A_{\neg\varphi} \vDash P_{pers}(A)$, onde P_{pers} refere-se à validade da propriedade.

Figura 18 – *Model Checking*: LTL e Autômato de Büchi

Sistema
Negação da Propriedade
Modelo do Sistema
Fórmula LTL $\neg\varphi$
model checker
Autômato de Büchi Generalizado
Sistema de Transição TS
Autômato de Büchi $\mathcal{A}_{\neg\varphi}$
Autômato Produto
Sistema de Transição
$TS \otimes \mathcal{A}_{\neg\varphi}$
$TS \otimes \mathcal{A} \vDash P_{pers(\mathcal{A})}$
Sim (Propriedade Verdadeira)
Não (Contra-exemplo)

Fonte: Traduzido de Baier e Katoen (2008)

4.5 *MODEL CHECKING PROGRAM*

O *model checking program* é um método derivado do *model checking* que foi desenvolvido para a verificação formal de programas em Java por meio da ferramenta JPF (Java *PathFinder*) (VISSER *et al.*, 2003). Aqui, a análise da corretude através da verificação de estados do

sistema é realizada diretamente na execução do sistema, e não em um modelo correspondente. O *model checking program* utiliza a abordagem *on the fly* para implementar o *model checking*.

Seus aspectos principais são a utilização de uma versão modificada da máquina virtual do Java (JVM, Java *Virtual Machine*) e o monitoramento de *threads* utilizadas por meio de *listeners*[5] (VISSER *et al.*, 2003).

A máquina virtual utilizada pelo JPF implementa o controle da execução do sistema (permitindo a exploração não-determinística da execução) e retroceder para estados anteriores da execução (*backtracking*), gravando as escolhas feitas pelo sistema (DENNIS; FISHER, 2016). Uma de suas desvantagens é que a máquina virtual modificada funciona por cima da máquina virtual padrão do Java, fazendo que o *model checking program* seja mais lento que o *model checking* (VISSER *et al.*, 2003).

Um *listener* está relacionado com um autômato que representa a propriedade sendo verificada. A cada passo da execução do código do sistema, o *listener* tenta reproduzi-los no seu autômato correspondente. Caso uma sequência de passos seja reconhecida no autômato, o *listener* reporta que a propriedade verificada é apresentada no sistema.

4.6 AJPF

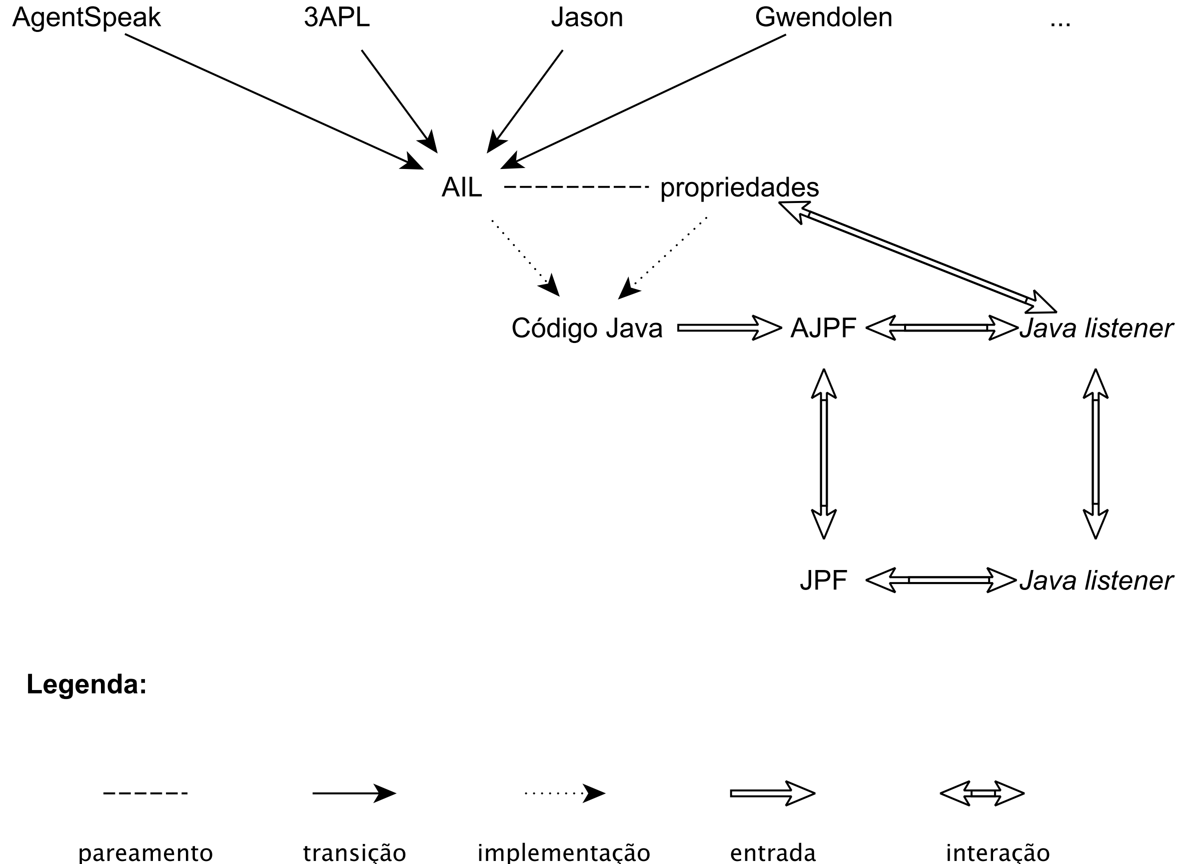
A ferramenta AJPF (*Agent* Java *PathFinder*) foi desenvolvida como parte do framework MCAPL (*Model Checking Agent Program Language*) por Dennis *et al.* (2012) para realizar a verificação formal de agentes. O AJPF é derivado do JPF e foi criado principalmente para ser utilizado em conjunto com APL que utilizam o AIL em sua implementação (BORDINI *et al.*, 2008). Assim como no JPF, a verificação formal é realizada diretamente no código do sistema, no entanto o foco é em verificar agentes racionais. Nessa ferramenta, as especificações formais das propriedades são descritas em uma Linguagem de Especificação de Propriedades baseada na lógica linear temporal (DENNIS *et al.*, 2012).

A arquitetura do AJPF é demonstrada na Figura 19. No processo do AJPF, um sistema desenvolvido em uma das linguagens de programação de agentes (Jason, Gwendolen, entre outras) é interpretado como sua representação correspondente na AIL. Simultaneamente, as propriedades desse sistema são descritas utilizando uma Linguagem de Especificação de Propriedades, e para cada uma destas é criado um *listener* correspondente. Ambos, o sistema e as suas propriedades são implementados em código Java, que é a entrada para a ferramenta AJPF, que atua em uma camada acima da execução do JPF.

A sintaxe utilizada nas especificações formais de propriedades pelo AJPF se baseia na lógica da LTL (DENNIS; FISHER, 2016). No início de cada verificação, é gerado um autômato de Büchi correspondente a negação da propriedade[6].

[5] O *listener* é um recurso em Java que permite que o desenvolvedor monitore eventos de um programa.
[6] O motivo da negação da propriedade é abordado anteriormente em 4.4

Figura 19 – Arquitetura do AJPF



Fonte: Traduzido de Bordini *et al.* (2008)

Durante a verificação de uma propriedade, o AJPF gera uma árvore[7] de estados do ambiente em tempo de execução. Qualquer sequência de nós desta árvore é denominada *caminho*. Todos os caminhos são verificados perante o autômato de Büchi que representa a propriedade. Para cada caminho, o AJPF irá verificar se transições ocorrem autômato da propriedade quando ambos, caminho e autômato são executados paralelamente. Se todos os possíveis caminhos de uma árvore de estados do ambiente atingirem um dos estados finais do autômato de Büchi, é dito que a propriedade sendo verificada é inconsistente. Caso contrário, se houver um caminho que não gere uma transição para um estado final, então a propriedade do agente é verdadeira. Se uma propriedade for verificada como verdadeira **antes** do final da execução do agente, o AJPF encerra a verificação.

4.6.1 Sintaxe das Especificações Formais

Os elementos da sintaxe utilizada para representação de especificações são: (i) ϕ uma representação de fórmula de propriedade a ser verificada; (ii) $_{ag}$ um agente específico dentro do

[7] Em casos onde o ambiente considerado é estático e não randômico, a árvore gerada será degenerada.

sistema; (iii) f uma fórmula atômica de primeira-ordem.

Então, os elementos e operações da sintaxe são os apresentados na definição 1[8].

Definição 1 – Sintaxe Abstrata das Especificações Formais do *AJPF*
$\phi ::= B_{ag}f \mid G_{ag}f \mid A_{ag}f \mid I_{ag}f \mid ID_{ag}f \mid P(f) \mid \phi \vee \phi \mid \phi \wedge \phi \mid \neg\phi \mid \phi\, U\, \phi \mid \phi\, R\, \phi \mid \Diamond\phi \mid \Box\phi$

Onde: (i) B_{ag}, representa uma crença do agente $_{ag}$; (ii) G_{ag}, representa um objetivo do agente $_{ag}$; (iii) A_{ag}, representa uma ação realizada pelo agente $_{ag}$; (iv) I_{ag}, representa uma intenção do agente $_{ag}$; (v) ID_{ag}, representa uma intenção o agente $_{ag}$ de realizar alguma ação; e, (vi) P, representa uma percepção vinda do ambiente.

A codificação das especificações formais que é utilizada para executar o *AJPF* é demonstrada na Definição 2. Note que, a ordem das proposições é respectiva a utilizada na Definição 1.

Definição 2 – Sintaxe Concreta das Especificações Formais do *AJPF*
$\phi ::= \texttt{B}(ag,\ f) \mid \texttt{G}(ag,\ f) \mid \texttt{D}(ag,\ f) \mid \texttt{I}(ag,\ f) \mid \texttt{ItD}(ag,\ f) \mid \texttt{P}(f) \mid \sim\phi$
$\phi' ::= \phi' \mid \phi'\ \texttt{||}\ \phi' \mid \phi'\ \texttt{\&}\ \phi' \mid \phi'\ \texttt{U}\ \phi' \mid \phi'\ \texttt{R}\ \phi' \mid \texttt{<>}\ \phi' \mid \texttt{[]}\ \phi'$

Logo, as equivalências entre a sintaxe e código de especificações é:

- $B_{ag}f$ e $\texttt{B}(ag,\ f)$;
- $G_{ag}f$ e $\texttt{G}(ag,\ f)$;
- $A_{ag}f$ e $\texttt{D}(ag,\ f)$;
- $I_{ag}f$ e $\texttt{I}(ag,\ f)$;
- $ID_{ag}f$ e $\texttt{ItD}(ag,\ f)$;
- $P(f)$ e $\texttt{P}(f)$.

É possível utilizar a regra ϕ `->` ψ para proposição de implicação, um atalho para $\sim\phi$ `||` ψ. E o símbolo _ é utilizado para omitir uma fórmula atômica de primeira-ordem.

4.6.2 Arquivo de Configuração

O arquivo de configuração `.jpf` do AJPF irá executar a verificação sobre um ambiente e seus agentes definidos em um arquivo `.ail`. Para a execução da verificação formal são necessários dois arquivos: o arquivo `.jpf` que especifica a configuração do AJPF e um arquivo `.psl` que define as especificações formais. O arquivo `.jpf`, apresentado no Código 10, é composto por três elementos essenciais do AJPF (DENNIS; FISHER, 2016):

[8] U é equivalente ao símbolo $\cup$ da LTL, e R equivale a $\bigcirc$.

- **@using = mcapl**: Caminho para o diretório MCAPL, presente em todos os arquivos `.jpf`;
- **target = ail.util.AJPF_w_AIL**: Arquivo a partir do qual o `model checking program` é executado, presente em todos os arquivos `.jpf`;
- **target.args =**: composto pelos seguintes três argumentos (separados por vírgula), caminho para arquivo `.ail`, caminho para arquivo `.pls` e identificação da propriedade.

Código 10 – Exemplo de um arquivo `.jpf`

```
@using = mcapl
target = ail.util.AJPF_w_AIL
target.args = "Caminho para o arquivo .ail", "Caminho para o arquivo .psl", " Identificacao
    da especificacao no arquivo .psl"
```

Fonte: Dennis e Fisher (2016)

No Código 11 é demonstrado um arquivo `.psl`, onde existe a seguinte regra $\Diamond A_{vehicle} honk$, e sua identificação é o termo `"1"`.

Código 11 – Exemplo de um arquivo `.psl`

```
1: <> D(vehicle, honk)
```

Fonte: Autoria própria

A cada verificação é criado um autômato de Büchi da negação da propriedade e um sistema de transição dos estados do ambiente e sua interação com o agente inserido em si é gerado em tempo de execução. Ao final do funcionamento do agente, o AJPF afirma se a propriedade é verdadeira ou não.

Dentro das principais opções de log da ferramenta, que servem para esclarecer o funcionamento da verificação, temos:

- **ajpf.product.Product**: passo-a-passo do caminho atual do sistema de transição sendo gerado;
- **ajpf.psl.buchi.BuchiAutomaton**: exibe o autômato de Büchi da negação da propriedade no início da verificação; e,
- **ail.mas.DefaultEnvironment**: exibe as ações executadas pelo agente.

4.7 RESUMO DO CAPÍTULO

Neste capítulo foram explorados os conceitos referentes a verificação formal. Aqui, o foco é explorar a realização do *model checking program* em propriedades de um agente racional.

O uso da verificação formal, mais especificamente o *model checking* em tempo de execução do sistema demonstra um enorme potencial de aplicação na área de agentes, onde o software realiza suas decisões de forma autônoma. Isto é possível com o uso do AJPF, que utiliza fórmulas LTL para especificar as propriedades de um agente, e autômatos de Büchi para verificar tais propriedades na execução do sistema. Para tal, é utilizado a abordagem *on-the-fly* para que a verificação formal seja menos custosa. Adiante, no Capítulo 6 é demonstrada a verificação formal com o uso da ferramenta AJPF.

5 DESENVOLVIMENTO

O foco deste trabalho é o desenvolvimento de um agente capaz de conduzir um veículo na linguagem Gwendolen, determinando suas funcionalidades por meio de especificações e verificá-las formalmente por meio da ferramenta AJPF.

A seguir, as seções deste capítulo irão explorar em mais detalhes o processo criativo para a implementação do agente, e em quais cenários este é capaz de atuar. Na seção 5.1 é apresentado uma visão geral do desenvolvimento do trabalho. Em 5.2, é demonstrado o agente que modela um veículo autônomo, seus planos e os cenários de atuação. Já o ambiente onde tal agente é inserido é abordado em 5.3, onde também são definidos os módulos UTIL e *Simulator*.

5.1 CONTEXTUALIZAÇÃO

O processo de desenvolvimento do agente e a definição de suas especificações é composto por cinco etapas. Em tais estágios, é realizado a implementação de ambos, agente e ambiente, de forma incremental.

Em um primeiro momento, o agente tem como seu único objetivo ativar a buzina do veículo. A simplicidade do agente deve-se ao fator de aprendizado da linguagem utilizada para implementá-lo, que é o foco desta etapa. Aqui, a especificação da funcionalidade do agente determina que é esperado que a ação *buzinar* seja executada em algum momento de sua execução. Tal agente está inserido no ambiente padrão disponibilizado pelo Gwendolen. Portanto, não há resposta do ambiente para ações executadas pelo agente.

Na segunda etapa é definido um ambiente e adicionado ao agente a capacidade de se deslocar. Sua posição é definida por coordenadas `(X,Y)`, considerando que as posições do ambiente são determinadas por uma grade com `N` células. O ambiente é capaz de armazenar internamente a posição do agente em termos de `(X,Y)` e implementa as ações *mover-se ao norte* e *buzinar*. A primeira ação irá deslocar o agente no eixo `Y` da coordenada que representa a posição do agente, enquanto a segunda ação faz com que o veículo emita um som. Aqui, o agente sempre irá possuir as coordenadas iniciais `(0,0)` e terá como objetivo se descolar até a posição `(0,3)`. Note que, o desenvolvimento é feito em um processo incremental e os valores e ações mudam a cada nova etapa.

Na sequência, a terceira etapa realiza um incremento em relação à fase anterior. O agente é capaz de mover-se em quatro direções: norte, sul, leste e oeste. Cabe ao ambiente informar ao agente qual é sua nova posição após o veículo se deslocar a alguma direção. É esperado que o veículo se desloque até o destino do trajeto, definido como uma crença inicial, independentemente de suas direções em relação à posição inicial do agente.

Anterior a versão final, a quarta fase modela o ambiente como uma matriz, onde cada

uma de sua posições são identificadas em termos de coordenadas `(X,Y)`, onde $\{X, Y \in \mathbb{N}| \; 0 \leq X$ e $0 \leq Y\}$. podem assumir Aqui, o agente é capaz de receber múltiplos passageiros e movê-los entre pontos distintos da matriz. Portanto, é esperado que o veículo atenda a todos transeuntes definidos pelo ambiente.

O etapa final de atuação do agente, baseada a partir dos estágios anteriores, será definido a seguir. Aqui, o agente não possui nenhuma crença inicial. O posicionamento dentro do ambiente é representado por uma matriz quadrada A de ordem N, $A_{N \times N} \mid N \in \mathbb{N}$. As células dessa grade são distinguidas por coordenadas `(X,Y)` $|$ `(X,Y)` $\in A$, onde os eixos X e Y são identificados pela expressão $\{X, Y \in \mathbb{N}| \; 0 \leq X < N$ e $0 \leq Y < N\}$. O ambiente impõe barreiras para que se garanta que o agente não consiga sair de dentro deste limite da sua matriz de posições. Qualquer coordenada $(i, \; j)$, onde $\{ \; i, j \in \mathbb{N} \mid i = -1$ ou $i = N$ ou $j = -1$ ou $j = N\}$, é caracteriza como uma barreira da matriz de posições e interpretada como um obstáculo pelo agente. Isto é utilizado para garantir que um agente nunca irá tentar um movimento para uma coordenada $(k, \; l) \mid (k, \; l) \notin A$, onde o eixo X ou Y não pertencem ao intervalo permitido por A. Cada posição pode conter em si um obstáculo. A Figura 20 apresenta uma abstração de uma matriz de posições de um ambiente de ordem 5. Note que nas extremidades superior e lateral esquerda é demonstrado a visão das barreiras impostas à movimentação do agente.

Figura 20 – Matriz de Posições do Ambiente

Fonte: Autoria Própria

No comando de um veículo, o agente modelado tem como sua principal função concluir todas as corridas de passageiros disponibilizadas pelo ambiente, comportando-se como um táxi. Estas corridas são compostas por posições distintas do ambiente denominadas por ponto

de partida e destino, posições nas quais o agente irá estacionar o veículo o embarque e desembarque do passageiro. Para concluir suas corridas, o agente deve se deslocar entre coordenadas, movendo-se nas direções norte e sul (eixo Y, linha da matriz) e leste e oeste (eixo X, coluna da matriz). A movimentação entre duas coordenadas quaisquer dentro do ambiente é denominada trajeto. Na Figura 21 é demonstrado um exemplo da movimentação de um agente em uma matriz de posições de ordem 3, onde:

(a) descreve a posição inicial do agente, coordenadas (1,1);

(b) representa a movimentação do veículo na direção norte da coordenada inicial, e após este deslocamento o agente estará localizado na posição (1,2);

(c) representa a movimentação do veículo na direção sul da coordenada inicial, e após este deslocamento o agente estará localizado na posição (1,0);

(d) representa a movimentação do veículo na direção leste da coordenada inicial, e após este deslocamento o agente estará localizado na posição (2,1); e,

(e) representa a movimentação do veículo na direção oeste da coordenada inicial, e após este deslocamento o agente estará localizado na posição (2,0);

Figura 21 – Representação da Movimentação e Posicionamento do Agente no Ambiente

(a) Posição Inicial (PI)

Legenda: Veículo

(b) Movimento ao Norte de (PI) (b) Movimento ao Sul de (PI) (c) Movimento ao Leste de (PI) (d) Movimento ao Oeste de (PI)

Fonte: Autoria Própria

Se o veículo se deparar com um obstáculo durante seu trajeto, o agente irá tentar realizar as manobras necessárias para evitar o impacto. No entanto, quando o agente se deparar com uma situação onde uma colisão é inevitável, o mesmo tomará decisões para que o nível de dano causado seja o mínimo possível e irá acionar seu plano de emergência. Na seção 5.3.5 serão

abordados outros formas de representações gráficas do ambiente e de sua matriz de posições. Mais detalhes sobre as especificações formais do funcionamento do agente são detalhadas no Capítulo 6, Seção 6.2 Verificação Formal.

Baseado no referencial teórico abordado na Seção 2.1, o agente modelado como veículo autônomo simulado neste trabalho possui um nível 4 de autonomia, pois em nenhum momento há interferência externa em seu processo de tomada de decisão.

A interação entre ambiente e veículo simula a existência de um GPS para que o agente localize sua posição dentro da matriz e determine em quais direções deve se mover para concluir um trajeto. Quando o agente se desloca para uma nova coordenada, é realizado um reconhecimento do ambiente sobre sua posição e as coordenadas imediatamente ao seu redor[1]. Quando um obstáculo é descoberto próximo ao veículo, uma crença é inserida sobre em qual direção o obstáculo está em relação a posição atual do agente. Se obstáculos obstruírem o caminho do veículo, o agente irá adaptar sua rota em outra direção. Portanto, ao decorrer de sua execução, o agente aprende a rota necessária para concluir um trajeto. Caso haja um obstáculo em algum ponto de partida ou destino, o agente irá abandonar aquela corrida e recusar pedidos futuros que requerer que o veículo se mova até tais coordenadas. Por fim, visando obter uma versão gráfica do ambiente e a execução do agente, é utilizado módulo *Simulator*.

Para isso, foram utilizados os seguintes módulos para compor este sistema: Agente (composto pelo agente que modela o comportamento de um veículo autônomo e o ambiente onde atua), Verificação Formal e MCALP. O diagrama de componentes apresentando pela Figura 22 ilustra o relacionamento os elementos deste sistema. Aqui, o agente, `autonomous_car.gwen`, é inserido dentro do ambiente, `AutonomousCarEnv.java`, por meio do arquivo `autonomous_car.ail` e então inicia sua execução. O ambiente por sua vez é responsável por enviar mensagens ao módulo do simulador, para que a visualização gráfica da execução do agente seja possível. Tanto ambiente quanto simulador utilizam classes do módulo UTIL que facilitam sua implementação.

A verificação formal das especificações das propriedades do agente, são definidas no arquivo `autonomous_car.psl`, e são executadas pelo AJPF por meio do arquivo `autonomous_car.jpf`. Ressalta-se que a execução da verificação é independente do simulador, logo, não há nenhuma interação entre o módulo Verificação Formal e o módulo do simulador.

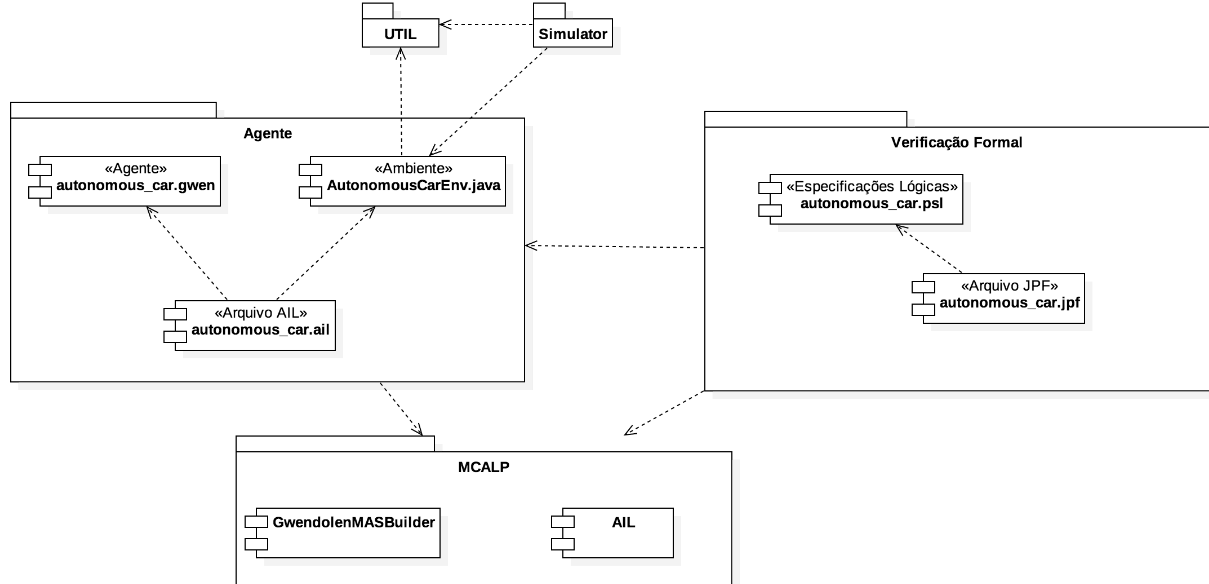
5.1.1 Delimitação do Trabalho

A fim de esclarecer possíveis ambiguidades referentes ao cenário de atuação do agente, certos esclarecimentos são necessários.

1. No início da sua execução, o agente não possui nenhuma crença, e ao decorrer de seu funcionamento, ele irá adquirir crenças referentes ao ambiente onde está inserido;

[1] Coordenadas nas direções norte, sul, leste e oeste.

Figura 22 – Diagrama de Componentes do Sistema



Fonte: Autoria Própria

2. Na implementação atual do trabalho, um conjunto de P passageiros é gerada aleatoriamente **somente** no início da execução do ambiente[2], onde $|P| \in \mathbb{N}$ e cada $c \in P$ representa uma corrida com pontos de partida e destino.

3. Os obstáculos considerados são estáticos e não possuem nenhuma diferença entre si. Embora, em situações onde o desvio de obstáculos é impossível, cada obstáculo irá possuir um nível de dano associado a ele, e esta classificação pode vir a ser diferente de outros obstáculos envolvidos no mesmo cenário. Antes do ambiente ser iniciado, podem ser definidos O obstáculos a serem inseridos aleatoriamente[3] em diferentes coordenadas da matriz de posições, onde $O \in \mathbb{N}$;

4. E por fim, o conceito *depósito* é empregado para representar a indisponibilidade do agente de receber novas corridas.

5.2 AGENTE MODELADO COMO UM VEÍCULO AUTÔNOMO

Esta seção têm como objetivo apresentar a parte do módulo Agente responsável por definir o código do agente autônomo, contidos no arquivo `autonomous_car.gwen`.

Neste trabalho, visando facilitar o entendimento do código do agente, uma nomenclatura é definida para identificar cada um dos planos. A identificação de um plano são as primeiras letras do nome do evento(*trigger*) que o acionou. Por exemplo, o nome de um plano que possui `+!finish_all_rides [achieve]` como evento será o prefixo `FAR`.

[2] Corridas podem ser definidas manualmente.
[3] Assim como corridas, obstáculos também podem ser definidos manualmente.

No caso de planos ativados por uma mesma *trigger*, é utilizado o prefixo e tais planos são diferenciados por uma numeração (1..*). A ordem da numeração é influenciada pela ordem que os planos são definidos no código. Logo, os planos:

- `+!finish_all_rides [achieve] : {~B at(X,Y)} ← localize, *at(LX, LY);` e,
- `+!finish_all_rides [achieve] : {B damaged(high)} ← refuse_ ride (car_ unavailable), +done_all_rides;`

serão identificados como `FAR 1` e `FAR 2`, respectivamente.

Quando planos são semelhantes em sua função (onde sua diferença consiste por ligeiras modificações em suas pré-condições e/ou corpo de tarefas) é adicionado uma nova numeração (1..*) após o prefixo e numeração que os identificam. Dessa forma, os planos:

- `+!finish_all_rides [achieve] : {B pick_up (X, Y), B obstacle (center, X, Y)} ← refuse_ride (pick_up), -ride_info;` e,
- `+!finish_all_rides [achieve] : {B drop_off (X, Y), B obstacle (center, X, Y), ~ B passenger)} ← refuse_ride (drop_off), -ride_info;`

serão identificados como `FAR 3.1` e `FAR 3.2`, respectivamente.

É importante ressaltar que na linguagem Gwendolen, é necessário organizar de forma clara os planos com um mesmo prefixo, pois o primeiro plano que tiver suas pré-condições satisfeitas terá o conjunto de tarefas definidas em seu corpo executado. E a definição de como uma ação é processada pelo ambiente é definido na Seção 5.3.

Serão utilizados fluxogramas para exemplificar melhor o funcionamento de alguns objetivos, onde o foco é proporcionar uma visão de alto nível de cada conjunto de planos do agente. Determinadas conexão possuem a identificação dos seus respectivos planos. A letra *S* é utilizada para abreviar *Sim*, enquanto *N* simboliza *Não*.

Dentro da linguagem Gwendolen, a ordem na qual os planos são implementados no código importa, pois o agente sempre escolhe o primeiro plano que tiver suas pré-condições cumpridas. Como a comunicação entre ambiente e agente não é instantânea, a suspensão de objetivos é necessária em alguns momentos da implementação.

O agente modelado como um veículo autônomo buscar atender uma lista de corridas de de passageiros, trafegando entre pontos de partida e destino ao realizar seus trajetos. Na ocorrência de um obstáculo impedindo o deslocamento do veículo em determinada direção, o agente irá tentar adaptar sua rota para tentar evitar uma colisão e buscará realizar um movimento naquela direção. Em situações onde o cenário imposto pelo ambiente impede o desvio de obstáculos e uma colisão é inevitável, o agente irá escolher pelo impacto que cause o menor dano possível ao

veículo. Há três classificações[4] de danos causados ao veículo, são estes: leve, médio e grave. Para cada um destes, o agente irá utilizar diferentes planos emergenciais para conter aquela situação.

A versão completa da implementação do agente é demonstrada no código 26 que pode ser encontrado no Apêndice A. Note que, o nome *vehicle* é dado para o agente desenvolvido por este trabalho.

A seguir, as subseções Planos Básicos (Seção 5.2.1), Planos para o Desvio de Obstáculos (Seção 5.2.3), Planos de Escolha do Menor Dano Possível em Caso de Colisões (Seção 5.2.4) e Planos de Controle e Recuperação em Casos de Colisões (Seção 5.2.5) agrupam diferentes características responsáveis pela atuação do agente sobre o ambiente. Para cada um destes, é apresentado seus respectivos fragmentos de código, sua função no contexto geral da execução do agente e um diagrama que demonstrando suas relações com outros objetivos do agente.

5.2.1 Planos Básicos

Aqui são definidos os conjuntos de planos básicos para o funcionamento do agente, entre estes: atendimento de passageiros, completar trajetos, limpar informações sobre trajetos anteriores, localização das direções do destino do trajeto movimentação em determinada direção, deslocamento até o destino do trajeto e regras de raciocínio sobre rotas conhecidas.

Conjunto de Planos para Atendimento de Passageiros

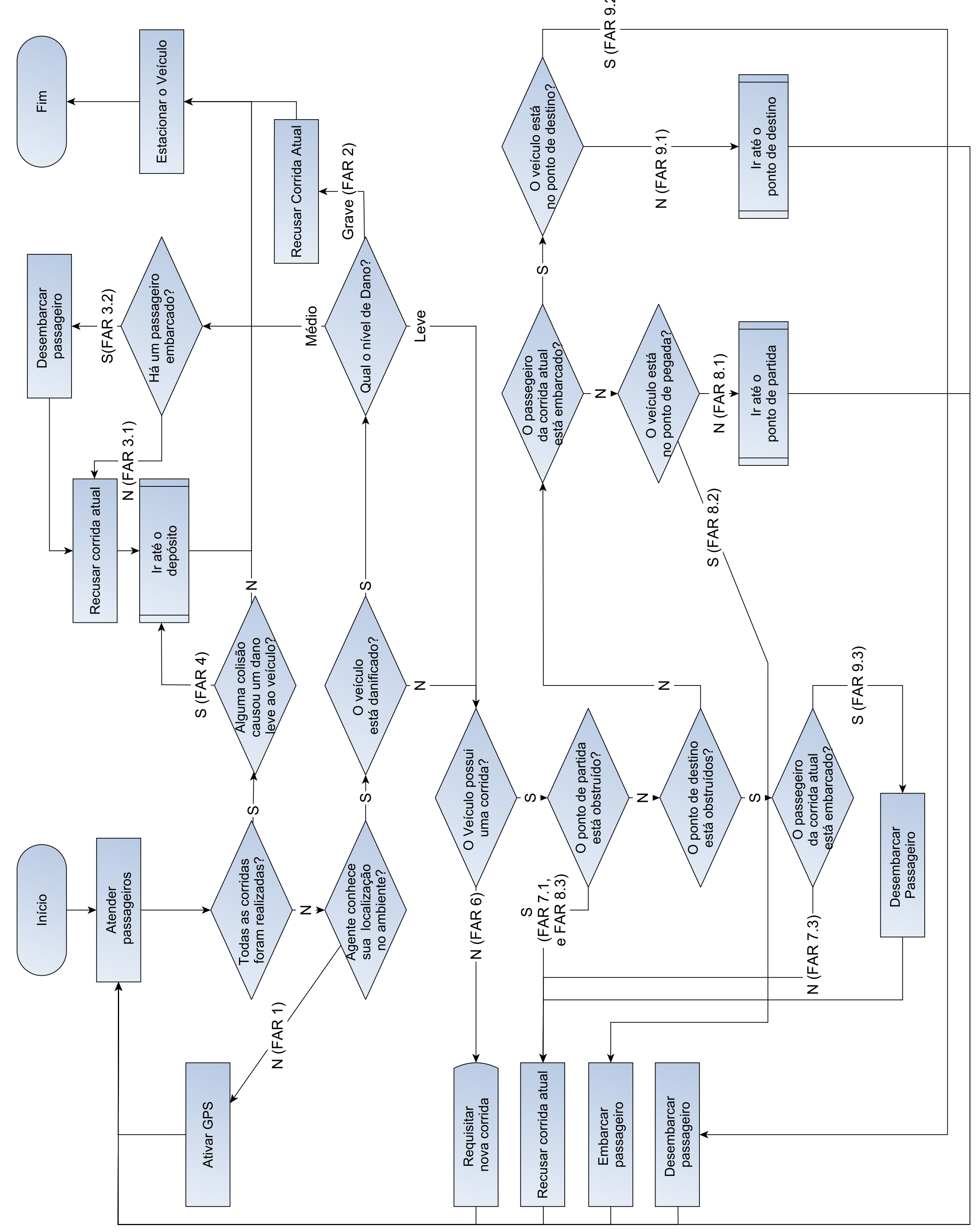
O objetivo inicial `!finish_all_rides` é o principal do agente. Têm como função principal receber informações sobre novos passageiros, acionar os objetivos necessários para que os trajetos aos pontos de partida e destino sejam realizados e decidir quando a execução do agente é finalizada. A representação deste objetivo é demonstrada em um fluxograma da Figura 23.

No Código 12 é demonstrado a primeira parte a implementação deste evento e os possíveis planos de ação. Primeiramente, este objetivo busca pela localização do agente dentro do ambiente com o plano `FAR 1` (ver linha 9), pois o mesmo não possui nenhuma crença inicial. É esperado que o ambiente adicione uma percepção `at(X,Y)`, que informa ao agente os eixos X e Y da sua posição atual[5].

Caso o agente não possua informações sobre um passageiro ou acabou de concluir uma corrida, o plano `FAR 6` (ver linha 11) será executado e este irá requerir uma nova corrida ao ambiente.

[4] Mais detalhes sobre estas categorias são dados em 5.3.3.
[5] Ver mais detalhes sobre a atualização da posição do agente na Subseção 5.3.1.

Figura 23 – Fluxograma do Conjunto de Planos para Atendimento de Passageiros



Fonte: Autoria Própria

Se o agente já souber que nas coordenadas do ponto de partida, `FAR 7.1` (ver linha 16), ou de destino, `FAR 7.2` (ver linha 19), de um passageiro existe um obstáculo, aquela corrida será recusada.

Para chegar nas coordenadas do ponto de partida do passageiro atual, o agente utiliza

`FAR 8.1` (ver linha 23) que irá adicionar um novo objetivo `complete_journey` para tentar completar tal trajeto a partir da sua localização atual. Se for possível realizar o percurso até tal ponto de partida, por meio do plano `FAR 8.2` (ver linha 26), o veículo será estacionado para o embarque do passageiro. Caso não tenha sido possível, `FAR 8.3` (ver linha 29) é selecionado e a corrida atual é recusada e descartada.

Após ter o passageiro embarcado no veículo, por meio de `FAR 9.1` (ver linha 33), o agente irá tentar realizar o trajeto até o ponto de destino. Ao atingir tais coordenadas, o agente irá estacionar e realizar o desembarque do passageiro com `FAR 9.2` (ver linha 36). Se o agente perceber que não será possível chegar no destino da corrida atual, o plano `FAR 9.3` (ver linha 39) irá recusar e descartar tal corrida, e o passageiro desembarca do veículo. Note que há uma semelhança entre os planos com prefixo `FAR 8` e `FAR 9`.

A verificação da conclusão de um trajeto é realizado por meio da regra de raciocínio `reach(X,Y)` (ver linha 2); verifica-se após tentar concluir um percurso, `try_to_reach(X,Y)`, o agente está nas coordenadas de destino do mesmo, `at(X,Y)`.

Código 12 – `!finish_all_rides [achieve]` : 1

```
1  :Reasoning Rules:
2  reach(X,Y) :- try_to_reach(X,Y), at(X,Y);
3
4  : Initial Goals:
5   finish_all_rides [achieve]
6
7  :Plans:
8  /* FAR 1 */
9  +! finish_all_rides [achieve] : {~B at(X,Y)} ← localize, *at(LX, LY);
10
11 /* FAR 6 */
12 +! finish_all_rides [achieve] : {~B ride_info, ~B no_possible_new_ride}
13          ← get_ride, *ride_info;
14
15 /* FAR 7.1 */
16 +! finish_all_rides [achieve] : {B pick_up(X,Y), B obstacle(center, X,Y)}
17          ← refuse_ride(pick_up), −ride_info;
18 /* FAR 7.2 */
19 +! finish_all_rides [achieve] : {B drop_off(X,Y), B obstacle(center, X,Y), ~B passenger}
20          ← refuse_ride(drop_off), −ride_info;
21
22 /* FAR 8.1 */
23 +! finish_all_rides [achieve] : {B pick_up(X,Y), ~B try_to_reach(X,Y), ~B passenger}
24          ← +!complete_journey (X,Y) [perform], +try_to_reach(X,Y);
25 /* FAR 8.2 */
```

```
26 +! finish_all_rides [achieve] : {B pick_up(X,Y), B reach(X,Y), ~B passenger}
27          ← park(pick_up), +passenger, −try_to_reach(X,Y);
28 /* FAR 8.3 */
29 +! finish_all_rides [achieve] : {B pick_up(X,Y), ~B reach(X,Y), ~B passenger}
30          ← refuse_ride(pick_up), −try_to_reach(X,Y), −ride_info;
31
32 /* FAR 9.1 */
33 +! finish_all_rides [achieve] : {B drop_off(X,Y), ~B try_to_reach(X,Y), B passenger}
34          ← +!complete_journey (X,Y) [perform], +try_to_reach(X,Y);
35 /* FAR 9.2 */
36 +! finish_all_rides [achieve] : {B drop_off(X,Y), B reach(X,Y), B passenger}
37          ← park(drop_off),
38          −passenger, −try_to_reach(X,Y), −ride_info;
39 /* FAR 9.3 */
40 +! finish_all_rides [achieve] : {B drop_off(X,Y), ~B reach(X,Y), B passenger}
41          ← refuse_ride(drop_off), −passenger, park(drop_off),
42          −try_to_reach(X,Y), −ride_info;
```

Fonte: Autoria Própria

A segunda parte da implementação de `!finish_all_rides` é apresentada no Código 13. Se a qualquer momento durante um trajeto, o agente obtiver o conhecimento de que as coordenadas do ponto de partida ou de destino de seu passageiro atual está bloqueada por um obstáculo, os objetivos do agente relacionados com a corrida atual serão reconsiderados. Isto é feito pelos planos planos `O 1` e `O 2` (ver linhas 23 e 26, respectivamente), que abandonarão qualquer objetivo atual responsável pela movimentação do veículo[6].

