Pontifícia Universidade Católica
do Rio de Janeiro

Francisco José Plácido da Cunha

Uma Abordagem de Teste Baseada em Modelo para Sistemas Normativos Autônomos

Tese de Doutorado

Tese apresentada como requisito parcial para obtenção do grau de Doutor pelo Programa de Pós–graduação em Informática da PUC-Rio.

Orientador: Prof. Carlos José Pereira de Lucena

Rio de Janeiro
Maio de 2019

Pontifícia Universidade Católica
do Rio de Janeiro

Francisco José Plácido da Cunha

Uma Abordagem de Teste Baseada em Modelo para Sistemas Normativos Autônomos

Tese apresentada como requisito parcial para obtenção do grau de Doutor pelo Programa de Pós–graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo.

Prof. Carlos José Pereira de Lucena
Orientador
Departamento de Informática – PUC-Rio

Prof. Helio Côrtes Vieira Lopes
Departamento de Informática – PUC–Rio

Prof. Simone Diniz Junqueira Barbosa
Departamento de Informática – PUC–Rio

Prof. Elder José Reioli Cirilo
Departamento de Informática – UFSJ

Prof. Paulo Sérgio Conceição Alencar
School of Computer Science – University of Waterloo

Rio de Janeiro, 23 de Maio de 2019

Francisco José Plácido da Cunha

Graduou-se em Ciência da Computação pela Universidade Federal Fluminense, UFF – Rio de Janeiro, Brasil. Recebeu o título de Mestre em Informática pelo Departamento de Informática da PUC-Rio em 2014. Atualmente pesquisa sobre o Desenvolvimento de Software Orientado a Agentes no Laboratório de Engenharia de Software (LES) da PUC-Rio.

Ficha Catalográfica

Cunha, Francisco José Plácido da

Uma Abordagem de Teste Baseada em Modelo para Sistemas Normativos Autônomos / Francisco José Plácido da Cunha; orientador: Carlos José Pereira de Lucena. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2019.

v., 161 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. Agentes Normativos;. 3. Agentes BDI;. 4. Teste em Sistemas Multiagente;. 5. Teste Baseado em Modelos.. I. Lucena, Carlos José Pereira de. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Gostaria de agradecer aqueles que, de alguma maneira, contribuíram para tornar esse sonho realidade!

Agradeço ao meu pai, Francisco Lemos da Cunha (*in memorian*) que sempre falou do filho estudando para ser doutor com orgulho e brilho nos olhos, e minha mãe Lizete Plácido da Cunha que, mesmo não compreendendo mais o significado dessa conquista, ainda intercede por mim em suas orações. Obrigado meu pai, obrigado meu amigo! Obrigado mamãe!

Agradeço a minha amada Tia Ceia pelas constantes orações e pelo impagável suporte familiar e a meu querido primo José Miguel pela ajuda que nem imagina que realiza.

Agradeço a minha fantástica esposa Carla Michele da Fonseca Soares da Cunha por todo carinho, apoio, incentivo e inigualável paciência, e ao meu *amigão* João Felipe Soares da Cunha que me obriga a ser melhor todos os dias ao copiar o papai em tudo. Afinal, "a gente é igualzinho né papai!"

Agradeço ao meu sogro Alenil Soares e minha sogra Elza da Fonseca Soares pela ajuda, carinho e incentivo. Essa conquista não seria possível sem vocês!

Agradeço ao meu orientador e amigo, Professor Carlos José Pereira de Lucena pela confiança, incentivo e por me tornar um pesquisador.

Agradeço ao amigo Marx Leles Viana por me ajudar a crescer a cada publicação e por me desafiar a uma constante evolução.

Agradeço ao Cláudio Silva, meu amigo e líder no Instituto TECGRAF por me ajudar a resgatar valores que nem sei como perdi, e também ao amigo Bernardo Tavares Breder que não imagina toda a inspiração que nosso convívio proporciona.

O presente trabalho foi realizado com o apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001 e do CNPq, sem os quais esta pesquisa não seria possível. Por fim, agradeço a PUC-Rio pela oportunidade de me tornar Doutor em Ciências de Informática exigindo para isso, apenas dedicação.

Resumo

Cunha, Francisco José Plácido da; Lucena, Carlos José Pereira de. **Uma Abordagem de Teste Baseada em Modelo para Sistemas Normativos Autônomos**. Rio de Janeiro, 2019. 161p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O uso de sistemas baseados em agentes é adequado à construção de software complexo. Para garantir uma ordem social desejável é preciso lidar com a autonomia e a diversidade de interesses dos agentes e as normas são mecanismos eficientes de controle usados para regular o comportamento dos agentes. O teste de software continua sendo amplamente aplicado para garantir a qualidade do software. No contexto de sistemas multiagentes normativos, os testes devem lidar com a necessidade dos agentes atuarem de forma robusta sob condições normativas dinâmicas as quais os desenvolvedores não consideraram. Neste contexto, propomos uma abordagem para testar agentes normativos que seguem o modelo *belief-desire-intention*. Como contribuições, esta tese apresenta: um framework para desenvolvimento de agentes BDI normativos, o NBDI4JADE; um modelo modelo de faltas para apoiar a identificação dos diferentes tipos de falhas em agentes normativos; um framework para testar agentes BDI normativos, o N-JAT4BDI e, um método para geração de casos de teste a partir de modelos ANA-ML dos agentes. Avaliamos o framework de teste através de um estudo experimental no qual discutimos a eficiência e a eficácia do framework. Avaliamos, também, a eficácia do método de geração de casos de teste, aplicando-o a dois cenários de uso: (i) um sistema para gerenciar a submissão e o processo de revisão de artigos em conferências, e (ii) um sistema de venda de pacotes turísticos de uma agência de viagens. Os resultados obtidos nas avaliações de ambos os frameworks apresentam indícios positivos da eficiência e eficácia na detecção e identificação de falhas em agentes normativos e eficiência na geração dos casos de teste.

Palavras-chave

Agentes Normativos; Agentes BDI; Teste em Sistemas Multiagente; Teste Baseado em Modelos.

Abstract

Cunha, Francisco José Plácido da; Lucena, Carlos José Pereira de (Advisor). **A Model-Based Testing Approach for Normative Autonomous Systems**. Rio de Janeiro, 2019. 161p. Tese de doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The use of agent-based systems is suitable for the construction of complex software. To guarantee a desirable social order one we must deal with the autonomy and diversity of interests of the agents and norms are efficient mechanisms of control used to regulate the behavior of the agents. Software testing still have been widely applied to ensure the software quality. In the context of normative multiagent systems, the test must deal with the need for the agents to act robustly under dynamic normative conditions in which the developers did not consider. In this context, we propose an approach to test normative agents that follow the *belief-desire-intention* model. Among the contributions, this thesis presents: a framework for development of normative BDI agents, the NBDI4JADE; a fault model to support the identification of different types of failures in normative agents; a framework for testing normative BDI agents, the N-JAT4BDI, and a method for generating test cases from ANA-ML models. We evaluated the test framework through an experimental study in which we discussed the efficiency and effectiveness of the framework. We also evaluate the effectiveness of the test case generation method, applying it to two usage scenarios: (i) a system to the manage submission and the review process of articles in conferences, and (ii) a system of sale of tour packages from a travel agency. The results obtained in the evaluations of both frameworks presented positive indications of efficiency and effectiveness in the detection and identification of failures in normative agents and efficiency in the generation of test cases.

Keywords

Normative Agents; BDI Agents; Testing in Multiagent Systems; Model-Based Testing.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

ACL – Agent Communication Language
ANA-ML – Adaptive Normative Agent - Modeling Language
AOSE – Agent Oriented Software Engineering
AUT – Agent Under Test
BOID – Belief-Obligation-Intention-Desire
BDI – Belief-Desire-Intention
EC – Expert Committee
FIPA - Foundation for Intelligent Physical Agents
GQM – Goal-Question-Metric
JADE – Java Agent DEvelopment Framework
SMA – Sistema Multiagente
TBM – Teste Baseado em Modelo

Eis que farei uma *coisa nova*, e, agora, sairá à luz;
porventura não a sabereis? Eis que porei um
caminho no deserto e *rios*, no ermo.

Isaías 43:19.

1
Introdução

O uso de *sistemas baseados em agentes* é adequado à construção de software complexo, que abrange desde sistemas distribuídos até "aplicações inteligentes" (Weiss, 1999) (Nwana, 1996). Um *agente* é uma entidade autônoma e proativa, que executa ações de forma a alcançar seus objetivos (Jennings e Wooldridge, 1996), possibilitando o uso de um paradigma onde são propostas novas formas de decompor, analisar, projetar, implementar e testar sistemas com base nas características centrais dos agentes (Oates et al., 1997) e em suas interações com outros agentes e com o ambiente.

Um *Sistema Multiagente* (SMA) é composto por *agentes* que, como mencionado, são entidades autônomas e dirigidas por metas (Wooldridge, 2009). No entanto, para garantir uma ordem social desejável, é preciso lidar com a autonomia e a diversidade de interesses dos agentes (Balke et al., 2013). *Normas* são mecanismos de controle social usadas para regular o comportamento dos agentes, definindo permissões, obrigações e proibições de suas ações, as quais o agente decide cumprir ou violar (Balke et al., 2013) (Lopez, 2003). Assim, um *sistema multiagente normativo* pode ser definido como um sistema onde um conjunto de normas são impostas aos agentes com a finalidade de regular seu comportamento para atingir os objetivos do grupo (Boella et al., 2006).

Diferentes domínios têm motivado um cenário emergente para o uso de sistemas multiagentes normativos (Criado et al., 2011) (Haynes et al., 2017) tais como: aplicações de comércio eletrônico, serviços bancários, controle de tráfego aéreo, simulações de cenários críticos, jogos, gerenciamento de informações, controle de rede e dispositivos e apoio a diagnósticos (Lucena et al., 2004) (Pěchouček e Mařík, 2008). Por exemplo, aplicações relacionadas a carros e caminhões autônomos (Manyika et al., 2013) geram a perspectiva de um grande impacto econômico no mercado nos próximos anos, e aplicações envolvendo robôs civis planejam investimentos na ordem de bilhões de euros até 2020 (Autefage et al., 2015). No entanto, ainda há uma lacuna nas metodologias de desenvolvimento orientadas a agentes em relação à tarefa de teste do software. O uso de sistemas normativos autônomos exige confiança em seu desempenho e a compreensão de como tais sistemas são afetados e reagem às normas aplicadas. Dessa forma, o teste de agentes é uma tarefa crucial e complexa, dado que

os agentes possuem comportamento dinâmico, proativo e orientado a objetivos (Padgham e Winikoff, 2005) (Winikoff e Cranefield, 2014).

Ainda sobre a perspectiva da qualidade, é indispensável que o processo de desenvolvimento de sistemas multiagentes adote métodos, técnicas e ferramentas que permitam o teste sistematizado e com fundamentação científica, de modo a aumentar a confiabilidade e a produtividade e diminuir os custos de produção do software (Delamaro et al., 2017). Entretanto, por serem os agentes, distribuídos, autônomos e possuir um mecanismo deliberativo sensível ao contexto, o teste de sistemas multiagentes torna-se uma tarefa desafiadora (Miles et al., 2010). Tais características, inerentes aos agentes, são conhecidas por serem difíceis não apenas de projetar e programar (Bergenti et al., 2006), mas também de testar, necessitando de métodos e estratégias capazes de lidar com sua natureza específica (Houhamdi, 2011).

Algumas abordagens propõem o uso da *verificação formal* como solução para verificar o comportamento dos agentes. Nesta abordagem, técnicas baseadas em lógica e matemática são usadas para demonstrar a conformidade entre o programa e sua especificação formal. Os sistemas que atendem ou cumprem suas especificações são considerados corretos (Bordini et al., 2006). Geralmente, a verificação formal utiliza a verificação de modelos (do inglês, *model checking*) ou a prova de programa por inferência lógica (Zheng e Alagar, 2005). No entanto, Bordini e Moreira enfatizam que a complexidade computacional de alguns tipos de agentes, torna o uso da verificação formal difícil e não prática para sistemas grandes (Bordini e Moreira, 2004). Miles *et al.* sugere como solução a combinação da verificação formal e dos testes de conformidade (Miles et al., 2010). O teste é uma alternativa prática e complementar à verificação formal (Zheng e Alagar, 2005) e o teste baseado em modelos é recomendado para a verificar a conformidade entre o comportamento do agente e sua especificação.

1.1 Motivação

O desenvolvimento de *sistemas complexos baseados em agentes* representa um mercado em crescimento onde são estimados grandes investimentos financeiros (Helle et al., 2016) e as razões de seu crescimento dependem do domínio onde são empregados. Normalmente, justifica-se o uso de agentes autônomos por questões de precisão em relação à capacidade humana e segurança do indivíduo na execução de tarefas, tais como: (i) *acesso a locais de difícil acesso*; (ii) *execução de tarefas perigosas*; (iii) *execução de tarefas longas e repetitivas ou*; (iv) *atividades que exijam baixo tempo de resposta.* Tais

atividades podem acarretar maior risco se executadas por pessoas, devido a fatores como fadiga ou estresse, motivando o uso dos sistemas autônomos. Entretanto, quanto maior o nível de autonomia empregada, mais rigorosa deve a verificação do sistema (Fisher et al., 2013).

Apesar de, normalmente, grande parte do custo total do desenvolvimento do software ser gasto em tarefas de verificação e validação, as pesquisas na área da Engenharia de Software Orientadas a Agentes têm concentrado seus esforços no desenvolvimento de abordagens disciplinadas para analisar, projetar e codificar um SMA e pouca atenção tem sido empregada na forma como tais sistemas poderiam ser testados (Caire et al., 2004).