O encerramento da execução do agente pode ocorrer a partir deste objetivo em cinco ocasiões:

1. Caso, durante todos trajetos percorridos o veículo não tenha tida qualquer tipo de colisão e todas as corridas tenham sido atendidas, por meio de `FAR 5` (ver linha 20) o veículo irá irá estacionar e encerrar suas atividades.

2. Se após todas as corridas terem sido atendidas e em algum momento houve uma colisão leve, o agente executa o plano `FAR 4` (ver linha 16) e realiza um trajeto para voltar ao depósito.

3. Na ocasião de uma colisão média e houver um passageiro dentro no veículo, `FAR 3.1` (ver linha 8) é acionado e o agente recusa a corrida atual e estaciona e desembarca o passageiro na coordenada onde ocorreu a colisão. Em seguida, a corrida é recusada e o veículo realiza um trajeto para voltar para o depósito para ser consertado.

[6] Estes planos são `!drive_to (X, Y)` e `!adapt_route (D, X, Y) [achieve]`, que serão discutidos nas Seções 5.2.3 e 5.2.1.

4. Similarmente, em `FAR 3.2` (ver linha 12) é definido que quando há uma colisão média e nenhum passageiro está embarcado no veículo, a corrida atual é recusada e o veículo realiza um trajeto para voltar ao depósito.

5. Se o agente sofreu uma colisão[7] grave em algum momento do último trajeto, por meio de `FAR 2` (ver linha 5) a corrida atual é recusada e o veículo torna-se indisponível para novas corridas, mesmo que ainda houverem passageiros a serem atendidos; isto é interpretado como o veículo está muito danificado para realizar qualquer percurso, e então o agente dever somente estacionar o veículo e parar sua execução.

Nestes casos, a crença `done_all_rides` é adicionada na base de crenças do agente, e de acordo com as regras de raciocínio, o objetivo `finish_all_rides` será satisfeito. Vale ressaltar que os planos de recuperação após uma colisão leve ou média `FAR 3.1`, `FAR 3.2` e `FAR 4` requisitam que o agente retorne ao depósito antes de parar sua totalmente sua execução, adicionando um jornada até sua posição por meio do objetivo `!complete_journey (X,Y)`. Sendo isto interpretado como o veículo estando indisponível para receber novas corridas.

Código 13 – `!finish_all_rides [achieve]` : 2

```
1  :Reasoning Rules:
2  finish_all_rides :- done_all_rides;
3  :Plans:
4  /* FAR 2 */
5  +!finish_all_rides [achieve] : {B damaged(high)}
6          <- refuse_ride(car_unavailable), +done_all_rides;
7  /* FAR 3.1 */
8  +!finish_all_rides [achieve] : {B damaged(moderate), B depot(X,Y), B passenger}
9          <- park(drop_off), -passenger, refuse_ride(drop_off),
10         +!complete_journey (X, Y) [perform], park(depot), +done_all_rides;
11 /* FAR 3.2 */
12 +!finish_all_rides [achieve] : {B damaged(moderate), B depot(X,Y), ~B passenger}
13         <- refuse_ride(pick_up), +!complete_journey (X, Y) [perform],
14         park(depot), +done_all_rides;
15 /* FAR 4 */
16 +!finish_all_rides [achieve] : {B no_possible_new_ride, B damaged(low), B depot(X,Y)}
17         <- print("Going back to Depot to be repaired."),
18         +!complete_journey (X, Y) [perform], park(depot), +done_all_rides;
19 /* FAR 5 */
20 +!finish_all_rides [achieve] : {B no_possible_new_ride, ~B damaged(DAMAGE_LEVEL)}
21         <- park(done_all_rides), +done_all_rides;
22
23 /* O 1 */
```

[7] Ver mais detalhes sobre os níveis de colisão na Subseção 5.2.4.

```
24 +obstacle(center, X, Y) : {B pick_up(X, Y), ~B temp_obstacle(X, Y)}
25         ← −moving, −adapt, −!drive_to(DX, DY) [achieve], −!adapt_route(D,DX,DY)
              [achieve];
26 /* O 2 */
27 +obstacle(center, X, Y) : {B drop_off(X, Y), ~B temp_obstacle(X, Y)}
28         ← −moving, −adapt, −!drive_to(DX, DY) [achieve], −!adapt_route(D,DX,DY)
              [achieve];
```

Fonte: Autoria Própria

Plano para Completar um Trajeto

O objetivo `!complete_journey (X,Y)` é adicionado ao agente para que as tarefas necessárias para a realização de um trajeto sejam executadas. É possível observar o fluxograma do objetivo `!complete_journey (X,Y)` na Figura 24. No Código 14 é exibido o plano que é ativado a partir deste evento, `CJ`, o qual irá adicionar novos objetivos ao agente, `!clear_travel_data` e `!drive_to(X,Y) [achieve]`. Antes do agente iniciar um trajeto, é necessário que crenças sobre trajetos antigos sejam removidas por meio do objetivo `!clear_travel_data` (ver linha 4), para que não haja interferência durante o novo percurso. Sua posição atual é interpretada como o ponto de partida (ver linha 5), enquanto as coordenadas `(X,Y)`, informadas quando este objetivo é adicionado, representam o ponto de destino do trajeto. Em seguida, o objetivo `!drive_to(X,Y) [achieve]` (ver linha 6) é ativado para que a movimentação necessária para realizar o trajeto seja executada. E por fim, as coordenadas do ponto de partida e destino são removidas, para que não haja conflitos com trajetos futuros (ver linha 7).

Figura 24 – Fluxograma do Plano para Completar um Trajeto

Fonte: Autoria Própria

Código 14 – `!complete_journey (X,Y) [perform]`

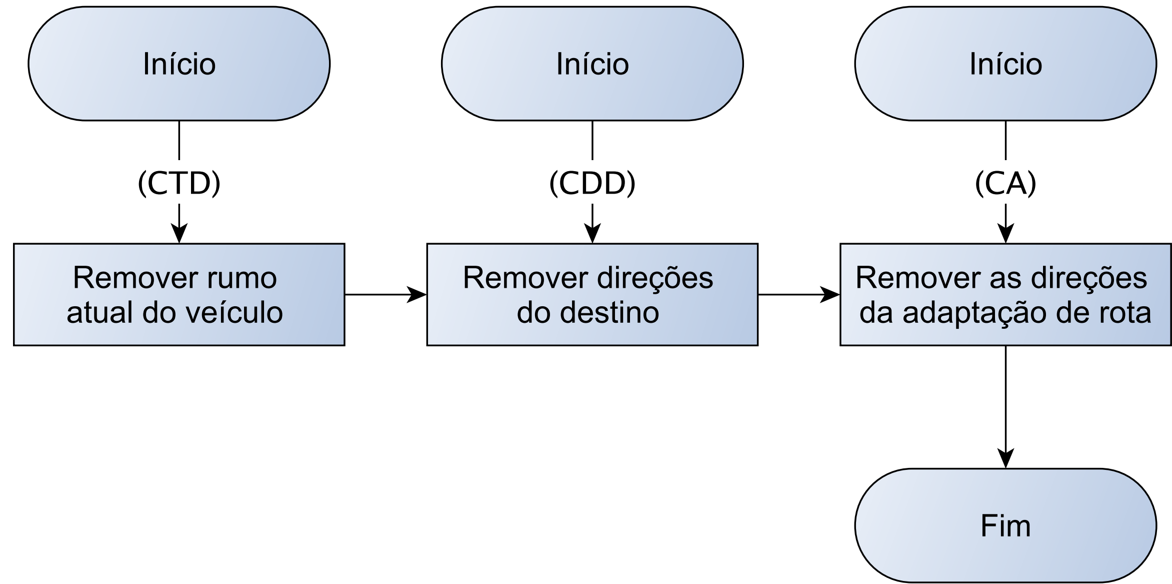
```
1  :Plans:
2  /* CJ */
3  +!complete_journey (X,Y) [perform] : {B at(F_X,F_Y)}
4            ← +!clear_travel_data [perform],
5            +from(F_X,F_Y), +moving,
6            +!drive_to(X,Y) [achieve],
7            −moving, −from(F_X,F_Y);
```

Fonte: Autoria Própria

Conjunto de Planos para Limpar Informações sobre Trajetos Anteriores

Os objetivos `CTD`, `CDD` e `CA`, têm como função remover crenças do agente para evitar conflitos entre percursos ou durante um mesmo trajeto. A implementação desses planos são demonstrados no Código 15, enquanto a Figura 25 ilustra o funcionamento desse conjunto de objetivos. Quando `!clear_travel_data` é acionada, seu único plano é `CTD` (ver linha 3), responsável por remover crenças relacionadas com o sentido que o agente deve se mover no ambiente, e também disparar o objetivo `!clear_direction_data`; que, possui o plano `CDD` (ver linha 6), utilizado para retirar informações referentes ao destino de um trajeto e adicionar o objetivo `!clear_adapt`; este último possui o plano `CA`, encarregado de apagar os dados referentes a adaptação de uma trajeto. Estas crenças relacionados a este conjunto de planos serão explorados no decorrer deste trabalho.

Figura 25 – Fluxograma do Conjunto de Planos para Limpar Informações sobre Trajetos Anteriores



Fonte: Autoria Própria

Código 15 – `!clear`

```
1  :Plans:
2  /* CTD */
3  +!clear_travel_data [perform] : {True} <- +!clear_direction_data [perform],
       -heading(north), -heading(south), -heading(east), -heading(west);
4
5  /* CDD */
6  +!clear_direction_data [perform] : {True}  <- +!clear_adapt [perform], -north, -south,
       -east, -west, -receive_direction;
7
8  /* CA */
9  +!clear_adapt [perform] : {True} <- -adapt(north), -adapt(south), -adapt(east),
       -adapt(west);
```

Fonte: Autoria Própria

Conjunto de Planos para Deslocamento até o Destino do Trajeto

O objetivo `!drive_to(X,Y)` realiza a movimentação necessária para que o agente atinja a posição `(X,Y)`, que é o destino do trajeto atual. O Código 16 demonstra a implementação do planos do evento deste objetivo e as regras de raciocínio envolvidas.

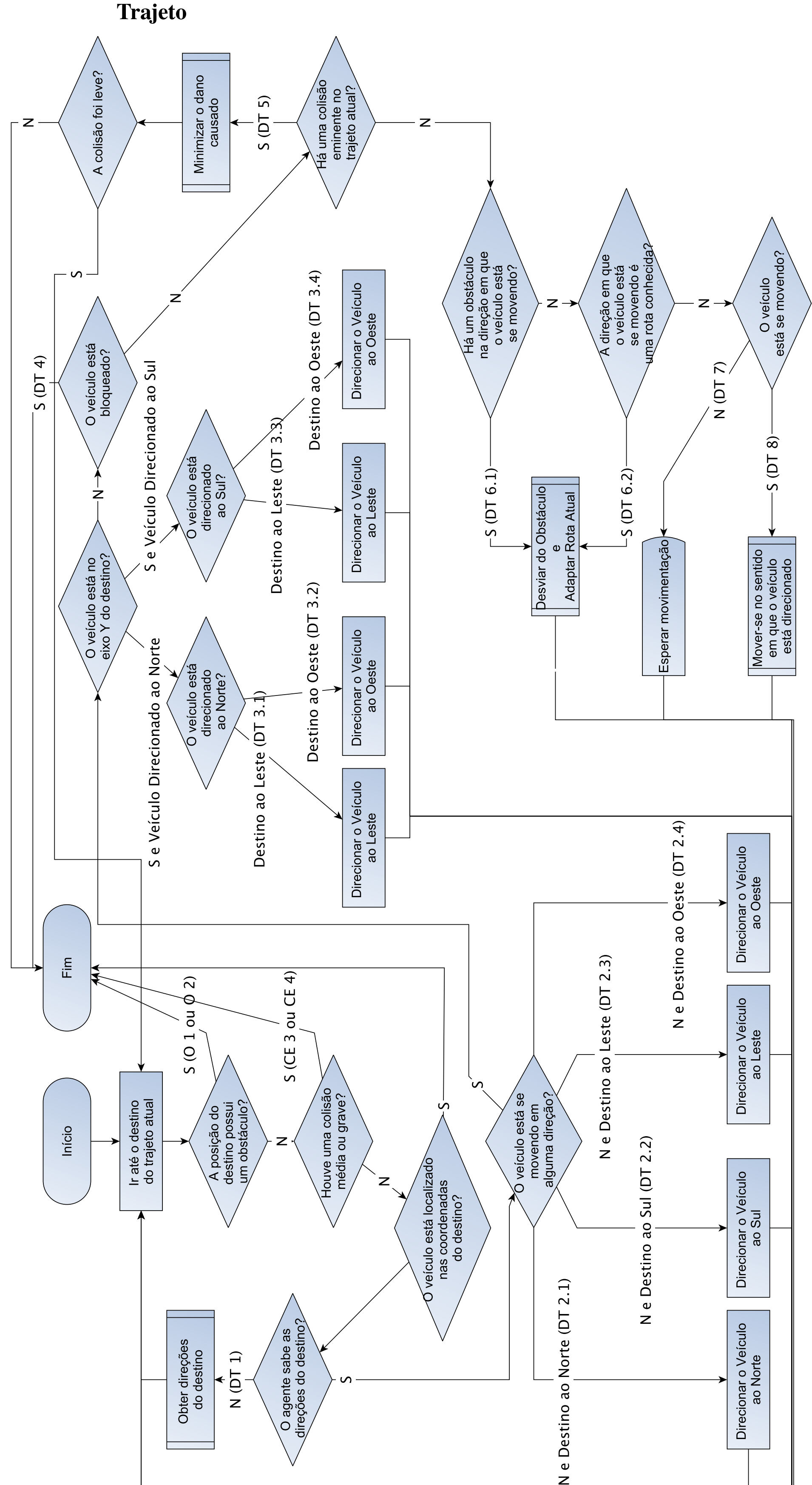
Na Figura 26 é demonstrado o fluxograma referente aos planos para deslocamento do veículo até o destino do trajeto.

É possível que o agente se mova uma posição em cada vez, em ambos os eixos X e Y, onde este deslocamento é classificado em termos de direções, tais que:

- Norte: Acréscimo no eixo Y, especificado pelo predicado `north`;
- Sul: Decréscimo no eixo Y, especificado pelo predicado `south`;
- Leste: Acréscimo no eixo X, especificado pelo predicado `east`;
- Oeste: Decréscimo no eixo X, especificado pelo predicado `west`.

A primeira prioridade deste objetivo é determinar com `DT 1` (ver linha 8) quais direções o agente deve-se mover para atingir as coordenadas do destino atual por meio do objetivo `!get_direction [perform]`. Em seguida, após a obtenção de tais informações e, se o veículo ainda não possuir um rumo, o agente decide uma direção a qual seguir, onde caso o destino se localize na direção: (i) norte, `DT 2.1` é acionado (ver linha 12); (ii) sul, `DT 2.2` é acionado (ver linha 14); (iii) leste, `DT 2.3` é acionado (ver linha 16); ou, (iv) oeste, `DT 2.4` é acionado (ver linha 18).

Figura 26 – Fluxograma de Conjunto Planos para Deslocamento até o Destino do Trajeto



Fonte: Autoria Própria

Vale ressaltar que neste trabalho, optou-se por dar prioridades as direções na vertical (norte e sul) da matriz do ambiente em relação a movimentação na horizontal (leste e oeste). Assim, o agente executa toda a movimentação necessária na vertical antes de se redirecionar na horizontal[8].

Na eventualidade do agente encontrar-se parado, o objetivo `drive_to(X,Y)` é suspenso por `DT 7` (ver linha 48), até que o contrário seja verdadeiro. Porém, quando o veículo está se movendo em uma direção, tal que não há um obstáculo diretamente na sua frente, e não tenha sido utilizada em algum momento para tentar concluir seu trajeto atual[9], o agente irá se deslocar naquela direção acionando o objetivo `!drive_direction(D)[perform]` pelo plano `DT 8` (ver linha 51).

Se o destino do trajeto atual do agente requerer a movimentação em ambos os eixos, e o agente tiver realizado o deslocamento necessário na vertical (eixo `Y`), o agente muda o rumo do agente para a horizontal (eixo `Y`). Este evento é desencadeado quando a coordenada do eixo `Y` da posição atual é igual ao eixo `Y` do destino do trajeto, dessa forma se o agente estiver se vindo no sentido: (i) norte e o destino está na direção leste, `DT 3.1` (ver linha 21) é disparado; (ii) norte e o destino está na direção oeste, `DT 3.2` (ver linha 24) é disparado; (iii) sul e o destino está na direção leste, `DT 3.3` (ver linha 27) é disparado; ou, (iv) sul e o destino está na direção oeste, `DT 3.4` (ver linha 30) é disparado.

Caso não seja possível continuar se movimentando no rumo atual, devido a colisões eminentes com obstáculos, quando `DT 6.1` (ver linha 41) é adicionado, ou quando o deslocamento na direção provou-se ineficaz para atingir o destino do trajeto atual, disparando `DT 6.2` (ver linha 44), o agente para de se movimentar e o objetivo `+!adapt_route(D,X,Y) [achieve]` para que o veículo adapte sua rota e eventualmente se mova naquela direção. A regra `obstacle_ahead(DIRECTION)` (ver linha 4) verifica se há um obstáculo na direção `DIRECTION` em relação a posição atual do agente; e, as regras de raciocínio `known_route`[10] informam ao agente em quais direções, a partir das coordenadas vigentes, já foi efetuada uma tentativa falha de completar o trajeto atual.

Quando em algum momento do percurso, o agente se deparar com um cenário onde uma colisão é inevitável, o objetivo `+!choose_obstacle_collision [perform]` é acionado por `DT 5` (ver linha 37). Porém se o agente estiver cercado por obstáculos em todas as direções possíveis o percurso atual é abandonado pelo plano `DT 4` (ver linha).

Por fim, `drive_to(X,Y)` é satisfeito quando a posição atual do agente são as coordenadas `(X,Y)`, que é o destino do trajeto vigente; e isto é determinado por uma regra de raciocínio homônima a este objetivo.

---

[8] No entanto, podem haver casos onde o destino está estritamente na vertical ou horizontal em relação a posição atual do veículo.

[9] Esta verificação é feita por meio da regra de raciocínio `can_adapt(CA_D,D_X,D_Y)`. Ver mais detalhes em na Subseção 5.2.3.

[10] Ver mais detalhes sobres as regras de raciocínio `known_route` na Subseção 5.2.2.

Código 16 – `!drive_to(X,Y) [achieve]`

```
1  :Reasoning Rules:
2  drive_to(X,Y) :- at(X,Y);
3
4  obstacle_ahead(DIRECTION) :- at(AT_X, AT_Y), obstacle(DIRECTION, AT_X, AT_Y);
5
6  :Plans:
7  /* DT 1 */
8  +!drive_to(X,Y) [achieve] : {~B north, ~B south, ~B east, ~B west}
9            ← +!get_direction [perform];
10
11 /* DT 2.1 */
12 +!drive_to(X,Y) [achieve] : {~B heading(H), B north} ← +heading(north);
13 /* DT 2.2 */
14 +!drive_to(X,Y) [achieve] : {~B heading(H), B south} ← +heading(south);
15 /* DT 2.3 */
16 +!drive_to(X,Y) [achieve] : {~B heading(H), B east} ← +heading(east);
17 /* DT 2.4 */
18 +!drive_to(X,Y) [achieve] : {~B heading(H), B west} ← +heading(west);
19
20 /* DT 3.1 */
21 +!drive_to(X,Y) [achieve] : {B heading(north), B at(AT_X,Y), B east}
22           ← -heading(north), +heading(east);
23 /* DT 3.2 */
24 +!drive_to(X,Y) [achieve] : {B heading(north), B at(AT_X,Y), B west}
25           ← -heading(north), +heading(west);
26 /* DT 3.3 */
27 +!drive_to(X,Y) [achieve] : {B heading(south), B at(AT_X,Y), B east}
28           ← -heading(south), +heading(east);
29 /* DT 3.4 */
30 +!drive_to(X,Y) [achieve] : {B heading(south), B at(AT_X,Y), B west}
31           ← -heading(south), +heading(west);
32
33 /* DT 4 */
34 +!drive_to(X,Y) [achieve] : {B blocked} ← -blocked, -!drive_to(X,Y) [achieve];
35
36 /* DT 5 */
37 +!drive_to(X,Y) [achieve] : {B at(AT_X, AT_Y), B unavoidable_collision(AT_X, AT_Y)}
38           ← -moving, +!choose_obstacle_collision [perform];
39
40 /* DT 6.1 */
41 +!drive_to(X,Y) [achieve] : {B heading(D), B obstacle_ahead(D)}
```

```
42          ← −moving, +adapt, +!adapt_route(D,X,Y) [achieve];
43  /* DT 6.2 */
44  +!drive_to(X,Y) [achieve] : {B heading(D), B known_route(D,X,Y)}
45          ← −moving, +adapt, +!adapt_route(D,X,Y) [achieve];
46
47  /* DT 7 */
48  +!drive_to(X,Y) [achieve] : {~B moving} ← *moving;
49
50  /* DT 8 */
51  +!drive_to(X,Y) [achieve] : {B heading(D), B can_adapt(D,X,Y)}
52          ← +!drive_direction(D) [perform];
```

Fonte: Autoria Própria

Plano para Localização das Direções do Destino do Trajeto

Para definir as direções nas quais o veículo deve se mover para concluir um percurso, o agente adiciona o objetivo `!get_direction [perform]`. Onde, baseado nas coordenadas do destino do trajeto atual informadas pelo objetivo `drive_to(X,Y)` atual, o plano `GD` (ver linha 3) do agente executa a ação `compass(X,Y)` e suspende o objetivo e espera pela resposta do ambiente sobre a localização de seu destino em relação a sua posição atual. A implementação deste evento é apresentada no Código 17, enquanto a Figura 27 ilustra seu funcionamento. Onde o plano `GD` também acionado o objetivo `!clear_travel_data` para garantir que nenhuma informação defasada afete o processo de decisão do agente.

Código 17 – `!get_direction [perform]`

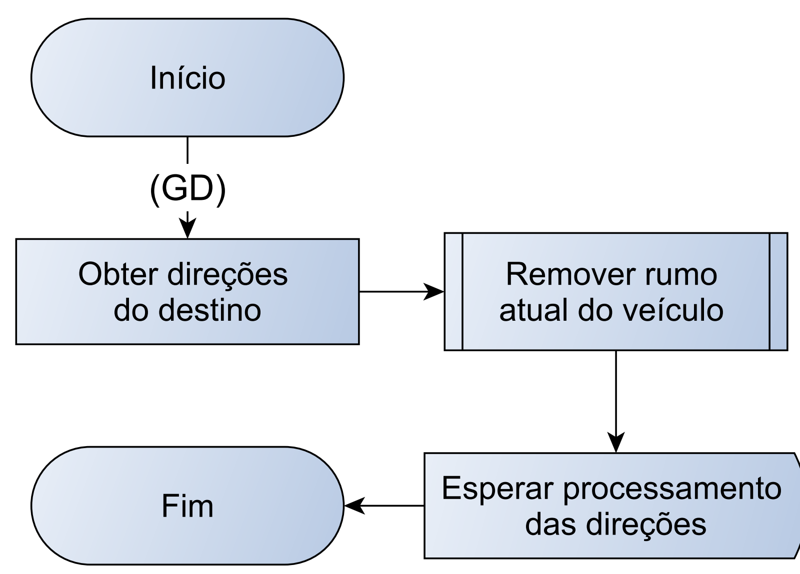
```
1  :Plans:
2  /* GD */
3  +!get_direction [perform] : {G drive_to(X, Y) [achieve]}
4          ← +!clear_travel_data [perform], compass(X, Y), *receive_direction ;
```

Fonte: Autoria Própria

Plano para a Movimentação em uma Direção

Quando o veículo deseja se deslocar de uma posição para outra, o objetivo `!drive_direction (D) [perform]` é adicionado. Como demonstrado no Código 18, para que a movimentação ocorra o plano `DD` (ver linha 3) é acionado, onde o agente executa a ação `drive`,

Figura 27 – Fluxograma do Plano para Localização das Direções do Destino do Trajeto



Fonte: Autoria Própria

que utiliza a posição inicial do trajeto e o destino final para manter um histórico sobre o percurso. O ambiente é responsável por enviar percepções a cada movimento agente referentes as coordenadas ao redor da nova posição[11].

Código 18 – `!drive_ direction (D) [perform]`

```
:Plans:
/* DD */
+!drive_direction(D) [perform] : {G drive_to(X, Y) [achieve], B from(F_X,F_Y)}
        ← −moving, drive(F_X,F_Y, D, X, Y);

/* A 1 */
+at(AT_X,AT_Y) : {~B obstacle(center, AT_X, AT_Y)} ← +moving;
```

Fonte: Autoria Própria

Se não houver nenhum obstáculo na nova posição do agente, a adição da crença `A 1` (ver linha 7) garante que o veículo irá continuar em movimento.

Cada posição pela qual o veículo percorre ao decorrer de um trajeto armazena informações que auxiliam o agente a determinar se uma direção será útil ou não para atingir o destino. Para isso, quando o agente se move a partir de uma coordenada descrita por `(X1,Y1)` em alguma direção `D`, `adapt_from_to(F_X,F_Y, X1,Y1, D, D_X,D_Y)` é inserido na base de crenças do agente, onde as coordenadas `(F_X,F_Y)` e `(D_X,D_Y)` representam respectivamente a posição inicial e o destino do trajeto atual. No momento em que o agente se desloca em alguma direção `D` e atinge uma posição `(X2,Y2)`, a crença `moved_from_to(F_X,F_Y, X2,Y2, D, D_X,D_Y)` é

[11] Ver mais na Subseção 5.3.1.

adicionada. Estas crenças podem ser interpretadas como a movimentação do agente em alguma direção, `adapt_from_to`, e em quais direções o agente chegou em uma mesma coordenada dentro de um mesmo trajeto, `moved_from_to`. O fluxograma deste objetivo é ilustrado na Figura 28, demonstrando os planos derivados dos eventos `!drive_direction(D) [perform]` e `at(AT_X,AT_Y)`.

Figura 28 – Fluxograma do Plano para a Movimentação em uma Direção

Fonte: Autoria Própria

5.2.2 Regras de Raciocínio para Rotas Conhecidas

Com base na lógica das crenças `adapt_from_to` e `moved_from_to`, são definidas as regras de raciocínio `known_route (DIRECTION, KR_X, KR_Y)`, determinando as tentativas falhas de concluir um trajeto. Neste trabalho, tais percursos serão denominados *rotas conhecidas*. Onde estas estabelecem que: se em percurso com ponto inicial em `(F_X,F_Y)` e destino `(KR_X,KR_Y)`, o veículo deslocar-se em uma direção `D1` a partir de uma posição `(X,Y)`, será adicionada a crença `adapt_from_to(F_X, F_Y, X, Y, D1, KR_X, KR_Y)`; e, caso em algum determinado momento durante o mesmo trajeto o agente retorne a tais coordenadas `(X,Y)` enquanto se movia em alguma direção `D2`, oposta a `D1`, haverá uma percepção `moved_from_to (F_X, F_Y, X, Y, D2, KR_X, KR_Y)`; a partir dessas crenças o agente aprende que não deve se deslocar na direção `D1` a partir de `(X,Y)` pois já foi verificado que esta movimentação não é válida pra atingir o destino atual `(KR_X,KR_Y)`. Logo, se o agente estiver localizado em uma coordenada `(X,Y)` e a base de crenças conter `adapt_from_to(F_X, F_Y, X, Y, D1, KR_X, KR_Y)` e `moved_from_to(F_X, F_Y, X, Y, D2, KR_X, KR_Y)`, o veículo não irá se descolar na direção `D1` para tentar concluir o trajeto de `(F_X,F_Y)` até `(KR_X,KR_Y)`.

Isto se faz necessário para que o agente evite movimentos duplicados em um mesmo trajeto. Assim, o agente irá aprender conforme sua movimentação no ambiente, e discernir quais direções a partir da sua posição são inválidas para concluir um trajeto. Este cenário é desencadeado pela adaptação de rotas para que haja o desvio de obstáculos.

Entende-se como direções opostas: norte em relação ao sul, e vice-versa; e, leste em relação ao oeste, e vice-versa. A implementação dessas regras de raciocínio podem ser obser-

vadas no Código 19; onde são demonstradas as regras para verificar se o agente pode se mover nas direções: (i) norte (ver linha 2); (ii) sul (ver linha 6); (iii) leste (ver linha 10); (iv) oeste (ver linha 14).

Código 19 – `known_route (DIRECTION, KR_X, KR_Y)`

```
1  :Reasoning Rules:
2  known_route(north,KR_X,KR_Y) :- at(AT_X,AT_Y), from(F_X,F_Y),
3            adapt_from_to(F_X,F_Y, AT_X,AT_Y, north, KR_X,KR_Y),
4            moved_from_to(F_X,F_Y,AT_X, AT_Y, south, KR_X, KR_Y);
5
6  known_route(south,KR_X,KR_Y) :- at(AT_X,AT_Y), from(F_X,F_Y),
7            adapt_from_to(F_X,F_Y, AT_X,AT_Y, south, KR_X,KR_Y),
8            moved_from_to(F_X,F_Y,AT_X, AT_Y, north, KR_X, KR_Y);
9
10 known_route(east,KR_X,KR_Y) :- at(AT_X,AT_Y), from(F_X,F_Y),
11           adapt_from_to(F_X,F_Y, AT_X,AT_Y, east, KR_X,KR_Y),
12           moved_from_to(F_X,F_Y,AT_X, AT_Y, west, KR_X, KR_Y);
13
14 known_route(west,KR_X,KR_Y) :- at(AT_X,AT_Y), from(F_X,F_Y),
15           adapt_from_to(F_X,F_Y, AT_X,AT_Y, west, KR_X,KR_Y),
16           moved_from_to(F_X,F_Y,AT_X, AT_Y, east, KR_X, KR_Y);
```

Fonte: Autoria Própria

5.2.3 Conjunto de Planos para o Desvio de Obstáculos

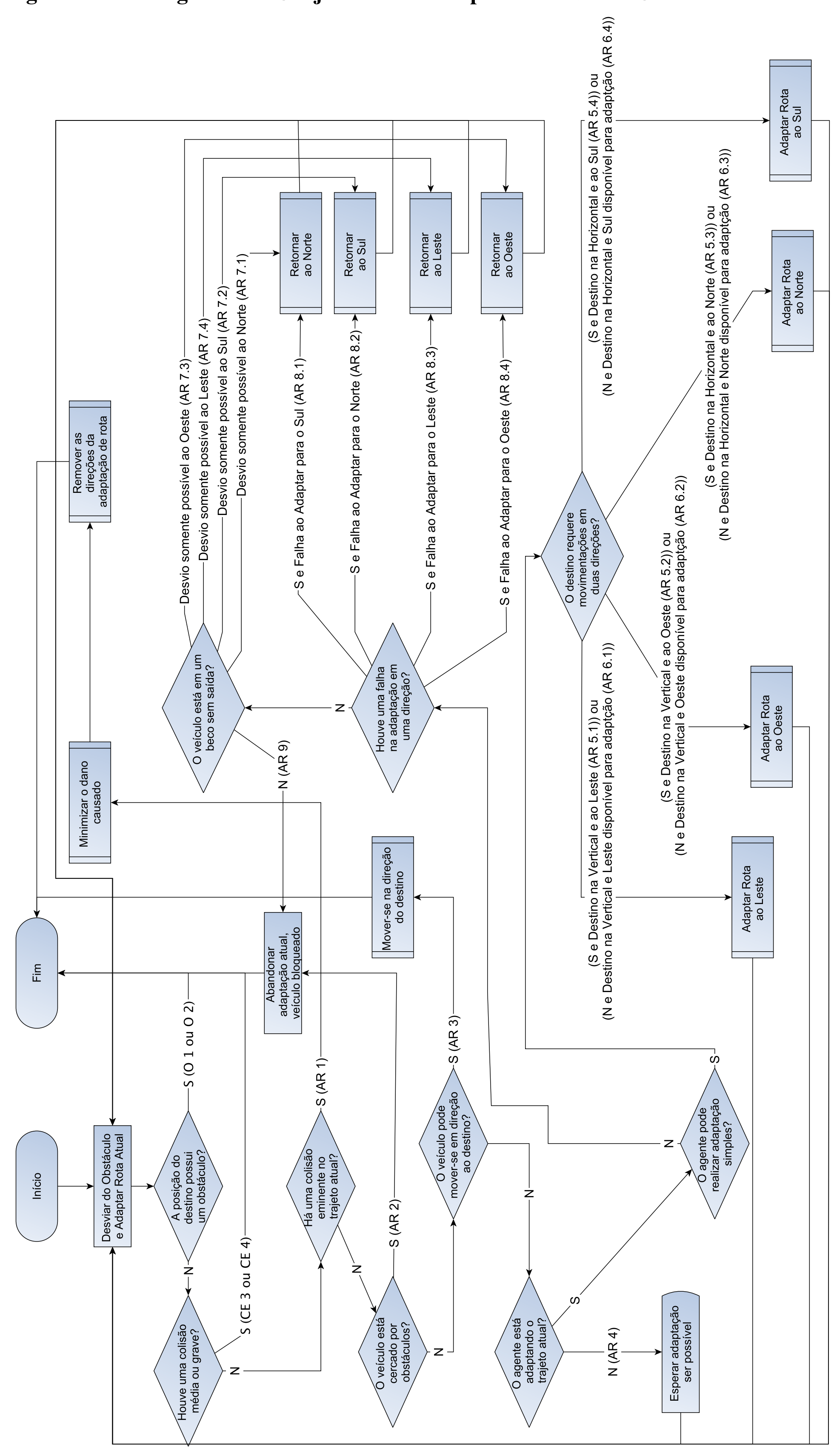
Quando, no meio de um percurso, o agente se deparar com um obstáculo diretamente na direção em que está se movimentando ou detectar uma rota conhecida, o objetivo `!adapt_route (D, X, Y) [achieve]` é disparado. O foco do objetivo é realizar um movimento na direção pela qual este foi disparado. Logo, se o veículo estiver movendo em um sentido `D` e `!adapt_route` for acionado, o objetivo é executado até que o agente realize um movimento na direção `D`, e portanto, é dito que o agente deseja se adaptar na direção `D`.

Existe um total de 21 planos, responsáveis por adaptar o trajeto do veículo para evitar colisões com obstáculos[12]. A implementação de `!adapt_route(D, X, Y) [achieve]` é apresentada no código 20. Um fluxograma do funcionamento do objetivo é demonstrado na Figura 29.

Alguns cenários devem ser considerados antes de explicar o funcionamento deste objetivo. Se for necessário dar prioridade a algum outro evento, a execução do objetivo pode ser sus-

[12] Em cenários onde a colisão pode ser evitada.

Figura 29 – Fluxograma do Conjunto de Planos para o Desvio de Obstáculos



Fonte: Autoria Própria

pensa através de AR 4 (ver linha 37). Caso o agente se depare com uma situação onde uma colisão seja inevitável, !choose_obstacle_collision [perform] é acionado por AR 1 (ver linha 21). E, se o veículo estiver cercado por obstáculos, com o plano AR 2 (ver linha 27) uma crença de que o veículo está bloqueado é inserida no agente. Nos últimos dois casos, !adapt_route(D, X, Y) deixa de ser executado pelo agente.

Uma das características fundamentais deste objetivo define que após o agente escolher uma direção para adaptar sua rota, o veículo irá se deslocar naquele sentido até que o objetivo seja satisfeito ou tal movimento não seja possível. Isto é possível por meio das regras de raciocínio go_adapt(CA_D) (ver linhas 11-17), onde é verificado se um adaptação simples no trajeto atual é possível pela direção CA_D. E, para verificar se o destino do trajeto do agente é na vertical da matriz, a regra north_south (ver linhas 2-3) é utilizada. Similarmente, usa-se a regra east_west (ver linhas 4-5) para o deslocamento na horizontal.

A regra de raciocínio can_adapt(CA_D, D_X, D_Y) (ver linha 7) é utilizada pelos planos deste objetivo para verificar quais direções o agente pode se deslocar. Tal regra se afirma quando: o veículo pode se mover na direção CA_D para atingir o destino (D_X, D_Y) a partir da sua posição atual se não houver um obstáculo no sentido CA_D e a rota na direção CA_D não é conhecida. Essa propriedade é referida como a capacidade de agente adaptar seu trajeto em alguma direção. Já can_adapt_route(CA_D, D_X, D_Y) (ver linha 9) é combinação das regras can_adapt(CA_D, D_X, D_Y) e go_adapt(CA_D).

Este objetivo dá prioridade em adaptações da rota convenientes ao agente. Logo, se o agente desejar adaptar seu trajeto no sentido vertical do matriz, e o destino do trajeto requerer que o agente mova-se em um sentido na horizontal em algum momento, o veículo irá deslocar-se naquele sentido na horizontal, para o leste através de AR 5.1 (ver linha 40) e para o oeste com AR 5.2 (ver linha 43). Semelhantemente, caso o agente queira ajustar sua rota na horizontal da matriz, e acreditar que em algum momento deslocou-se em um sentido na vertical para atingir o destino do trajeto, o veículo irá ajustar seu percurso naquele sentido na vertical; caso o veículo esteve em algum momento do trajeto direcionado ao norte, o plano AR 5.3 (ver linha 46) é escolhido, e no caso tenha se descolado ao sul AR 5.4 (ver linha 49) será ativado. Quando o destino do trajeto for estritamente na vertical ou horizontal, não será possível realizar um movimento conveniente e o veículo irá se mover em uma direção que o permita adaptar sua rota. Nesse caso, uma direção horizontal sem obstrução é escolhida quando o destino está na vertical em relação ao veículo, em AR 6.1 (ver linha 53) para casos onde leste esteja livre, e para o oeste por AR 6.2 (ver linha 56). Se o destino estiver na horizontal do veículo, uma direção vertical sem obstáculo é escolhida, AR 6.3 (ver linha 56) é acionado quando o norte está livre e AR 6.4 (ver linha 62) dispara para o sul. Nestas situações, utiliza-se o objetivo !adapt_drive_direction(A_D, X, Y) [perform] para selecionar a direção escolhida para adaptar a rota e mover o veículo. Vale ressaltar que para adaptar trajetos na vertical o agente prioritariamente tentará se deslocar na horizontal, similarmente, no caso de ajustes na horizontal, a preferência é por movimentos na vertical.

Quando a adaptação do trajeto em alguma direção falhar, a prioridade do agente é retornar na direção oposta quando possível, onde:

- Quando o agente tentou e falhou uma adaptação ao norte, `AR 8.1` (ver linha 88) é acionado e o veículo moverá ao norte;
- Quando o agente tentou e falhou uma adaptação ao sul, `AR 8.2` (ver linha 92) é acionado e o veículo moverá ao sul;
- Quando o agente tentou e falhou uma adaptação ao leste, `AR 8.3` (ver linha 96) é acionado e o veículo moverá ao oeste; e,
- Quando o agente tentou e falhou uma adaptação ao oeste, `AR 8.4` (ver linha 100) é acionado e o veículo moverá ao leste.