1.2 Definição do Problema

Se por um lado a tecnologia de agentes ajuda na decomposição e entendimento dos requisitos de sistemas grandes e complexos, por outro lado, as características inerentes aos agentes, aliadas à flexibilidade e variabilidade comportamental decorrentes do uso de normas sociais, trazem obstáculos à sua testabilidade (Voas e Miller, 1995) (Miles et al., 2010). Segundo Voas e Miller (1995), a testabilidade do software está associada a dois problemas práticos: (i) *controlabilidade*, que é a capacidade de controlar os dados utilizados pelo teste e, (ii) *observabilidade*, que é a capacidade de observar os resultados obtidos do componente que está sob teste (Voas e Miller, 1995). O comportamento autônomo dificulta a *observabilidade* do teste, uma vez que algum grau de não determinismo pode ser empregado. De forma semelhante, definir um conjunto coerente de dados de entrada para os testes não é uma tarefa trivial, uma vez que o comportamento do agente depende não apenas do ambiente, mas também das interações e mensagens trocadas com outros agentes, ou seja, de difícil *controlabilidade*.

Apesar dos esforços de algumas abordagens da *Engenharia de Software Orientada à Agentes* (do inglês AOSE, Agent-Oriented Software Engineering) em lidar com o teste de agentes, no que se refere aos agentes BDI (belief-desire-intention) regulados por normas, tais abordagens precisam considerar vários fatores, ainda desconsiderados, tais como:

(F1) A falta de abordagens que explicite como as normas afetam o comportamento dos agentes e como estes entendem e tomam suas decisões sobre uma norma.

(F2) A ausência de um modelo de faltas que represente os diferentes tipos de falha e em quais condições podem ocorrer.

(F3) A falta de abordagens capazes de lidar com as muitas possibilidades de comportamentos decorrentes da presença de uma norma.

(F4) A falta de mecanismos capazes lidar com a cobertura dos testes.

(F5) A falta de mecanismos para apoiar a identificação das falhas encontradas.

Diante do apresentado, nota-se a necessidade de uma abordagem de teste com mecanismos capazes de: **(F1)** explicitar como a norma afeta o comportamento dos agentes; **(F2)** identificar quais tipos de falhas são relevantes e em que condições podem ocorrer; **(F3)** lidar com a variabilidade dos comportamentos dos agentes; **(F4)** garantir a cobertura na identificação dos diferentes tipos de falhas nos comportamentos afetados pela norma e, **(F5)** executar testes que permitam identificar as falhas encontradas.

1.3 Limitações das Abordagens Existentes

Embora seja possível encontrar na literatura abordagens de testes de agentes, tais como: (Abushark et al., 2017) (Abushark et al., 2015) (Abushark et al., 2014) (Thangarajah et al., 2014), que até consideram fatores importantes como a geração de casos de teste, nenhuma dessas abordagens apresenta uma solução para o teste de agentes BDI normativos, verificando se estes estão atuando em conformidade com a norma. Alguns trabalhos (Bulling e Dastani, 2011) (Knobbout et al., 2016) adotam uma abordagem que analisa formalmente a corretude de agentes normativos através da *verificação formal*. Entretanto, tais abordagens não garantem a conformidade quando ocorre variação de normas no ambiente.

1.4 Questões de Pesquisa

Baseando-se na definição do problema e limitações das abordagens existentes apresentadas anteriormente, foram definidas as seguintes questões de pesquisa:

Questão de Pesquisa (QP_1): *Como podemos apoiar o teste de sistemas multiagentes normativos através da construção e manutenção de casos de testes para agentes BDI?*

(i) *Como **verificar** se os agentes agem em conformidade com as normas endereçadas a eles?*

(ii) *Como **controlar** as ações executadas durante o ciclo de raciocínio na execução do caso de teste?*

(iii) *Como **observar** as decisões tomadas pelo agente durante a execução do caso de teste?*

Questão de Pesquisa (QP_2): *Como podemos utilizar o teste baseado em modelos para apoiar o teste de agentes BDI normativos?*

(i) *Como **lidar** com a variabilidade de comportamentos regulados pelas normas?*

(ii) *Como **cobrir** a identificação dos tipos de falhas nos comportamentos?*

(iii) *Como **executar cenários** para representar as decisões tomadas pelo agente?*

1.5 Solução Proposta

Este trabalho apresenta uma abordagem que suporta o desenvolvimento e o teste de agentes BDI normativos, como pode ser visto na Figura 1.1.

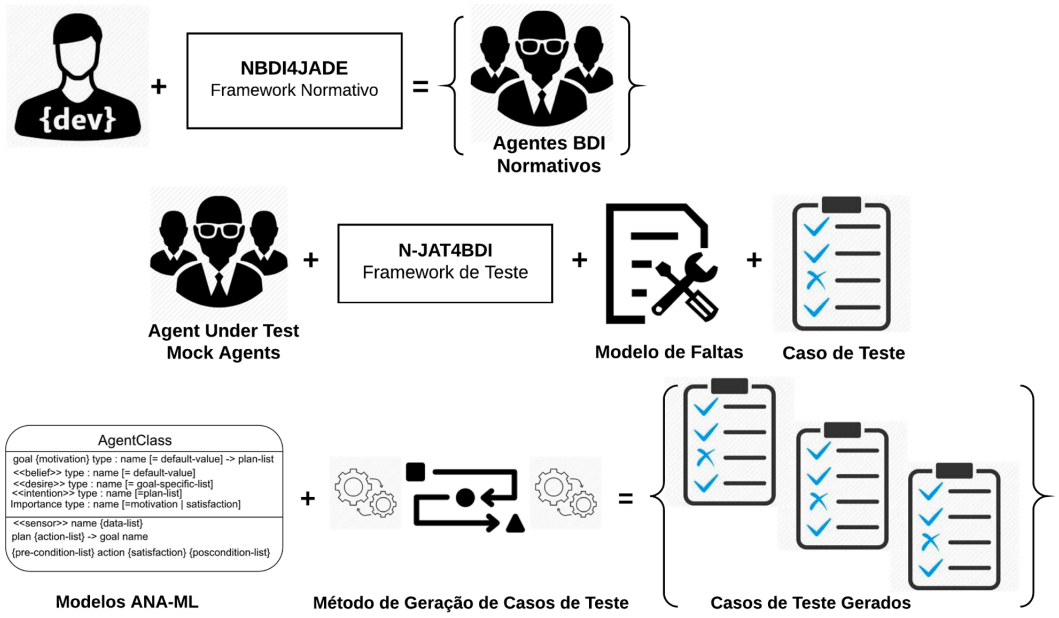


Figura 1.1: Visão Geral da Solução Proposta.

A solução compreende um framework que estende a arquitetura *belief-desire-intention* (Rao et al., 1995), incluindo funções para lidar com o raciocínio normativo. Além disso, é proposto um framework para o teste de agentes normativos e um método para gerar e executar casos de teste, apoiando a identificação de falhas causadas pela ação das normas sobre o comportamento dos agentes. Por fim, propomos um modelo de faltas no qual definimos os tipos de falhas e as condições em que tais falhas podem ocorrer. Em resumo, a proposta apresentada nesta tese é composta de:

- Modelo de Faltas
  - Que define quais características da norma são relevantes para o teste de agentes normativos;
  - Que define quais tipos de falhas podem ser identificadas;
  - Que define as condições em que cada tipo de falha pode ocorrer;
- Framework de Testes para Agentes Normativos
  - Capaz de controlar a entrada dos testes;
  - Capaz de observar os resultados obtidos pelos testes;
  - Capaz de monitorar a execução dos agentes permitindo entender as decisões tomadas;
  - Fornece mecanismos para apoiar o desenvolvedor na identificação de falhas;
- Método para Geração de Casos de Teste
  - Define um método para sistematizar o teste baseado em modelos de agentes normativos;
  - Capaz de lidar com a variabilidade de comportamentos que sofrem a ação da norma;
- Mecanismo para Desenvolvimento de Agentes Normativos
  - Revisar as crenças do agente a partir da percepção do ambiente, incluindo a percepção de novas normas ou a extinção de normas não mais existentes;
  - Gerar objetivos a partir de suas crenças e em conformidade com as normas adotadas;
  - Mecanismos para cumprir ou violar normas;
  - Selecionar objetivos para serem atingidos, levando em consideração as normas endereçadas;
- Ferramenta de Teste
  - Que permite a construção de casos de teste;
  - Que gera casos de teste a partir de informações do modelo ANA-ML dos agentes;
  - Que permite a execução desses casos de teste;
  - Que permite a visualização e análise dos casos de teste executados;

1.6 Contribuições

Esta seção sumariza as principais contribuições desta tese.

- **Modelo de Faltas**. Definição de um modelo de faltas que descreve os tipos de falhas e as condições nas quais as falhas podem ocorrer;
- **Framework de Testes**. Um framework para teste de agentes normativos que permite controlar e observar o teste de agentes BDI normativos;
- **Método para Geração de Casos de Teste**. Definição de um método para geração de casos de testes a partir de informações extraídas dos modelos ANA-ML;
- **Mecanismos para Desenvolvimento de Agentes Normativos**. Um framework para o desenvolvimento de agentes BDI normativos;
- **Ferramenta de Teste**. Uma ferramenta para construção e execução de casos de teste;
- **Cenário de Uso**. Aplicação prática da abordagem proposta em um cenário de uso;

1.7 Organização da Tese

O documento está estruturado da seguinte forma:

- O Capítulo 2 apresenta a *fundamentação teórica* dos assuntos abordados e define os principais conceitos e terminologias utilizadas no documento.
- O Capítulo 3 apresenta os *trabalhos relacionados*, discutindo os aspectos importantes e suas limitações.
- O Capítulo 4 apresenta o *modelo de faltas normativo*, classifica os tipos de falhas e descreve as condições em que tais falhas ocorrem.
- O Capítulo 5 apresenta o framework de teste *N-JAT4BDI* e o framework *NBDI4JADE* para desenvolvimento de agentes normativos.
- O Capítulo 6 apresenta o *método de geração de casos de testes*.
- O Capítulo 7 apresenta o *cenário de uso* utilizado na avaliação.
- O Capítulo 8 apresenta a *avaliação* da abordagem.
- O Capítulo 9 apresenta as *conclusões* e *trabalhos futuros*.
- O Apêndice A apresenta os *artefatos* utilizados no estudo experimental e a *produção científica* da pesquisa realizada.

2
Fundamentação Teórica

Neste capítulo são definidos os principais conceitos, a terminologia básica e os principais termos do jargão envolvidos e utilizados ao longo do documento para o seu melhor entendimento. A Seção 2.1 apresenta a *terminologia básica* utilizada em teste de software. A Seção 2.2 apresenta uma visão geral sobre *agente*, *agente inteligentes* e *sistema multiagente*. A Seção 2.3 apresenta outro conceito central deste trabalho: *normas* e sua aplicação a *sistemas multiagentes normativos*. A Seção 2.4 apresenta uma visão geral sobre *teste de software* e a Seção 2.5 apresenta os desafios e particularidades do *teste de sistemas baseados em agentes*. Por fim, a Seção 2.6 apresenta uma visão geral da *linguagem de modelagem ANA-ML*.

2.1
Terminologia Básica

Um assunto tão vasto quanto o proposto nesta tese exige, primeiramente, estabelecer uma linguagem comum. Como acontece em muitas áreas da computação e das ciências em geral, diversos termos comumente utilizados adquirem significados específicos e particulares quando usados tecnicamente (Delamaro et al., 2017). Os termos e definições relacionados ao teste de software e utilizados neste documento estão de acordo com o "*Standard Glossary of Terms used in Software Testing v.2.4, July, 2014*", documento de referência do ISTQB.[1]

Diversos "problemas" podem emergir durante o desenvolvimento de software e, normalmente, nos referimos a esses problemas como "erros". A literatura tradicional estabelece significados específicos para este e outros termos relacionados como: *falha*, *defeito*, *erro*, *engano* (Delamaro et al., 2017).

Define-se "defeito" (do inglês, **fault**) como sendo um passo, processo ou definição de dados incorretos e "engano" (**mistake**) como a ação humana que produz um defeito. Assim, esses dois conceitos são estáticos, pois estão

[1] O ISTQB® (International Software Testing Qualifications Board) foi fundado em novembro de 2002 e é uma associação sem fins lucrativos legalmente registrada na Bélgica. Tal associação definiu o ISTQB Certified Tester®, que se tornou líder mundial na certificação de competências em testes de software.

associados a um determinado programa ou modelo e não dependem de uma execução particular (Delamaro et al., 2017).

A existência de um defeito pode ocasionar a ocorrência de um "erro" (**error**) durante uma execução do programa, que se caracteriza por um estado inconsistente ou inesperado. Tal estado pode levar a uma "falha" (**failure**), ou seja, pode fazer com que o resultado produzido pela execução seja diferente do resultado esperado (Delamaro et al., 2017).

Tais definições não são unanimidade entre os pesquisadores e engenheiros de software, principalmente em situações informais do cotidiano. Em particular, utiliza-se o termo "erro" de uma maneira bastante flexível, muitas vezes significando defeito, erro ou até falha (Delamaro et al., 2017).