Se o veículo não conseguir adaptar sua rota em três direções, independentemente se alguma destas tenha sido escolhida previamente para adaptar o trajeto, o agente irá se mover naquela direção onde é possível o deslocamento, sendo esta situação interpretada como um beco sem saída. Caso: (i) somente a direção norte está livre, `AR 7.1` (ver linha 72) será disparado; (ii) somente a direção sul está livre, `AR 7.2` (ver linha 77) será disparado; (iii) somente a direção oeste está livre, `AR 7.3` (ver linha 82) será disparado; ou se (iv) somente a direção leste está livre, `AR 7.4` (ver linha 87) será disparado.

Em ambos os últimos casos, o objetivo `!invalid_coordinate(TD) [perform]` é acionado para identificar a posição atual do agente como inválida para atingir o destino atual e realizar a movimentação do agente.

Finalmente, se o veículo estiver localizado em alguma posição onde é possível se movimentar na direção pela qual `!adapt_route(D, X, Y) [achieve]` foi acionado, o agente irá realizar tal deslocamento e o objetivo será concluído por `AR 3` (ver linha 33). Se for constatado que não há direção válida para a movimentação do veículo, por `AR 9` (ver linha 104) a adaptação é abandonada e o veículo irá interpretada esta situação como um bloqueio para atingir o destino do trajeto atual.

Código 20 – `!adapt_route (D, X, Y) [achieve]`

```
1  :Reasoning Rules:
2  north_south :- north;
3  north_south :- south;
4  east_west :- east;
5  east_west :- west;
6
7  can_adapt(CA_D, D_X, D_Y) :- at(AT_X, AT_Y),
8  ~obstacle(CA_D, AT_X, AT_Y), ~known_route(CA_D, D_X, D_Y);
9  can_adapt_route(CA_D, D_X, D_Y) :- can_adapt(CA_D, D_X, D_Y), go_adapt(CA_D);
```

```
10
11  go_adapt(north) :− ∼adapt(south), go_adapt(north_south);
12  go_adapt(south) :− ∼adapt(north), go_adapt(north_south);
13  go_adapt(east) :− ∼adapt(west), go_adapt(east_west);
14  go_adapt(west) :− ∼adapt(east), go_adapt(east_west);
15
16  go_adapt(north_south)  :− ∼heading(north), ∼heading(south), ∼adapt(east), ∼adapt(west);
17  go_adapt(east_west)  :− ∼heading(east), ∼heading(west), ∼adapt(north), ∼adapt(south);
18
19  :Plans:
20  /∗ AR 1 ∗/
21  +!adapt_route(D,X,Y) [achieve] :
22              {B at(AT_X, AT_Y), B unavoidable_collision(AT_X, AT_Y)}
23              ← −adapt, +!choose_obstacle_collision [perform],
24              +!clear_adapt [perform],  −!adapt_route(D,X,Y) [achieve];
25
26  /∗ AR 2 ∗/
27  +!adapt_route(D,X,Y) [achieve] :
28              {B obstacle_ahead(north), B obstacle_ahead(south),
29              B obstacle_ahead(east), B obstacle_ahead(west)}
30                  ← +blocked, −!adapt_route(D,X,Y) [achieve];
31
32  /∗ AR 3 ∗/
33  +!adapt_route(D,X,Y) [achieve] : {B adapt, B can_adapt(D,X,Y)}
34                  ← −adapt, +!drive_direction(D) [perform],  +!get_direction [perform],
                        −!adapt_route(D,X,Y) [achieve];
35
36  /∗ AR 4 ∗/
37  +!adapt_route(D,X,Y) [achieve] : {∼B adapt} ← ∗adapt;
38
39  /∗ AR 5.1 ∗/
40  +!adapt_route(D,X,Y) [achieve] : {B can_adapt_route(east,X,Y), B east, B north_south}
41                  ← −adapt, +!adapt_drive_direction(east,X,Y) [perform];
42  /∗ AR 5.2 ∗/
43  +!adapt_route(D,X,Y) [achieve] : {B can_adapt_route(west,X,Y), B west, B north_south}
44                  ← −adapt, +!adapt_drive_direction(west,X,Y) [perform];
45  /∗ AR 5.3 ∗/
46  +!adapt_route(D,X,Y) [achieve] : {B north,  B can_adapt_route(north,X,Y), B east_west}
47                  ← −adapt,  +!adapt_drive_direction(north,X,Y) [perform];
48  /∗ AR 5.4 ∗/
49  +!adapt_route(D,X,Y) [achieve] : {B south,  B can_adapt_route(south,X,Y), B east_west}
50                  ← −adapt,  +!adapt_drive_direction(south,X,Y) [perform];
```

```
51
52 /∗ AR 6.1 ∗/
53 +!adapt_route(D,X,Y) [achieve] : {B can_adapt_route(east,X,Y), B north_south}
54              ← −adapt, +!adapt_drive_direction(east,X,Y) [perform];
55 /∗ AR 6.2 ∗/
56 +!adapt_route(D,X,Y) [achieve] : {B can_adapt_route(west,X,Y), B north_south}
57              ← −adapt, +!adapt_drive_direction(west,X,Y) [perform];
58 /∗ AR 6.3 ∗/
59 +!adapt_route(D,X,Y) [achieve] : {B can_adapt_route(north,X,Y), B east_west}
60              ← −adapt, +!adapt_drive_direction(north,X,Y) [perform];
61 /∗ AR 6.4 ∗/
62 +!adapt_route(D,X,Y) [achieve] : {B can_adapt_route(south,X,Y), B east_west}
63              ← −adapt, +!adapt_drive_direction(south,X,Y) [perform];
64
65
66 /∗ AR 7.1 ∗/
67 +!adapt_route(D,X,Y) [achieve] :
68           {B can_adapt(north,X,Y), ∼B can_adapt(south,X,Y),
69           ∼B can_adapt(east,X,Y), ∼B can_adapt(west,X,Y)}
70              ← −adapt, +!invalid_coordinate(north) [perform];
71 /∗ AR 7.2 ∗/
72 +!adapt_route(D,X,Y) [achieve] :
73           {∼B can_adapt(north,X,Y), B can_adapt(south,X,Y),
74           ∼B can_adapt(east,X,Y), ∼B can_adapt(west,X,Y)}
75              ← −adapt, +!invalid_coordinate(south) [perform];
76 /∗ AR 7.3 ∗/
77 +!adapt_route(D,X,Y) [achieve] :
78           {∼B can_adapt(north,X,Y), ∼B can_adapt(south,X,Y),
79           ∼B can_adapt(east,X,Y), B can_adapt(west,X,Y)}
80              ← −adapt, +!invalid_coordinate(west) [perform];
81 /∗ AR 7.4 ∗/
82 +!adapt_route(D,X,Y) [achieve] :
83           {∼B can_adapt(north,X,Y), ∼B can_adapt(south,X,Y),
84           B can_adapt(east,X,Y), ∼B can_adapt(west,X,Y)}
85              ← −adapt, +!invalid_coordinate(east) [perform];
86
87 /∗ AR 8.1 ∗/
88 +!adapt_route(D,X,Y) [achieve] :
89           {∼B can_adapt(north,X,Y), B can_adapt(south,X,Y)}
90              ← −adapt, +!invalid_coordinate(south) [perform];
91 /∗ AR 8.2 ∗/
92 +!adapt_route(D,X,Y) [achieve] :
```

```
93            {~B can_adapt(south,X,Y), B can_adapt(north,X,Y)}
94                ← −adapt, +!invalid_coordinate(north) [perform];
95 /* AR 8.3 */
96 +!adapt_route(D,X,Y) [achieve] :
97            {~B can_adapt(east,X,Y), B can_adapt(west,X,Y)}
98                ← −adapt, +!invalid_coordinate(west) [perform];
99 /* AR 8.4 */
100 +!adapt_route(D,X,Y) [achieve] :
101           {~B can_adapt(west,X,Y), B can_adapt(east,X,Y)}
102               ← −adapt, +!invalid_coordinate(east) [perform];
103 /* AR 9*/
104 +!adapt_route(D,X,Y) [achieve] : {B at(AT_X, AT_Y)} ←
105                   +missing_adapt_route(AT_X, AT_Y, D, X, Y),
106                   +blocked, −!adapt_route(D,X,Y) [achieve];
```

Fonte: Autoria Própria

Plano para Adaptar de um Trajeto

O objetivo `!adapt_drive_direction(A_D, X, Y) [perform]` tem como função inserir na base de crenças a direção escolhida para adaptar um trajeto e mover o veículo em tal direção através do objetivo `!drive_direction(A_D) [perform]`. A Figura 30 ilustra o fluxograma deste objetivo e o código 21 apresenta a implementação do plano `ADD` (ver linha 3) referente a este objetivo.

Código 21 – `!adapt_drive_direction(A_D, X, Y) [perform]`

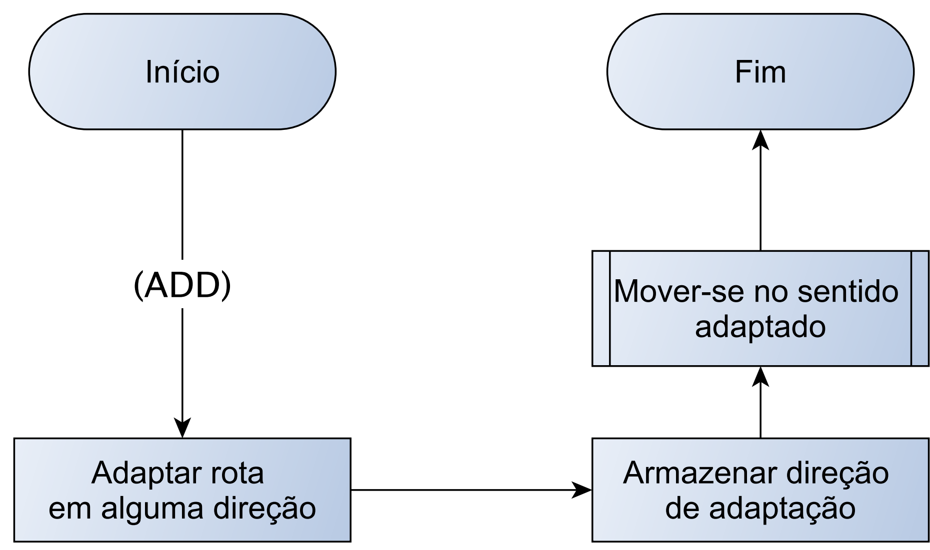
```
1 :Plans:
2 /* ADD */
3 +!adapt_drive_direction(A_D, X, Y) [perform] : {True}
4                ← +adapt(A_D), +!drive_direction(A_D) [perform], −moving, +adapt;
```

Fonte: Autoria Própria

Plano de Coordenadas Inválidas

Quando o agente está tentando adaptar sua rota e está localizado em uma posição `(AT_X, AT_Y)` pela qual não é possível atingir o destino do seu trajeto, `!invalid_coordinate (TD) [perform]` é acionado. Isto é ocasionado em situações onde o agente se encontra em um beco sem saída, onde o veículo está cercado por obstáculos em três direções ou uma adaptação

Figura 30 – Fluxograma do Plano para Adaptar de um Trajeto



Fonte: Autoria Própria

da rota em uma direção falhou e o agente teve que retornar no sentido oposto. Na Figura 31 é demonstrado o fluxograma deste objetivo, que pela execução do plano `DD` (ver linha 3) irá limpar os dados atuais referentes a sua rota e mover-se em um direção.

Figura 31 – Fluxograma do Plano de Coordenadas Inválidas

Fonte: Autoria Própria

Para que a coordenada atual do veículo seja dada como inválida, a ação `no_further_from (F_X, F_Y, AT_X, AT_Y, X, Y)` é executada pelo agente. Dessa forma, as posições em todas as direções da `(AT_X, AT_Y)` são notificadas para identificar que tal coordenada não permite que o veículo adapte seu trajeto de `(F_X, F_Y)` até o destino `(X, Y)`. Por fim, o agente se movimenta na direção `T_D` por meio de `!drive_direction(T_D) [perform]`. No fragmento de Código 22 é apresentado a implementação de `!invalid_coordinate (TD) [perform]`.

Código 22 – `!invalid_coordinate (TD) [perform]`

```
1  :Plans:
2  /* IC */
3  +!invalid_coordinate(TD) [perform] :
4          {G adapt_route(D,X,Y) [achieve], B at(AT_X, AT_Y), B from(F_X,F_Y)}
5            ← no_further_from(F_X,F_Y, AT_X,AT_Y, X, Y), *no_further(F_X,F_Y,
                 AT_X,AT_Y, X, Y),
6            +!clear_adapt [perform], +!drive_direction(TD) [perform], −moving, +adapt;
```

Fonte: Autoria Própria

5.2.4 Conjunto de Planos de Escolha do Menor Dano Possível em Caso de Colisões

A qualquer momento durante um trajeto que o agente venha a detectar que uma colisão com obstáculo é inevitável, o objetivo `!choose_obstacle_collision [perform]` é disparado para escolher o menor impacto possível ao veículo. A implementação deste objetivo é demonstrada no Código 23.

Os tipos de colisões detectados são: leve (`low`), médio (`moderate`) e grave (`high`), sendo a primeira o menor nível de dano e a último o maior. A direção escolhida pelo agente para evitar danos independe da localização do destino de seu trajeto, portanto, o veículo pode vir a se deslocar em qualquer sentido. Cada obstáculo disposto no ambiente é classificado individualmente, sendo assim possível que diferentes obstáculos possuam níveis diferentes de dano.

Código 23 – `!choose_obstacle_collision [perform]`

```
1  :Plans:
2  /* COC 1.1 */
3  +!choose_obstacle_collision [perform] :
4            {B at(AT_X, AT_Y), B obstacle_damage(AT_X, AT_Y, DIRECTION, low)}
5            ← +!colide_obstacle(DIRECTION, low) [perform];
6  /* COC 1.2 */
7  +!choose_obstacle_collision [perform] :
8            {B at(AT_X, AT_Y), B obstacle_damage(AT_X, AT_Y, DIRECTION, moderate)}
9            ← +!colide_obstacle(DIRECTION, moderate) [perform];
10 /* COC 1.3 */
11 +!choose_obstacle_collision [perform] :
12           {B at(AT_X, AT_Y), B obstacle_damage(AT_X, AT_Y, DIRECTION, high)}
13           ← +!colide_obstacle(DIRECTION, high) [perform];
```

Fonte: Autoria Própria

Como a prioridade é escolher uma colisão que cause o menor estrago ao veículo, se

houver ao menos uma colisão do tipo leve, esta será escolhida pelo agente através do plano `COC 1.1` (ver linha 3). Quando não haver nenhum obstáculo que causará um dano leve, e existir ao menos um obstáculo de nível médio, este será escolhido pelo agente através de `COC 1.2` (ver linha 7). Somente quando todos os obstáculos possuirem um nível de dano grave, é que este tipo de colisão irá ocorrer, `COC 1.3` (ver linha 11). Como mostra a Figura 32, em todos estes casos, o plano `CO`, abordado a seguir neste trabalho, é utilizado para lidar a colisão em si.

Figura 32 – Fluxograma de Conjuntos de Plano de Escolha do Menor Dano Possível em Caso de Colisões

Fonte: Autoria Própria

Em todos estes casos, o objetivo `!colide_obstacle(DIRECTION, DAMAGE_LEVEL) [perform]` é acionado após a escolha de um tipo de colisão. A coordenada atual do agente, a qual possibilitou este cenário de colisão, é marcada como inválida para completar o trajeto atual do agente, o nível de dano que será causado ao veículo é adicionado na base de crenças e o agente irá se deslocar naquela direção escolhida pelo objetivo `!choose_obstacle_collision [perform]`. E por fim, como demonstra a implementação no código 24, o objetivo `!get_direction [perform]` é executado para atualizar a direções nas quais o agente deve se mover

para completar o percurso[13].

O fluxograma na Figura 32 exemplifica o funcionamento de `!colide_obstacle(DIRECTION, DAMAGE_LEVEL)[perform]` por meio do seu plano `CO` (ver linha 3), que ativa o objetivo `!drive_ direction(D) [perform]` para se mover na direção do obstáculo de dano mínimo e, a seguir, atualizar as crenças do agente referentes a direção do destino do trajeto com `!get_ direction [perform]`.

Código 24 – `!colide_obstacle(DIRECTION, DAMAGE_LEVEL) [perform]`

```
1 :Plans:
2 /* CO */
3 +!colide_obstacle(DIRECTION, DAMAGE_LEVEL) [perform] :
4             {B from(F_X,F_Y), B at(AT_X, AT_Y), G drive_to(X, Y) [achieve]}
5             ← no_further_from(F_X,F_Y, AT_X, AT_Y, X,Y),
6             *no_further(F_X,F_Y, AT_X, AT_Y, X,Y),
7             +damaged(DAMAGE_LEVEL),
8             +!drive_direction(DIRECTION) [perform], −moving,
9             +!get_direction [perform], +moving;
```

Fonte: Autoria Própria

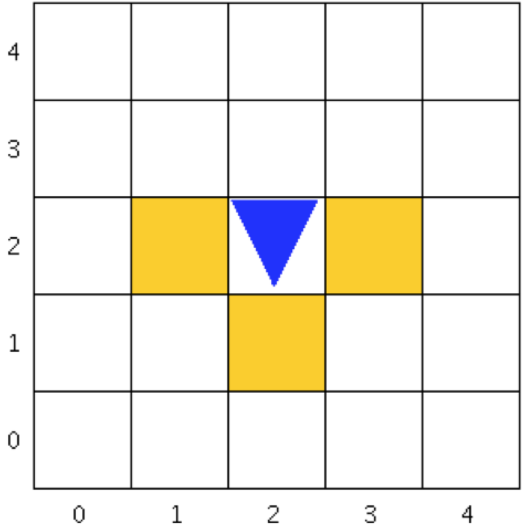
Na Figura 33 é ilustrado um exemplo de cenário onde o agente detectou uma colisão inevitável e escolheu pelo obstáculo que causaria mais dano ao veículo. No item (a) é demonstrado um veículo que ao se mover na direção sul, da coordenada `(2,3)` para `(2,2)` detectou que não haveria como desviar dos obstáculos ao seu redor ou retornar a sua posição anterior[14]. Na sequência, no item (b) o agente percebe o nível de dano que cada dano irá causar ao veículo, sendo estes leve para as coordenadas `(2,1)` e `(2,3)`, médio para `(2,2)` e grave na posição `(2,3)`. E por fim, no item (c), é exibido que o agente se moveu para o leste, mostrando a escolha do agente por um obstáculo que causará somente um dano leve ao veículo. Frisando que a escolha pela direção que haverá uma colisão com obstáculo é tarefa do objetivo `!choose_obstacle_collision [perform]` e `!colide_obstacle(DIRECTION, low) [perform]` irá realizar a movimentação necessário na direção escolhida.

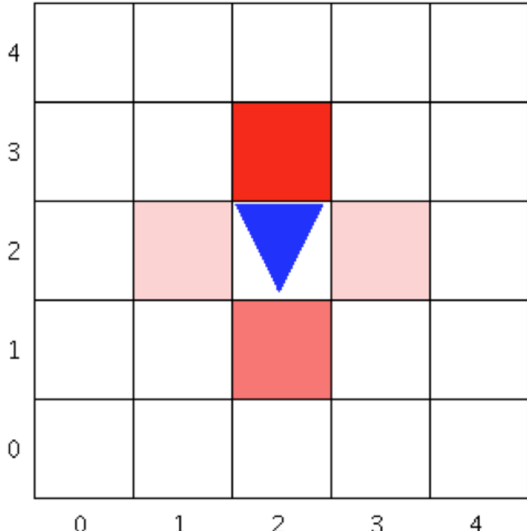
5.2.5 Conjunto de Planos de Controle e Recuperação em Casos de Colisões

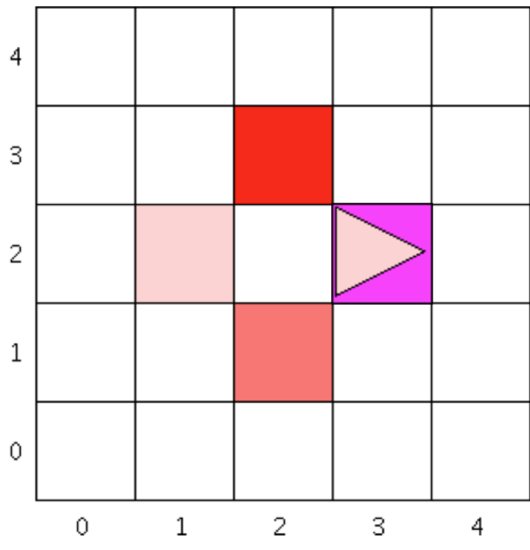
A implementação dos planos de controle de colisões pode ser encontrada no Código 25. Onde é demonstrado que se a qualquer momento o veículo venha a se mover para uma coordenada que contenha um obstáculo em si, o agente reconhece tal situação como uma colisão, por meio do plano `A 2` (ver linha 6) da adição de um crença `at(AT_X, AT_Y)`, e para toda sua

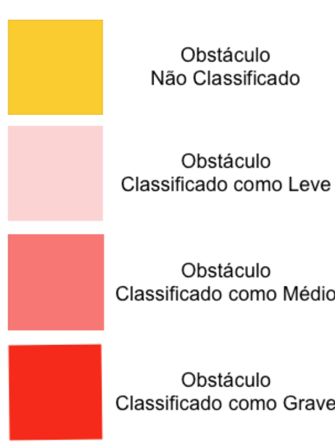
[13] Quando possível. Ver mais sobre planos de controle de emergências na Subseção 5.2.5.
[14] Onde tal posição será interpretada como um obstáculo temporariamente.

Figura 33 – Representação da Escolha por Danos Mínimos







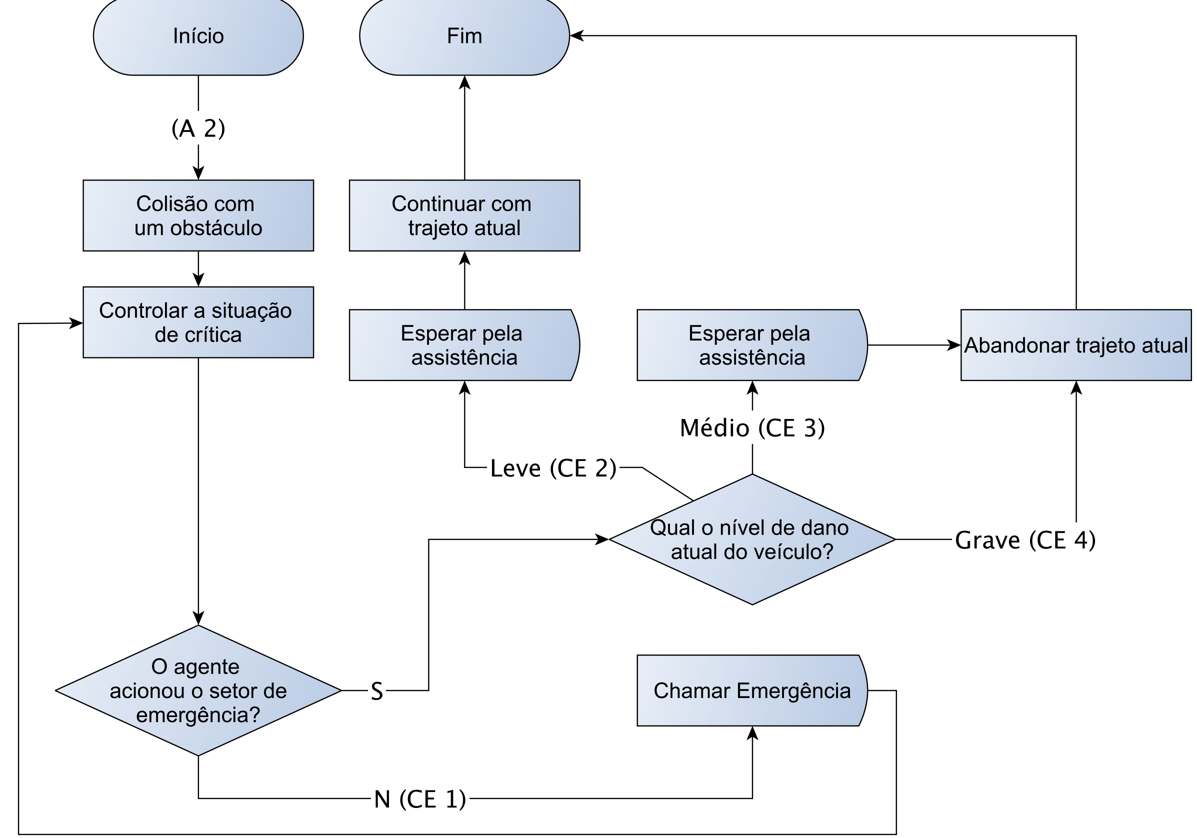


Fonte: Autoria Própria

movimentação e acionando o objetivo `!control_emergency (AT_X, AT_Y) [achieve]`. A Figura 34 ilustra tal relação e demonstra o funcionamento deste objetivo; ilustrando sua invocação pelo plano `A 2`, e quais planos irão abandonar os objetivos `!drive_to (X, Y)` e `!adapt_route (D, X, Y) [achieve]` responsáveis pela movimentação do veículo no ambiente.

Após a colisão, a prioridade de `!control_emergency (AT_X, AT_Y) [achieve]` é executar `CE 1` (ver linha 11) para ligar para a emergência e espera até que a mesma chegue nas coordenada atual do veículo. Quando o nível de danificação do veículo for leve, válido para qualquer outro impacto passado, através de `CE 2` (ver linha 16) o agente espera que a emer-

Figura 34 – Fluxograma do Conjunto de Planos de Controle e Recuperação em Casos de Colisões



Fonte: Autoria Própria

gência acionada auxilie o veículo e seus passageiros, e a qualquer outro indivíduo afetado na colisão, antes de continuar percorrendo seu trajeto[15]. Semelhantemente, se o nível de dano atual do veículo for médio, no plano ativado `CE 3` (ver linha 23) espera-se pela ajuda da emergência aos envolvidos na colisão, no entanto, devido ao estrago causado, a rota atual é abandonada[16]. Para estas últimas duas situações, o objetivo é concluído quando o veículo não estiver envolvido na colisão pela qual `!control_emergency (AT_X, AT_Y) [achieve]` foi acionado. Isto é definido pela regra de raciocínio `control_emergency (X,Y)` (ver linha 2).

Por fim, quando uma colisão ocasionar um dano grave, o veículo abandona toda sua movimentação imediatamente e também o plano de controle de emergências por meio de `CE 4` (ver linha 30)[17].

Código 25 – `!control_emergency (AT_X, AT_Y) [achieve]`

```
1  :Reasoning Rules:
2  control_emergency (X,Y) :− ~crashed(X,Y);
3
```

[15] Como definido pelo plano `FAR 4` (código 13, linha 16), quando há uma colisão leve no decorrer da execução do agente, o veículo retorna ao depósito após atender todos seus passageiros.

[16] Após uma colisão média, o veículo recusa a corrida atual e retorna para o depósito imediatamente, de acordo com os planos `FAR 3.1` (código 13, linha 8) e `FAR 3.2` (Código 13, linha 12).

[17] Como definido no plano `FAR 2` (código 13, linha 5), esta ocasião irá fazer com o que o veículo fique indisponível.

```
4  :Plans:
5  /* A 2 */
6  +at(AT_X,AT_Y) : {B obstacle(center, AT_X,AT_Y)}
7                ← −moving, −adapt, +crashed(AT_X,AT_Y),
8                +!control_emergency (AT_X,AT_Y) [achieve];
9
10 /* CE 1 */
11 +!control_emergency (AT_X, AT_Y) [achieve] :
12             {B crashed(AT_X, AT_Y), ~B emergency(AT_X, AT_Y)}
13             ← call_emergency(AT_X,AT_Y), *emergency(AT_X, AT_Y);
14
15 /* CE 2 */
16 +!control_emergency (AT_X, AT_Y) [achieve] :
17                {B emergency(AT_X, AT_Y), B damaged(low),
18                ~B damaged(moderate), ~B damaged(high)}
19                ←  get_assistance(AT_X,AT_Y), *assisted(AT_X,AT_Y),
20                −crashed(AT_X,AT_Y), +moving, +adapt;
21
22 /* CE 3 */
23 +!control_emergency (AT_X,AT_Y) [achieve] :
24                {B emergency(AT_X,AT_Y), B damaged(moderate), ~B damaged(high)}
25                ← get_assistance(AT_X,AT_Y), *assisted(AT_X,AT_Y),
26                −crashed(AT_X,AT_Y), −!adapt_route(D,X,Y) [achieve],
27                −!drive_to(X,Y) [achieve];
28
29 /* CE 4 */
30 +!control_emergency (AT_X,AT_Y) [achieve] :
31                {B emergency(AT_X,AT_Y), B damaged(high)}
32                ← −!drive_to(X,Y) [achieve], −!adapt_route(D,X,Y) [achieve],
33                −!control_emergency (AT_X,AT_Y) [achieve];
```

Fonte: Autoria Própria

5.3 AMBIENTE

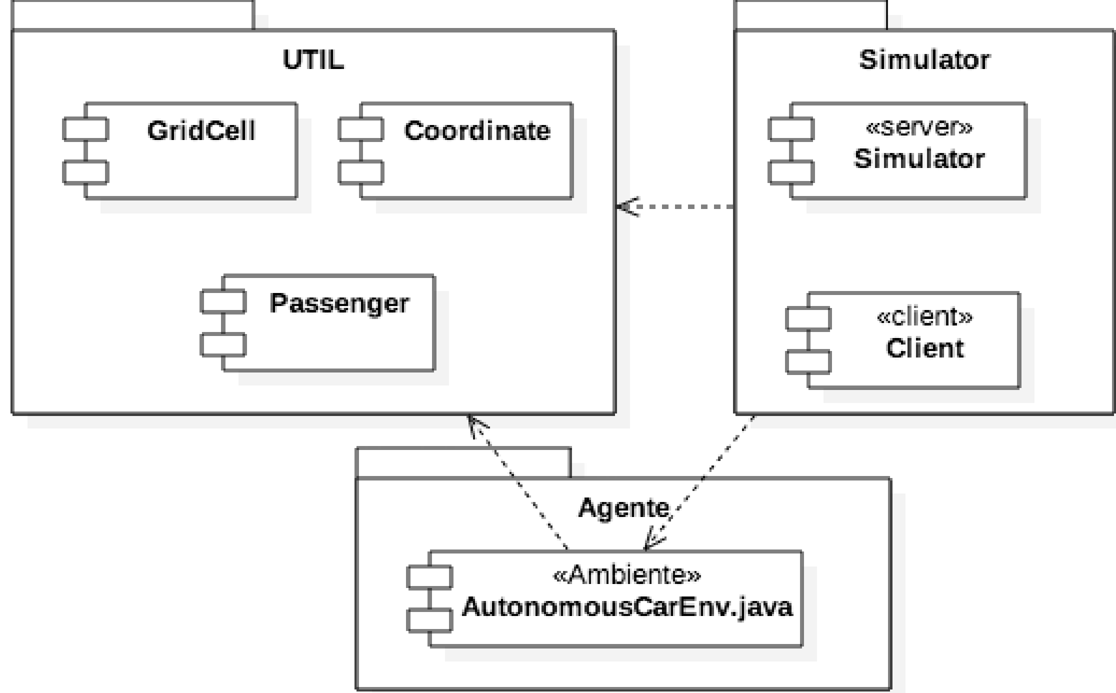
Esta seção detalha o ambiente onde o agente está situado, sendo este o componente `AutonomousCarEnv.java` do módulo Agente. O módulo *Simulator* depende de mensagens vindas do ambiente para que venha a ser executado, e tanto este quanto o ambiente utilizam o módulo UTIL. Esta relação é ilustrada no diagram de componentes na Figura 35.

Dentro do cenário considerado neste trabalho, as propriedades do ambiente desenvol-

vido o classificam como:

- Parcialmente observável: O agente somente obtêm informações referente as coordenadas da matriz conforme sua movimentação entre as posições do ambiente. Embora o veículo possa receber corridas com pontos de partida e destino em qualquer posição do ambiente, o agente não adquire nenhuma percepção sobre o estado daquelas coordenadas. No entanto, as crenças obtidas durante sua execução são mantidas na base de crenças do agente;
- Não-determinístico: Como será detalhado na sequência deste trabalho, a colisão ou não do veículo é determinado por meio de uma probabilidade. Similarmente, é especificado uma probabilidade para cada tipo de classificação do dano que um obstáculo pode causar ao agente;
- Estático: Modificações no ambiente são ocasionadas somente pelo agente. Logo, obstáculos mantêm sua posição durante toda a existência do ambiente.
- Discreto: Para qualquer tamanho `N` da matriz de posições, há um número finito de possíveis estados do ambiente;
- Sequencial: As ações realizadas pelo agente podem vir a afetam seu desempenho no futuro.

Figura 35 – Fragmento do Diagrama de Componentes do Sistema: Ambiente e suas relações



Fonte: Autoria Própria

A implementação completa do ambiente encontra-se no código 27 do Apêndice B. O ambiente armazena internamente informações que vem a auxiliar a execução do agente. Entre

estas, a matriz de posições com todas as coordenadas para as quais o veículo pode se deslocar é mapeada pela variável Map <String, GridCell> environmentGrid (ver linha 21). Posições são identificadas por uma String (X,Y), onde X e Y representam respectivamente eixos X e Y daquela coordenada na matriz; e, GridCell é um objeto que representa a coordenada dessa matriz, que pode representar: (i) localização do agente; (ii) localização do depósito; (iii) posição de um ponto de partida de uma corrida; (iv) posição de um ponto de destino de uma corrida; (v) obstáculos em uma coordenada.

A lista de passageiros que o veículo virá a atender está contida em ArrayList <Passenger> passengers (ver linha 36), onde cada objeto Passenger possui coordenadas para um ponto de partida e um de destino. A informação sobre a localização (X,Y) inicial do veículo está contida em Coordinate car[18] (ver linha 26), e esta variável é atualizada conforme a movimentação do agente pela matriz. E, uma coordenada (X,Y) definida em Coordinate depotLocation representa a posição do depósito dentro do ambiente [19]. Na variável int max GridSize (ver linha 32) é determinado o tamanho N da matriz de posições.

As configurações iniciais do ambiente são executadas a partir do método void setMAS (MAS m) (ver linha 45). Aqui, são realizadas chamadas de métodos e definições de variáveis, tais como:

- Inicialização da matriz de posições: são criadas um total de N x N células no ambiente pelo método initGridInformation() (ver linha 88), onde N é determinado por maxGridSize;

- Lista de Passageiros: A função initPassengerList() (ver linha 99) cria um total de P corridas, onde P é determinado pela variável int nPassengers (ver linha 37). Para cada passageiro existem duas coordenadas distintas, definidas aleatoriamente pelo ambiente, especificando o ponto de partida e de destino;

- Obstáculos: Um total de O obstáculos, onde $\{O \mid 0 \leq O \leq N-2\}$, definido por int nObstacles (ver linha 38), são criados aleatoriamente por initObstacles() (ver linha 112) em diferentes posições da matriz. Por padrão, a posição inicial do agente e o depósito nunca irão ter um obstáculo em suas coordenadas, e podem haver situações onde há um obstáculo no ponto de partida ou destino de uma corrida;

- Enviar informações iniciais sobre o ambiente ao simulador, referente ao tamanho do N do ambiente.

A função Unifier executeAction(String agName, Action act) (ver linha 127) interpreta as atuações do agente sobre o ambiente. As ações realizadas pelo agente são:

[18] Antes da execução do ambiente, pode ser definido qualquer posição dentro da matriz do ambiente para ser a inicial do agente.

[19] Pode ser definido qualquer posição dentro da matriz do ambiente.

- Dirigir (drive) (ver linha 130);
- Bússola (compass) (ver linha 138);
- Localizar (localize) (ver linha 143);
- Buscar corrida (get_ride) (ver linha 146);
- Recusar corrida (refuse_ride) (ver linha 149);
- Estacionar (park) (ver linha 153);
- Beco sem saída (no_further_from) (ver linha 157);
- Chamar emergência (call_emergency) (ver linha 166);
- Esperar ajuda (get_assistence) (ver linha 171).

Após a identificação do predicado da ação, seus argumentos são atribuídos a variáveis e um método correspondente a aquela ação é invocado.

Na sequência dessa seção será abordada como é feito a atualização da posição do agente, a interpretação das ações realizadas pelo agente pelo ambiente, as situações onde uma colisão é inevitável e a comunicação entre o ambiente e o simulador.

5.3.1 Atualizar a Localização do Agente

O método updateLocation() (ver linha 198) é responsável pela atualização da posição do agente dentro do ambiente, onde este recebe como argumentos as coordenadas antigas do veículo e sua nova localização.

Durante este processo, é removida a percepção sobre a posição antiga do veículo e as novas coordenadas são inseridas na base de crenças do agente e atualizadas no simulador. A seguir, o método scanSurroundings() (ver linha 226) é invocado para obter informações sobre as redondezas do veículo; cuja função principal é verificar a existência de obstáculos em qualquer direção que o agente possa vir a se movimentar à partir da sua nova localização. Entende-se como as posições vizinhas de (X,Y) aquelas coordenadas na sua redondeza, diretamente nas direções norte, sul, leste e oeste definidas como, respectivamente, (X,Y+1), (X,Y-1), (X+1,Y) e (X-1,Y).

Para tal, verifyObstacle() (ver linha 298) é acionado em cada uma das posições vizinhas em relação a posição atual (X,Y) do veículo. Na ocorrência de um obstáculo em determinada direção direction, é adicionado uma nova percepção (obstacle(direction, X,

Y)) no agente informando-o a direção em que o obstáculo se encontra em relação a sua localização e uma outra percepção (obstacle(center, directionX, directionY)) referente à aquela posição[20] que contém o obstáculo.

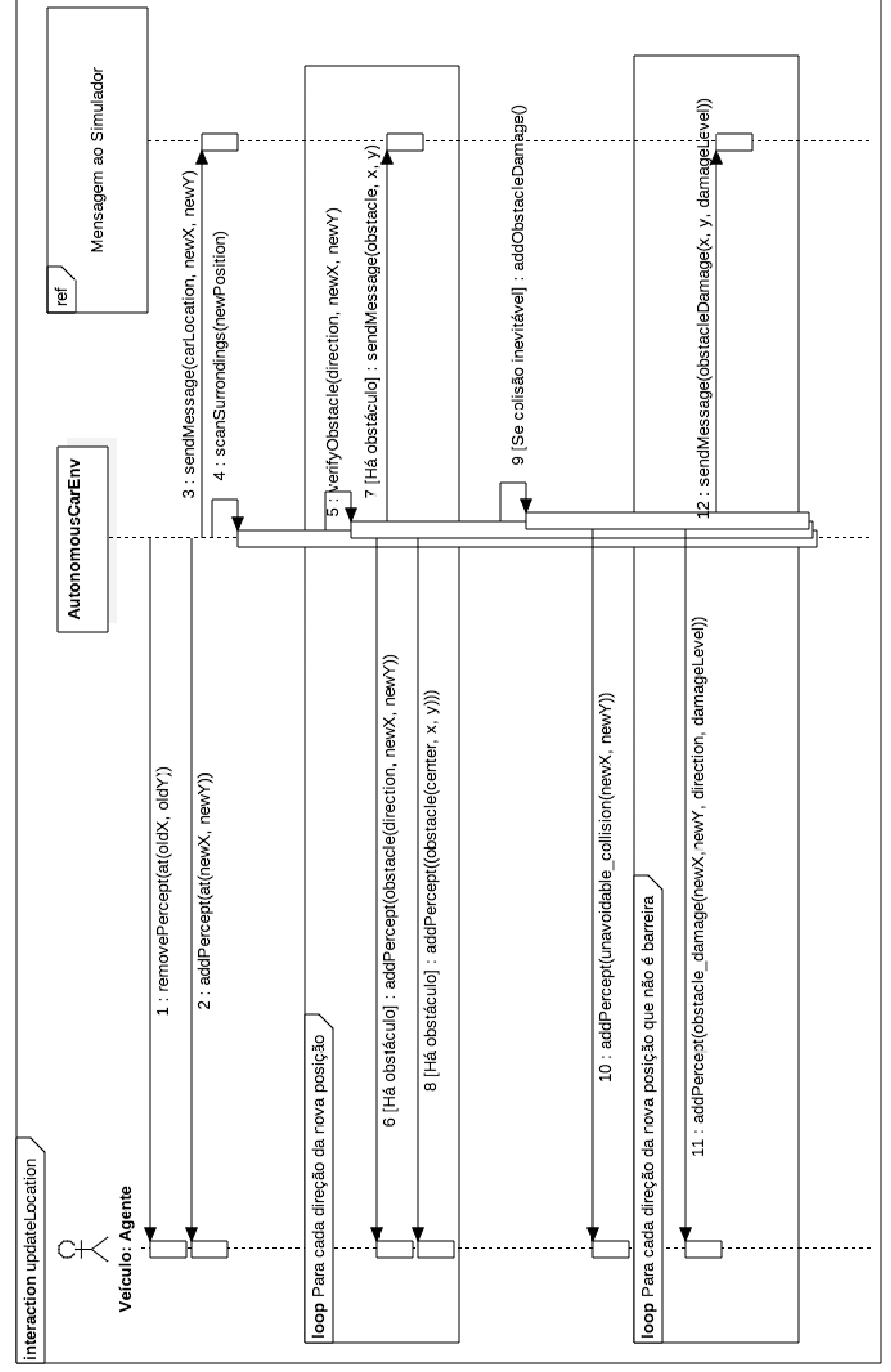
Como o ambiente é estático, nenhum obstáculo irá mudar de posição e nenhum novo obstáculo será adicionado ao ambiente após o início de sua execução. Consequentemente, o estado de uma coordenada, seja este estado livre ou obstruído por um obstáculo, será mantido durante todo do o funcionamento do agente. Portanto, o agente aprende novos conhecimentos sobre a matriz do ambiente conforme percorre diferentes trajetos. As coordenadas fora do limite da matriz ambiente são interpretadas como barreiras. Vale lembrar que, como o ambiente é estático, o agente mantém suas crenças sobre obstáculos conhecidos. As coordenadas de cada obstáculo são enviadas ao simulador.