Outros termos importantes que, de alguma forma contribuem para o entendimento do trabalho, são apresentados, como segue:

- **Caso de teste**: Um conjunto de valores de entrada, pré-condições de execução, resultados esperados e pós-condições de execução, desenvolvidos para objetivos específicos ou uma condição de teste, tais como: exercitar um determinado caminho em um programa ou verificar o cumprimento de um requisito específico;
- **Cenário de teste**: A sequência de operações entre um ator e um componente ou sistema, com um resultado tangível;
- **Cenário de uso**: é uma narrativa textual ou pictórica de uma situação (de uso de uma aplicação), envolvendo usuários, processos e dados reais ou potenciais (Erickson e Carroll, 1995);
- **Error-guessing**: uma técnica para projetos de testes onde a experiência do testador é usada para antecipar quais defeitos podem estar presentes no componente ou no sistema em teste;
- **Execução do teste**: O processo de executar um teste sobre o componente ou sistema em teste, produzindo um resultado real;
- **Injeção de falhas**: processo de adição de defeitos intencionalmente a um sistema com o objetivo de descobrir se o sistema pode detectar e, possivelmente, se recuperar de um defeito. A injeção de falhas destina-se a simular falhas que possam realmente ocorrer.
- **Objetivo do teste**: Uma razão ou finalidade para a concepção e execução de um teste;
- **Resultado esperado**: é o comportamento previsto pela especificação, ou por alguma outra fonte, do componente ou sistema sob condições específicas;

- **Suíte de teste**: Um conjunto de vários casos de teste para um componente ou sistema em teste, em que a pós-condição de um teste é, muitas vezes, usada como uma pré-condição para o próximo teste.
- **Testabilidade**: A capacidade de o software ser testado;
- **Teste automatizado**: O uso de software para executar ou apoiar as atividades de teste, por exemplo, gerenciamento de testes, design de teste, execução de testes e a verificação dos resultados;
- **Teste de caixa branca**: Teste com base em uma análise da estrutura interna do componente ou sistema;
- **Teste de caixa preta**: Teste, funcional ou não funcional, sem referência à estrutura interna do componente ou sistema;
- **Verificação do comportamento**: confirmação por exame ou através de evidência objetiva de que os requisitos solicitados foram cumpridos.

2.2 Agente, Agente Inteligente e Sistema Multiagente

Segundo Choren e Lucena, tecnologias tradicionais de desenvolvimento de software como a orientação a objetos falham ao fornecerem técnicas de decomposição e abstração adequadas à modelagem e desenvolvimento de sistemas complexos e baseados em rede (Lucena, 1987) (Choren e Lucena, 2005). O uso de agentes de software é visto como uma tecnologia adequada para lidar com a complexidade inerente a tais sistemas (Lucena, 1987) (Choren e Lucena, 2005) (Weiss, 1999) (Silva e Lucena, 2007).

Agente

Apesar de não existir uma definição única para agentes, Wooldridge e Jennings estabeleceram uma que vem sendo cada vez mais adotada e comumente utilizada por pesquisadores da área (Padgham e Winikoff, 2005), que é dada por:

"*Um agente é um sistema de computador que está situado em um ambiente e é capaz de realizar ações autônomas neste ambiente a fim de alcançar os objetivos projetados.*" (Wooldridge e Jennings, 1995).

Wooldridge e Jennings caracterizam ainda, como principais propriedades dos agentes (Wooldridge e Jennings, 1995):

- *Autonomia* – agentes devem ser capazes de solucionar problemas delegados a eles sem a intervenção externa (pessoas ou outros agentes), ou seja, deve ter certo grau de controle sobre suas ações e seu estado interno;
- *Habilidade Social* – agentes devem ser capazes de interagir, quando julgarem apropriado, com outros agentes, a fim de resolver problemas;
- *Reatividade* – agentes devem perceber modificações ocorridas no ambiente no qual estão situados e reagir, em tempo oportuno, a tais modificações;
- *Proatividade* – agentes não devem simplesmente reagir às modificações do ambiente, eles também devem ser capazes de visualizar oportunidades, manter um comportamento orientado ao alcance de suas metas e tomar a iniciativa quando apropriado;

Agente Inteligente

Wooldridge também faz distinção entre um agente *reativo* e um *agente inteligente* ou *racional*, afirmando que este último precisa, necessariamente, ser ainda mais reativo, proativo e social do que o primeiro (Wooldridge, 2009). Uma análise mais detalhada do significado de "racional" é encontrada na obra de Bratman (Bratman, 1987) a qual constitui a base do mecanismo deliberativo dos *agentes inteligentes* (Rao e Georgeff, 1991).

No que se refere ao desenvolvimento de agentes inteligentes, uma arquitetura amplamente conhecida e utilizada é a arquitetura BDI (*belief–desire–intention*), proposta por Rao e Georgeff (Rao et al., 1995) e inspirada no modelo filosófico do comportamento humano de Bratman (Bratman, 1987). Nesta arquitetura, três estados mentais, *crenças*, *desejos* e *intenções*, compõem a base do mecanismo deliberativo dos agentes permitindo que estes decidam o que fazer, baseados no conhecimento que possuem sobre si e sobre o ambiente no qual estão situados, a fim de atingirem seus objetivos. Os três estados mentais que compõem o modelo BDI são descritos abaixo (Bordini et al., 2007):

- *Belief*: são as informações que o agente possui sobre si e sobre o ambiente no qual está situado, formando a sua base de conhecimento;
- *Desire*: são todos os possíveis objetivos que o agente gostaria de realizar. Ter um desejo, no entanto, não implica que um agente vai agir para realizá-lo. O desejo é um potencial influenciador das ações do agente;
- *Intention*: representa o desejo com o qual o agente está comprometido em alcançar;

Assim, crenças, desejos e intenções são as principais estruturas dos agentes inteligentes (Bordini et al., 2007), fundamentais ao processo de tomada de decisão subjacente ao modelo BDI, conhecido como *raciocínio prático*. O raciocínio prático é o raciocínio voltado para as ações do agente, ou seja, o processo de descobrir o que deve ser feito.

"*O raciocínio prático é uma questão de pesar as opções conflitantes favoráveis e contrárias de uma escolha, onde as considerações relevantes são fornecidas por aquilo que o agente deseja, seus valores e aquilo em que o agente acredita.*" (Bratman, 1990).

Duas atividades distintas constituem o raciocínio prático dos agentes: (i) *deliberação*, no qual os agentes decidem o que fazer, ou seja, consideram suas preferências, pesam as alternativas existentes e escolhem qual objetivo irão alcançar, e (ii) *meios-fins*, no qual decidem como irão alcançar seus objetivos, ou seja, quais sãos as melhores opções, qual sequência de ações (planos) devem executar, etc.

A literatura de agentes comumente se utiliza do termo *agente inteligente* para fazer referência aos agentes que seguem o processo de raciocínio prático do modelo BDI. A Figura 2.1 apresenta o processo de raciocínio prático em um agente BDI. Cada componente é descrito abaixo, como segue:

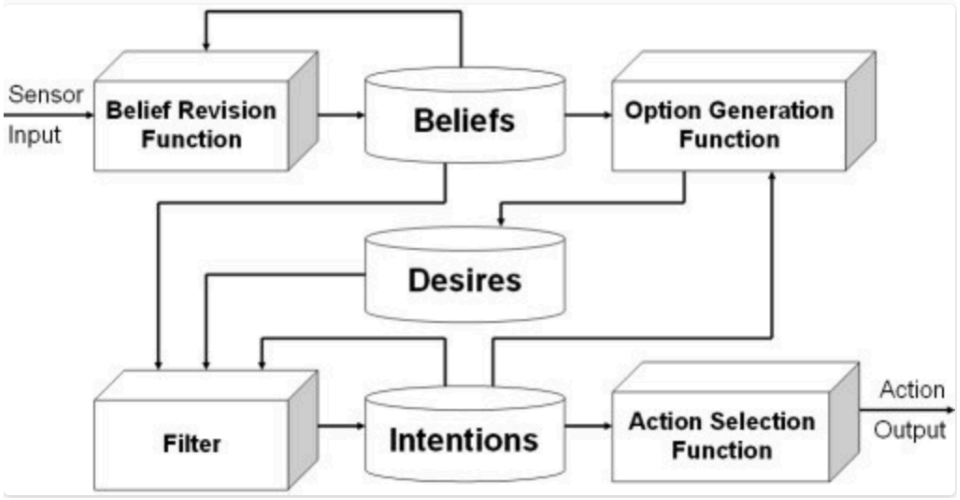


Figura 2.1: Arquitetura BDI Genérica (Wooldridge, 2009).

- *Belief Revision Function*: revisa as crenças do agente, determinando um novo conjunto de crenças a partir da percepção do ambiente e das crenças existentes;
- *Beliefs*: representam as informações atualizadas do agente e do ambiente no qual está situado;

- *Option Generation Function*: determina as opções disponíveis aos agentes, ou seja, seus desejos, considerando as crenças e intenções do agente;
- *Desires*: representam o estado motivacional dos agentes, ou seja, aquilo que o agente gostaria de alcançar;
- *Filter*: representa o processo de deliberação, que determina as novas intenções do agente com base nas suas crenças, desejos e intenções atuais;
- *Intentions*: representa o foco atual do agente, isto é, aqueles objetivos que o agente está comprometido em alcançar;
- *Action Selection Function*: determina a ação a ser executada para alcançar suas intenções atuais;

Sistema Multiagente

Geralmente, uma solução baseada em agentes de software envolve um conjunto de outros agentes de tipos iguais ou diferentes, formando uma sociedade ou organização (Wooldridge, 2009). A esta sociedade dá-se o nome de *Sistema Multiagente* (SMA). Logo, um SMA consiste de uma sociedade de agentes capazes de interagir entre si e, para interagirem com sucesso, são necessárias as habilidades de cooperação, coordenação e negociação entre eles, assim como na sociedade humana (Wooldridge, 2009).

2.3 Normas e Sistema Multiagente Normativo

Nesta seção, primeiramente é definido o conceito de norma e seu objetivo. Em seguida, as normas são contextualizadas aos sistemas normativos e, por fim, é apresentada a aplicação das normas aos sistemas multiagentes.

Normas

O termo *normas sociais* é usado para regular os comportamentos dos membros de uma sociedade (Mahmoud et al., 2014) e, de acordo com Cialdini e Trost (Cialdini e Trost, 1998), são "*regras e padrões que guiam e/ou restringem os comportamentos dos membros de um grupo sem a força das leis*". Estar em conformidade com as normas reduz a ocorrência de atritos sociais e facilita a coordenação entre os indivíduos (Sen e Airiau, 2007). Com o passar do tempo, podem acontecer mudanças nas normas devido a circunstâncias objetivas ou mudanças nas percepções e expectativas subjetivas (Young, 2007). Entretanto, a norma ainda representa o comportamento esperado em relação

a uma situação específica (Ahmad, 2012) dentro de uma comunidade, sendo meios comumente aceitos como eficientes para regular tais comportamentos (Alberti et al., 2011). Geralmente, as normas são entendidas como regras que indicam ações as quais se esperam que sejam obrigatórias, proibitivas ou permissivas com base em um conjunto específico de fatos. Dessa forma, as normas estabelecem princípios socialmente compartilhados para os comportamentos dos membros da sociedade (Mahmoud et al., 2014).

Também é necessário considerar que as normas de um sistema não são isoladas umas das outras. As vezes, a ativação, desativação, cumprimento ou violação de uma norma pode levar a ativação, desativação, cumprimento ou violação de outras normas (Lopez, 2003). Nesse contexto, também vale ressaltar a possibilidade de existirem conflitos entre normas, que acontece quando duas normas regulando o mesmo comportamento estão ativadas, mas uma delas obriga a realização do comportamento enquanto a outra proíbe a realização do comportamento (Vasconcelos et al., 2009).

A definição de norma utilizada nesta tese (Silva, 2008) é representada pelas seguintes propriedades: *Addressees*, *Condition* (por exemplo, Activation, Expiration), *Motivation* (por exemplo, Rewards, Punishments), *Deontic Concept* e *State*. Cada propriedade é descrita da seguinte forma: (i) ***Addressees*** é usado para especificar os agentes ou funções responsáveis pelo cumprimento da norma; (ii) ***Activation*** é a condição para que a norma se torne ativa; (iii) ***Expiration*** é a condição para que a norma se torne inativa; (iv) ***Rewards*** é usado para representar o conjunto de recompensas atribuídas ao agente pelo cumprimento da norma; (v) ***Punishments*** é o conjunto de punições atribuídas ao agente por violar uma norma; (vi) ***Deontic Concept*** é usado para indicar se a norma estabelece uma obrigação, uma permissão ou uma proibição, e (vii) ***State*** é usado para descrever o conjunto de estados ou ações que estão sendo regulados.

Sistema Normativo

Um *sistema normativo* é responsável por avaliar se as obrigações foram completamente atingidas ou se proibições foram descumpridas pelas entidades participantes (Boella e Torre, 2004). Dessa maneira, um sistema normativo precisa continuamente: (i) monitorar o comportamento dos participantes; (ii) avaliar o efeito de cada participante no ambiente; (iii) aplicar sanções quando elas forem violadas.

Sistema Multiagente Normativo

No contexto de sistemas multiagentes, as normas são mecanismos comumente aceitos como meios eficientes para regular o comportamento dos agentes e representar a maneira pela qual os agentes entendem as responsabilidades de outros agentes (Alberti et al., 2011) (Lopez, 2003). Tais mecanismos são as vezes representados como ações a serem executadas (Silva, 2008), ou restrições a serem impostas sobre as ações de um agente (Silva, 2008). Em outras ocasiões, os comportamentos regulados são especificados através de objetivos que devem ser satisfeitos ou evitados por agentes (Lopez, 2003). Além disso, normalmente as normas não são aplicadas o tempo todo e podem vigorar no ambiente por diferentes períodos de tempo, podendo afetar um agente enquanto este permanecer ativo na sociedade, apenas em circunstâncias especiais ou apenas dentro de um contexto específico como, por exemplo, até que uma meta social seja alcançada (Luck et al., 2002). Assim, as normas devem especificar o contexto no qual devem ser ativadas de maneira que os agentes possam cumpri-las e, da mesma forma, devem especificar o contexto no qual devem ser desativadas, de forma que os agentes podem desconsiderá-las (Lopez, 2003).