Após o reconhecimento da existência de obstáculos, é verificado se o cenário atual do veículo é propício para a ocorrência de um acidente[21]. Caso uma colisão seja inevitável, o nível de dano que cada obstáculo pode causar ao veículo é averiguado pelo método addObstacle Damage() (ver linha 262), e a classificação da danificação é transferida para o simulador. Note que, em hipótese alguma o veículo irá se deslocar para coordenadas na barreira do ambiente. O funcionamento de updateLocation() é apresentado no diagrama de sequência[22] da Figura 36. Neste é exibido o seguinte fluxo de ações:

1. O ambiente remove a percepção do agente referente a sua posição antiga at(oldX, oldY);
2. A percepção at(newX, newY) é inserida, referente a nova localização do agente;
3. A posição do veículo (newX, newY) é atualizado no simulador;
4. O método scanSurroundings() é acionado;
5. É verificado se há obstáculos com verify Obstacle() para cada direção direction ao redor de (newX, newY), onde
6. se houver um obstáculo naquela direção direction, a percepção obstacle(direction, newX, newY) é inserida, e
7. nas coordenadas (x, y) é inserido uma perceção obstacle(center, x, y), e
8. é enviado para o simulador as coordenadas (x, y) do obstáculo;
9. Caso o agente se encontre em um situação onde uma colisão é inevitável, o método addObsta cle Damage() é invocado, onde

[20] (X,Y+1), (X,Y-1), (X+1,Y) ou (X-1,Y).
[21] Ver mais na Subseção 5.3.3
[22] Diagrama UML (Unified Modeling Language) para representar uma sequência de processos computacionais.

Figura 36 – Diagrama de Sequência do método `updateLocation()`



Fonte: Autoria Própria

10. a percepção `unavoidable_colision(newX, newY)` é inserida, pois a localização atual apresenta um cenário onde não é possível desviar dos obstáculos, e

11. cada direção `direction` dentro do limite do ambiente ao redor de `(newX, newY)`, será

classificada com um nível de dano `damageLevel` e a percepção `obstacle_damage(newX, newY, direction, damageLevel)` é inserida,

12. e por fim, o ambiente irá informar ao simulador o dano que pode vir a ser causado por cada obstáculo.

5.3.2 Ações do Agente sobre o Ambiente

As ações do agente que o ambiente é capaz de interpretar são definidas a seguir.

Localizar

O método `localize()` (ver linha 451) é invocado após o agente realizar a ação `localize`. Tem como objetivo adicionar percepções referentes a posição inicial do veículo, onde estas coordenadas são inseridas por meio do método `updateLocation()`, e também a localização do depósito. Ambas informações são enviadas ao simulador. Na figura 37 é possível observar o diagrama de sequência desta função, onde:

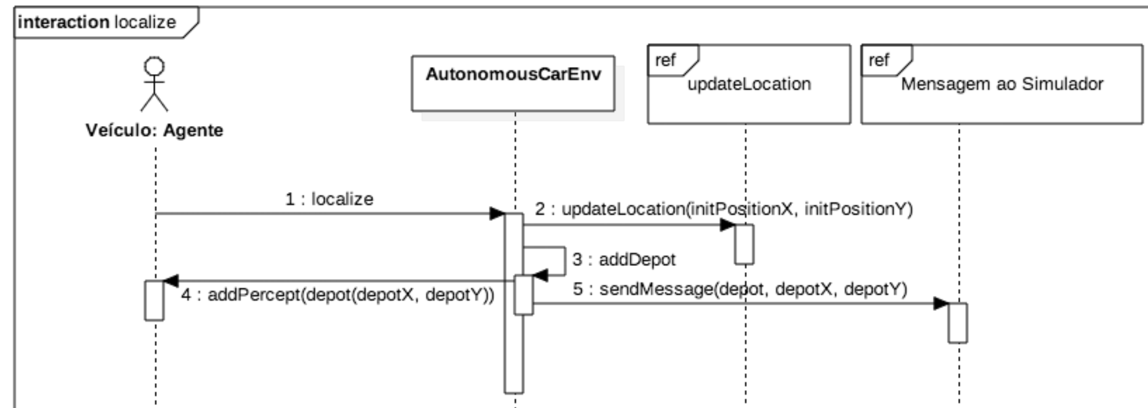
1. Agente realiza ação `localize`;
2. O ambiente invoca o método `updateLocation()` com as coordenadas iniciais `(initPositionX, initPositionY)` do agente;
3. o método `addDepot` (ver linha 458) é chamado;
4. A percepção `depot(depotX, depotY)` é inserida, referindo-se as coordenadas do depósito;
5. E por fim, as coordenadas `(depotX, depotY)` do depósito são enviadas ao simulador.

Dirigir

A movimentação do veículo é realizada por meio da ação `drive`, que desencadeia a chamada do método `drive()` (ver linha 183); aqui é calculada a nova posição baseada na direção em que o agente deseja se movimentar. Seja `(X,Y)` a coordenada atual do agente, o cálculo de sua movimentação é feito da seguinte forma:

- Caso o movimento seja ao norte, a nova posição do agente será `(X,Y+1)`;

Figura 37 – Diagrama de Sequência do método `localize()`



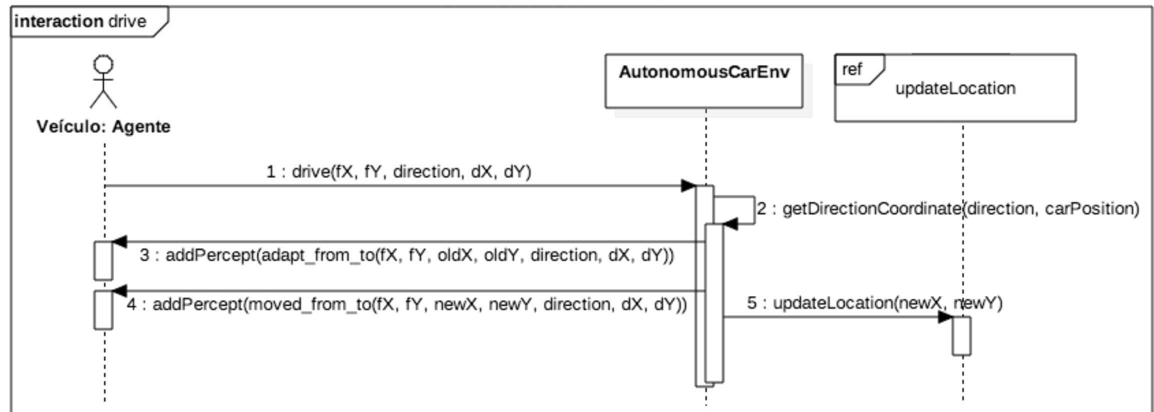
Fonte: Autoria Própria

- Caso o movimento seja ao sul, a nova posição do agente será `(X,Y-1)`;
- Caso o movimento seja ao leste, a nova posição do agente será `(X+1,Y)`; e,
- Caso o movimento seja ao oeste, a nova posição do agente será `(X-1,Y)`.

Também são inseridas percepções referentes ao trajeto sendo realizado pelo veículo, tanto na coordenada antiga quanto na nova posição. E por fim, é invocado o método `updateLocation()` para atualizar a posição do agente. No diagrama de sequência exibido na Figura 38 é possível verificar que:

1. Agente realiza ação `drive`, informando as coordenadas do início do trajeto `(fX,fY)`, a direção do movimento `direction` e o destino do percurso `(dX,dY)`;
2. O ambiente calcula as novas coordenadas do agente ao mover-se em uma direção `direction` na matriz de posições;
3. A percepção `adapt_from_to(fX, fY, oldX, oldY, direction, dX, dY)` é inserida na posição antiga `(oldX, oldY)`;
4. A percepção `moved_from_to(fX, fY, newX, newY, direction, dX, dY)` é inserida nova posição do agente `(newX, newY)`;
5. E por fim, o ambiente invoca o método `updateLocation()` com as novas coordenadas `(newX, newY)` do agente.

Figura 38 – Diagrama de Sequência do método `drive()`



Fonte: Autoria Própria

Buscar Corrida

Após realizar o pedido de uma nova corrida com a ação `get_ride`, o método `getRide()` (ver linha 468) é invocado para selecionar um novo passageiro. Esta função remove as percepções referentes ao ponto de partida e o ponto de destino, da última corrida realizada, e na sequência, é verificado se ainda há algum passageiro disponível aguardando o veículo. Caso exista alguma corrida, o ambiente envia ao agente os dados sobre o tal passageiro e atualiza as informações do simulador sobre a nova corrida. Se não, o agente é informado que não há mais nenhuma corrida disponível. A figura 39 demonstra o diagrama de sequência do método `getRide()`, onde:

1. Agente realiza ação `get_ride`, para obter informações sobre uma nova corrida;
2. A percepção `pick_up(oldPX, oldPY)` é removida, sendo `(oldPX, oldPY)` a coordenada do ponto de partida da última corrida realizada;
3. A percepção `drop_off(oldDX, oldDY)` é removida, sendo `(oldPX, oldPY)` a coordenada do ponto de destino da última corrida realizada;
4. Se houver um passageiros disponível, a percepção da coordenada do ponto de partida da nova corrida `pick_up(pX, pY)`, o é inserida e,
5. A percepção da coordenada do ponto de destino da nova corrida `drop_off(dX, dY)` é inserida, e
6. Estas informações referentes a corrida, `pick_up(pX, pY)` e `drop_off(dX, dY)` são enviadas ao simulador;
7. Caso não haja uma corrida disponível, uma percepção `no_possible_new_ride` é inserida;

8. E, por fim, uma percepção `ride_info` é inserida no agente, o notificando que todas os dados referentes a uma corrida já foram informados.

Figura 39 – Diagrama de Sequência do método `getRide()`

Fonte: Autoria Própria

Bússola

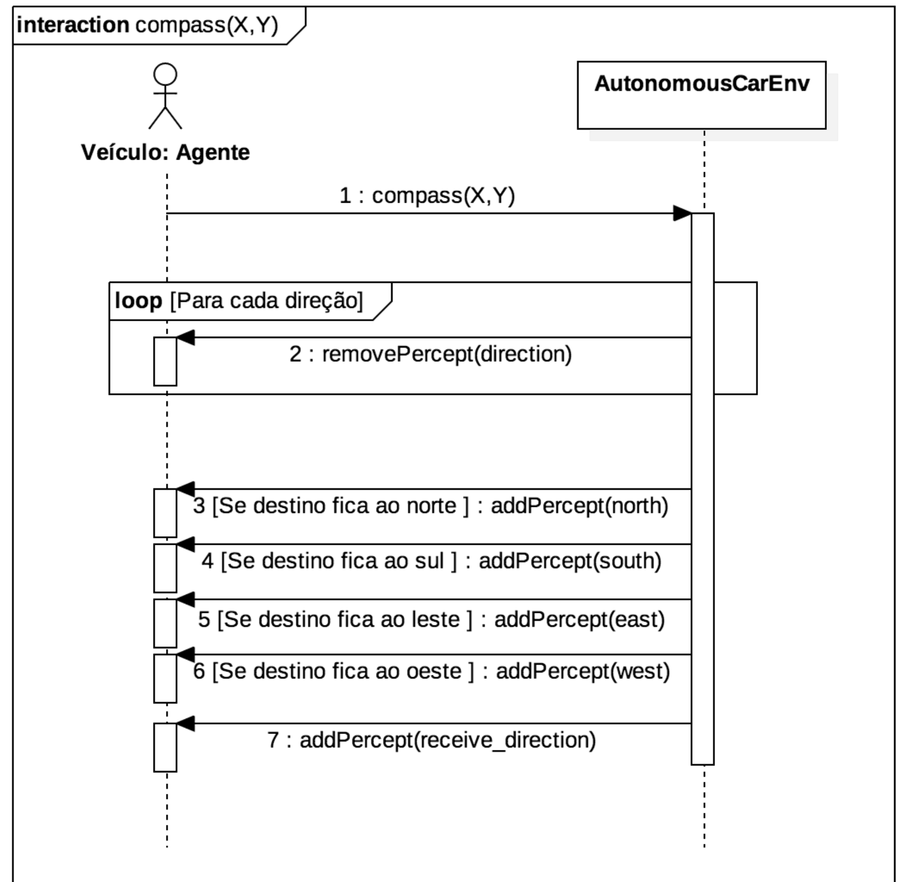
Para se orientar e descobrir o sentido do destino do trajeto atual, o agente realiza a ação `compass`, que é interpretada com o método `compass()` (ver linha 374) pelo ambiente. Aqui, todas as crenças referentes a alguma direção são removidas do agente, e verifica-se quais direções do destino atual com base na posição atual do agente. Após a verificação, novas percepções sobre as direções são adicionadas ao agente. Estes eventos são exemplificados por meio do diagrama de sequência contido na Figura 40. Este diagrama descreve que:

1. Agente realiza ação `compass(X,Y)`, onde `(X,Y)` é a coordenada do destino atual;

2. Todas as percepções referentes a uma direção `direction` são removidas do agente;

3. Se o destino fica ao norte da posição atual do veículo, a percepção `north` é inserida;
4. Se o destino fica ao sul da posição atual do veículo, a percepção `south` é inserida;
5. Se o destino fica ao leste da posição atual do veículo, a percepção `east` é inserida;
6. Se o destino fica ao oeste da posição atual do veículo, a percepção `west` é inserida;
7. E, por fim, a percepção `receive_direction` é adicionado, informando-o que qualquer informação sobre uma direção já foi inserida.

Figura 40 – Diagrama de Sequência do método `compass()`



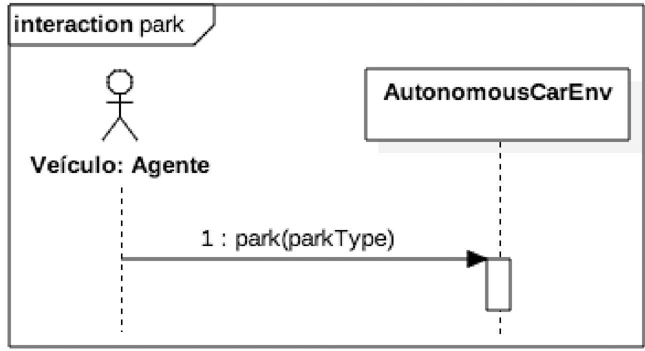
Fonte: Autoria Própria

Estacionar

Quando o veículo deseja estacionar, seja para o embarque ou desembarque de passageiros, a ação `park` é realizada. O diagrama de sequência na Figura 41 exibe a execução realizada pelo ambiente nestas situações, onde o método `park()` (ver linha 434) é invocado.

1. Agente realiza ação `park(parkType)`, onde `parkType` é o motivo que levou o veículo a estacionar;

Figura 41 – Diagrama de Sequência do método `park()`



Fonte: Autoria Própria

Recusar Corrida

Se não for possível concluir uma corrida por algum motivo, entre estes: quando há uma colisão média ou grave durante o trajeto, existe um obstáculo no ponto de partida ou de destino da corrida, ou não foi possível encontrar uma rota para concluir um trajeto; o agente realiza a ação `refuse_ride`. O ambiente interpreta tal ação e invoca o método `refuseRide()` (ver linha 399), e a sequência desses eventos é demonstrado pelo diagrama de sequência na Figura 42, onde:

1. Agente realiza ação `refuse_ride(refuseType)`, onde `refuseType` é o motivo que ocasionou a recusa da corrida;

2. O ambiente informa ao simulador que a corrida atual foi cancelado por causa de `refuse Type`.

Beco Sem Saída

Ao descobrir que a partir de uma coordenada não é possível concluir um trajeto, o agente executa a ação `no_further_from`. Quando assimilada pelo ambiente por meio do método `noFurtherFrom()` (ver linha 355), as posições ao redor daquela coordenada são atualizadas para não permitir que o veículo se desloque até lá. Isto é possível pois o ambiente insere

Figura 42 – Diagrama de Sequência do método `refuseRide()`

Fonte: Autoria Própria

percepções que serão interpretadas pelo agente através das regras de raciocínio `known_route`. Na Figura 43 é possível observar o diagrama de sequência deste método, e:

1. O agente realiza ação `no_further_from(fX, fY, x, y, dX, dY)`, onde `(fX,fY)` e `(dX,dY)` são respectivamente as coordenadas do início e destino de um trajeto e `(x,y)` é uma posição a partir da qual não é possível atingir `(dX,dY)`;
2. Para cada direção `direction` ao redor de `(x,y)`, é adicionado percepções `adapt_from_to` e `moved_from_to` que caracterizam o movimento até `(x,y)` como uma rota conhecida;
3. A percepção `no_further(fX, fY, x, y, dX, dY)` é inserida, informando o agente que as coordenadas vizinhas de `(x,y)` foram atualizadas.

Chamar Emergência

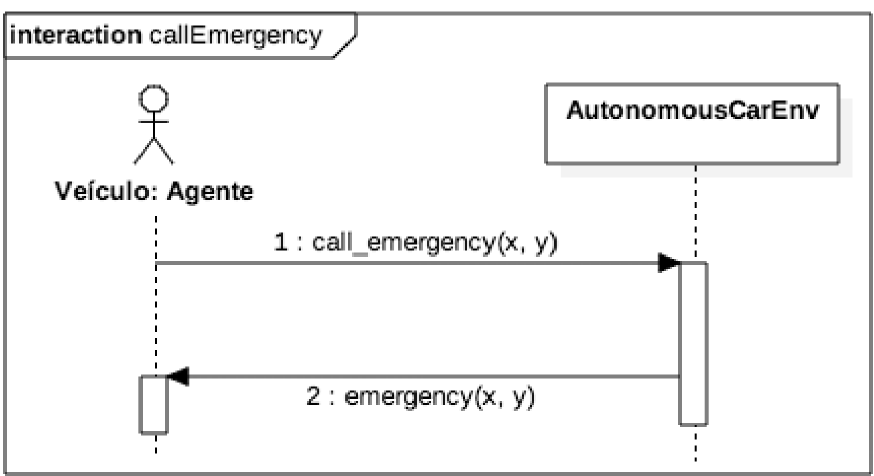
Quando o veículo colide com um obstáculo de nível de dano leve ou médio, a ação `call_emergency` é executada. E nestas ocasiões, a função `callEmergency()` (ver linha 519) é chamada. Esta adiciona uma percepção no agente de que a emergência na sua posição atual está sendo atendida. Como demonstrado no diagrama da Figura 44:

1. Agente realiza ação `call_emergencyx,y)` para acionar a emergência até sua posição atual `(x,y)`;
2. A percepção `emergency(x,y)` é inserida para representar a chegada de um setor de emergência.

Figura 43 – Diagrama de Sequência do método `noFurtherFrom()`

Fonte: Autoria Própria

Figura 44 – Diagrama de Sequência do método `callEmergency()`



Fonte: Autoria Própria

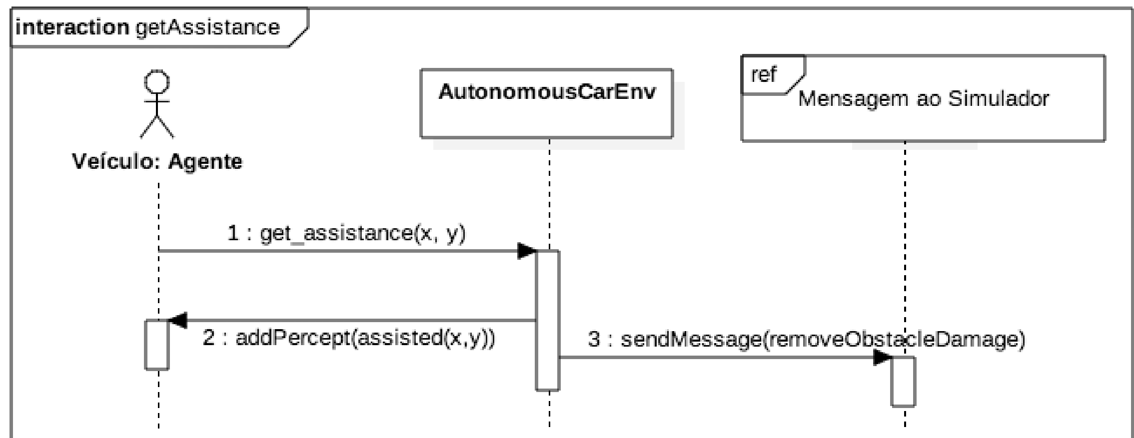
Esperar Ajuda

Logo após a chegada da emergência, em decorrência de uma colisão do veículo, o agente requisita que a assistência necessária seja realizada por meio da ação `get_assistance`.

O ambiente processa este pedido com o método `getAssistance()` (ver linha 529), que depois da assistência ser realizada, adiciona uma percepção referente ao fato de que o veículo está pronto para retomar seu percurso. Nessa ocasião, é informado ao simulador para remover os dados sobre os níveis de dano de cada obstáculo envolvido na colisão. O funcionamento deste método é explorado no diagrama de sequência da Figura 45, onde:

1. Agente realiza ação `get_assistance(x,y)` para esperar qualquer assistência necessária por causa de uma na sua posição atual `(x,y)`;
2. A percepção `assisted(x, y)` é inserida quando o agente for auxiliado;
3. E por fim, as informações dos níveis de dano de dano obstáculo são removidos do simulador.

Figura 45 – Diagrama de Sequência do método `getAssistance()`



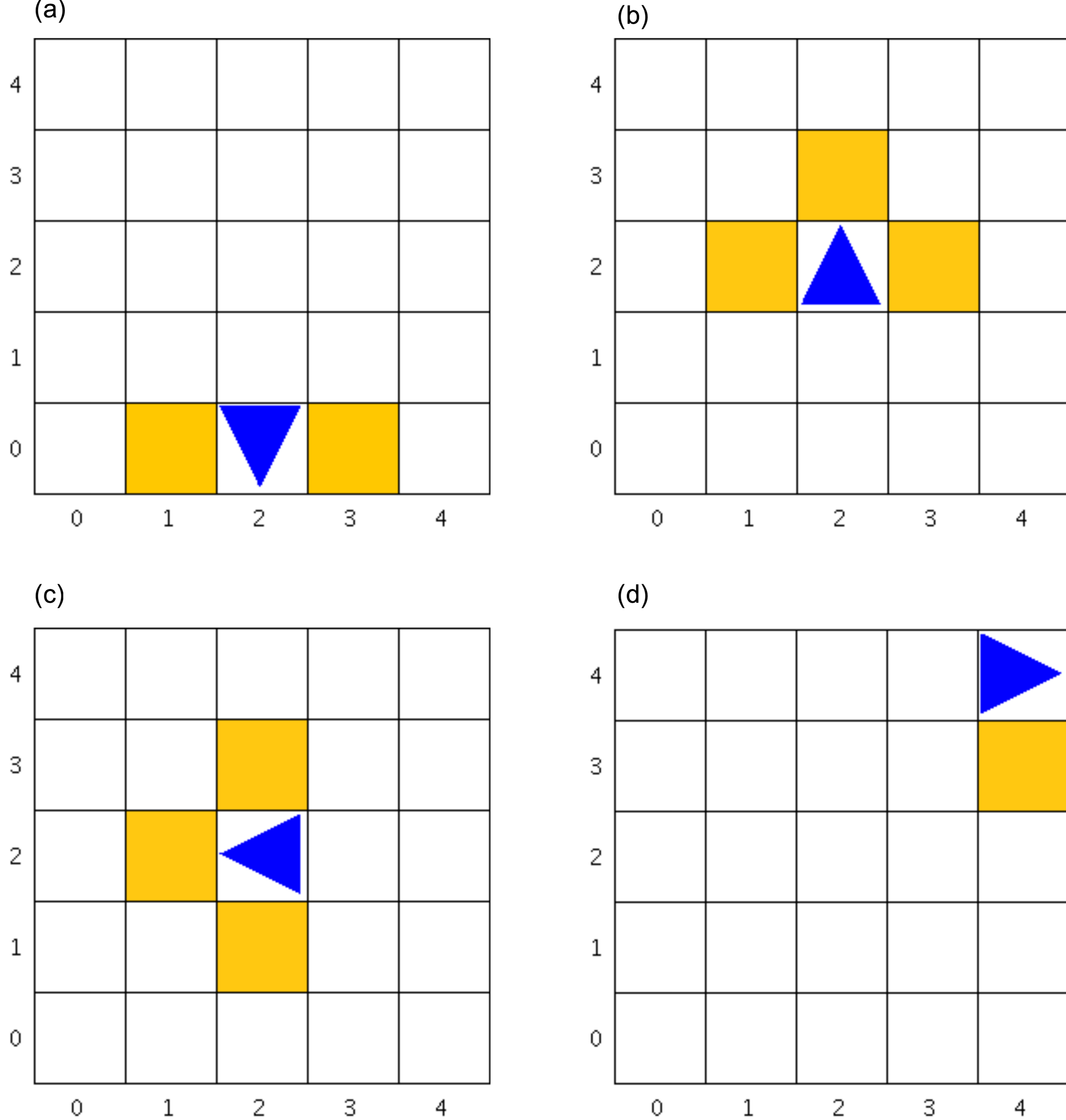
Fonte: Autoria Própria

5.3.3 Colisões Inevitáveis

Como foi abordado anteriormente, há cenários onde a colisão do veículo é inevitável a partir de sua posição atual, onde o mesmo irá buscar escolher uma colisão que cause o menor dano ao veículo. Tais situações possuem como pré-condição a existência de três obstáculos ao redor do agente. Ressalta-se que, como abordado 5.1, o agente nunca irá sair dentro dos limites da matriz de posição do ambiente e qualquer coordenada do tipo barreira é considerado um obstáculo pelo agente.

O ambiente implementa na função `setMAS()` (ver linhas 48-50) uma probabilidade T para que, com a presença da pré-condição, a colisão do veículo será inevitável. Na outra

Figura 46 – Pré-condições para a Ocorrência de Colisões



Fonte: Autoria Própria

porcentagem de casos definidos por F, mesmo com a pré-condição sendo verdadeira, o desvio de obstáculos será possível. Onde $\{T, F \in \mathbb{R} \mid 1 - T = F\}$.

A verificação da existência da pré-condição e da probabilidade do desvio não ser possível é implementado pelo método `scanSurroundings()` (ver linha 238), que é invocado sempre que a posição do agente é atualizada. As possíveis pré-condições, demonstradas na Figura 46, são as seguintes:

(a) Obstáculos ao leste, oeste, e sul do veículo, única posição sem obstáculo é ao norte. No exemplo dado, o veículo está localizado em `(2,0)`, e há obstáculos em `(1,0)` e `(3,0)`, e existe uma barreira ao sul;

(b) Obstáculos ao leste, oeste, e norte do veículo, única posição sem obstáculo é ao sul. No exemplo dado, o veículo está localizado em `(2,2)` e há obstáculos em `(1,2)`, `(2,3)` e `(3,2)`;

(c) Obstáculos ao norte, sul e oeste do veículo, única posição sem obstáculo é ao leste. No exemplo dado, o veículo está localizado em `(2,2)` e há obstáculos em `(1,2)`, `(2,3)` e `(2,1)`; e,

(d) Obstáculos ao norte, sul e leste do veículo, única posição sem obstáculo é ao oeste. No exemplo dado, o veículo está localizado em `(4,4)` e há um obstáculo em `(4,3)`, e existem barreiras ao leste e ao norte.

O ambiente define no método `setMAS()` (ver linhas 52-55) três tipos de danos que um obstáculo pode causar ao veículo: leve (`low`), médio (`moderate`) e grave (`high`). A classificação dos obstáculos é realizada pelo método `addObstacleDamage()` (ver linha 262), que também irá inserir as respectivas percepções sobre o nível de dano do obstáculo no agente. A cada uma destas categorias é atribuído um valor que determina a probabilidade de um obstáculo receber tal classificação, onde as chances de um obstáculo causar uma colisão:

- grave é definido por H;
- média é definido por M; e
- leve é definido por L.

E, $\{H, M, L \in \mathbb{R} \mid H + M + L = 1\}$. Como as chances de classificação de um obstáculo são irrelevantes para a tomada de decisão do agente.

No caso de uma direção não possuir um obstáculo, a mesma é considerada como obstruída temporariamente. Logo, o agente não possui opções de descolamento na qual não haverá uma colisão. Obstáculos do tipo barreira não são classificados, pois como o agente não deve sair da matriz de posições do ambiente tais coordenadas não devem ser consideradas como uma opção viável para minimizar o dano da colisão iminente. Na Figura 47 são utilizados os cenários da figura 46 para ilustrar a classificação de dano de obstáculos, onde:

(a) Os obstáculos são classificados por:

- dano leve, `(1,0)` e `(3,0)`; e,
- dano médio, `(2,1)`.

(b) Os obstáculos são classificados por:

- dano leve, `(1,2)` e `(3,2)`;
- dano médio, `(2,1)`; e,
- dano grave, `(2,3)`.

(c) Os obstáculos são classificados por:

- dano leve, `(2,1)`, `(2,3)` e `(3,2)`; e,

- dano grave, `(2,1)`.

(d) Os obstáculos são classificados por:

- dano leve, `(3,4)`; e,
- dano médio, `(4,3)`.

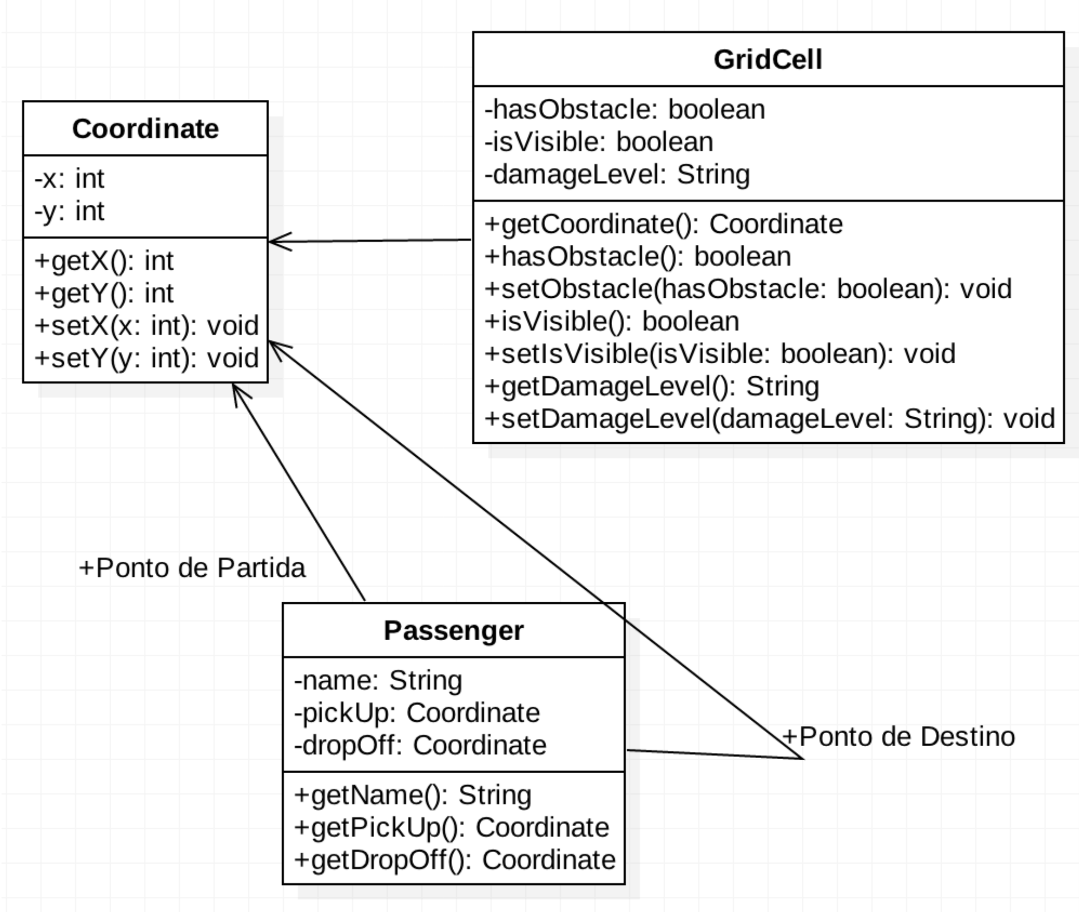
Figura 47 – Classificação de Dano de Obstáculo

Fonte: Autoria Própria

5.3.4 UTIL

Este módulo é composto por três classes: `Coordinate`, `GridCell` e `Passenger`. O diagrama de classes mostrado na Figura 48 descreve os elementos deste módulo e como estes se relacionam. A implementação desse módulo pode ser encontrada no Apêndice C - UTIL.

Figura 48 – Diagrama de Classes: Módulo UTIL



Fonte: Autoria Própria

A classe `Coordinate` (ver código 28) é utilizada para identificar a coordenada `(X,Y)` de uma posição, enquanto as instâncias de `GridCell` (ver código 29) representam células de uma matriz; onde é especificado por meio do atributo `hasObstacle` se aquela localização contém ou não uma obstáculo. Por virtude do simulador, cada célula pode estar visível ou não, atributo `isVisible`, representando o conhecimento obtido no decorrer dos percursos realizados pelo agente sobre a matriz. E, em situações de colisão eminentes, `damageLevel` especifica o dano causado pelo impacto do veículo com o obstáculo localizado em uma determinada coordenada. E por fim, os objetos do tipo `Passenger` (ver código 30) representam corridas de passageiros, por meio de um ponto de partida, `pickUp`, e um de destino, `dropOff`.

5.3.5 Simulador

Neste trabalho foi desenvolvido um simulador para representar graficamente o funcionamento do agente implementado em Gwendolen e sua respectiva interação com o ambiente.

A comunicação entre o ambiente e o simulador é feita por mensagens enviadas com intermédio da classe `Client`. Tais trocas são feitas ao decorrer da execução dos seguintes métodos

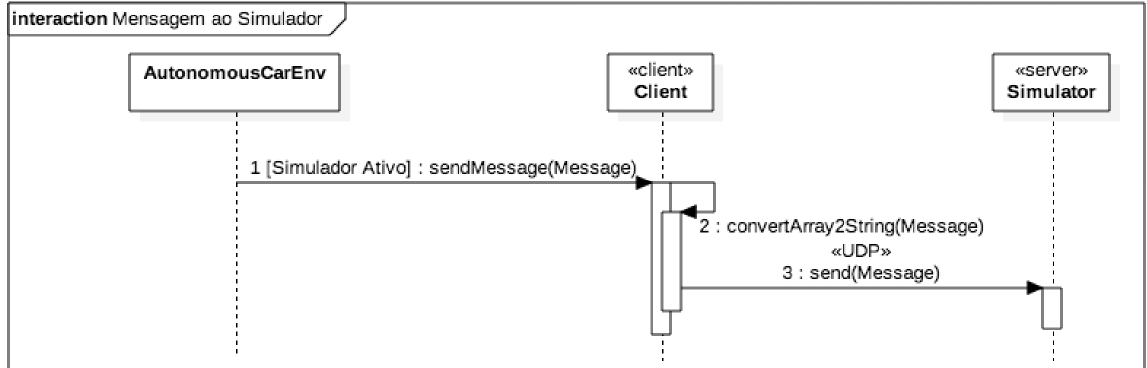
do ambiente (ver código 27):

- `addDepot()` (ver linha 459);
- `getRide()` (ver linhas 494-497);
- `updateLocation()` (ver linha 206);
- `verifyObstacle()` (ver linhas 314-315);
- `addObstacleDamage()` (ver linhas 290-291);
- `refuseRide()` (ver linha 424); e,
- `getAssistance()` (ver linha 544).

É possível determinar se esta troca deve acontecer a partir da variável `boolean simulate` (ver linha 40), enquanto `int waitTimeDefault` (ver linha 41) e `int waitTimeLocation` (ver linha 42) estabelecem a frequência de tempo entre transferências. Na figura 49 é apresentado o diagrama de sequência do envio de mensagem ao simulador pelo ambiente, onde este fluxo ocorre da seguinte forma:

1. O ambiente envia uma mensagem no formato de um vetor de `String` com os dados de uma atualização para a classe `Client`;
2. A mensagem recebida é convertida em `String`, em um padrão que será reconhecido pelo simulador;
3. e por fim, a classe `Client` transforma a mensagem do formato `String` para `Bytes`, e a destina este datagrama gerado para o simulador, utilizando um protocolo de rede UDP[23].

Figura 49 – Diagrama de Sequência: Comunicação Ambiente-Simulador



Fonte: Autoria Própria

[23] `User Datagram Protocol`, protocolo de rede que permite o envio de mensagens.

Como demonstrado no diagrama de implantação na figura 50, o ambiente envia mensagens ao simulador pela rede por meio da classe `Client` até o servidor contido em `Simulator`, portanto, essas classes se comunicam em uma relação cliente-servidor.

Figura 50 – Diagrama de Implantação: Comunicação Ambiente-Simulador

Fonte: Autoria Própria

As implementações desses módulos são disponibilizadas no Apêndice D. A conexão estabelecida para a comunicação cliente-servidor possui a seguinte configuração:

- Nome do endereço da rede: `localhost` (ver código 31, linha 21);
- Porta do soquete recebendo pacotes: `9999` (ver código 32, linha 322);
- Tamanho máximo da mensagem enviada pelo cliente: 1KB (ver código 31, linha 22);
- Tamanho máximo da mensagem recebida pelo servidor: 1KB (ver código 32, linha 323);

O cliente (classe `Client`) (ver código 31) implementa o método estático `sendMessage()` (ver linha 18), responsável por estabelecer a comunicação com o servidor, através de uma porta localizada na rede local, e pela configuração do envio dos dados. E, ao término de cada transmissão de mensagens, a conexão com o servidor é encerrada. A criação de uma instância da classe `Simulator` (ver código 32) inicializa o servidor por meio do método `main()` (ver linha 316). Então, o servidor começa uma rotina (ver linha 289) para receber mensagens enviadas pelo cliente até a sua porta de entrada.