Para motivar o cumprimento das normas pelos agentes, são utilizadas *recompensas* como forma de promover seu cumprimento e *punições* como meio de inibir sua violação. Tais recompensas e punições podem estar associadas ao atingimento de objetivos (Lopez, 2003), a realização de ações ou ao estabelecimento de outras normas (Silva, 2008). Dignum *et al.* (2002), considera que para um agente capaz de lidar com normas (conhecido como *agentes normativos*) ser verdadeiramente autônomo, ele deve ser capaz de raciocinar sobre as normas que são de sua responsabilidade e, ocasionalmente, optar por violar tais normas se estas forem de encontro aos interesses individuais do agente (Dignum et al., 2002).

Segundo Mahmoud *et al.* (2014), a literatura sugere três tipos diferentes de normas para sistemas multiagentes normativos (Caire, 2007, Mahmoud et al., 2014) tais como: (i) *normas reguladoras* que especificam o comportamento de um sistema usando obrigações, proibições e permissões (Caire, 2007); (ii) *normas constitutivas* que, além de regular seu próprio comportamento, podem também criar novas normas derivadas de outras normas existentes (Boella e Torre, 2004, Rubino et al., 2005, Boella e Torre, 2006); (iii) *normas processuais*, que são endereçadas aos agentes do sistema normativo para regular seus comportamentos (Boella e Torre, 2008).

Por fim, um agente normativo deve ser capaz de realizar dois importantes processos: *deliberação* – que diz respeito à decisão por selecionar uma norma

para ser cumprida ou violada, e tal decisão pode ser realizada de diferentes maneiras, tais como: simplesmente decidir por cumprir todas as normas ou violar quaisquer normas do sistema; e *cumprimento* – diz respeito a como tal decisão por cumprir ou violar uma norma influenciará no raciocínio do agente, por exemplo, objetivos podem deixar de serem atingidos pela decisão por cumprir uma norma de proibição.

2.4 Teste de Software

De acordo com Pezzè e Young (Pezzè e Young, 2009), as disciplinas de engenharia alinham atividades de projeto e construção com atividades que verificam produtos intermediários e finais de forma que os defeitos possam ser identificados e removidos. O mesmo acontece com a Engenharia de Software: a construção de software de alta qualidade requer a combinação de atividades de projeto e verificação ao longo do desenvolvimento.

O software é um dos mais complexos e variáveis artefatos construídos de forma regular. Os requisitos de qualidade de softwares usados em um ambiente podem ser muito diferentes e incompatíveis para outro ambiente ou domínio de aplicação, e sua estrutura evolui e frequentemente se deteriora à medida que o sistema cresce (Pezzè e Young, 2009).

A verificação do software é uma importante atividade que engloba todo o processo de desenvolvimento e manutenção (Adrion et al., 1982). O objetivo é encontrar defeitos nas especificações, no projeto dos artefatos e na implementação. Por outro lado, um outro objetivo também é prevenir defeitos. O projeto de teste pode descobrir e eliminar tais defeitos em todas as etapas do processo de construção do software (Schach, 1996).

Entretanto, o custo da verificação do software frequentemente corresponde a mais da metade do custo total do desenvolvimento e manutenção. Técnicas avançadas de desenvolvimento e poderosas ferramentas de suporte podem reduzir a frequência de algumas classes de erros (Pezzè e Young, 2009).

A variedade de problemas e a riqueza de abordagens fazem com que seja um desafio escolher e planejar a combinação correta de técnicas para atingir o nível exigido de qualidade satisfazendo requisitos de custo. Não existem fórmulas prontas para abordar o problema de verificar um produto de software. Mesmo os especialistas mais experientes não possuem soluções predefinidas, necessitando projetar uma solução personalizada, que seja adequada ao problema, aos requisitos e ao ambiente (Pezzè e Young, 2009).

O objetivo do teste e análise do software é avaliar a qualidade do software ou possibilitar melhorias no software, revelando defeitos. Segundo Pezzè, não

existem técnicas de testes ou de análise perfeitas, nem uma "técnica melhor" para todas as circunstâncias. Para ele, as técnicas possuem capacidades e fraquezas complementares (Pezzè e Young, 2009).

A Atividade de Teste

A atividade de teste é complexa e diversos os fatores podem colaborar para a ocorrência de erros (Delamaro et al., 2017). Por isso a atividade de teste é dividida em fases com objetivos distintos. De uma forma geral, pode-se estabelecer como fases: o *teste de unidade*, o *teste de integração* e o *teste de sistemas*. O teste de unidade foca nas menores unidades de um programa a qual é testada separadamente. Esse tipo de teste pode ser aplicado à medida que ocorre a implementação das unidades e pelo próprio desenvolvedor, sem a necessidade de dispor-se do sistema totalmente finalizado (Delamaro et al., 2017). O teste de integração deve ser realizado após serem testadas as unidades individualmente e dá ênfase na construção da estrutura do sistema. À medida que diversas partes do software são colocadas para trabalhar juntas, é preciso verificar se a interação entre elas funciona de maneira adequada e não leva a erros (Delamaro et al., 2017). Por fim, com todas as suas partes integradas, inicia-se o teste de sistema cujo objetivo é verificar se as funcionalidades especificadas nos documentos de requisitos estão todas corretamente implementadas. Aspectos de correção, completude e coerência devem ser explorados, bem como requisitos não funcionais como segurança, performance e robustez (Delamaro et al., 2017).

Independentemente da fase de teste, existem algumas etapas bem definidas para a execução da atividade de teste. São elas: (i) planejamento; (ii) projeto de casos de teste; (iii) execução; e (iv) análise (Delamaro et al., 2017).

Teste Baseado em Modelos

De acordo com Delamaro *et al.* (2017), uma especificação é um documento que representa o comportamento e as características de um sistema e pode ser definida de diversas formas, como por exemplo, através de uma descrição textual em linguagem natural (Delamaro et al., 2017). Delamaro *et al.* (2017) afirmam ainda que o uso de outras formas de especificação torna-se importante em contextos nos quais imprecisões e ambiguidades podem causar problemas. A modelagem, por exemplo, permite que o conhecimento sobre o sistema seja capturado e reutilizado durante diversas etapas do desenvol-

vimento. O modelo utilizado é muito importante nessa tarefa pois, se bem desenvolvido, captura o que é essencial no sistema (Delamaro et al., 2017).

O *American Heritage Dictionary*[2] (Mifflin, 2000), apresenta, dentre outras, as seguintes definições para *modelo*: (i) um pequeno objeto usualmente construído para escalar, que representa em detalhes outro objeto frequentemente maior, e (ii) uma descrição semântica de um sistema, teoria ou fenômeno que é responsável por suas propriedades conhecidas ou inferidas e pode ser usado para mais estudo de suas características. Essas definições mostram duas importantes características dos modelos para o *teste baseado em modelo*: os modelos devem ser pequenos em relação ao tamanho do sistema que está sendo testado, e suficientemente detalhados para descrever precisamente as características daquilo que se deseja testar (Utting e Legeard, 2010).