As mensagens transmitidas do ambiente para o cliente são do tipo `String`. Para a transmissão, o cliente converte a mensagem para o formato `bytes` no método. No recebimento, o servidor transforma o conteúdo do tipo `bytes` recebido para `String` novamente e envia a mensagem para o método `readReceivedMessage()` (ver código 32, linha 203); este último é responsável pela interpretação das informações da mensagem.

Um padrão definido neste trabalho especifica o conteúdo de uma mensagem contento a identificação da atualização a ser realizado no simulador e todos os dados necessários para que isto ocorra; onde cada um desses fragmentos devem ser separados pelo caractere ponto-e-vírgula (`;`). No Quadro 3 é listado as mensagens que são transmitidas pelo ambiente e interpretadas pelo simulador no `readReceivedMessage()`.

Quadro 3 – Mensagens da Comunicação Ambiente-Simulador

Notificação	Identificação	Conteúdo Enviado	Mensagem
Nova simulação	`clear`	Tamanho N da matriz de posições	`clear;`N`;`
Localização do depósito	`depot`	Coordenada `(DX,DY)` do depósito	`depot;DX;DY;`
Nova posição do veículo	`carLocation`	Coordenada `(X,Y)` do veículo e direção `direction` em que o veículo se move	`carLocation;X;Y; direction;`
Obstáculo	`obstacle`	Coordenada `(OX,OY)` do obstáculo	`obstacle;OX;OY;`
Ponto de partida da corrida atual	`pickUp`	Coordenada `(PUX,PUY)` do ponto de partida	`pickUp;PUX;PUY;`
Ponto de destino da corrida atual	`dropOff`	Coordenada `(DOX,DOY)` do ponto de destino	`dropOff;DOX;DOY;`
Nível de dano causado por um obstáculo	`obstacle Damage`	Coordenada `(OX,OY)` do veículo e a classificação `damageLevel` referente ao dano	`obstacleDamage;OX;OY; damageLevel;`
Remover níveis de dano dos obstáculos	`remove Obstacle Damage`	-	`removeObstacle Damage;`
Recusar corrida atual	`refuseRide`	Coordenada `(X,Y)` do veículo e o motivo `refuseType` da recusa	`refuseRide;X;Y; refuseType;`

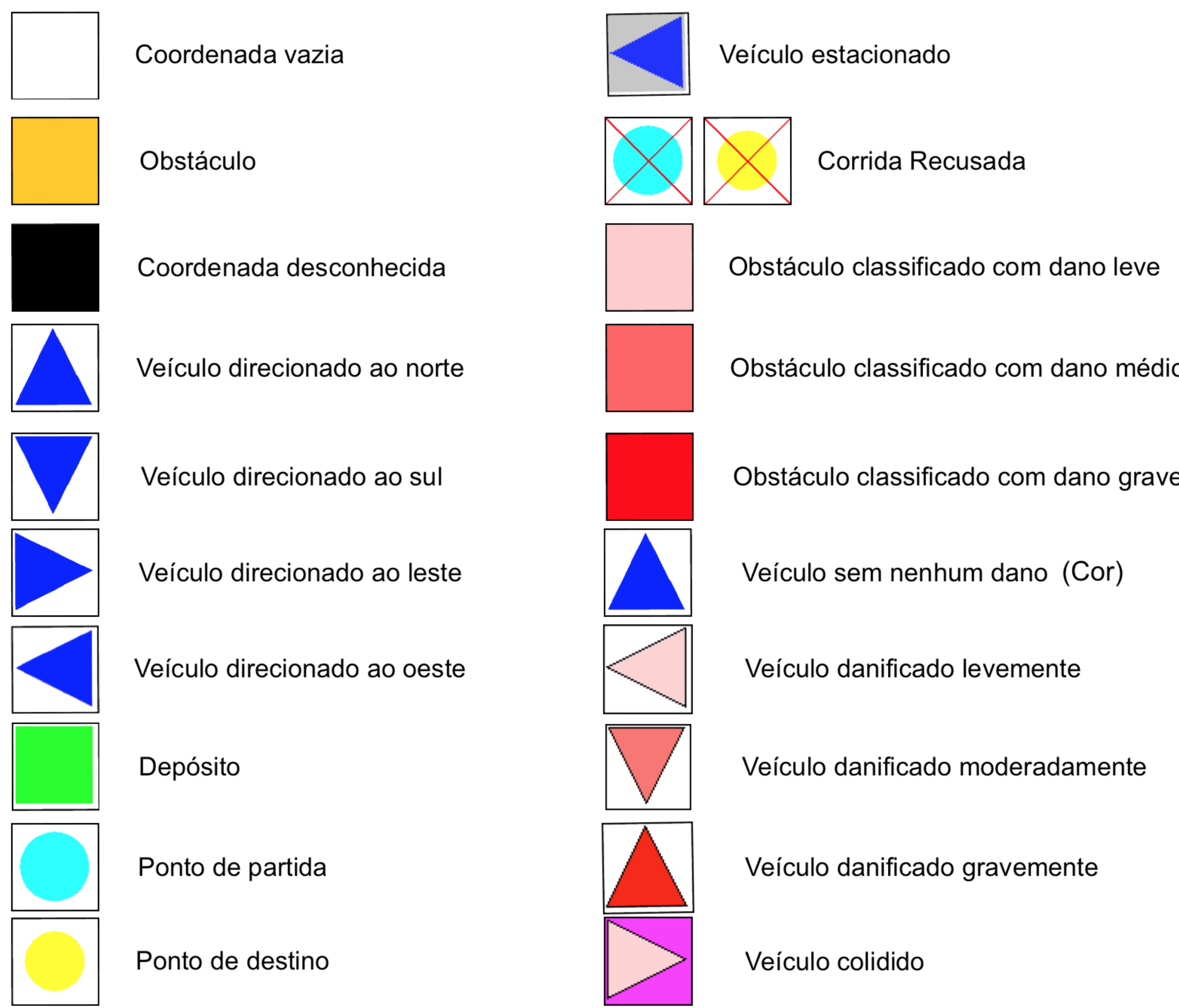
Fonte: Autoria Própria

A exibição gráfica do simulador é desenvolvida com o auxílio da biblioteca *2D Graphics* da linguagem Java. Além de atuar como um servidor de mensagens do ambiente, a classe `Simulator` (ver código 32) também é responsável pela criação de um janela que demonstra o funcionamento do agente; onde, conforme a interpretação de mensagens, a janela será atualizada

através do método (ver linha 95). Ressaltando que, não serão explorados neste trabalho a implementação referente ao desenho dos elementos do simulador na tela.

Na Figura 51 é apresentado a legenda do simulador, contendo o significado de possíveis condições das coordenadas da matriz de posições ambiente. Esclarecendo que a legenda define somente elementos básicos exibidos pelo simulador. Portanto sobreposições entre elementos podem ocorrer e devem ser interpretadas como a combinação de tais elementos. Em representação de elementos envolvendo o veículo, qualquer direção e quaisquer níveis de dano devem ser considerada.

Figura 51 – Legenda das Representações do Ambiente Criadas pelo Simulador



Fonte: Autoria Própria

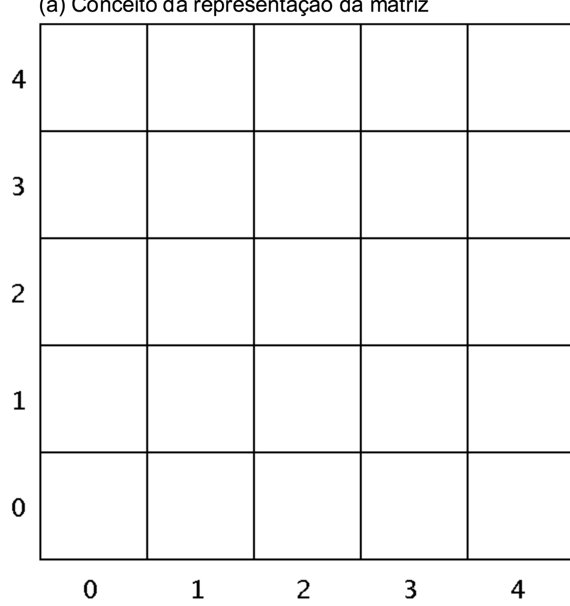
Na sequência desta seção, serão explorados em detalhes as representações gráficas referentes ao funcionamento do agente e as respectivas mensagens responsáveis por atualizar o simulador. Ao final da seção, é apresentado uma representação do funcionamento do agente no simulador.

Nova simulação

Uma mensagem identificada por `clear` (ver linha 216) envia ao simulador o tamanho N de uma matriz de posições para que as configurações iniciais da representação do ambiente sejam realizadas.

A reprodução da matriz do ambiente é feita por meio de um quadriculado de posições. A numeração dos eixos X e Y é informada respectivamente na horizontal inferior e na vertical esquerda da representação. Podemos observar no item (a) da Figura 52 o conceito de um ambiente cujo tamanho N é 5. No entanto, uma vez que o agente não iniciou sua execução, o mesmo não possui nenhuma informação referente ao ambiente onde está inserido. Consequentemente, visando realizar tal representação, posições da matriz exibida pelo simulador são preenchidas pela cor preta, significando que o agente não possui nenhuma crença referente aquelas posições, caracterizando assim, coordenadas desconhecidas. O item (b) da Figura 52 demonstra a janela real exibida pelo ambiente ao receber a mensagem *clear* e um tamanho 5 do ambiente.

Figura 52 – Representação da Matriz do Ambiente



Fonte: Autoria Própria

Localização do Depósito

Uma mensagem `depot` (ver linha 222) acompanhado quaisquer coordenadas `(X,Y)` da matriz de posições do informa ao simulador a localização do depósito do veículo. A representação deste elemento do ambiente é feita por meio de um quadrado na cor verde naquela posição informada. Na representação do ambiente pelo simulado demonstrado na figura 53, o depósito está localizado nas coordenadas `(0,0)`.

Figura 53 – Representação da Coordenada do Depósito

Fonte: Autoria Própria

Nova Posição do Veículo

Para atualizar a posição do veículo no simulador, espera-se que uma mensagem `car Location` (ver linha 227) seja enviada, juntamente com as novas coordenadas `(X,Y)` do agente e a direção `direction` na qual está se movendo. A cada nova posição do agente enviada ao simulador, aquela localização e as coordenadas ao seu redor[24] tem seu estado revelado, seja este livre ou obstruído por um obstáculo, e se manterá dessa forma. Isso visa a representação da base de crenças do agente referente à seus trajetos percorridos[25], que pode ser observado no exemplo na Figura 54, onde é demonstrado o trajeto do veículo entre as coordenadas `(0,0)` e `(2,3)`.

O veículo é representado pelo simulador como um triângulo na cor azul, onde o ângulo oposto a base deste trilátero está direcionado para a direção em que o agente se desloca. Desta forma, como representado na Figura 55, onde o veículo está localizado em `(2,2)` em todos os cenários apresentados, se o triângulo aponta para:

(a) Cima, significa que o agente está se movimentando na direção norte;

(b) Baixo, significa que o agente está se movimentando na direção sul;

(c) Direita, significa que o agente está se movimentando na direção leste;

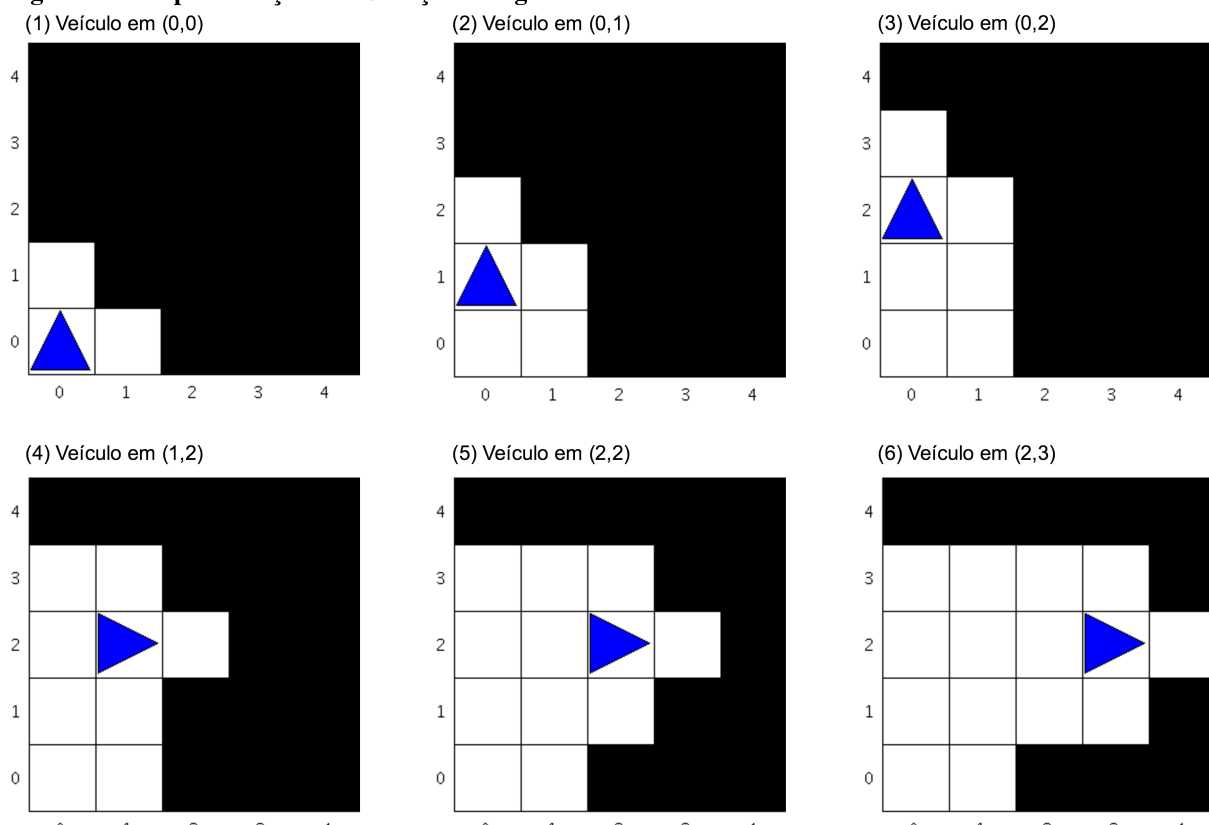
(d) Esquerda, significa que o agente está se movimentando na direção oeste.

---

[24] Nas direções norte, sul, leste e oeste.

[25] Abordado nas Seções 5.3.1 e 5.3.2

Figura 54 – Representação das Crenças do Agente Referentes ao Ambiente



Fonte: Autoria Própria

Obstáculo

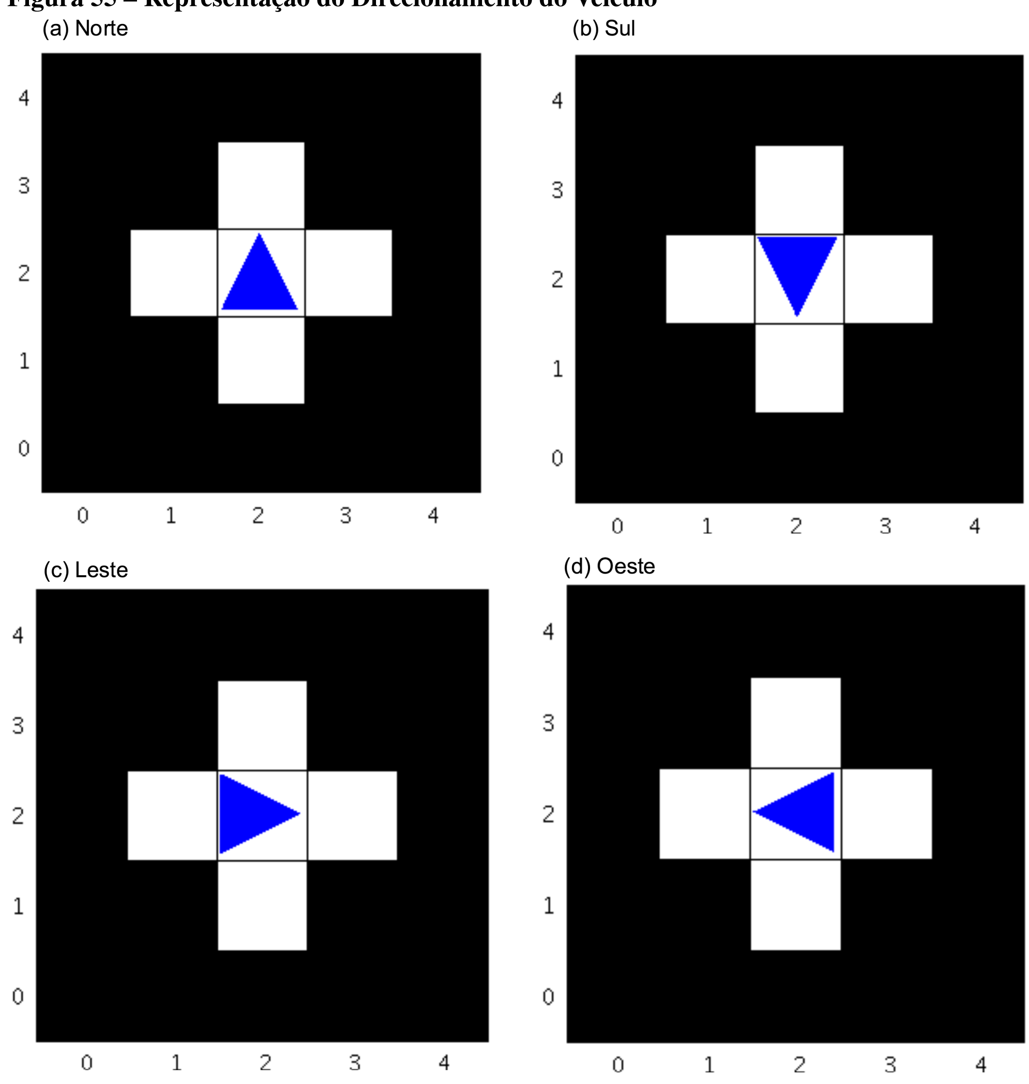
Todo novo obstáculo percebido pelo agente têm suas coordenadas `(X,Y)` enviadas ao simulador em uma mensagem identificada por `obstacle` (ver linha 270). Um obstáculo dentro do simulador é representado através do preenchimento de todo o quadriculado que identifica sua localização na cor laranja. O ambiente demonstrado na Figura 56 possui um agente nas coordenadas `(2,2)` e obstáculos ao norte `(2,3)` e ao leste `(3,2)` em relação a posição do veículo.

Na Figura 57 é demonstrado uma representação do agente obtendo percepções referentes a obstáculos ao seu redor conforme seu descolamento. Neste cenário apresentado, o agente realiza dois trajetos, primeiro das coordenadas `(0,0)` até `(4,4)`, e depois de `(4,4)` até `(3,2)`.

Ponto de Partida e Destino da Corrida Atual

Os pontos de partida e de destino referentes a corrida atual do agente são informados por meio das mensagens `pickUp` (ver linha 273) e `dropOff` (ver linha 278), respectivamente, acompanhados por coordenadas `(X,Y)` referentes as suas posições. Ambos os pontos são representados por uma circunferência, no entanto o ponto de partida possui a cor ciano, enquanto o

Figura 55 – Representação do Direcionamento do Veículo



Fonte: Autoria Própria

ponto de destino apresenta a cor amarelo. Na Figura 58 é possível observar um ponto de partida em `(1,1)` e um ponto de destino em `(4,4)`.

Nível de Dano Causado por um Obstáculo

O nível de dano que um obstáculo pode vir a causar ao veículo é notificado ao simulador por meio de uma mensagem `obstacleDamage` (ver linha 282), com sua posição `(X,Y)` e a categoria de seu possível estrago `damageLevel`.

Tais danos são representados pelo simulador por meio do preenchimento de todo o quadriculado da coordenada do obstáculo em diferentes tons da cor vermelho. Na Figura 59 é demonstrado uma situação [26] onde no próximo movimento do veículo localizado em `(2,2)` uma

[26] Vale lembrar que as situações contidas nesta figura são meramente ilustrativos, e podem haver cenários onde os

Figura 56 – Representação de Obstáculos

Fonte: Autoria Própria

colisão será inevitável, e:

- Os danos classificados como leve possuem a cor salmão para representá-los, como nas coordenadas `(2,1)` e `(1,2)`;
- Os danos classificados como médio possuem a cor cereja para representá-los, como na coordenada `(3,2)`;
- Os danos classificados como grave possuem a cor vermelho puro para representá-los, como na coordenada `(2,3)`.

Remover Níveis de Dano dos Obstáculos

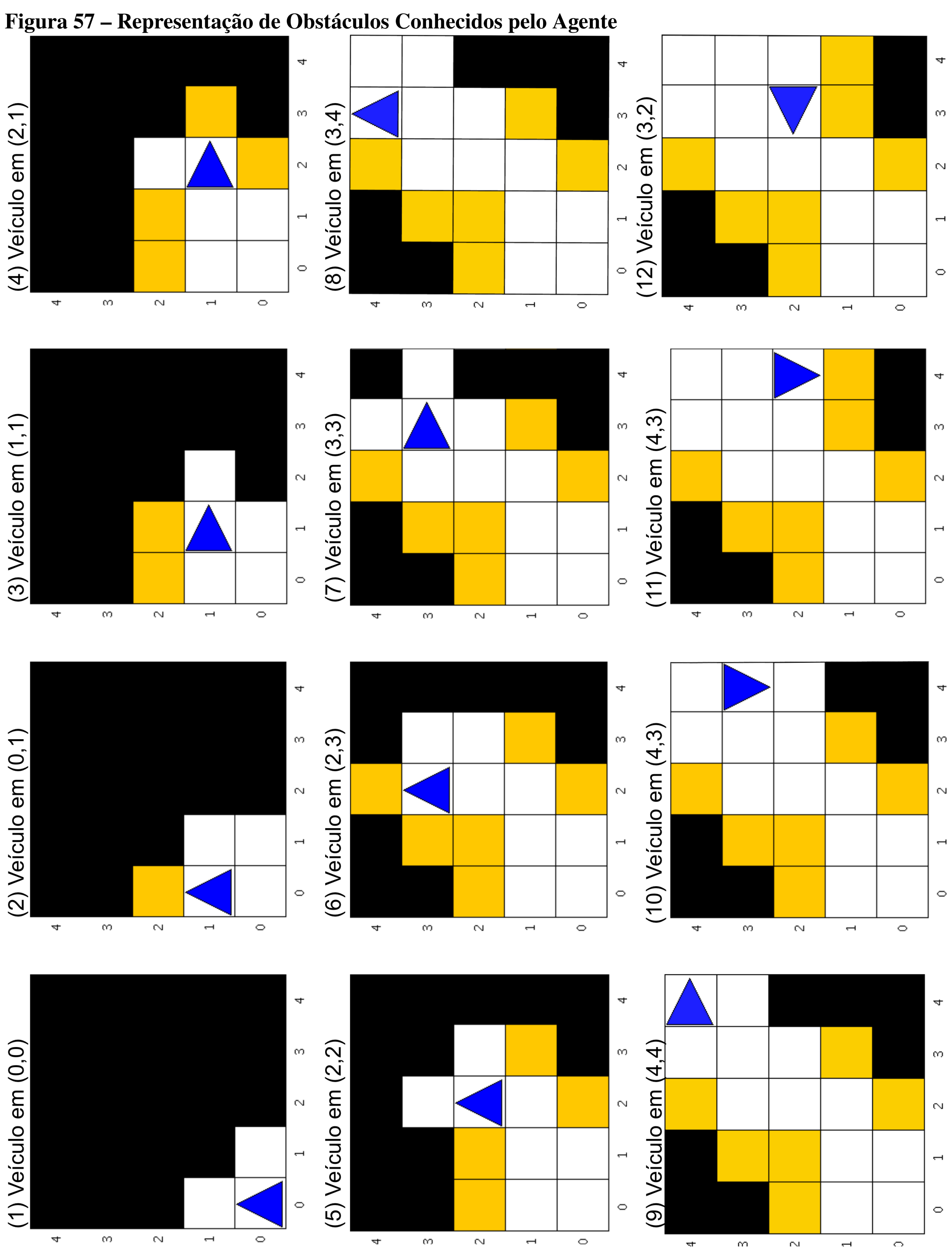
O ambiente irá enviar uma mensagem `removeObstacleDamage` (ver linha 285) para que o simulador remova a classificação de dano dos obstáculos.

Recusar Corrida Atual

Ao recusar uma corrida, uma mensagem `refuseRide` (ver linha) será enviada ao simulador com o motivo `refuseType` da recusa. Há três possíveis causas que ocasionam na recusa de uma corrida, estes são: ponto de partida inacessível (`pick_up`), ponto de destino inacessível (`drop_off`), ou veículo indisponível (`car_unavailable`).

obstáculos são classificados diferentemente uns aos outros.

Figura 57 – Representação de Obstáculos Conhecidos pelo Agente



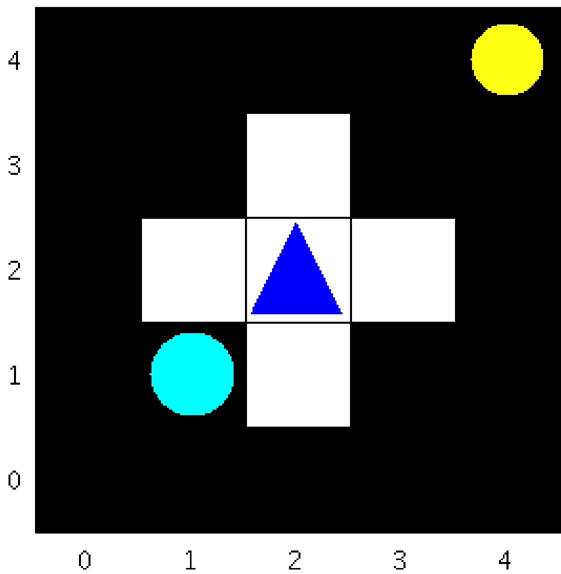
Fonte: Autoria Própria

Os dois primeiros casos são representados por um X na cor vermelha, nas coordenadas referentes aos pontos da corrida atual. Isto é demonstrado na Figura 60, onde o agente localizado em `(2,2)` percebe que:

(a) O ponto de partida `(1,2)` possui um obstáculo;

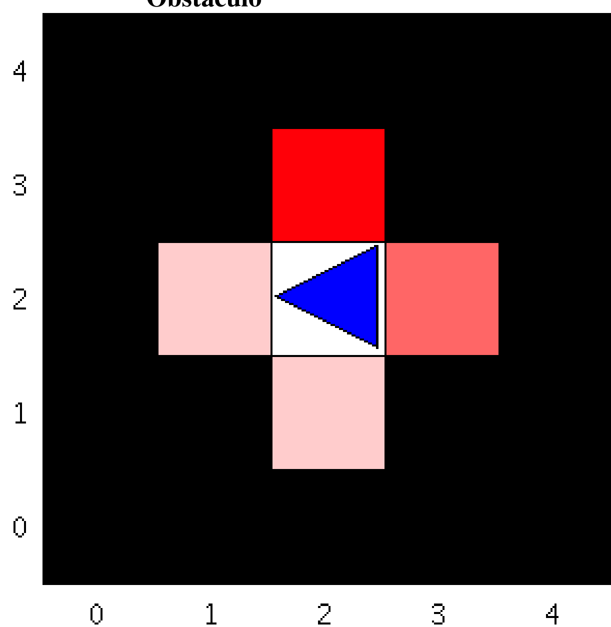
(b) há um obstáculo no ponto de destino localizado em `(1,2)`.

Figura 58 – Representação dos Pontos de Partida e de Destino



Fonte: Autoria Própria

Figura 59 – Representação dos Níveis de Dano de um Obstáculo

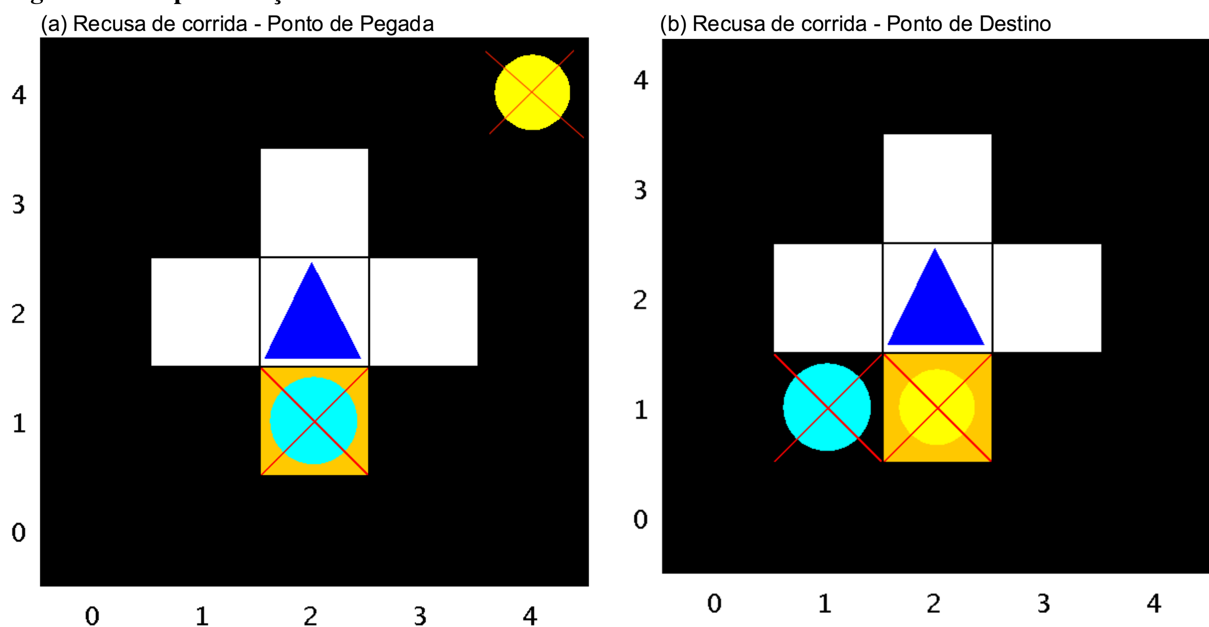


Fonte: Autoria Própria

A indisponibilização do veículo é ocasionada por uma colisão do tipo grave[27]. Esse tipo de situação é representada pelo simulador através do preenchimento de toda a matriz do ambiente na cor vermelho puro exceto a posição do veículo, que será preenchida na cor branca. Na Figura 61 é apresentado a janela do simulador após uma colisão grave do veículo nas coordenadas `(3,2)`.

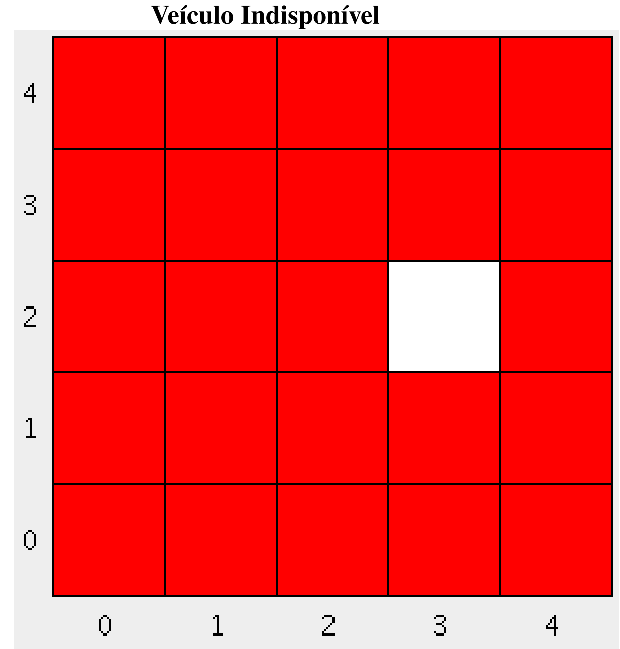
[27] Abordado na Seção 5.2.1

Figura 60 – Representação da Recusa de Corridas - Pontos de Partida e Destino Inacessíveis



Fonte: Autoria Própria

Figura 61 – Representação: Recusar a Corrida - Veículo Indisponível



Fonte: Autoria Própria

6 RESULTADOS

Através do desenvolvimento foi obtido como resultados um simulador capaz de interpretar mensagens e representar graficamente a interação agente-ambiente, e além disso, um agente modelo como um veículo autônomo. Com base em tal agente foram definidas propriedades que o mesmo deve possuir e que serão verificadas pelo *model checking* da ferramenta AJPF. Assim, a Seção 6.1 visa demonstrar o funcionamento do simulador e a seção 6.2 explora as propriedades do agente implementado, suas especificações formais e a verificação realizada pelo AJPF.

6.1 EXEMPLO DO FUNCIONAMENTO DO SIMULADOR

Baseado na legenda do simulador apresentada anteriormente na Figura 51 e do conteúdo abordado sobre o simulador, é apresentado a seguir, na Figura 62, o funcionamento do agente e a execução da simulação. Este cenário foi escolhido por demonstrar um conjunto significativo de elementos da simulação e do processo de tomada de decisão do agente. Neste cenário:

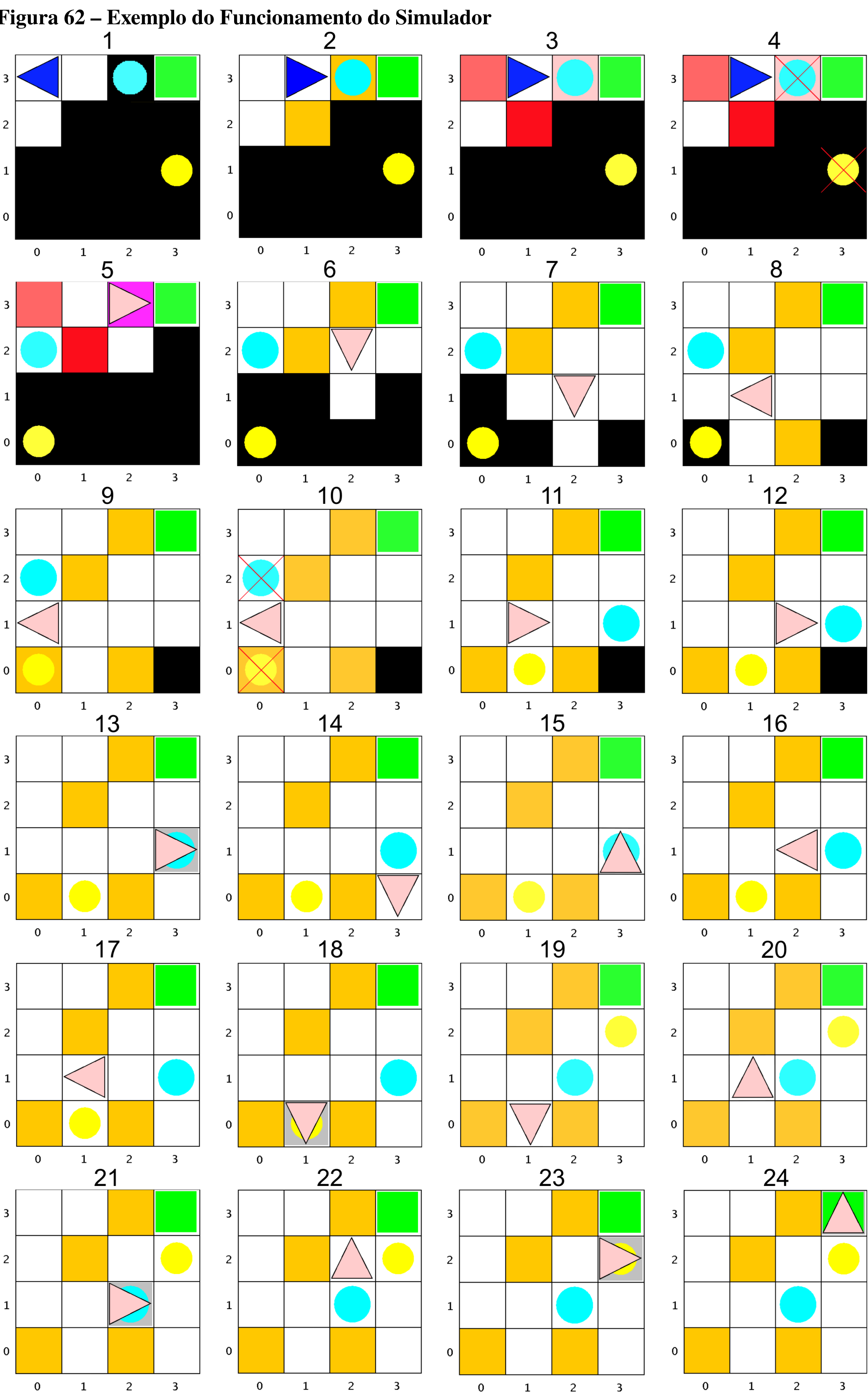
1. Agente inicializa sua execução, recebendo sua posição inicial, a localização do depósito, e uma nova corrida;
2. O agente direciona sua rota para o sentido leste para tentar chegar no ponto de pegada e identifica obstáculos nas direções leste e sul, inclusive percebendo que o ponto da corrida atual está obstruído. O cenário atual apresenta uma pré-condição para uma colisão inevitável;
3. O veículo irá colidir em seu próximo movimento. Os obstáculos ao seu redor possuem as classificações leve (leste), médio (oeste) e grave (sul). Note que o norte é uma barreira.
4. A corrida atual é recusada, pois o ponto de partida possui um obstáculo[1].
5. O veículo movimenta-se para o leste e colide com o obstáculo, ficando levemente danificado, onde há uma mudança na cor de sua representação. O dano causado ao veículo estará presente até o fim de todas as jornadas. Após o controle da colisão, o agente recebe uma nova corrida;
6. O agente move-se ao sul para chegar no ponto de partida da nova corrida;
7. Embora o ponto de partida esteja estritamente ao oeste do veículo, há uma obstáculo naquela direção. O agente realiza o desvio de obstáculo adaptando sua rota para o sul;

[1] Não está relacionado com o fato de uma colisão ser inevitável.

8. O agente consegue finalizar o plano de adaptação de rota deslocando-se para o oeste, em direção ao destino de seu trajeto atual. A direção do destino atual é atualizada;
9. Como não foi possível se descolar para o norte[2], direção do seu destino, o agente adapta sua rota indo para o oeste;
10. O ponto de destino da corrida atual está obstruído por um obstáculo, então a corrida é recusada.
11. Após receber uma nova corrida, o agente move-se para o leste, em direção ao ponto de partida;
12. Veículo continua se movendo ao leste pois não há nada obstruindo aquela direção;
13. Agente chega no ponto de partida e estaciona o veículo para o embarque do passageiro;
14. O agente se descola ao sul, buscando chegar no ponto de destino;
15. A rota atual é adaptada e o veículo move-se para o norte, a única direção disponível;
16. O agente concluí a adaptação da rota pois foi possível se deslocar para o oeste;
17. Por meio uma nova adaptação, o veículo continua movendo-se ao oeste;
18. Depois de um movimento ao sul, o agente chega no ponto de destino da sua corrida atual e estaciona para o desembarque de seu passageiro;
19. O agente recebe uma nova corrida;
20. O veículo move-se ao norte, em direção ao ponto de partida atual;
21. Após de um movimento ao leste, o veículo chega no ponto de partida e estacionado para o embarque do passageiro;
22. O veículo move-se ao norte, em direção ao ponto de destino atual;
23. O agente realiza um movimento ao leste para atingir o ponto de destino, e estaciona o veículo para o desembarque;
24. Como o veículo está levemente danificado[3], o agente realiza um trajeto até o depósito.

[2] O agente dá preferência por direções na vertical.
[3] Caso o veículo estivesse com um nível de dano médio ou grave, sua executa teria sido encerrada anteriormente

Figura 62 – Exemplo do Funcionamento do Simulador



Fonte: Autoria Própria

6.2 VERIFICAÇÃO FORMAL

A verificação formal das especificações do agente é necessária para garantir que o mesmo está executando seus planos corretamente. Busca-se confirmar o funcionamento correto de planos e objetivos essenciais para a execução do agente.

Todas as propriedades verificadas nesta seção produzem uma árvore degenerada de estados do ambiente, exceto pela propriedade descrita na subseção 6.2.8: Colisões Inevitáveis, onde são consideradas as diversas probabilidades de classificação de obstáculos e a árvore possui diversas ramificações.

Certas verificações têm o seu cenário delimitado pelos seguintes motivos: hardware disponível para execução de testes e pelo fato do AJPF não permitir o uso de variáveis nas especificações formais.

No primeiro caso, a limitação do hardware disponível para a realização deste trabalho, não permite a verificação de cenários onde vários estados do ambiente são considerados. A máquina disponível possui as seguintes especificações de hardware: MacBook Pro (Modelo *Early* 2015); Processador Intel Core i5 2.7 Ghz; e, Memória RAM DDR3 1867 MHz 8 GB.

Já a segunda limitação é imposta pela base do AJPF, o *JPF*, que não permite o uso de variáveis na verificação, obrigando a declaração explícita de valores ou sua omissão. No entanto, apesar do uso de valores nas especificações, o conceito, é aplicável para qualquer valor. Devido a semelhança entre alguns planos e especificações do agente, como os que envolvem o embarque e desembarque de veículo, determinadas verificações serão suprimidas.

As seções a seguir irão abordar as especificações formais do agente, os cenários onde houve verificação e o resultado obtido AJPF. Nota-se que, o código do agente implementado por este trabalho é identificado como `vehicle` e, o formato utilizado para representar corridas de passageiros é: (Coordenada do ponto de partida - Coordenada do ponto de destino). Cada especificação apresentada será acompanhada por dados relevantes ao cenário utilizado. E por fim, será abordado uma situação onde a verificação formal falha e como investigar o erro através da ferramenta. O arquivo `psl` com os códigos de cada especificação pode ser encontrado no Código 33 do Apêndice E Verificação Formal. Opta-se pela omissão de parte dos dados gerados pelo *log* da ferramenta AJPF, devido a ser uma quantidade extensiva de informações que não são necessárias neste trabalho. Vale ressaltar que, **todas** propriedades verificadas aqui demonstraram-se presentes no agente proposto por este trabalho[4] após a verificação formal.

[4] Apresentado anteriormente na Seção 5.2.

6.2.1 Localizar Veículo no Ambiente

Esta verificação busca afirmar a seguinte propriedade: "A qualquer momento que o agente venha a executar uma ação para detectar sua localização, então haverá uma resposta do ambiente em algum momento sobre uma percepção referente a suas coordenadas atuais `(0,0)`". O Quadro 4 apresenta os dados referentes a esta especificação.

Quadro 4 – Verificação Formal do AJPF: Localizar Veículo no Ambiente

Cenário Utilizado	
Propriedade	**Valor**
Posição inicial do agente	`(0,0)`
Número de passageiros	0
Número de obstáculos	0
Verificação realizada pelo AJPF	
Especificação formal `FAR_1`: $\Diamond A_{vehicle} location \implies \Diamond P(at(0,0))$	
Propriedade	***Feedback***
Número de estados do ambiente	46
Número de estados do autômato de Büchi gerado pelo AJPF	3
Tempo gasto na execução	00:00:06
Memória RAM utilizada	228MB

Fonte: Autoria Própria

6.2.2 Buscar uma Nova Corrida

Esta verificação busca afirmar a seguinte propriedade: "Durante toda sua a execução, o agente sempre irá executar uma ação para obter uma nova corrida assim que perceber que não possui informações sobre uma". O Quadro 5 apresenta os dados referentes a esta especificação.

6.2.3 Embarque de Passageiros

Esta verificação busca afirmar a seguinte propriedade: "O veículo irá tentar completar um trajeto até `(1,1)` a partir da sua coordenada atual caso em algum momento o agente receba uma corrida com o ponto de partida `(1,1)`. Então, em algum momento o veículo estará localizado naquela coordenada e estacionará o veículo para o embarque do passageiro". O Quadro 6 apresenta os dados referentes à esta especificação.

Note que é possível fazer a verificação do desembarque dos passageiros, porém opta-se por não a demonstrar devido a sua similaridade com o embarque.

Quadro 5 – Verificação Formal do AJPF: Buscar uma Nova Corrida

Cenário Utilizado	
Propriedade	**Valor**
Posição inicial do agente	`(1,0)`
Número de passageiros	2
Coordenadas dos Passageiros	{ `((1,1)-(1,2))`, `((1,3)-(1,4))` }
Número de obstáculos	0
Verificação realizada pelo AJPF	
Especificação formal `FAR_6`: $\square(\neg B_{vehicle} ride_info\ U\ A_{vehicle} get_ride)$	
Propriedade	***Feedback***
Número de estados do ambiente	738
Número de estados do autômato de Büchi gerado pelo AJPF	7
Tempo gasto na execução	00:03:35
Memória RAM utilizada	2895MB

Fonte: Autoria Própria

Quadro 6 – Verificação Formal do AJPF: Embarque de Passageiros

Cenário Utilizado	
Propriedade	**Valor**
Posição inicial do agente	`(1,0)`
Número de passageiros	1
Coordenadas dos Passageiros	{ `((1,1)-(1,2))` }
Número de obstáculos	0
Verificação realizada pelo AJPF	
Especificação formal `FAR_8_1_8_2`: $(\diamond B_{vehicle} pick_up(1,1) \implies \diamond G_{vehicle} complete_journey(1,1)) \implies (B_{vehicle} at(1,1) \wedge A_{vehicle} park(pick_up))$	
Propriedade	***Feedback***
Número de estados do ambiente	187
Número de estados do autômato de Büchi gerado pelo AJPF	15
Tempo gasto na execução	00:00:34
Memória RAM utilizada	1013MB

Fonte: Autoria Própria

6.2.4 Recusar Nova Corrida

O objetivo dessa verificação é garantir que o agente irá recusar corridas cujo ponto de partida está obstruído por um obstáculo. Na ocorrência de um obstáculo no ponto de partida, esta verificação busca afirmar a seguinte propriedade: "Se o agente em algum momento receber uma nova corrida com o ponto de partida em `(1,2)` e já souber da existência de um obstáculo na coordenada `(1,2)`, então, a corrida será recusada". O Quadro 7 apresenta os dados referentes a esta especificação.

Quadro 7 – Verificação Formal do AJPF: Recusar Nova Corrida

Cenário Utilizado	
Propriedade	**Valor**
Posição inicial do agente	`(1,0)`
Número de passageiros	2
Coordenadas dos Passageiros	{ `((1,1)-(2,1))`, `((1,2)-(2,1))` }
Número de obstáculos	1
Coordenadas dos obstáculos	{`(1,2)`}
Verificação realizada pelo AJPF	
Especificação formal `FAR_7`: $\Diamond(B_{vehicle} obstacle(center, 1, 2) \land P(pick_up(1, 2))) \implies \Diamond A_{vehicle} refuse_ride(pick_up)$	
Propriedade	***Feedback***
Número de estados do ambiente	416
Número de estados do autômato de Büchi gerado pelo AJPF	11
Tempo gasto na execução	00:01:07
Memória RAM utilizada	1940MB

Fonte: Autoria Própria

6.2.5 Recusar Corrida Atual

Esta verificação busca afirmar a seguinte propriedade: "A corrida atual será recusada pelo agente se, a qualquer momento durante o trajeto até o ponto de partida em `(1,2)` de tal corrida o agente perceber que há um obstáculo naquela coordenada". O Quadro 8 apresenta os dados referentes a esta especificação.

Corridas também podem ser recusadas pela existência de obstáculos em seu ponto de destino, no entanto, devido a sua similaridade com a recusa de corridas pela obstrução do ponto de partida, essa verificação não será demonstrada.

Quadro 8 – Verificação Formal do AJPF: Recusar Corrida Atual

Cenário Utilizado	
Propriedade	**Valor**
Posição inicial do agente	`(1,0)`
Número de passageiros	1
Coordenadas dos Passageiros	{ `((1,2)-(1,3))` }
Número de obstáculos	1
Coordenadas dos obstáculos	{`(1,2)`}

Verificação realizada pelo AJPF	
Especificação formal `FAR_8_1_3`: $\Diamond(B_{vehicle} pick_up(1,2) \wedge G_{vehicle} complete_up(1,2) \wedge P(obstacle(center,1,2))) \implies \Diamond A_{vehicle} refuse_ride(pick_up)$	
Propriedade	***Feedback***
Número de estados do ambiente	202
Número de estados do autômato de Büchi gerado pelo AJPF	4
Tempo gasto na execução	00:00:38
Memória RAM utilizada	1069MB

Fonte: Autoria Própria

6.2.6 Concluir Trajetos

Aqui não se verifica as direções que o agente deve se mover explicitamente, e sim, são impostos trajetos que obriguem o veículo a deslocar-se em determinados sentidos. As verificações realizadas buscam afirmar a seguinte propriedade: "O veículo irá realizar os movimentos necessários para se locomover dentro do ambiente e concluir seus percursos". Todas as especificações demonstradas nesta seção verificam que se em algum momento, o agente adquirir um objetivo para se movimentar até determinada coordenada, o veículo terá que se deslocar até aquela posição. Vale ressaltar que, para cada cenário abordado aqui, o agente irá realizar todos os percursos disponíveis até que uma especificação seja verificada. Verificações dentro de um mesmo cenário são executadas sequencialmente.

No primeiro cenário proposto apresentado no Quadro 9, o veículo possui as seguintes corridas a serem realizadas:

- Corrida 1: com ponto de partida em `(0, 0)` e ponto de destino em `(0, 2)`, onde o veículo deverá mover-se estritamente ao norte;
- Corrida 2: com ponto de partida em `(2, 2)` e ponto de destino em `(2, 0)`, onde o veículo deverá mover-se estritamente ao sul;
- Corrida 3: com ponto de partida em `(0, 2)` e ponto de destino em `(2, 2)`, onde o veículo deverá mover-se estritamente ao leste; e,

- Corrida 4: com ponto de partida em `(2, 0)` e ponto de destino em `(1, 0)`, onde o veículo deverá mover-se estritamente ao oeste.

Quadro 9 – Verificação Formal do AJPF: Concluir Trajeto - Movimentação em Somente uma Direção

Cenário Utilizado	
Propriedade	**Valor**
Posição inicial do agente	`(0,0)`
Número de passageiros	4
Coordenadas dos Passageiros	`{((0,0)-(0,2)), ((0,2)-(2,2)), ((2,2)-(2,0)),((2,0)-(1,0))}`
Número de obstáculos	0
Verificação realizada pelo AJPF	
1. Especificação formal `DT_DIRECTIONS_NORTH`: $\lozenge G_{vehicle} drive_to(0,2) \implies \lozenge P(at(0,2))$	
Propriedade	***Feedback***
Número de estados do ambiente	244
Número de estados do autômato de Büchi gerado pelo AJPF	4
Tempo gasto na execução	00:00:53
Memória RAM utilizada	1391MB
2. Especificação formal `DT_DIRECTIONS_EAST`: $\lozenge G_{vehicle} drive_to(2,2) \implies \lozenge P(at(2,2))$	
Propriedade	***Feedback***
Número de estados do ambiente	526
Número de estados do autômato de Büchi gerado pelo AJPF	4
Tempo gasto na execução	00:01:43
Memória RAM utilizada	2193MB
3. Especificação formal `DT_DIRECTIONS_SOUTH`: $\lozenge G_{vehicle} drive_to(2,0) \implies \lozenge P(at(2,0))$	
Propriedade	***Feedback***
Número de estados do ambiente	808
Número de estados do autômato de Büchi gerado pelo AJPF	4
Tempo gasto na execução	00:03:05
Memória RAM utilizada	2617MB
4. Especificação formal `DT_DIRECTIONS_WEST`: $\lozenge G_{vehicle} drive_to(1,0) \implies \lozenge P(at(1,0))$	
Propriedade	***Feedback***
Número de estados do ambiente	1064
Número de estados do autômato de Büchi gerado pelo AJPF	4
Tempo gasto na execução	00:04:16
Memória RAM utilizada	3207MB

Fonte: Autoria Própria

O segundo cenário abordado, Quadro 10, propõe que o veículo realize os seguintes percursos:

- Corrida 1: com ponto de partida em `(2, 0)` e ponto de destino em `(4, 3)`, onde o veículo deverá mover-se nas direções norte e leste;
- Corrida 2: com ponto de partida em `(4, 3)` e ponto de destino em `(2, 1)`, onde o veículo deverá mover-se nas direções sul e oeste;
- Corrida 3: com ponto de partida em `(2, 1)` e ponto de destino em `(0, 3)`, onde o veículo deverá mover-se nas direções norte e oeste; e,
- Corrida 4: com ponto de partida em `(0, 3)` e ponto de destino em `(1, 1)`, onde o veículo deverá mover-se nas direções sul e leste;

6.2.7 Desvio de Obstáculos

As verificações dos planos de desvio de obstáculo serão agrupados em razão de sua semelhança; nesses casos, a alteração das direções de uma especificação formal para que um novo plano do mesmo grupo seja verificado.

O primeiro grupo a ser verificado são as adaptações de um percurso que possa vir se conveniente em relação ao destino do trajeto atual do agente. Este grupo consiste dos planos `AR 5.1`, `AR 5.2`, `AR 5.3` e `AR 5.4` apresentados no Código 20 (ver linhas 40, 43, 46 e 49) na Subseção 5.2.3, página 88.

No Quadro 11 são apresentados os dados referentes a esta especificação, e neste exemplo a verificação busca afirmar que: Durante toda a execução do agente, se houver uma percepção de um obstáculo bloqueando o caminho do veículo ao norte enquanto o mesmo se desloca para esta direção e o destino de seu trajeto estiver localizado nas direções norte e leste da sua posição atual, então o agente irá adaptar sua rota ao leste até que seja possível se mover para o norte novamente; e com isso, em nenhum momento haverá uma colisão com um obstáculo. Considerando o cenário utilizado, esta verificação só será desencadeada quando o agente receber uma percepção da presença de um obstáculo ao norte da coordenada `(1,1)`.

Esta próxima verificação busca afirmar que o agente consegue encontrar novas rotas para finalizar um trajeto por meio da movimentação em direções alheias ao seu destino. Este grupo consiste dos planos `AR 6.1`, `AR 6.2`, `AR 6.3` e `AR 6.4` apresentados no Código 20 (ver linhas 53, 56, 59 e 62) na Subseção 5.2.3, página 88.

Essa verificação é demonstrada no Quadro 12. A especificação diz que: Durante toda a execução do agente, se a seguinte situação ocorrer: há uma percepção sobre um obstáculo bloqueando o caminho do veículo ao sul enquanto o mesmo se desloca para esta direção e o destino de seu trajeto estiver localizado estritamente ao sul da sua posição atual, não requerendo uma movimentação na horizontal, mas a direção leste está obstruída e o sentido oeste livre; então, o agente irá adaptar sua rota ao oeste até que seja possível se mover novamente ao sul.

Quadro 10 – Verificação Formal do AJPF: Concluir Trajeto - Movimentação em Duas Direções

Cenário Utilizado	
Propriedade	**Valor**
Posição inicial do agente	`(0,0)`
Número de passageiros	4
Coordenadas dos Passageiros	`{((2,0)-(4,3)), ((4,3)-(2,1)), ((2,2)-(2,0)),((2,0)-(1,0))}`
Número de obstáculos	0
Verificação realizada pelo AJPF	
1. Especificação formal `DT_DIRECTIONS_NORTH_EAST`: $\Diamond G_{vehicle} drive_to(4,3)\ R\ \Diamond P(at(4,3))$	
Propriedade	***Feedback***
Número de estados do ambiente	333
Número de estados do autômato de Büchi gerado pelo AJPF	6
Tempo gasto na execução	00:01:39
Memória RAM utilizada	1522MB
2. Especificação formal `DT_DIRECTIONS_SOUTH_WEST`: $\Diamond G_{vehicle} drive_to(2,1)\ R\ \Diamond P(at(2,1))$	
Propriedade	***Feedback***
Número de estados do ambiente	808
Número de estados do autômato de Büchi gerado pelo AJPF	6
Tempo gasto na execução	00:03:13
Memória RAM utilizada	3097MB
3. Especificação formal `DT_DIRECTIONS_NORTH_WEST`: $\Diamond G_{vehicle} drive_to(0,3)\ R\ \Diamond P(at(0,3))$	
Propriedade	***Feedback***
Número de estados do ambiente	1025
Número de estados do autômato de Büchi gerado pelo AJPF	6
Tempo gasto na execução	00:05:45
Memória RAM utilizada	3599MB
4. Especificação formal `DT_DIRECTIONS_SOUTH_EAST`: $\Diamond G_{vehicle} drive_to(1,1)\ R\ \Diamond P(at(1,1))$	
Propriedade	***Feedback***
Número de estados do ambiente	1345
Número de estados do autômato de Büchi gerado pelo AJPF	6
Tempo gasto na execução	00:09:12
Memória RAM utilizada	3942MB

Fonte: Autoria Própria

Quadro 11 – Verificação Formal do AJPF: Desvio de Obstáculos, Adaptações Convenientes

Cenário Utilizado	
Propriedade	**Valor**
Posição inicial do agente	(1,0)
Número de passageiros	1
Coordenadas dos Passageiros	{ ((1,0)-(2,2)) }
Número de obstáculos	1
Coordenadas dos obstáculos	{(1,2)}

Verificação realizada pelo AJPF	
Especificação formal AR_5: $\square(((B_{vehicle}east \wedge B_{vehicle}north \wedge B_{vehicle}heading(north) \wedge B_{vehicle}obstacle(north,_,_)) \implies (\Diamond B_{vehicle}adapt(east)\ R\ \Diamond A_{vehicle}drive(_,_,north,_,_))) \wedge (\neg B_{vehicle}crashed(_,_)))$	
Propriedade	***Feedback***
Número de estados do ambiente	458
Número de estados do autômato de Büchi gerado pelo AJPF	10
Tempo gasto na execução	00:01:46
Memória RAM utilizada	1546MB

Fonte: Autoria Própria

Dessa forma, em nenhum momento haverá uma colisão do veículo com um obstáculo. No cenário utilizado, esta verificação só será desencadeada quando o agente obtém uma percepção referente a presença do obstáculo ao sul da coordenada (1,3).

O terceiro grupo de verificação de desvio de obstáculos faz uma análise sobre a capacidade do veículo de se recuperar de uma tentativa falha de se adaptar e criar uma nova adaptação para trajeto atual. Este grupo consiste dos planos AR 8.1, AR 8.2, AR 8.3 e AR 8.4 apresentados no Código 20 (ver linhas 88, 92, 96 e 100) na Subseção 5.2.3, página 88.

No Quadro 13 são apresentados os dados da verificação e a especificação, que busca afirmar que: Se o agente agente estiver no meio de uma adaptação de rota ao norte e for percebido que aquela direção está obstruída por um obstáculo (caracterizando-se uma tentativa falha), então o agente retornará a coordenada anterior movendo-se ao sul e tentará adaptar seu trajeto atual ao sul para alcançar seu destino ao leste. Como a verificação de que o veículo nunca irá colidir já foi realizada para os grupos anteriores, não se vê necessário demonstra-la novamente.

E finalmente, o último grupo verifica a capacidade do agente detectar a necessidade de executar uma movimentação desfavorável a seu trajeto atual ao adaptar sua rota atual para evitar múltiplos obstáculos em seu redor. Tais cenários são caracterizados como becos sem saída, situações onde o agente possui obstáculos em três das quatro direções disponíveis para movimentação. Este grupo consiste dos planos AR 7.1, AR 7.2, AR 7.3 e AR 7.4 apresentados no Código 20 (ver linhas 72, 77, 82 e 87) na Subseção 5.2.3, página 88.

Essa verificação é demonstrada no Quadro 14, onde se busca afirmar que: "Se o destino

Quadro 12 – Verificação Formal do AJPF: Desvio de Obstáculos, Adaptações Estritas

Cenário Utilizado	
Propriedade	**Valor**
Posição inicial do agente	`(1,3)`
Número de passageiros	1
Coordenadas dos Passageiros	{ `((1,3)-(1,0))` }
Número de obstáculos	2
Coordenadas dos obstáculos	{`(1,2), (2,3)`}
Verificação realizada pelo AJPF	
Especificação formal `AR_6`: $\square(((B_{vehicle} south \land \neg B_{vehicle} east \land \neg B_{vehicle} west \land B_{vehicle} heading(south) \land B_{vehicle} obstacle(south, _, _) \land B_{vehicle} obstacle(east, _, _) \land \neg B_{vehicle} obstacle(west, _, _)) \implies (\diamondsuit B_{vehicle} adapt(west)\ R\ \diamondsuit A_{vehicle} drive(_, _, south, _, _))) \land (\neg B_{vehicle} crashed(_, _)))$	
Propriedade	***Feedback***
Número de estados do ambiente	561
Número de estados do autômato de Büchi gerado pelo AJPF	10
Tempo gasto na execução	00:03:58
Memória RAM utilizada	2954MB

Fonte: Autoria Própria

Quadro 13 – Verificação Formal do AJPF: Desvio de Obstáculos, Readaptação

Cenário Utilizado	
Propriedade	**Valor**
Posição inicial do agente	`(1,1)`
Número de passageiros	1
Coordenadas dos Passageiros	{ `((1,1)-(3,1))` }
Número de obstáculos	3
Coordenadas dos obstáculos	{`(2,1), (2,2), (1,3)`}
Verificação realizada pelo AJPF	
Especificação formal `AR_8`: $\diamondsuit(B_{vehicle} east \land B_{vehicle} adapt(north) \land \neg B_{vehicle} can_adapt(north)) \implies (\diamondsuit A_{vehicle} drive(_, _, south, _, _)\ R\ \diamondsuit B_{vehicle} adapt(south))$	
Propriedade	***Feedback***
Número de estados do ambiente	368
Número de estados do autômato de Büchi gerado pelo AJPF	15
Tempo gasto na execução	00:09:17
Memória RAM utilizada	3506MB

Fonte: Autoria Própria

de um trajeto do agente for localizado no sentido oeste da sua posição atual, e o veículo estiver bloqueado por obstáculos nas direções norte, sul e oeste, então o veículo irá em algum momento se deslocará no sentido leste".

Apesar desta situação ser uma pré-condição para um acidente, este não é o objetivo da verificação e portanto o cenário de colisão inevitável não será considerado.

Quadro 14 – Verificação Formal do AJPF: Desvio de Obstáculos, Retornando a Coordenada Anterior

Cenário Utilizado	
Propriedade	**Valor**
Posição inicial do agente	`(4,1)`
Número de passageiros	1
Coordenadas dos Passageiros	{ `((4,1)-(1,1))` }
Número de obstáculos	3
Coordenadas dos obstáculos	`{(2,1),(3,2),(3,0)}`
Verificação realizada pelo AJPF	
Especificação formal `AR_7`: $\Diamond(B_{vehicle}west \land \neg B_{vehicle}can_adapt(north) \land \neg B_{vehicle}can_adapt(south) \land \neg B_{vehicle}can_adapt(west)) \implies (\Diamond G_{vehicle}drive_direction(east))$	
Propriedade	***Feedback***
Número de estados do ambiente	319
Número de estados do autômato de Büchi gerado pelo AJPF	4
Tempo gasto na execução	00:05:19
Memória RAM utilizada	3351MB

Fonte: Autoria Própria

6.2.8 Colisões Inevitáveis

Esta verificação é a mais importante do sistema, pois aqui, garante que em caso de uma colisão inevitável, o agente irá escolher por se deslocar até alguma coordenada onde o dano é mínimo. Com o auxílio do AJPF é possível averiguar todas as probabilidades possíveis de classificação de dano.

No cenário utilizado, a verificação busca afirmar que: Durante toda a execução do agente, na ocorrência de uma colisão inevitável onde há ao menos uma direção onde, caso o agente se mova para lá, a colisão causará um nível de dano leve ao veículo, então aquela será a direção escolhida pelo agente, **e** caso não haja nenhuma opção onde o dano será leve mas há um obstáculo com dano médio, a direção de tal será obstáculo será escolhida pelo agente.

A especificação formal `COC` no Quadro 15 se refere a propriedade de minimização de danos ao veículo. A propriedade do AJPF *profundidade de busca máxima* exibida no Quadro

15 demonstra que a ferramenta faz a verificação em todas as possibilidades possíveis no cenário estipulado.

Quadro 15 – Verificação Formal do AJPF: Colisões Inevitáveis, Escolha pelo Menor Dano

Cenário Utilizado	
Propriedade	**Valor**
Posição inicial do agente	`(2,1)`
Posição do depósito	`(4,0)`
Número de passageiros	1
Coordenadas dos Passageiros	{ `((4,1)-(4,0))` }
Número de obstáculos	3
Coordenadas dos obstáculos	`{(1,1),(2,2),(2,0)}`

Verificação realizada pelo AJPF	
Especificação formal COC: $\Box((\Diamond(B_{vehicle}unavoidable_collision(_,_) \wedge B_{vehicle}obstacle_damage(_,_,_,low)) \implies \Diamond G_{vehicle}colide_obstacle(_,low)) \wedge (\Diamond(B_{vehicle}unavoidable_collision(_,_) \wedge \sim B_{vehicle}obstacle_damage(_,_,_,low) \wedge B_{vehicle}obstacle_damage(_,_,_,moderate)) \implies \Diamond G_{vehicle}colide_obstacle(_,moderate)))$	
Propriedade	***Feedback***
Número de estados do ambiente	25465
Profundidade de busca máxima	444
Número de estados do autômato de Büchi gerado pelo AJPF	12
Tempo gasto na execução	01:09:58
Memória RAM utilizada	3866MB

Fonte: Autoria Própria

6.2.9 Investigação de Erros da Verificação Formal

Além de garantir o funcionamento correto do agente, é possível investigar as causas de uma falha em determinada especificação por meio da ferramenta AJPF. Devido à complexidade do código do agente e o número de estados do ambiente gerado nas verificações anteriores, será utilizado uma especificação simples nesta seção.

Considere a seguinte propriedade: Em algum momento durante sua execução, o agente ativar a buzina do veículo. No entanto, o agente desenvolvido neste trabalho não possui esta ação em nenhum de seus planos, portanto, ele nunca irá buzinar (executar ação $honk$).

No Quadro 16 são apresentados os dados desta verificação, onde o cenário utilizado foi criado aleatoriamente. Com base na propriedade $\Diamond A_{vehicle}honk$, o AJPF gera um autômato de Büchi correspondente a negação dessa especificação, que é $\Box\neg A_{vehicle}honk$. Este autômato é demonstrado na Figura 63.

Figura 63 – Autômato de Büchi: $\Box\neg A_{vehicle}honk$

0
Início

2
~A(vehicle,
honk())

Fonte: Autoria Própria

Quando há um erro na verificação de uma propriedade de um agente, o AJPF notifica o motivo que levou ao erro. Para isto, é informado a sequência de estados do ambiente gerados pela execução do agente onde tal propriedade foi verificada como falsa, e o respectivo estado final do autômato que foi atingido. Desta forma, é possível analisar as condições do ambiente que geraram as transições no autômato da propriedade.

A verificação é feita por meio da interpretação de cada estado do ambiente, exibido no *log* da execução do AJPF e, em casos de erros, pela rota apresentada ao final do *model checking*. Uma rota é uma sequência[5] de tuplas composta pelo estado do ambiente chamado MS e o estado do autômato de Büchi BS. Nesse exemplo, a rota é: $[MS : 0, BS : 2, UN : 0], [MS : 1, BS : 2, UN : 0], \ldots, [MS : 284, BS : 2, UN : 0]$. Podemos interpretá-la como: em todos os caminhos possíveis da árvore de estados do ambiente (há 285 estados nesse exemplo, identificados de 0 a 284), o estado final 2 do autômato da propriedade foi atingido. Logo, durante toda a execução do agente, a propriedade $\Diamond A_{vehicle}honk$ foi falsa.

Quadro 16 – Verificação Formal do AJPF: Investigação de Erros

Cenário Utilizado	
Propriedade	**Valor**
Posição inicial do agente	`(2,2)`
Número de passageiros	1
Coordenadas dos Passageiros	{ `((0,1)-(2,1))` }
Número de obstáculos	1
Coordenadas dos obstáculos	{`(0,1)`}
Verificação realizada pelo AJPF	
Especificação formal `IFV`: $\Diamond A_{vehicle}honk$	
Propriedade	***Feedback***
Número de estados do ambiente	285
Número de estados do autômato de Büchi gerado pelo AJPF	2
Tempo gasto na execução	00:01:07
Memória RAM utilizada	1667MB

Fonte: Autoria Própria

[5] Há um terceiro componente nesta tupla denominado método *until*. Tal dado não é relevante para o propósito desta seção.

7 CONCLUSÃO

Os veículos autônomos demonstram um enorme potencial e uma tecnologia promissora em um futuro próximo. Atualmente, dezenas de companhias estão pesquisando suas próprias versões destes automóveis. Empresas como a Tesla já disponibilizarem veículos parcialmente autônomo para seus consumidores. No entanto, a regulamentação necessária para permitir a circulação de veículos sem motoristas ainda é uma questão em aberto para governos no mundo todo. Além disso, questões éticas, como a apresentada na introdução deste trabalho, devem ser respondidas antes que tais automóveis possam trafegar dentro de cidades e rodovias. Portanto é imprescindível que o sistema autônomo se comporte corretamente ao comandar as funções do veículo e ao reagir a situações que ocorrem no tráfego, sendo tais cenários cotidianos ou possivelmente fatais. Consequentemente, deve ser realizado uma verificação precisa sobre o design do sistema.

Este trabalho está inserido no projeto AVIA do Grupo de Pesquisa em Agentes de Software, e aqui são abordadas as primeiras contribuições para o uso da verificação formal em agente modelados como veículo autônomo.

A implementação de um agente BDI modelado como um veículo autônomo. Essa escolha de arquitetura é motivada pelo fato de que esta categoria permitir a observação das razões que levam o agente a escolher determinada sequência de ações. Dessa forma, é possível observar o processo de tomada de decisão do agente através da análise de suas crenças, desejos e intenções, sendo isto essencial para a verificação de agentes.

Por sua vez, a implementação do agente BDI foi realizada por meio da linguagem Gwendolen, e tal agente possui planos que definem seu comportamento e capacidades. O uso dessa linguagem demonstra-se ideal para o escopo apresentado por este trabalho, pois além de oferecer meios para a implementação de agentes BDI, é a única atualmente que oferece integração com o *model checking program* do AJPF.

O agente desenvolvido não possui nenhum conhecimento inicial, e suas características consistem em: (i) localizar o veículo dentro do ambiente onde está inserido; (ii) realizar múltiplas corridas no ambiente onde está inserido, atendendo a uma lista pré-definida de passageiros; (iii) recusar corridas quando perceber que seu o ponto de partida ou destino está obstruído por um obstáculo; (iv) definir trajetórias para chegar em uma coordenada do ambiente; (v) desviar de obstáculos que bloqueiam seu percurso atual; (vi) aprender sobre rotas que se demonstram ineficazes para atingir o destino de um trajeto; (vii) escolher um plano que minimize o dano causado ao veículo em situações onde uma colisão é inevitável, dentre as classificações leve, médio e grave; e, (viii) recuperação de uma colisão e restauração do trajeto atual, quando for possível.

Tendo construído o agente era necessário a elaboração de um ambiente sobre o qual o mesmo irá atuar; dentro de tal ambiente, o agente irá perceber o cenário ao seu redor conforme

seu deslocamento sobre a matriz de posições, permitindo criar um modelo interno sobre as condições de cada coordenada. Na inicialização do ambiente é possível definir um número de passageiros, que são gerados aleatoriamente, para os quais o agente irá realizar corridas. Também há como determinar um número de obstáculos a serem inseridos no ambiente. No ambiente foram implementadas as pré-condições para que um cenário apresente uma colisão inevitável, bem como a classificação dos possíveis danos que os obstáculos podem causar ao veículo. Este ambiente é caracterizado como parcialmente observável, não-determinístico, estático, discreto e sequencial.

Visando ilustrar o ambiente e o agente desenvolvidos, é apresentando a implementação de um simulador gráfico em Java integrado com o ambiente. Neste, é possível observar passo-a-passo das interações ocorridas. O simulador é atualizado por meio de mensagens enviadas pelo ambiente na rede local, que informa a localização atual de cada um de seus elementos. Note que, o funcionamento do agente e do ambiente é independente da execução do simulador, mas o oposto não é verdade. A partir desse simulador, há como realizar uma análise visual da interação entre o agente e o ambiente.

A especificação formal das propriedades que são esperadas que agente possua foi realizada por meio de uma sintaxe variante da lógica de linguagem temporal. Por meio da imposição de cenários no ambiente, para os quais os planos do agente foram implementados, foi verificado formalmente a veracidade de cada uma das especificações definidas durante o funcionamento do agente; entre as quais, a propriedade do agente que assegura a escolha do obstáculo de menor impacto em caso de colisões. Frisando que o erro não é simulado, e sim o cenário onde se espera que o agente se comporte corretamente; este último é realizado por meio do framework AJPF, que disponibiliza informações referentes a cada nova mudança no ambiente, estado interno do agente e um autômato de Büchi referente a propriedade sendo verificada. Com esses dados é possível investigar a causa da falha de uma propriedade do agente.

Conclui-se que com o desenvolvimento deste trabalho é possível utilizar agentes racionais para modelar um veículo autônomo nas condições apresentadas por este trabalho. A linguagem Gwendolen pode ser aplicada para a implementação de tal agente, embora durante sua comunicação com o ambiente haja uma demora mínima para o tempo de resposta e o processamento interno do agente. A utilização do simulador é útil para observar as ações do agente sobre o ambiente onde estava inserido, uma vez que a linguagem Gwendolen não oferece representação gráfica. Além disso, constatou-se que a verificação formal pode garantir o funcionamento correto do agente nos cenários onde ele irá atuar.

Uma vez discutido as principais contribuições deste trabalho, ainda faz-se necessário destacar na Seção 7.1 a continuidade deste trabalho, bem como na Seção 7.2 elencar alguns trabalhos que possuem similaridades com o trabalho apresentado aqui.

7.1 TRABALHOS FUTUROS

Ainda faz-se necessário destacar os possíveis trabalhos futuros que podem ser realizados a partir dos resultados obtidos aqui.

- Expansão dos cenários de atuação do agente implementado;
- Simulação de cenários reais do tráfego e ajustar o simulador para representa-los;
- Adaptação do AJPF para outras linguagens de agentes em Java para o uso da verificação formal;
- Extensão do simulador para lidar com outras questões, como: obstáculos dinâmicos e veículos trafegando em faixas;
- Verificação formal de obstáculos dinâmicos;
- Integração de Gwendolen e do ambiente Java com outros tipos de simuladores, específicos para simulação veicular;
- Criação de um sistema multiagente onde veículos autônomos cooperem entre si;

7.2 TRABALHOS RELACIONADOS

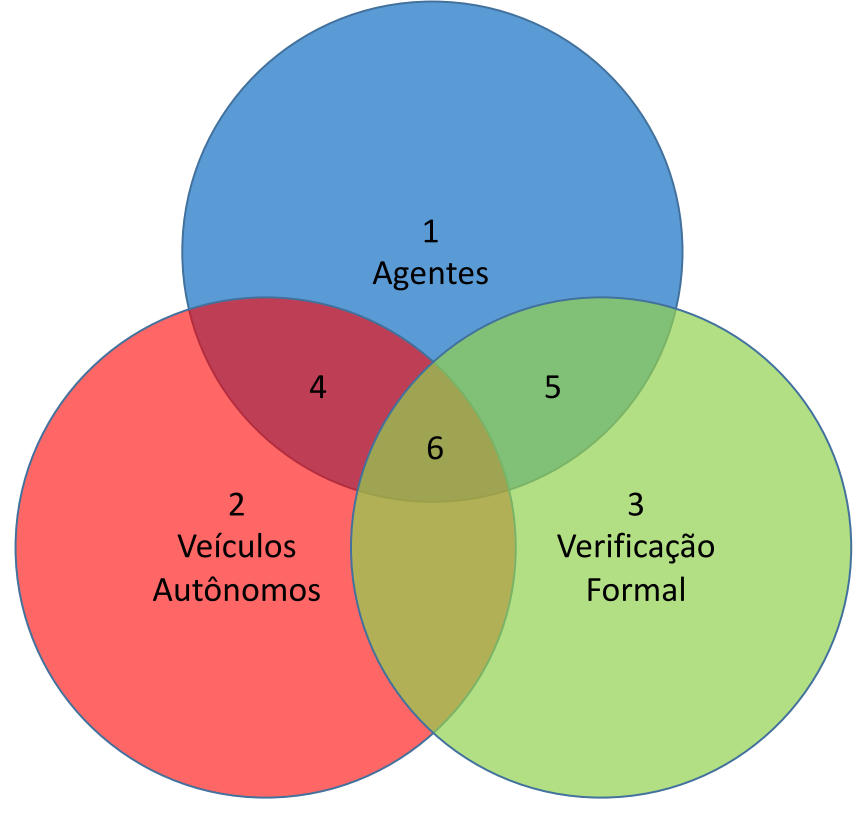
Atualmente existem diferentes pesquisas nas áreas de agentes, veículos autônomos e verificação formal. Existe uma intersecção entre estes campos de estudo, a partir das quais novas vertentes de pesquisas surgem. A Figura 64 apresenta um diagrama de Venn demonstrando as três áreas presentes neste trabalho, são estas: agentes (1), veículos Autônomos (2) e verificação formal (3). A classificação deste trabalho é a convergência destas três áreas (6). Outras intersecções discutidas aqui são:

- Agentes e Veículos Autônomos (4);
- Agentes e Verificação Formal (5).

É relatado a seguir trabalhos similares a este, onde os autores propõem diferentes soluções para problemas utilizando o conhecimento das áreas citadas anteriormente. E, no Quadro 17 são referenciados a quais campos de estudos tais trabalhos pertencem.

O Waymo desenvolvido pela Google (2017) é dos projetos sobre veículos autônomos mais avançados na área. Como não há divulgação de informações referentes ao software utilizado, não há como supor que abordagem de sistema autônomo a empresa utiliza, logo sua área

Figura 64 – Diagrama de Venn referentes as áreas de pesquisa



Fonte: Autoria Própria

de estudos é assumida como somente veículos Autônomos (2). Este projeto é discutido detalhadamente na Subseção 2.3.2.

Em seu trabalho, Dafflon *et al.* (2015) discutem o desenvolvimento de um agente inteligente capaz de controlar um veículo em ambientes desordenados, tais como os cenários urbanos. Dessa forma, este trabalho pode ser caracterizado na intersecção (4).

Webster *et al.* (2011) abordam a utilização do *model checking* para verificar um sistema que implementa o controle de um aeronave não-tripulada por meio do uso de agentes BDI. Este trabalho pode ser classificado na intersecção (5).

A verificação formal de comboios de veículos autônomos é o foco central em Kamali *et al.* (2016). Aqui, busca-se garantir que as decisões tomadas nunca violem requisitos mínimos de segurança. Logo, este trabalho é classificado na intersecção entre todas as áreas (6).

Fernandes, Custodio e Alves (2016) propõem uma implementação de um agente racional no controle de um veículo autônomo. Nesse trabalho, é discutido o uso da verificação formal para garantir a corretude no processo de tomada de decisão do agente. Este pode ser classificado na intersecção entre todas as áreas (6).

Quadro 17 – Comparação dos Trabalhos

	Áreas de Pesquisa		
Identificação do Trabalho	**Agentes**	**Veículos Autônomos**	**Verificação Formal**
Waymo, por Google (2017)	Desconhecido	Sim	Desconhecido
Adaptive autonomous navigation using reactive multi-agent system for control law merging, por Dafflon *et al.* (2015)	Sim	Sim	Não
Formal methods for the certification of autonomous unmanned aircraft systems, por Webster *et al.* (2011)	Sim	Não	Sim
Formal verification of autonomous vehicle platooning, por Kamali *et al.* (2016)	Sim	Sim	Sim
Implementação e Verificação Formal de Estratégias para Desvio de Obstáculos de Veículos Autônomos Modelados como Agentes Racionais, por Fernandes, Custodio e Alves (2016)	Sim	Sim	Sim

Fonte – Autoria própria

REFERÊNCIAS

BAIER, Christel; KATOEN, Joost-Pieter. **Principles of Model Checking**. [S.l.]: The MIT Press, 2008.

BARENDREGT, Henk. **The Quest for Correctness**. 1996.

BATTELLE, John. **Sorry, But Driverless Cars Aren't Right Around the Corner**. 2016. Disponível em: <*https://www.linkedin.com/pulse/sorry-driverless-cars-arent-right-around-corner-john-battelle*>. Acesso em: 30 out. 2016.

BBC Radio. **The Trolley Problem**. 2014. Disponível em: <*https://www.youtube.com/watch?v=bOpf6KcWYyw&t=38s*>. Acesso em: 14 nov. 2016.

BORDINI, R. H. *et al.* Automated verification of multi-agent programs. In: **Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2008. (ASE '08), p. 69–78. ISBN 978-1-4244-2187-9. Disponível em: <*http://dx.doi.org/10.1109/ASE.2008.17*>.

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

BÜCHI, J. Richard. On a decision method in restricted second order arithmetic. In: ____. **The Collected Works of J. Richard Büchi**. New York, NY: Springer New York, 1990. p. 425–435. ISBN 978-1-4613-8928-6.

CBINSIGHTS. **33 Corporations Working On Autonomous Vehicles**. 2016. Disponível em: <*https://www.cbinsights.com/blog/autonomous-driverless-vehicles-corporations-list/*>. Acesso em: 30 ago. 2016.

CLARKE JR., Edmund M.; GRUMBERG, Orna; PELED, Doron A. **Model Checking**. Cambridge, MA, USA: MIT Press, 1999. ISBN 0-262-03270-8.

CLIFFE, Mark. **Driverless cars – the route to more than smart cities**. 2016. Disponível em: <*https://ingworld.ing.com/en/2016-1Q/12-column-m-cliffe*>. Acesso em: 28 ago. 2016.

CROTHERS, Brooke. **The Self-Driving Tesla Model S: Autopilot And 'Summoning' In Focus**. 2015. Disponível em: <*https://electrek.co/2016/08/11/tesla-autopilot-2-0-next-gen-radar-triple-camera-production/*>. Acesso em: 26 out. 2016.

DAFFLON, Baudouin *et al.* Adaptive autonomous navigation using reactive multi-agent system for control law merging. **International Conference On Computational Science**, v. 51, p. 423–432, 2015.

DENNIS, Louise *et al.* **MCAPL:Model Checking Agent Programming Languages**. 2017. Disponível em: <*http://cgi.csc.liv.ac.uk/MCAPL/index.php/Main_Page*>. Acesso em: 2 mai. 2017.

DENNIS, Louise; FISHER, Michael. **Draft Verifiable Autonomous Systems**. 1. ed. [S.l.]: John Wiley Sons, 2016.

DENNIS, Louise A.; FARWER, Berndt. **Gwendolen: A BDI Language for Verifiable Agents**. [S.l.]: University of Aberdeen, 2008.

DENNIS, Louise A. *et al.* Model checking agent programming languages. **Automated Software Engg.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 19, n. 1, p. 5–63, mar. 2012.

DOCTOROW, Cory. **Sorry, But Driverless Cars Aren't Right Around the Corner**. 2015. Disponível em: <*https://www.theguardian.com/technology/2015/dec/23/the-problem-with-self-driving-cars-who-controls-the-code*>. Acesso em: 28 ago. 2016.

FAGNANT, Daniel J.; KOCKELMAN, Kara M. Preparing a nation for autonomous vehicles: Opportunities, barriers and policy recommendations. **Eno Center for Transportation**, v. 2, 2013.

FERBER, Jacques. **Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence**. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201360489.

FERNANDES, Lucas E. R.; CUSTODIO, Vinicius; ALVES, Gleifer V. Implementação e verificação formal de estratégias para desvio de obstáculos de veículos autônomos modelados como agentes racionais. **WPCCG - Workshop de Pesquisas em Computação dos Campos Gerais**, Ponta Grossa, v. 1, p. 12–15, 2016.

FISHER, Michael; DENNIS, Louise; WEBSTER, Matt. Verifying autonomous systems. **Commun. ACM**, ACM, New York, NY, USA, v. 56, n. 9, p. 84–93, set. 2013.

FOOT, Philippa. The problem of abortion and the doctrine of double effect. **Oxford Review**, v. 5, p. 5–15, 1967.

GOOGLE. **Waymo**. 2017. Disponível em: <*https://waymo.com/*>. Acesso em: 29 mai. 2017.

HARRIS, Mark. **Google reports self-driving car mistakes: 272 failures and 13 near misses**. 2016. Disponível em: <*https://www.theguardian.com/technology/2016/jan/12/google-self-driving-cars-mistakes-data-reports*>. Acesso em: 11 jan. 2017.

KAMALI, Maryam *et al.* Formal verification of autonomous vehicle platooning. **CoRR**, 2016.

LAMBERT, Fred. **Tesla Autopilot 2.0: next gen Autopilot powered by more radar, new triple camera, some equipment already in production**. 2016. Disponível em: <*http://www.forbes.com/sites/brookecrothers/2015/08/19/the-self-driving-tesla-model-s-autopilot-and-summoning-in-focus/#508f944ded95*>. Acesso em: 26 out. 2016.

LIN, Patrick *et al.* **The ethical dilemma of self-driving cars**. 2015. Disponível em: <*http://ed.ted.com/lessons/the-ethical-dilemma-of-self-driving-cars-patrick-lin*>. Acesso em: 22 ago. 2016.

LUBELL, sam. **Here's How Self-Driving Cars Will Transform Your City**. 2016. Disponível em: <*https://www.wired.com/2016/10/heres-self-driving-cars-will-transform-city/*>. Acesso em: 30 out. 2016.

MARSHALL, Aarian. **Columbus Just Won $50 Million To Become The City Of The Future**. 2016. Disponível em: <*https://www.wired.com/2016/06/columbus-wins-50-million-become-city-future/*>. Acesso em: 28 out. 2016.

MARSHALL, Aarian; DAVIES, Alex. **How Pittsburgh birthed the age of the self-driving car**. 2016. Disponível em: <*https://www.wired.com/2016/08/pittsburgh-birthed-age-self-driving-car/*>. Acesso em: 26 ago. 2016.

MCGEE, Jaime. **Five cities chosen for self-driving car test**. 2016. Disponível em: <*http://www.usatoday.com/story/money/cars/2016/10/26/five-cities-chosen-self-driving-car-test/92757834/*>. Acesso em: 29 out. 2016.

MORRIS, David Z. **Only 6% of Cities are Preparing for Driverless Cars**. 2015. Disponível em: <*http://fortune.com/2015/12/02/somerville-driverless-car/*>. Acesso em: 29 out. 2016.

MUKUND, Madhavan. **Finite-state Automata on Infinite Inputs**. 1996.

NATIONS, United. **Department of Economic and Social Affairs: Population Division. World Urbanization Prospects: The 2014 Revision**. [S.l.]: United Nations, 2015.

NAUGHTON, Keith. **Google's Driverless-Car Czar on Taking the Human Out of the Equation**. 2016. Disponível em: <*http://www.bloomberg.com/features/2016-john-krafcik-interview-issue/*>. Acesso em: 29 out. 2016.

NHTSA. **Preliminary Statement of Policy Concerning Automated Vehicles**. 2012. Disponível em: <*http://www.autoalliance.org/index.cfm?objectid=CC9678B0-A415-11E5-997E000C296BA163*>. Acesso em: 25 nov. 2016.

NUNES, Ingrid Oliveira de *et al.* **BDI Architecture**. 2016. Disponível em: <*http://www.inf.ufrgs.br/prosoft/bdi4jade/?page_id=46*>. Acesso em: 15 nov. 2016.

PEDEN, Margie *et al.* **World report on road traffic injury prevention. World Health Organization**. [S.l.: s.n.], 2004.

PULLEN, John Patrick. **You Asked: How Do Driverless Cars Work?** 2015. Disponível em: <*http://time.com/3719270/you-asked-how-do-driverless-cars-work/*>. Acesso em: 29 out. 2016.

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

ROUSE, Margaret. **DEFINITION: geo-fencing (geofencing)**. 2015. Disponível em: <*http://whatis.techtarget.com/definition/geofencing*>. Acesso em: 25 set. 2016.

RUSSELL, Stuart J.; NORVIG, Peter. **Artificial Intelligence: A Modern Approach**. 3. ed. [S.l.]: Pearson Education, 2010.

SHOHAM, Yoav. Agent-oriented programming. **Artificial intelligence**, Elsevier, v. 60, n. 1, p. 51–92, 1993.

TESLA. **Tesla Models**. 2016. Disponível em: <*https://www.tesla.com/models/*>. Acesso em: 29 ago. 2016.

VISSER, Willem *et al.* Model checking programs. **Automated Software Engg.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 10, n. 2, p. 203–232, abr. 2003.

WALLACE, Richard; SILBERG, Gary. **Self-Driving cars: The next revolution**. [S.l.]: KPMG Center for Automotive Research, 2012.

WEBSTER, Matt *et al.* Formal methods for the certification of autonomous unmanned aircraft systems. In: **Proceedings of the 30th International Conference on Computer Safety, Reliability and Security**. [S.l.]: Springer-Verlag, 2011. (SAFECOMP'11), p. 228–242.

WOOLDRIDGE, Michael. **Reasoning about Rational Agents**. [S.l.]: MIT Press, 2000.

____. **An Introduction to MultiAgent Systems**. 2. ed. [S.l.]: John Wiley Sons, 2009.

WOOLDRIDGE, Michael; RAO, Anand. **Foundations of Rational Agency**. [S.l.]: Kluwer Academic Publishers, 1999.

APÊNDICE A - AGENTE

A seguir, o Código 26 apresentado o código do agente.

Código 26 – Agente *Autonomous Car*

```
1  GWENDOLEN
2
3  :name: vehicle
4
5  : Initial  Beliefs :
6
7  :Reasoning Rules:
8
9   finish_all_rides  :- done_all_rides;
10
11 reach(X,Y) :- try_to_reach(X,Y), at(X,Y);
12
13 drive_to(X,Y) :- at(X,Y);
14
15 obstacle_ahead(DIRECTION) :- at(AT_X, AT_Y), obstacle(DIRECTION, AT_X, AT_Y);
16
17 known_route(north,KR_X,KR_Y) :- at(AT_X,AT_Y), from(F_X,F_Y),
       adapt_from_to(F_X,F_Y, AT_X,AT_Y, north, KR_X,KR_Y),
       moved_from_to(F_X,F_Y,AT_X, AT_Y, south, KR_X, KR_Y);
18 known_route(south,KR_X,KR_Y) :- at(AT_X,AT_Y), from(F_X,F_Y),
       adapt_from_to(F_X,F_Y, AT_X,AT_Y, south, KR_X,KR_Y),
       moved_from_to(F_X,F_Y,AT_X, AT_Y, north, KR_X, KR_Y);
19 known_route(east,KR_X,KR_Y) :- at(AT_X,AT_Y), from(F_X,F_Y),
       adapt_from_to(F_X,F_Y, AT_X,AT_Y, east, KR_X,KR_Y),
       moved_from_to(F_X,F_Y,AT_X, AT_Y, west, KR_X, KR_Y);
20 known_route(west,KR_X,KR_Y) :- at(AT_X,AT_Y), from(F_X,F_Y),
       adapt_from_to(F_X,F_Y, AT_X,AT_Y, west, KR_X,KR_Y),
       moved_from_to(F_X,F_Y,AT_X, AT_Y, east, KR_X, KR_Y);
21
22 north_south :- north;
23 north_south :- south;
24 east_west :- east;
25 east_west :- west;
26
27
28 can_adapt(CA_D, D_X, D_Y) :- at(AT_X, AT_Y), ~obstacle(CA_D, AT_X, AT_Y), ~
```

```
        known_route(CA_D, D_X, D_Y);
29  can_adapt_route(CA_D, D_X, D_Y) :− can_adapt(CA_D, D_X, D_Y), go_adapt(CA_D);
30
31  go_adapt(north) :− ~adapt(south), go_adapt(north_south);
32  go_adapt(south) :− ~adapt(north), go_adapt(north_south);
33  go_adapt(east) :− ~adapt(west), go_adapt(east_west);
34  go_adapt(west) :− ~adapt(east), go_adapt(east_west);
35
36  go_adapt(north_south)  :− ~heading(north), ~heading(south), ~adapt(east), ~adapt(west);
37  go_adapt(east_west)  :− ~heading(east), ~heading(west), ~adapt(north), ~adapt(south);
38
39  control_emergency (X,Y) :− ~crashed(X,Y);
40
41  : Initial  Goals:
42
43   finish_all_rides  [achieve]
44
45  :Plans:
46  /* FAR 1 */
47  +! finish_all_rides  [achieve]  :  {~B at(X,Y)} ← localize, *at(LX, LY);
48
49  /* FAR 2 */
50  +! finish_all_rides  [achieve]  :  {B damaged(high)} ← refuse_ride(car_unavailable),
        park(car_unavailable), +done_all_rides;
51  /* FAR 3.1 */
52  +! finish_all_rides  [achieve]  :  {B damaged(moderate), B depot(X,Y), B passenger} ←
        park(drop_off), −passenger, refuse_ride(drop_off), +!complete_journey (X, Y) [perform],
        park(depot), +done_all_rides;
53  /* FAR 3.2 */
54  +! finish_all_rides  [achieve]  :  {B damaged(moderate), B depot(X,Y), ~B passenger} ←
        refuse_ride(pick_up), +!complete_journey (X, Y) [perform], park(depot), +done_all_rides;
55  /* FAR 4 */
56  +! finish_all_rides  [achieve]  :  {B no_possible_new_ride, B damaged(low), B depot(X,Y)} ←
        +!complete_journey (X, Y) [perform], park(depot), +done_all_rides;
57  /* FAR 5 */
58  +! finish_all_rides  [achieve]  :  {B no_possible_new_ride, ~B damaged(DAMAGE_LEVEL)}
        ← park(done_all_rides), +done_all_rides;
59
60  /* FAR 6 */
61  +! finish_all_rides  [achieve]  :  {~B ride_info, ~B no_possible_new_ride} ← get_ride,
        *ride_info;
62
```

```
63 /* FAR 7.1 */
64 +! finish_all_rides [achieve] : {B pick_up(X,Y), B obstacle(center, X,Y)} ←
       refuse_ride(pick_up), −ride_info;
65 /* FAR 7.2 */
66 +! finish_all_rides [achieve] : {B drop_off(X,Y), B obstacle(center, X,Y), ~B passenger} ←
       refuse_ride(drop_off), −ride_info;
67
68 /* FAR 8.1 */
69 +! finish_all_rides [achieve] : {B pick_up(X,Y), ~B try_to_reach(X,Y), ~B passenger} ←
       +!complete_journey (X,Y) [perform], +try_to_reach(X,Y);
70 /* FAR 8.2 */
71 +! finish_all_rides [achieve] : {B pick_up(X,Y), B reach(X,Y), ~B passenger} ←
       park(pick_up), +passenger, −try_to_reach(X,Y);
72 /* FAR 8.3 */
73 +! finish_all_rides [achieve] : {B pick_up(X,Y), ~B reach(X,Y), ~B passenger} ←
       refuse_ride(pick_up), −try_to_reach(X,Y), −ride_info;
74
75 /* FAR 9.1 */
76 +! finish_all_rides [achieve] : {B drop_off(X,Y), ~B try_to_reach(X,Y), B passenger} ←
       +!complete_journey (X,Y) [perform], +try_to_reach(X,Y);
77 /* FAR 9.2 */
78 +! finish_all_rides [achieve] : {B drop_off(X,Y), B reach(X,Y), B passenger} ←
       park(drop_off), −passenger, −try_to_reach(X,Y), −ride_info;
79 /* FAR 9.3 */
80 +! finish_all_rides [achieve] : {B drop_off(X,Y), ~B reach(X,Y), B passenger} ←
       refuse_ride(drop_off), −passenger, park(drop_off), −try_to_reach(X,Y), −ride_info;
81
82
83 /* CJ */
84 +!complete_journey (X,Y) [perform] : {B at(F_X,F_Y)}
85                                        ←    +!clear_travel_data [perform],
86                                             +from(F_X,F_Y), +moving,
87                                             +!drive_to(X,Y) [achieve],
88                                             −moving, −from(F_X,F_Y);
89
90
91 /* CTD */
92 +!clear_travel_data [perform] : {True} ← +!clear_direction_data [perform],
       −heading(north), −heading(south), −heading(east), −heading(west);
93
94
95 /* CDD */
```

```
96  +!clear_direction_data [perform] : {True}  ← +!clear_adapt [perform], −north, −south,
        −east, −west, −receive_direction;
97
98
99  /∗ CA ∗/
100 +!clear_adapt [perform] : {True} ← −adapt(north), −adapt(south), −adapt(east),
        −adapt(west);
101
102
103 /∗ DT 1 ∗/
104 +!drive_to(X,Y) [achieve] : {∼B north, ∼B south, ∼B east, ∼B west} ← +!get_direction
        [perform];
105
106 /∗ DT 2.1 ∗/
107 +!drive_to(X,Y) [achieve] : {∼B heading(H), B north} ← +heading(north);
108 /∗ DT 2.2 ∗/
109 +!drive_to(X,Y) [achieve] : {∼B heading(H), B south} ← +heading(south);
110 /∗ DT 2.3 ∗/
111 +!drive_to(X,Y) [achieve] : {∼B heading(H), B east} ← +heading(east);
112 /∗ DT 2.4 ∗/
113 +!drive_to(X,Y) [achieve] : {∼B heading(H), B west} ← +heading(west);
114
115 /∗ DT 3.1 ∗/
116 +!drive_to(X,Y) [achieve] : {B heading(north), B at(AT_X,Y), B east} ← −heading(north),
        +heading(east);
117 /∗ DT 3.2 ∗/
118 +!drive_to(X,Y) [achieve] : {B heading(north), B at(AT_X,Y), B west} ← −heading(north),
        +heading(west);
119 /∗ DT 3.3 ∗/
120 +!drive_to(X,Y) [achieve] : {B heading(south), B at(AT_X,Y), B east} ← −heading(south),
        +heading(east);
121 /∗ DT 3.4 ∗/
122 +!drive_to(X,Y) [achieve] : {B heading(south), B at(AT_X,Y), B west} ← −heading(south),
        +heading(west);
123
124 /∗ DT 4 ∗/
125 +!drive_to(X,Y) [achieve] : {B blocked} ← −blocked, −!drive_to(X,Y) [achieve];
126
127 /∗ DT 5 ∗/
128 +!drive_to(X,Y) [achieve] : {B at(AT_X, AT_Y), B unavoidable_collision(AT_X, AT_Y)} ←
        −moving, +!choose_obstacle_collision [perform];
129
```

```
130 /* DT 6.1 */
131 +!drive_to(X,Y) [achieve] : {B heading(D), B obstacle_ahead(D)} ← −moving, +adapt,
        +!adapt_route(D,X,Y) [achieve];
132 /* DT 6.2 */
133 +!drive_to(X,Y) [achieve] : {B heading(D), B known_route(D,X,Y)} ← −moving, +adapt,
        +!adapt_route(D,X,Y) [achieve];
134
135 /* DT 7 */
136 +!drive_to(X,Y) [achieve] : {~B moving} ← *moving;
137
138 /* DT 8 */
139 +!drive_to(X,Y) [achieve] : {B heading(D), B can_adapt(D,X,Y)} ← +!drive_direction(D)
        [perform];
140
141
142 /* GD */
143 +!get_direction [perform] : {G drive_to(X, Y) [achieve]} ← +!clear_travel_data [perform],
        compass(X, Y), *receive_direction ;
144
145
146 /* DD */
147 +!drive_direction(D) [perform] : {G drive_to(X, Y) [achieve], B from(F_X,F_Y)} ←
        −moving, drive(F_X,F_Y, D, X, Y);
148
149
150 /* COC 1.1 */
151 +!choose_obstacle_collision [perform] :
152             {B at(AT_X, AT_Y), B obstacle_damage(AT_X, AT_Y, DIRECTION, low)}
153             ← +!colide_obstacle(DIRECTION, low) [perform];
154 /* COC 1.2 */
155 +!choose_obstacle_collision [perform] :
156             {B at(AT_X, AT_Y), B obstacle_damage(AT_X, AT_Y, DIRECTION, moderate)}
157             ← +!colide_obstacle(DIRECTION, moderate) [perform];
158 /* COC 1.3 */
159 +!choose_obstacle_collision [perform] :
160             {B at(AT_X, AT_Y), B obstacle_damage(AT_X, AT_Y, DIRECTION, high)}
161             ← +!colide_obstacle(DIRECTION, high) [perform];
162
163
164 /* CO */
165 +!colide_obstacle(DIRECTION, DAMAGE_LEVEL) [perform] :
166             {B from(F_X,F_Y), B at(AT_X, AT_Y), G drive_to(X, Y) [achieve]}
```

```
167            ← no_further_from(F_X,F_Y, AT_X, AT_Y, X,Y), *no_further(F_X,F_Y,
                   AT_X, AT_Y, X,Y),
168            +damaged(DAMAGE_LEVEL),
169            +!drive_direction(DIRECTION) [perform], −moving,
170            +!get_direction [perform], +moving;
171
172
173 /* AR 1 */
174 +!adapt_route(D,X,Y) [achieve] : {B at(AT_X, AT_Y), B unavoidable_collision(AT_X,
        AT_Y)}
175            ← −adapt, +!choose_obstacle_collision [perform], +!clear_adapt [perform],
                   −!adapt_route(D,X,Y) [achieve];
176
177 /* AR 2 */
178 +!adapt_route(D,X,Y) [achieve] : {B obstacle_ahead(north), B obstacle_ahead(south), B
        obstacle_ahead(east), B obstacle_ahead(west)}
179            ← +blocked, −!adapt_route(D,X,Y) [achieve];
180 /* AR 3 */
181 +!adapt_route(D,X,Y) [achieve] : {B adapt, B can_adapt(D,X,Y)}
182            ← −adapt, +!drive_direction(D) [perform], +!get_direction [perform],
                   −!adapt_route(D,X,Y) [achieve];
183
184 /* AR 4 */
185 +!adapt_route(D,X,Y) [achieve] : {∼B adapt} ← *adapt;
186
187 /* AR 5.1 */
188 +!adapt_route(D,X,Y) [achieve] : {B can_adapt_route(east,X,Y), B east, B north_south}
189            ← −adapt, +!adapt_drive_direction(east,X,Y) [perform];
190 /* AR 5.2 */
191 +!adapt_route(D,X,Y) [achieve] : {B can_adapt_route(west,X,Y), B west, B north_south}
192            ← −adapt, +!adapt_drive_direction(west,X,Y) [perform];
193 /* AR 5.3 */
194 +!adapt_route(D,X,Y) [achieve] : {B north, B can_adapt_route(north,X,Y), B east_west}
195            ← −adapt, +!adapt_drive_direction(north,X,Y) [perform];
196 /* AR 5.4 */
197 +!adapt_route(D,X,Y) [achieve] : {B south, B can_adapt_route(south,X,Y), B east_west}
198            ← −adapt, +!adapt_drive_direction(south,X,Y) [perform];
199
200 /* AR 6.1 */
201 +!adapt_route(D,X,Y) [achieve] : {B can_adapt_route(east,X,Y), B north_south}
202            ← −adapt, +!adapt_drive_direction(east,X,Y) [perform];
203 /* AR 6.2 */
```

```
204 +!adapt_route(D,X,Y) [achieve] : {B can_adapt_route(west,X,Y), B north_south}
205             ← −adapt, +!adapt_drive_direction(west,X,Y) [perform];
206 /∗ AR 6.3 ∗/
207 +!adapt_route(D,X,Y) [achieve] : {B can_adapt_route(north,X,Y), B east_west}
208             ← −adapt, +!adapt_drive_direction(north,X,Y) [perform];
209 /∗ AR 6.4 ∗/
210 +!adapt_route(D,X,Y) [achieve] : {B can_adapt_route(south,X,Y), B east_west}
211             ← −adapt, +!adapt_drive_direction(south,X,Y) [perform];
212
213
214 /∗ AR 7.1 ∗/
215 +!adapt_route(D,X,Y) [achieve] : {B can_adapt(north,X,Y), ∼B can_adapt(south,X,Y), ∼B
        can_adapt(east,X,Y), ∼B can_adapt(west,X,Y)}
216             ← −adapt, +!invalid_coordinate(north) [perform];
217 /∗ AR 7.2 ∗/
218 +!adapt_route(D,X,Y) [achieve] : {∼B can_adapt(north,X,Y), B can_adapt(south,X,Y), ∼B
        can_adapt(east,X,Y), ∼B can_adapt(west,X,Y)}
219             ← −adapt, +!invalid_coordinate(south) [perform];
220 /∗ AR 7.3 ∗/
221 +!adapt_route(D,X,Y) [achieve] : {∼B can_adapt(north,X,Y), ∼B can_adapt(south,X,Y), ∼B
        can_adapt(east,X,Y), B can_adapt(west,X,Y)}
222             ← −adapt, +!invalid_coordinate(west) [perform];
223 /∗ AR 7.4 ∗/
224 +!adapt_route(D,X,Y) [achieve] : {∼B can_adapt(north,X,Y), ∼B can_adapt(south,X,Y), B
        can_adapt(east,X,Y), ∼B can_adapt(west,X,Y)}
225             ← −adapt, +!invalid_coordinate(east) [perform];
226
227
228 /∗ AR 8.1 ∗/
229 +!adapt_route(D,X,Y) [achieve] : {∼B can_adapt(north,X,Y), B can_adapt(south,X,Y)}
230             ← −adapt, +!invalid_coordinate(south) [perform];
231 /∗ AR 8.2 ∗/
232 +!adapt_route(D,X,Y) [achieve] : {∼B can_adapt(south,X,Y), B can_adapt(north,X,Y)}
233             ← −adapt, +!invalid_coordinate(north) [perform];
234 /∗ AR 8.3 ∗/
235 +!adapt_route(D,X,Y) [achieve] : {∼B can_adapt(east,X,Y), B can_adapt(west,X,Y)}
236             ← −adapt, +!invalid_coordinate(west) [perform];
237 /∗ AR 8.4 ∗/
238 +!adapt_route(D,X,Y) [achieve] : {∼B can_adapt(west,X,Y), B can_adapt(east,X,Y)}
239             ← −adapt, +!invalid_coordinate(east) [perform];
240 /∗ AR 9∗/
241 +!adapt_route(D,X,Y) [achieve] : {B at(AT_X, AT_Y)} ←
```

```
242                     print("Missing Adapt Route to reach destination"),
243                     +missing_adapt_route(AT_X, AT_Y, D, X, Y),
244                     +blocked, −!adapt_route(D,X,Y) [achieve];
245
246
247 /∗ ADD ∗/
248 +!adapt_drive_direction(A_D,X,Y) [perform] : {True}
249               ← +adapt(A_D), +!drive_direction(A_D) [perform], −moving, +adapt;
250
251
252 /∗ IC ∗/
253 +!invalid_coordinate(TD) [perform] :
254               {G adapt_route(D,X,Y) [achieve], B at(AT_X, AT_Y), B from(F_X,F_Y)}
255               ← no_further_from(F_X,F_Y, AT_X,AT_Y, X, Y), ∗no_further(F_X,F_Y,
                    AT_X,AT_Y, X, Y),
256               +!clear_adapt [perform], +!drive_direction(TD) [perform], −moving, +adapt;
257
258
259 /∗ O 1 ∗/
260 +obstacle(center, X, Y) : {B pick_up(X, Y), ∼B temp_obstacle(X, Y)} ← −moving, −adapt,
        −!drive_to(DX, DY) [achieve], −!adapt_route(D,DX,DY) [achieve];
261 /∗ O 2 ∗/
262 +obstacle(center, X, Y) : {B drop_off(X, Y), ∼B temp_obstacle(X, Y)} ← −moving, −adapt,
        −!drive_to(DX, DY) [achieve], −!adapt_route(D,DX,DY) [achieve];
263
264
265 /∗ A 1 ∗/
266 +at(AT_X,AT_Y) : {∼B obstacle(center, AT_X,AT_Y)} ← +moving;
267
268
269 /∗ A 2 ∗/
270 +at(AT_X,AT_Y) : {B obstacle(center, AT_X, AT_Y)}
271               ← −moving, −adapt, +crashed(AT_X, AT_Y),
272               +!control_emergency (AT_X,AT_Y) [achieve];
273
274
275 /∗ CE 1 ∗/
276 +!control_emergency (AT_X, AT_Y) [achieve] : {B crashed(AT_X, AT_Y), ∼B
        emergency(AT_X, AT_Y)}
277               ← call_emergency(AT_X,AT_Y), ∗emergency(AT_X,AT_Y);
278
279 /∗ CE 2 ∗/
```

```
280 +!control_emergency (AT_X, AT_Y) [achieve] :
281             {B emergency(AT_X,AT_Y), B damaged(low),
282             ∼B damaged(moderate), ∼B damaged(high)}
283             ←  get_assistance(AT_X,AT_Y), ∗assisted(AT_X,AT_Y),
284             −crashed(AT_X,AT_Y), +moving, +adapt;
285 
286 /∗ CE 3 ∗/
287 +!control_emergency (AT_X, AT_Y) [achieve] :
288             {B emergency(AT_X, AT_Y), B damaged(moderate), ∼B damaged(high)}
289             ← get_assistance(AT_X, AT_Y), ∗assisted(AT_X, AT_Y),
290             −crashed(AT_X,AT_Y), −!adapt_route(D,X,Y) [achieve],
291             −!drive_to(X,Y) [achieve];
292 
293 /∗ CE 4 ∗/
294 +!control_emergency (AT_X, AT_Y) [achieve] :
295             {B emergency(AT_X, AT_Y), B damaged(high)}
296             ← −!drive_to(X,Y) [achieve],  −!adapt_route(D,X,Y) [achieve],
297             −!control_emergency (AT_X, AT_Y) [achieve];
```

Fonte: Autoria Própria

APÊNDICE B - AMBIENTE

O Código 27 apresenta o código referente a implementação do ambiente.

Código 27 – Ambiente, Classe `AutonomousCarEnv`

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.TimeUnit;

import ail.mas.DefaultEnvironmentwRandomness;
import ail.mas.MAS;
import ail.syntax.Action;
import ail.syntax.NumberTermImpl;
import ail.syntax.Predicate;
import ail.syntax.Unifier;
import ail.util.AILexception;
import ajpf.util.choice.Choice;
import main.Client;
import util.Coordinate;
import util.GridCell;
import util.Passenger;
import util.Util;

public class AutonomousCarEnv extends DefaultEnvironmentwRandomness {
    private Map<String, GridCell> environmentGrid;

    private Choice<Boolean> collisionChance;
    private Choice<String> damageLevel;

    private Coordinate car;
    private Coordinate depotLocation;
    private String currentDirection;

    private int minGridSize;
    private int maxGridSize;

    private ArrayList<Predicate> obstacleDamageRelated = new ArrayList<>();

    private ArrayList<Passenger> passengers = new ArrayList<Passenger>();
    private int nPassengers;
```

```
37    private Passenger currentPassenger;
38    private int nObstacles;
39
40    private boolean simulate;
41    private int waitTimeDefault;
42    private int waitTimeLocation;
43
44    @Override
45    public void setMAS(MAS m) {
46        super.setMAS(m);
47
48        this.car = new Coordinate(0, 0);
49        this.depotLocation = new Coordinate(0, 0);
50        this.currentDirection = "north";
51        this.minGridSize = 0;
52        this.maxGridSize = 0;
53        this.nPassengers = 0;
54        this.currentPassenger = new Passenger(new Coordinate(0, 0), new
              Coordinate(0, 0));
55
56        this.simulate = true;
57        this.waitTimeDefault = 2000;
58        this.waitTimeLocation = 300;
59
60         collisionChance = new Choice<Boolean>(m.getController());
61         collisionChance.addChoice(0.9, false);
62        ollisionChance.addChoice(0.1, true);
63
64         damageLevel = new Choice<String>(m.getController());
65         damageLevel.addChoice(0.33, "high");
66         damageLevel.addChoice(0.33, "moderate");
67         damageLevel.addChoice(0.34, "low");
68
69         initGridInformation();
70         initPassengerList();
71         initObstacles();
72
73
74        if(simulate) {
75            Client.sendMessage( new String[] {"clear",
                  String.valueOf(maxGridSize)} );
76
```

```
77          Client.sendMessage(
78          new String[] {"pickUp",
               String.valueOf(currentPassenger.getPickUp().getX()),
               String.valueOf(currentPassenger.getPickUp().getY())} );
79          Client.sendMessage(
80          new String[] {"dropOff",
               String.valueOf(currentPassenger.getDropOff().getX()),
               String.valueOf(currentPassenger.getDropOff().getY())});
81
82          try {
83            TimeUnit.MILLISECONDS.sleep(100);
84          } catch(Exception e) {
85            System.err.println(e);
86          }
87       }
88      environmentGrid.get(GridCell.getIndex(car.getX(),
           car.getY())).setObstacle(false);
89      environmentGrid.get(GridCell.getIndex(depotLocation.getX(),
           depotLocation.getY())).setObstacle(false);
90
91      showAllPassengerList();
92  }
93
94  private void showAllPassengerList() {
95      for (Passenger passenger : passengers) {
96       System.err.println(String.format("Passenger %s: PickUp(%d,%d) - Drop
            Off (%d,%d)", passenger.getName(), passenger.getPickUp().getX(),
            passenger.getPickUp().getY(),passenger.getDropOff().getX(),
            passenger.getDropOff().getY()));
97      }
98  }
99
100 private void initGridInformation() {
101     environmentGrid = new HashMap<String, GridCell>();
102
103     for (int x = minGridSize; x < maxGridSize; x++) {
104         for (int y = minGridSize; y < maxGridSize; y++) {
105             String cellName = GridCell.getIndex(x, y);
106             environmentGrid.put(cellName, new GridCell(x, y, false));
107         }
108     }
109 }
```

```
110
111     private void initPassengerList() {
112         for (int i = 0; i < nPassengers; i++) {
113             int pickUpX = (int) (Math.random() * (maxGridSize));
114             int pickUpY = (int) (Math.random() * (maxGridSize));
115             int dropOffX, dropOffY;
116             do {
117                 dropOffX = (int) (Math.random() * (maxGridSize));
118                 dropOffY = (int) (Math.random() * (maxGridSize));
119             }while(pickUpX == dropOffX && pickUpY == dropOffY);
120             passengers.add(new Passenger("" + i, new Coordinate(pickUpX,
                    pickUpY), new Coordinate(dropOffX, dropOffY)));
121         }
122     }
123
124     private void initObstacles() {
125         int x, y;
126         for(int i = 0; i < this.nObstacles; i++) {
127             do {
128             x = (int) (Math.random() * maxGridSize);
129             y = (int) (Math.random() * maxGridSize);
130            }while(environmentGrid.get(GridCell.getIndex(x, y)).hasObstacle() ||
                   (x == this.depotLocation.getX() && y ==
                  this.depotLocation.getY()));
131             environmentGrid.get(GridCell.getIndex(x, y)).setObstacle(true);
132         }
133     }
134
135     public Unifier executeAction(String agName, Action act) throws AILexception
            {
136         String actionName = act.getFunctor();
137
138         int fromX, fromY, x, y, destinationX, destinationY;
139         String direction;
140
141         switch(actionName) {
142             case "drive":
143                 fromX = Util.getIntTerm(act.getTerm(0));
144                 fromY = Util.getIntTerm(act.getTerm(1));
145                 direction = act.getTerm(2).getFunctor();
146                 destinationX = Util.getIntTerm(act.getTerm(3));
147                 destinationY = Util.getIntTerm(act.getTerm(4));
```

```
148                 drive(agName, new Coordinate(fromX, fromY), direction, new
                        Coordinate(destinationX, destinationY));
149             break;
150             case "compass":
151                 x = Util.getIntTerm(act.getTerm(0));
152                 y = Util.getIntTerm(act.getTerm(1));
153                 compass(agName, new Coordinate(x,y));
154             break;
155             case "localize":
156                 localize(agName);
157             break;
158             case "get_ride":
159                 getRide(agName);
160             break;
161             case "refuse_ride":
162                 String refuseType = act.getTerm(0).getFunctor();
163                 refuseRide(agName, refuseType);
164             break;
165             case "park":
166                 String parkType = act.getTerm(0).getFunctor();
167                 park(agName, parkType);
168             break;
169             case "no_further_from":
170                 fromX = Util.getIntTerm(act.getTerm(0));
171                 fromY = Util.getIntTerm(act.getTerm(1));
172                 x = Util.getIntTerm(act.getTerm(2));
173                 y = Util.getIntTerm(act.getTerm(3));
174                 destinationX = Util.getIntTerm(act.getTerm(4));
175                 destinationY = Util.getIntTerm(act.getTerm(5));
176                 noFurtherFrom(agName, new Coordinate(fromX, fromY), new
                        Coordinate(x, y), new Coordinate(destinationX, destinationY));
177             break;
178             case "call_emergency":
179                 x = Util.getIntTerm(act.getTerm(0));
180                 y = Util.getIntTerm(act.getTerm(1));
181                 callEmergency(agName, new Coordinate(x, y));
182             break;
183             case "get_assistance":
184                 x = Util.getIntTerm(act.getTerm(0));
185                 y = Util.getIntTerm(act.getTerm(1));
186                 getAssistance(agName, new Coordinate(x, y));
187             break;
```

```
188                 default:
189             }
190 
191             return super.executeAction(agName, act);
192 
193     }
194 
195     private void drive(String agName, Coordinate from, String direction,
            Coordinate destination) {
196         this.currentDirection = direction;
197 
198         Coordinate newPosition = getDirectionCoordinate(direction, car);
199 
200         Predicate adaptedFromTo = new Predicate("adapt_from_to");
201         Predicate movedFromTo = new Predicate("moved_from_to");
202         addFromTo(agName, adaptedFromTo, from, car, direction, destination);
203         addFromTo(agName, movedFromTo, from, newPosition, direction,
                destination);
204 
205         System.err.println(String.format("Moving %s from (%d,%d) to (%d,%d)",
                direction, car.getX(), car.getY(), newPosition.getX(),
                newPosition.getY()));
206 
207         updateLocation(agName, car, newPosition);
208     }
209 
210     private void updateLocation(String agName, Coordinate currentPosition,
            Coordinate newPosition) {
211         if(simulate) {
212             try {
213                 TimeUnit.MILLISECONDS.sleep(waitTimeLocation);
214             } catch(Exception e) {
215                 System.err.println(e);
216             }
217 
218             Client.sendMessage( Client.convertArray2String( new String[]
                    {"carLocation", String.valueOf(newPosition.getX()),
                    String.valueOf(newPosition.getY()), this.currentDirection} ) );
219         }
220 
221         Predicate oldLocation = new Predicate("at");
222         oldLocation.addTerm(new NumberTermImpl(currentPosition.getX()));
```

```
223            oldLocation.addTerm(new NumberTermImpl(currentPosition.getY()));
224
225            Predicate at = new Predicate("at");
226            at.addTerm(new NumberTermImpl(newPosition.getX()));
227            at.addTerm(new NumberTermImpl(newPosition.getY()));
228
229            car.setX(newPosition.getX());
230            car.setY(newPosition.getY());
231
232            scanSurroundings(agName, newPosition);
233
234            removePercept(agName, oldLocation);
235            addPercept(agName, at);
236        }
237
238        private void scanSurroundings(String agName, Coordinate currentPosition) {
239            char north = verifyObstacle(agName, "north", currentPosition);
240            char south = verifyObstacle(agName, "south", currentPosition);
241            char east = verifyObstacle(agName, "east", currentPosition);
242            char west = verifyObstacle(agName, "west", currentPosition);
243
244            int obstacleAround = 0;
245            if(north != 'N')  obstacleAround++;
246            if(south != 'N')  obstacleAround++;
247            if(east != 'N')   obstacleAround++;
248            if(west != 'N')   obstacleAround++;
249
250            if(obstacleAround == 3 && collisionChance.get_choice()) {
251                addObstacleDamage(agName, north, "north", currentPosition);
252                addObstacleDamage(agName, south, "south", currentPosition);
253                addObstacleDamage(agName, east, "east", currentPosition);
254                addObstacleDamage(agName, west, "west", currentPosition);
255
256                Predicate unavoidableCollision = new
                       Predicate("unavoidable_collision");
257                unavoidableCollision.addTerm(new
                       NumberTermImpl(currentPosition.getX()));
258                unavoidableCollision.addTerm(new
                       NumberTermImpl(currentPosition.getY()));
259                obstacleDamageRelated.add(unavoidableCollision);
260
261                addPercept(agName, unavoidableCollision);
```

```
262
263             if(simulate) {
264                 try {
265                     TimeUnit.MILLISECONDS.sleep(waitTimeDefault);
266                 } catch(Exception e) {
267                     System.err.println(e);
268                 }
269             }
270
271         }
272     }
273
274     private void addObstacleDamage(String agName, char typeObstacle, String
        direction, Coordinate currentPosition) {
275         String currentDamageLevel = " ";
276
277       if(typeObstacle != 'F') {
278             currentDamageLevel = this.damageLevel.get_choice();
279
280             if (typeObstacle == 'N') {
281             Coordinate directionPosition = getDirectionCoordinate(direction,
                    currentPosition);
282
283             Predicate newObstacle = addObstacle("center", directionPosition);
284             addPercept(agName, newObstacle);
285
286             Predicate tempObstacle = new Predicate("temp_obstacle");
287             tempObstacle.addTerm(new NumberTermImpl(directionPosition.getX()));
288             tempObstacle.addTerm(new NumberTermImpl(directionPosition.getY()));
289             addPercept(agName, tempObstacle);
290
291             obstacleDamageRelated.add(newObstacle);
292             obstacleDamageRelated.add(tempObstacle);
293         }
294
295         Predicate obstacleDamage = new Predicate("obstacle_damage");
296         obstacleDamage.addTerm(new NumberTermImpl(currentPosition.getX()));
297         obstacleDamage.addTerm(new NumberTermImpl(currentPosition.getY()));
298         obstacleDamage.addTerm(new Predicate(direction));
299             obstacleDamage.addTerm(new Predicate( currentDamageLevel ));
300
301         if(simulate) {
```

```
302                 Coordinate directionPosition = getDirectionCoordinate(direction,
                        currentPosition) ;
303             Client.sendMessage( new String[] {"obstacleDamage",
                    String.valueOf(directionPosition.getX()),
                    String.valueOf(directionPosition.getY()), currentDamageLevel} );
304         }
305         obstacleDamageRelated.add(obstacleDamage);
306         addPercept(agName, obstacleDamage);
307     }
308 }
309
310 private char verifyObstacle(String agName, String direction, Coordinate
        currentPosition) {
311
312     Coordinate surroundingPosition = getDirectionCoordinate(direction,
            currentPosition);
313
314     char typeObstacle = 'N';
315     boolean hasObstacle = false;
316
317     if ( surroundingPosition.getX() < minGridSize ||
            surroundingPosition.getX() >= maxGridSize ||
            surroundingPosition.getY() < minGridSize || surroundingPosition.getY()
            >= maxGridSize) {
318         hasObstacle = true;
319         typeObstacle = 'F';
320     } else {
321         if (environmentGrid.get(GridCell.getIndex(surroundingPosition.getX(),
                surroundingPosition.getY())).hasObstacle()) {
322             hasObstacle = true;
323             typeObstacle = 'O';
324
325             if(simulate) {
326                 Client.sendMessage( new String[] {"obstacle",
327                         String.valueOf(surroundingPosition.getX()),
                            String.valueOf(surroundingPosition.getY())} );
328             }
329         }
330     }
331
332     if(hasObstacle) {
333         addPercept(agName, addObstacle(direction, currentPosition));
```

```
334            addPercept(agName, addObstacle("center", surroundingPosition));
335        }
336
337        return typeObstacle;
338    }
339
340    private Predicate addObstacle(String direction, Coordinate coordinate) {
341        Predicate obstacle = new Predicate("obstacle");
342        obstacle.addTerm(new Predicate(direction));
343        obstacle.addTerm(new NumberTermImpl(coordinate.getX()));
344        obstacle.addTerm(new NumberTermImpl(coordinate.getY()));
345        return obstacle;
346    }
347
348    private void addFromTo(String agName, Predicate predicate, Coordinate from,
         Coordinate current, String direction, Coordinate destination) {
349        predicate.addTerm(new NumberTermImpl(from.getX()));
350        predicate.addTerm(new NumberTermImpl(from.getY()));
351        predicate.addTerm(new NumberTermImpl(current.getX()));
352        predicate.addTerm(new NumberTermImpl(current.getY()));
353        predicate.addTerm(new Predicate(direction));
354        predicate.addTerm(new NumberTermImpl(destination.getX()));
355        predicate.addTerm(new NumberTermImpl(destination.getY()));
356        addPercept(agName, predicate);
357    }
358
359
360    private void addAdaptAndMovedFromTo(String agName, Coordinate from,
         Coordinate currentPosition,Coordinate destination, String moved, String
         adapt) {
361        Predicate adaptFromTo = new Predicate("adapt_from_to");
362        Predicate movedFromTo = new Predicate("moved_from_to");
363        addFromTo(agName, adaptFromTo, from, getDirectionCoordinate(moved,
             currentPosition), adapt, destination);
364        addFromTo(agName, movedFromTo, from, getDirectionCoordinate(moved,
             currentPosition), moved, destination);
365    }
366
367    private void noFurtherFrom(String agName, Coordinate from, Coordinate
         currentPosition, Coordinate destination) {
368         System.err.println( "Can't come here:
               at("+currentPosition.getX()+","+currentPosition.getY()+") " + "to
```

```
            get to ("+destination.getX()+","+destination.getY()+")");
369
370        addAdaptAndMovedFromTo(agName, from, currentPosition, destination,
               "north", "south");
371        addAdaptAndMovedFromTo(agName, from, currentPosition, destination,
               "south", "north");
372        addAdaptAndMovedFromTo(agName, from, currentPosition, destination, "east",
               "west");
373        addAdaptAndMovedFromTo(agName, from, currentPosition, destination, "west",
               "east");
374
375
376        Predicate noFurther = new Predicate("no_further");
377        noFurther.addTerm(new NumberTermImpl(from.getX()));
378        noFurther.addTerm(new NumberTermImpl(from.getY()));
379        noFurther.addTerm(new NumberTermImpl(currentPosition.getX()));
380        noFurther.addTerm(new NumberTermImpl(currentPosition.getY()));
381        noFurther.addTerm(new NumberTermImpl(destination.getX()));
382        noFurther.addTerm(new NumberTermImpl(destination.getY()));
383        addPercept(agName, noFurther);
384    }
385
386    private void compass(String agName, Coordinate coordinate) {
387         removeDirection(agName, "north");
388         removeDirection(agName, "south");
389         removeDirection(agName, "east");
390         removeDirection(agName, "west");
391
392         if (coordinate.getY() > car.getY()) addDirection(agName, "north");
393         if (coordinate.getY() < car.getY()) addDirection(agName, "south");
394         if (coordinate.getX() > car.getX()) addDirection(agName, "east");
395         if (coordinate.getX() < car.getX()) addDirection(agName, "west");
396
397         Predicate receiveDirection = new Predicate("receive_direction");
398         addPercept(agName, receiveDirection);
399    }
400
401    private void addDirection(String agName, String direction) {
402         Predicate go = new Predicate(direction);
403        addPercept(agName, go);
404    }
405
```

```
406  private void removeDirection(String agName, String direction) {
407    Predicate go = new Predicate(direction);
408    removePercept(agName, go);
409  }
410
411  private void refuseRide(String agName, String refuseType) {
412      if(simulate){
413       try {
414             TimeUnit.MILLISECONDS.sleep(this.waitTimeDefault);
415       } catch(Exception e) {
416          System.err.println(e);
417       }
418    }
419
420    String type = "";
421    switch (refuseType) {
422          case "pick_up":
423          type = "Pick up";
424       break;
425        case "drop_off":
426             type = "Drop off";
427       break;
428          case "car_unavailable":
429             type = "Car Unavailable - Total Loss";
430       break;
431        default: break;
432    }
433
434    if(simulate)
435    {
436        Client.sendMessage( new String[] {"refuseRide",
           String.valueOf(car.getX()), String.valueOf(car.getY()),
           refuseType} );
437        try {
438             TimeUnit.MILLISECONDS.sleep(this.waitTimeDefault);
439        } catch(Exception e) {
440             System.err.println(e);
441        }
442    }
443    System.err.println(String.format("%s: %s can't finish ride for %s", type,
        agName, currentPassenger.getName()));
444  }
```

```
445
446    private void park(String agName, String parkType) {
447       switch (parkType) {
448           case "pick_up":
449                 System.err.println(String.format("Pick Up Passanger %s in
                     (%d,%d)", currentPassenger.getName(), car.getX(),
                     car.getY()));
450          break;
451             case "drop_off":
452                 System.err.println(String.format("Drop Off Passanger %s in
                     (%d,%d)\n", currentPassenger.getName(), car.getX(),
                     car.getY()));
453          break;
454       case "depot":
455          if(car.getX() == depotLocation.getX() && car.getY() ==
                 depotLocation.getY())
456             System.err.println(String.format("%s is back to the depot.",
                   agName));
457          break;
458       default:
459          break;
460       }
461    }
462
463    private void localize(String agName) {
464         System.err.println("Initializing GPS");
465       System.err.println(String.format("Agent %s is at (%d,%d)", agName,
             car.getX(), car.getY()));
466       addDepot(agName);
467       updateLocation(agName, new Coordinate(minGridSize, minGridSize), car);
468    }
469
470    private void addDepot(String agName) {
471       if(simulate) Client.sendMessage(new String[] {"depot",
             String.valueOf(depotLocation.getX()),
             String.valueOf(depotLocation.getY())});
472
473       Predicate depot = new Predicate("depot");
474       depot.addTerm(new NumberTermImpl(depotLocation.getX()));
475       depot.addTerm(new NumberTermImpl(depotLocation.getY()));
476
477       addPercept(agName, depot);
```

```
478     }
479
480     private void getRide(String agName) {
481          Predicate pickUpLast = new Predicate("pick_up");
482        pickUpLast.addTerm(new
               NumberTermImpl(currentPassenger.getPickUp().getX()));
483        pickUpLast.addTerm(new
               NumberTermImpl(currentPassenger.getPickUp().getY()));
484
485        Predicate dropOffLast = new Predicate("drop_off");
486        dropOffLast.addTerm(new
               NumberTermImpl(currentPassenger.getDropOff().getX()));
487        dropOffLast.addTerm(new
               NumberTermImpl(currentPassenger.getDropOff().getY()));
488
489        removePercept(agName, pickUpLast);
490        removePercept(agName, dropOffLast);
491
492        if (passengers.isEmpty()) {
493           Predicate noPossibleRide = new Predicate("no_possible_new_ride");
494           addPercept(agName, noPossibleRide);
495           System.err.println("No more available rides!");
496        } else {
497           currentPassenger = passengers.get(0);
498           passengers.remove(0);
499
500           int pickUpX = currentPassenger.getPickUp().getX();
501           int pickUpY = currentPassenger.getPickUp().getY();
502
503           int dropOffX = currentPassenger.getDropOff().getX();
504           int dropOffY = currentPassenger.getDropOff().getY();
505
506           if(simulate){
507              Client.sendMessage( new String[] {"pickUp", String.valueOf(pickUpX),
                     String.valueOf(pickUpY)} );
508              Client.sendMessage( new String[] {"dropOff",
                     String.valueOf(dropOffX), String.valueOf(dropOffY)});
509           }
510
511
512           System.err.println(String.format("\n%s going to pick up %s in (%d,%d)",
                  agName, currentPassenger.getName(),pickUpX, pickUpY));
```

```
513         System.err.println(String.format("%s going to drop off %s in (%d,%d)",
                agName, currentPassenger.getName(),dropOffX, dropOffY));
514
515         Predicate pickUp = new Predicate("pick_up");
516         pickUp.addTerm(new NumberTermImpl(pickUpX));
517         pickUp.addTerm(new NumberTermImpl(pickUpY));
518
519         Predicate dropOff = new Predicate("drop_off");
520         dropOff.addTerm(new NumberTermImpl(dropOffX));
521         dropOff.addTerm(new NumberTermImpl(dropOffY));
522
523         addPercept(agName, pickUp);
524         addPercept(agName, dropOff);
525       }
526
527       Predicate rideInfo = new Predicate("ride_info");
528       addPercept(agName, rideInfo);
529     }
530
531     private void callEmergency(String agName, Coordinate currentPosition) {
532       System.err.println(String.format("%s crashed in (%d,%d). Calling
            Emergency.", agName, currentPosition.getX(), currentPosition.getY()));
533
534       Predicate emergency = new Predicate("emergency");
535       emergency.addTerm(new NumberTermImpl(currentPosition.getX()));
536       emergency.addTerm(new NumberTermImpl(currentPosition.getY()));
537       this.obstacleDamageRelated.add(emergency);
538       addPercept(agName, emergency);
539     }
540
541     private void getAssistance(String agName, Coordinate currentPosition) {
542       for(Predicate obstacleDamage : this.obstacleDamageRelated) {
543         removePercept(agName, obstacleDamage);
544       }
545       this.obstacleDamageRelated.clear();
546
547       if(simulate) {
548         try {
549           TimeUnit.MILLISECONDS.sleep(this.waitTimeDefault);
550         } catch(Exception e) {
551           System.err.println(e);
552         }
```

```
553     }
554
555     if(simulate) {
556        Client.sendMessage( new String[] {"removeObstacleDamage"} );
557     }
558
559     Predicate assisted = new Predicate("assisted");
560     assisted.addTerm(new NumberTermImpl(currentPosition.getX()));
561     assisted.addTerm(new NumberTermImpl(currentPosition.getY()));
562     addPercept(agName, assisted);
563  }
564
565
566  private Coordinate getDirectionCoordinate(String direction, Coordinate
         coordinate) {
567     Coordinate newCoordinate;
568
569     switch (direction) {
570        case "north":
571           newCoordinate = new Coordinate(coordinate.getX(),
                  coordinate.getY()+1);
572        break;
573        case "south":
574           newCoordinate = new Coordinate(coordinate.getX(),
                  coordinate.getY()-1);
575        break;
576        case "east":
577           newCoordinate = new Coordinate(coordinate.getX()+1,
                  coordinate.getY());
578        break;
579        case "west":
580           newCoordinate = new Coordinate(coordinate.getX()-1,
                  coordinate.getY());
581        break;
582        default:
583           newCoordinate = new Coordinate(coordinate.getX(), coordinate.getY());
584        break;
585     }
586
587     return newCoordinate;
588  }
589 }
```

Fonte: Autoria Própria

APÊNDICE C - UTIL

A implementação das classes *Coordinate*, *GridCell* e *Passenger* do módulo UTIL são apresentadas respectivamente nos códigos 28, 29 e 30.

Código 28 – Classe `Coordinate`

```
package util;

public class Coordinate {

   private int x;
   private int y;

   public Coordinate(int x, int y) {
      this.x = x;
      this.y = y;
   }

   public int getX() {
      return this.x;
   }

   public int getY() {
      return this.y;
   }

   public void setX(int x) {
      this.x = x;
   }

   public void setY(int y) {
      this.y = y;
   }

}
```

Fonte: Autoria Própria

Código 29 – Classe `GridCell`

```
package util;

```

```
3  public class GridCell {
4
5     private Coordinate coordinate;
6
7     private boolean hasObstacle;
8     private boolean isVisible;
9     private String damageLevel = "none";
10
11    public GridCell(Coordinate coordinate) {
12       this.coordinate = coordinate;
13    }
14
15    public GridCell(int gridX, int gridY) {
16       this.coordinate = new Coordinate(gridX, gridY);
17    }
18
19    public GridCell(Coordinate coordinate, boolean hasObstacle) {
20       this.coordinate = coordinate;
21       this.hasObstacle = hasObstacle;
22    }
23
24    public GridCell(int gridX, int gridY, boolean hasObstacle) {
25       this.coordinate = new Coordinate(gridX, gridY);
26       this.hasObstacle = hasObstacle;
27    }
28
29    public GridCell(int gridX, int gridY, boolean hasObstacle, boolean
          isVisible) {
30       this.coordinate = new Coordinate(gridX, gridY);
31       this.hasObstacle = hasObstacle;
32       this.isVisible = isVisible;
33    }
34
35    public GridCell(int gridX, int gridY, boolean hasObstacle, boolean
          isVisible, String damageLevel) {
36       this.coordinate = new Coordinate(gridX, gridY);
37       this.hasObstacle = hasObstacle;
38       this.isVisible = isVisible;
39       this.damageLevel = damageLevel;
40    }
41
42
```

```
43    public Coordinate getCoordinate() {
44        return this.coordinate;
45    }
46
47    public void setObstacle(boolean hasObstacle) {
48        this.hasObstacle = hasObstacle;
49    }
50    public boolean hasObstacle() {
51        return this.hasObstacle;
52    }
53
54    public void setIsVisible(boolean isVisible) {
55        this.isVisible = isVisible;
56    }
57    public boolean isVisible() {
58        return this.isVisible;
59    }
60
61    public String getDamageLevel() {
62        return this.damageLevel;
63    }
64    public void setDamageLevel(String damageLevel) {
65        this.damageLevel = damageLevel;
66    }
67
68    public static String getIndex (int x, int y) {
69        return String.format("%d,%d", x, y);
70    }
71
72 }
```

Fonte: Autoria Própria

Código 30 – Classe Passenger

```
1 package util;
2
3 public class Passenger {
4
5     private String name;
6
7     private Coordinate pickUp;
8     private Coordinate dropOff;
```

```
9
10    public Passenger(Coordinate pickUp, Coordinate dropOff) {
11       this.pickUp = pickUp;
12       this.dropOff = dropOff;
13    }
14
15    public Passenger(String name, Coordinate pickUp, Coordinate dropOff) {
16       this.name = name;
17
18       this.pickUp = pickUp;
19       this.dropOff = dropOff;
20    }
21
22    public String getName() {
23       return name;
24    }
25
26    public Coordinate getPickUp() {
27       return pickUp;
28    }
29
30    public Coordinate getDropOff() {
31       return dropOff;
32    }
33
34 }
```

Fonte: Autoria Própria

APÊNDICE D - SIMULADOR

A implementação das classes `Client` e `Simulator`, responsáveis pela comunicação do ambiente com o agente são apresentados respectivamente nos Códigos 31 e 32.

Código 31 – Classe `Client`

```
1  18package main;
2
3  import java.net.*;
4
5  public class Client {
6
7     public static String convertArray2String(String[] stringArray) {
8
9        String stringConverted = "";
10
11       for( String string : stringArray ) {
12          stringConverted += string + ";";
13       }
14
15       return stringConverted;
16    }
17
18    public static void sendMessage (String message) {
19          try {
20             DatagramSocket client = new DatagramSocket();
21             InetAddress IPAddress = InetAddress.getByName("localhost");
22             byte[] sendData = new byte[1024];
23             sendData = message.getBytes();
24
25             DatagramPacket sendPacket = new DatagramPacket(sendData,
                   sendData.length, IPAddress, 9999);
26             client.send(sendPacket);
27
28             client.close();
29          }
30          catch (Exception e) {
31             System.out.println(e);
32          }
33       }
34
```

```
35        public static void sendMessage(String[] stringArray) {
36            sendMessage (convertArray2String (stringArray) );
37        }
38  }
```

Fonte: Autoria Própria

Código 32 – Classe `Simulator`

```
1   package main;
2
3   import java.awt.Color;
4   import java.awt.Graphics;
5   import java.awt.HeadlessException;
6   import javax.swing.JComponent;
7   import javax.swing.JFrame;
8   import javax.swing.WindowConstants;
9   import java.net.*;
10  import java.util.HashMap;
11  import java.util.Map;
12  import util.Coordinate;
13  import util.GridCell;
14
15  public class Simulator extends JComponent{
16      private static final long serialVersionUID = 1L;
17
18      private JFrame window = new JFrame();
19
20      private int width = 800;
21      private int height = 800;
22
23      private int gridSize = 13;
24      private int gridAmplifier = 55;
25
26      private Coordinate car = new Coordinate(0, 0);
27      private Color carColor = Color.BLUE;
28      private String direction = "north";
29      private boolean ableToMove = true;
30      private boolean isCrashed = false;
31
32      private Coordinate depot = new Coordinate(0, 0);
33      private Color depotColor = Color.green;
34      private Map<String, GridCell> environmentGrid;
```

```
35
36  private Coordinate pickUp = new Coordinate(gridSize-1, gridSize-1);
37  private Color pickUpColor = Color.cyan;
38
39  private Coordinate dropOff = new Coordinate(gridSize-2, gridSize-2);
40  private Color dropOffColor = Color.yellow;
41
42  private boolean refuseRide = false;
43  private boolean parked = true;
44
45  private Color high = new Color(255,0,0);
46  private Color moderate = new Color(255,102,102);
47  private Color low = new Color(255,204,204);
48
49
50  private static DatagramSocket server;
51
52  private Simulator() throws HeadlessException {
53     this.window.setTitle("Autonomous Car Simulator");
54     this.window.setSize(width, height);
55     this.window.setLocationRelativeTo(null);
56     this.window.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
57
58     initGridInformation();
59     this.window.add(this);
60     this.window.setVisible(true);
61  }
62
63  private void initGridInformation() {
64     car = new Coordinate(0, 0);
65     depot = new Coordinate(0, 0);
66
67     environmentGrid = new HashMap<String, GridCell>();
68     for (int x = 0; x < gridSize; x++) {
69        for (int y = 0; y < gridSize; y++) {
70
71           String cellName = GridCell.getIndex(x, y);
72           environmentGrid.put(cellName, new GridCell(x, y, false, false));
73
74        }
75     }
76  }
```

```
77
78    protected void drawLineGrid(Graphics g) {
79       g.setColor( new Color(0, 0, 0) );
80       for (int x = 0; x <= this.gridSize ; x++){
81          g.drawLine(convertX(amplify(x)), convertY(0), convertX(amplify(x)),
              convertY( amplify(gridSize) ));
82       }
83       for (int y = 0; y <= this.gridSize ; y++){
84          g.drawLine(convertX(0), convertY( amplify(y) ), convertX(
              amplify(gridSize) ), convertY( amplify(y) ));
85       }
86       for (int x = 0; x < this.gridSize; x++){
87          g.drawString(String.valueOf(x), convertX( amplify(x) + (int)
              (this.gridAmplifier * 0.4)), convertY( -20 ));
88       }
89       for (int y = 0; y < this.gridSize; y++) {
90          g.drawString(String.valueOf(y), convertX( -15 ), convertY( amplify(y) +
              (int) (this.gridAmplifier * 0.4) ));
91       }
92    }
93
94    @Override
95    protected void paintComponent(Graphics g) {
96       if (!this.ableToMove) {
97          g.setColor(Color.RED);
98          for (int x = 0; x < this.gridSize; x++) {
99             for (int y = 0; y < this.gridSize; y++) {
100               g.fillRect( getGridX( x ), getGridY( y ), this.gridAmplifier,
                    this.gridAmplifier);
101
102            }
103         }
104         g.setColor(Color.WHITE);
105         g.fillRect( getGridX( car.getX() ), getGridY( car.getY() ),
              this.gridAmplifier, this.gridAmplifier);
106
107         drawLineGrid(g);
108         return;
109      }
110
111      g.setColor(Color.WHITE);
112      g.fillRect(0, 0, this.width, this.height);
```

```
113
114        boolean isVisible, hasObstacle, hasDamageLevel;
115
116        for (int x = 0; x < this.gridSize; x++) {
117           for (int y = 0; y < this.gridSize; y++) {
118              isVisible = this.environmentGrid.get(GridCell.getIndex(x,
                     y)).isVisible();
119              hasObstacle = this.environmentGrid.get(GridCell.getIndex(x,
                     y)).hasObstacle();
120              hasDamageLevel = !this.environmentGrid.get(GridCell.getIndex(x,
                     y)).getDamageLevel().equals("none");
121
122              if(hasObstacle) {
123                 g.setColor(Color.orange);
124                 g.fillRect( getGridX( x ), getGridY( y ), this.gridAmplifier,
                        this.gridAmplifier);
125              }
126              if( hasDamageLevel ){
127                 String damageLevel =
                        this.environmentGrid.get(GridCell.getIndex(x,
                        y)).getDamageLevel();
128                 switch(damageLevel) {
129                 case "high":
130                    g.setColor(high);
131                    break;
132                 case "moderate":
133                    g.setColor(moderate);
134                    break;
135                 case "low":
136                    g.setColor(low);
137                    break;
138                 }
139                 g.fillRect( getGridX( x ), getGridY( y ), this.gridAmplifier,
                        this.gridAmplifier);
140              }
141              if( !isVisible ) {
142                 g.setColor(Color.BLACK);
143                 g.fillRect( getGridX( x ), getGridY( y ), this.gridAmplifier,
                        this.gridAmplifier);
144              }
145           }
146        }
```

```
147
148         if(parked) {
149             g.setColor(Color.LIGHT_GRAY);
150             g.fillRect(getXReduced(car.getX(), 95), getYReduced(car.getY(), 95),
                    reducedSize(95), reducedSize(95));
151         }
152 
153         if(isCrashed) {
154             g.setColor(Color.MAGENTA);
155             g.fillRect( getGridX( car.getX() ) , getGridY( car.getY() ),
                    this.gridAmplifier, this.gridAmplifier);
156         }
157 
158         g.setColor(depotColor);
159         g.fillRect(getXReduced(depot.getX(), 90), getYReduced(depot.getY(), 90),
                reducedSize(90), reducedSize(90));
160 
161         g.setColor(pickUpColor);
162         g.fillOval(getXReduced(pickUp.getX(), 80), getYReduced(pickUp.getY(), 80),
                reducedSize(80), reducedSize(80));
163 
164         g.setColor(dropOffColor);
165         g.fillOval(getXReduced(dropOff.getX(), 70), getYReduced(dropOff.getY(),
                70), reducedSize(70), reducedSize(70));
166 
167         g.setColor(Color.RED);
168         if( this.refuseRide ) {
169             g.drawLine(getGridX(pickUp.getX()), getGridY(pickUp.getY()),
                    getGridX(pickUp.getX()) + this.gridAmplifier,
                    getGridY(pickUp.getY()) + this.gridAmplifier);
170             g.drawLine(getGridX(pickUp.getX()), getGridY(pickUp.getY()) +
                    this.gridAmplifier, getGridX(pickUp.getX()) + this.gridAmplifier,
                    getGridY(pickUp.getY()));
171             g.drawLine(getGridX(dropOff.getX()), getGridY(dropOff.getY()),
                    getGridX(dropOff.getX()) + this.gridAmplifier,
                    getGridY(dropOff.getY()) + this.gridAmplifier);
172             g.drawLine(getGridX(dropOff.getX()), getGridY(dropOff.getY()) +
                    this.gridAmplifier, getGridX(dropOff.getX()) + this.gridAmplifier,
                    getGridY(dropOff.getY()));
173         }
174         int [] xPoints = null;
175         int [] yPoints = null;
```

```
176
177     switch(direction) {
178        case "north":
179           xPoints = new int[]{ getXReduced(car.getX(), 90), (int)
                 (getXReduced(car.getX(), 90) + reducedSize(90)/2),
                 getXReduced(car.getX(), 90) + reducedSize(90) };
180           yPoints = new int[]{ getYReduced(car.getY(), 90) + reducedSize(90),
                 getYReduced(car.getY(), 90) , getYReduced(car.getY(), 90) +
                 reducedSize(90)};
181        break;
182        case "south":
183           xPoints = new int[]{ getXReduced(car.getX(), 90), (int)
                 (getXReduced(car.getX(), 90) + reducedSize(90)/2),
                 getXReduced(car.getX(), 90) + reducedSize(90) };
184           yPoints = new int[]{ getYReduced(car.getY(), 90),
                 getYReduced(car.getY(), 90) + reducedSize(90),
                 getYReduced(car.getY(), 90)};
185        break;
186        case "east":
187           xPoints = new int[]{ getXReduced(car.getX(), 90),
                 getXReduced(car.getX(), 90) + reducedSize(90),
                 getXReduced(car.getX(), 90)};
188           yPoints = new int[]{ getYReduced(car.getY(), 90), (int)
                 (getYReduced(car.getY(), 90) + reducedSize(90)/2),
                 getYReduced(car.getY(), 90) + reducedSize(90)};
189           break;
190        case "west":
191           xPoints = new int[]{ getXReduced(car.getX(), 90)+ reducedSize(90),
                 getXReduced(car.getX(), 90) , getXReduced(car.getX(), 90)+
                 reducedSize(90)};
192           yPoints = new int[]{ getYReduced(car.getY(), 90), (int)
                 (getYReduced(car.getY(), 90) + reducedSize(90)/2),
                 getYReduced(car.getY(), 90) + reducedSize(90)};
193        break;
194     }
195     g.setColor(carColor);
196     g.fillPolygon(xPoints, yPoints, 3);
197     g.setColor(Color.BLACK);
198     g.drawPolygon(xPoints, yPoints, 3);
199
200     drawLineGrid(g);
201  }
```

```
202
203 private void readReceivedMessage(String message) {
204    String[] messageArray = message.split(";");
205    String switchMessage = messageArray[0];
206
207    int x = 0, y = 0;
208    String d;
209
210    if( !(switchMessage.equals("clear") ||
        switchMessage.equals("removeObstacleDamage")) ) {
211       x = Integer.parseInt( messageArray[1] );
212       y = Integer.parseInt( messageArray[2] );
213    }
214
215    switch(switchMessage) {
216       case "clear":
217          this.gridSize = Integer.parseInt( messageArray[1] );
218          this.ableToMove = true;
219          carColor = Color.blue;
220          initGridInformation();
221          break;
222       case "depot":
223          depot.setX( Integer.parseInt( messageArray[1] ) );
224          depot.setY( Integer.parseInt( messageArray[2] ) );
225          this.environmentGrid.get( GridCell.getIndex(x, y)
              ).setIsVisible(true);
226          break;
227       case "carLocation":
228          d = messageArray[3];
229
230          car.setX( x );
231          car.setY( y );
232
233          this.direction = d;
234
235          this.environmentGrid.get( GridCell.getIndex(x, y)
              ).setIsVisible(true);
236
237          if(y < (this.gridSize-1))
238             this.environmentGrid.get( GridCell.getIndex(x, y+1)
                 ).setIsVisible(true);
239
```

```
240         if(y > 0)
241           this.environmentGrid.get( GridCell.getIndex(x, y-1)
                 ).setIsVisible(true);
242
243         if(x < (this.gridSize-1))
244           this.environmentGrid.get( GridCell.getIndex(x+1, y)
                 ).setIsVisible(true);
245
246         if(x > 0)
247           this.environmentGrid.get( GridCell.getIndex(x-1, y)
                 ).setIsVisible(true);
248
249         if( (car.getX() == pickUp.getX() && car.getY() == pickUp.getY()) ||
250           (car.getX() == dropOff.getX() && car.getY() == dropOff.getY()) ||
251           (car.getX() == depot.getX() && car.getY() == depot.getY())
252         )
253            parked = true;
254         else
255            parked = false;
256
257         isCrashed = false;
258
259         String damageLevel = this.environmentGrid.get( GridCell.getIndex(x,
               y) ).getDamageLevel();
260
261         if( !damageLevel.equals("none") ){
262
263           if(damageLevel.equals("high")) carColor = high;
264             else if(damageLevel.equals("moderate") && carColor != high)
                   carColor = moderate;
265             else if(damageLevel.equals("low") && carColor != moderate)
                   carColor = low;
266
267           isCrashed = true;
268         }
269         break;
270       case "obstacle":
271         this.environmentGrid.get( GridCell.getIndex(x, y)
               ).setObstacle(true);
272         break;
273       case "pickUp":
274         this.pickUp.setX( x );
```

```
275            this.pickUp.setY( y );
276            this.refuseRide = false;
277            break;
278         case  "dropOff":
279            this.dropOff.setX( x );
280            this.dropOff.setY( y );
281            break;
282         case "obstacleDamage":
283            this.environmentGrid.get( GridCell.getIndex(x, y)
                   ).setDamageLevel(messageArray[3]);
284            break;
285         case "removeObstacleDamage":
286            for (int X = 0; X < this.gridSize; X++) {
287               for (int Y = 0; Y < this.gridSize; Y++) {
288                  this.environmentGrid.get( GridCell.getIndex(X,
                         Y)).setDamageLevel("none");
289               }
290            }
291            break;
292         case "refuseRide":
293            String type = messageArray[3];
294            switch(type) {
295            case "pick_up":
296               this.refuseRide = true;
297               break;
298            case "drop_off":
299                  parked = true;
300               this.refuseRide = true;
301               break;
302            case "car_unavailable":
303               this.ableToMove = false;
304               break;
305            }
306            break;
307         default:
308            System.out.println("Erro");
309            System.out.println(messageArray[0]);
310      }
311
312      repaint();
313   }
314
```

```
315
316    public static void main(String args[]){
317
318        Simulator sim = new Simulator();
319
320        try {
321
322            server = new DatagramSocket(9999);
323            byte[] receive = new byte[1024];
324
325            while(true)
326            {
327                DatagramPacket receivePacket = new DatagramPacket(receive,
                       receive.length); // Pacote Recebido
328                server.receive(receivePacket);
329
330                String sentence = new String( receivePacket.getData() );
331                sim.readReceivedMessage(sentence);
332            }
333        }
334        catch (Exception e) {
335            System.out.println(e);
336        }
337     }
338
339    private int getGridX(int x)
340    {
341        return convertX(0) + (x * this.gridAmplifier);
342    }
343
344    private int getGridY(int y)
345    {
346        return convertY(0) - this.gridAmplifier - (y * this.gridAmplifier);
347    }
348
349    private int getXReduced(int x, int percentage) {
350        return getGridX(x) + reducedSizeSpace(percentage);
351    }
352
353    private int getYReduced(int y, int percentage) {
354        return getGridY(y) + reducedSizeSpace(percentage);
355    }
```

```
356
357    private int reducedSize(int percentage) {
358       return (int) ( this.gridAmplifier * ( percentage/100.0 ));
359    }
360
361    private int reducedSizeSpace(int percentage) {
362       return (int) (this.gridAmplifier * ( (1 - percentage/100.0) / 2 ));
363    }
364
365    private int amplify(int number) {
366       return number * this.gridAmplifier;
367    }
368
369    private int convertX (int x) {
370       return x + 50;
371    }
372
373    private int convertY (int y) {
374       return this.getHeight() - y - 50;
375    }
376 }
```

Fonte: Autoria Própria

APÊNDICE E - VERIFICAÇÃO FORMAL

O código referentes as propriedades do agente é apresentado no Código 33.

Código 33 – Código da Especificação Formal das Propriedades do Agente

```
1   1:  [] ~B(vehicle, name(vehicle))
2
3   FAR_1: <> D(vehicle, location) -> <> P (at(0,0))
4
5   FAR_6: [] ( ~B(vehicle, ride_info) U D(vehicle, get_ride) )
6
7   FAR_7: <> ( B(vehicle, obstacle(center, 1, 2)) & P(pick_up(1, 2)) ) -> <> D(vehicle,
        refuse_ride(pick_up))
8
9   FAR_8_1_8_2: ( <> B(vehicle, pick_up(1,1)) -> <> G(vehicle, complete_journey(1,1)) )
        -> <> ( B(vehicle, at(1,1)) & D(vehicle, park(pick_up)) )
10
11  FAR_8_1_8_3: <>(B(vehicle,pick_up(1,2)) & G(vehicle,complete_journey(1,2)) &
        P(obstacle(center, 1, 2))) -> <> D (vehicle, refuse_ride(pick_up))
12
13  DT_DIRECTIONS_NORTH: <> G(vehicle, drive_to(0, 2)) -> <> P(at(0,2))
14  DT_DIRECTIONS_EAST: <> G(vehicle, drive_to(2, 2)) -> <> P(at(2,2))
15  DT_DIRECTIONS_SOUTH: <> G(vehicle, drive_to(2, 0)) -> <> P(at(2,0))
16  DT_DIRECTIONS_WEST: <> G(vehicle, drive_to(1, 0)) -> <> P(at(1,0))
17
18  DT_DIRECTIONS_NORTH_EAST: <> G(vehicle, drive_to(4, 3)) R <> P(at(4,3))
19  DT_DIRECTIONS_SOUTH_WEST: <> G(vehicle, drive_to(2, 1)) R <> P(at(2,1))
20  DT_DIRECTIONS_NORTH_WEST: <> G(vehicle, drive_to(0, 3)) R <> P(at(0,3))
21  DT_DIRECTIONS_SOUTH_EAST: <> G(vehicle, drive_to(1, 1)) R <> P(at(1,1))
22
23  AR_5: [] (  ( ( B(vehicle, east) & B(vehicle, north)  & B(vehicle, heading(north)) &
        B(vehicle, obstacle(north,_,_)) ) -> ( <> B(vehicle, adapt(east)) R <> D(vehicle,
        drive(_,_,north,_,_) ) ) ) & ~B(vehicle, crashed(_,_)) )
24
25  AR_6: [] (  ( ( B(vehicle, south) & ~B(vehicle, east) & ~B(vehicle, west)  & B(vehicle,
        heading(south)) & B(vehicle, obstacle(south,_,_)) & B(vehicle, obstacle(east,_,_)) &
        ~B(vehicle, obstacle(west,_,_)) ) -> ( <> B(vehicle, adapt(west)) R <> D(vehicle,
        drive(_,_,south,_,_) ) ) ) & ~B(vehicle, crashed(_,_)) )
26
27  AR_7: <> ( B(vehicle, west) & ~B(vehicle, can_adapt(north,_,_)) & ~B(vehicle,
        can_adapt(south,_,_)) & ~B(vehicle, can_adapt(west,_,_)) & B(vehicle,
```

```
      can_adapt(east,_,_)) ) −> ( <>G(vehicle, drive_direction(east) ) )
28
29  AR_8: <> ( B(vehicle, east) & B(vehicle, adapt(north)) & ~B(vehicle, can_adapt(north,_,_))
      ) −> ( <> D(vehicle, drive(_,_,south,_,_) ) R <> B(vehicle, adapt(south)) )
30
31  COC: []( (<> ( B(vehicle, unavoidable_collision (_,_)) & B(vehicle,
      obstacle_damage(_,_,_,low) ) ) −> <> G(vehicle, colide_obstacle(_, low)) ) &
32  (<> ( B(vehicle, unavoidable_collision (_,_)) & ~B(vehicle, obstacle_damage(_,_,_,low) ) &
      B(vehicle, obstacle_damage(_,_,_,moderate) ) ) −> <> G(vehicle, colide_obstacle(_,
      moderate))))
33
34
35  IFV: <> D(vehicle, honk)
36
37
38  TRIVIAL: <> B(vehicle, done_all_rides)
```

Fonte: Autoria Própria