O *teste baseado em modelo* surgiu com o intuito de tratar as atividades de teste de maneira sistemática, exceto tarefas de planejamento, sendo bastante discutidos no meio acadêmico e despertando o interesse na indústria de software (Conrad et al., 2002).

Em geral, o *teste baseado em modelo* apresenta quatro principais abordagens, e são elas (Utting e Legeard, 2010):

(i) geração de dados de entrada dos testes a partir de um modelo de domínio;

(ii) geração de casos de teste a partir de um modelo do ambiente;

(iii) geração de casos de teste com oráculos a partir de um modelo de comportamento;

(iv) geração de scripts de teste a partir de testes abstratos.

Ao ser usado para geração dos dados de entrada dos testes, o *modelo* representa as informações sobre o domínio dos valores de entrada e a geração do teste requer a seleção e combinação "inteligente" desses valores para produzir os dados de entrada. A geração automática da entrada dos testes é, obviamente, de grande importância prática, mas ela não resolve por completo o problema do projeto de testes porque ela não ajuda a saber se um teste passou ou falhou (Utting e Legeard, 2010).

A segunda abordagem utiliza um tipo diferente de modelo que descreve o *ambiente esperado* do sistema em teste. A partir desses modelos do ambiente, é possível gerar sequências de chamadas para o sistema em teste. Entretanto, da mesma forma como mostrado anteriormente, as sequências geradas não especificam as saídas esperadas do sistema em teste. Assim, não é possível

[2] *The American Heritage Dictionary of the English Language*

prever os valores de saída porque os modelos de ambiente não modelam o comportamento do sistema em teste, e nem determinar precisamente se um teste passou ou falhou (Utting e Legeard, 2010).

O terceiro uso é a geração de casos de teste que incluem um *oráculo* de informações, tais como os valores de saídas esperadas do sistema em teste ou alguma verificação automática sobre os valores de saída atuais para ver se eles estão corretos. Esta é, obviamente, uma tarefa mais desafiadora do que apenas gerar os dados de entrada ou a sequência de testes que chamam o sistema em teste mas não verificam os resultados. Para gerar testes com oráculos, o gerador de casos de teste deve conhecer o comportamento esperado do sistema em teste para ser capaz de prever ou verificar os valores de saída. Em outras palavras, o modelo deve descrever o comportamento esperado do sistema em teste, assim como o relacionamento entre suas entradas e saídas. A vantagem desta abordagem é que, dentre as quatro, ela é a única que trata todo o problema do projeto de teste, desde escolher os valores de entrada até a geração de casos de testes executáveis que inclui o veredicto da execução (Utting e Legeard, 2010).

O quarto uso é ligeiramente diferente. É assumido que é conhecida uma descrição abstrata de um caso de teste como, por exemplo, um diagrama de sequência ou uma sequência de chamadas de procedimentos em alto nível. Essa abordagem foca na transformação do caso de teste abstrato em um script de teste executável de baixo nível (Utting e Legeard, 2010).

Nesta tese, o foco está na terceira abordagem do *teste baseado em modelo*: a geração de casos de teste executáveis que incluem um oráculo de informações extraídas de modelos descritos através da linguagem de modelagem ANA-ML (Viana et al., 2016) representam os comportamentos do agente normativo em teste. Este processo de geração inclui a geração dos valores de entrada e o sequenciamento de chamadas dos testes, mas também inclui a geração de oráculos que verificam as saídas do sistema em teste. Este uso do *teste baseado em modelo* é mais sofisticado e complexo do que os outros usos (Utting e Legeard, 2010).

Uma vez construído o modelo do sistema que se deseja testar, pode-se, então, usar a ferramenta de teste NJAT4BDI para gerar os testes a partir do modelo. A saída do gerador de casos de teste é um conjunto de *casos de teste* cada um dos quais é uma sequência de operações com os valores de entrada e saídas esperados associados. Os testes executáveis resultantes podem ser produzidos diretamente em um formato como, por exemplo, os testes JUnit. Finalmente, os casos de testes podem ser executados para tentar detectar falhas no sistema em teste. A execução dos testes é controlada e monitorada através

da própria ferramenta de *execução dos testes* (Utting e Legeard, 2010).

2.5 Teste em Sistemas Multiagente

De acordo com (Voas e Miller, 1995), dois aspectos são cruciais à testabilidade: (i) *controlabilidade* – a capacidade de controlar os dados utilizados pelo teste e, (ii) *observabilidade* – a capacidade de observar os resultados obtidos do componente em teste.

O teste de agentes está relacionado à capacidade de testar as propriedades particulares dos agentes. Algumas dessas propriedades, tais como autonomia, consciência do contexto e proatividade, tornam o teste de agentes uma tarefa complexa e desafiadora (Rouff, 2002). Somando-se ainda, a habilidade social dos agentes e o uso de normas para regular comportamentos "socialmente inapropriados", pode-se resultar em um comportamento emergente imprevisível (Miles et al., 2010). Se por um lado o uso de sistemas baseados em agentes são adequados aos requisitos de aplicativos complexos, por outro lado, as características intrínsecas dos agentes trazem novos obstáculos para a testabilidade do software (Voas e Miller, 1995). São discutidas abaixo, porque algumas dessas características dos agentes são difíceis de testar.

- *Ambiente complexo*: O grande número de entradas válidas e o contexto dinâmico no qual o sistema opera onde, muitas vezes, ambos podem ser parcialmente desconhecidos no tempo de design, fazem com que estratégias de testes exaustivas sejam inviáveis (Brat e Jonsson, 2005), (Schumann e Visser, 2006), (Clapper et al., 2007), (Micskei et al., 2012), (Cheng et al., 2009) e (Pouly e Jouanneau, 2012).
- *Software complexo*: A complexidade interna do software, que é causada por características inerentes a sistemas autônomos, como montar seu próprio curso de ações para um determinado objetivo, adaptar-se a diferentes ambientes, aprender, diagnosticar-se e reconfigurar-se para manter sua funcionalidade (Schumann e Visser, 2006), (Mikaelian, 2010) e (Brat e Jonsson, 2005). Brat e Jonsson (2005), concluíram que "verificar um software que planeja suas ações é um enorme desafio, considerando que tais softwares devem encontrar soluções complexas em espaços de estado muito grandes" (Brat e Jonsson, 2005).
- *Comportamento não determinístico*: Como frequentemente o comportamento autônomo é baseado em algum tipo de mecanismo deliberativo, os agentes podem reagir de maneira diferente às mesmas entradas ao longo do tempo (Nguyen et al., 2012), (Mikaelian, 2010),

(Schumann e Visser, 2006), (Cukic, 2001) e (Menzies e Pecheur, 2005). Isso também significa que um teste bem-sucedido não garante que o sistema passará no mesmo teste na próxima vez. Kurd (2005), identifica como um desafio central, nessas situações, a dificuldade de entender o modelo de como sistema se adaptou (transparência e representação do comportamento) (Kurd, 2005).

A verificação de sistemas multiagentes é importante para detectar violações no comportamento dos agentes e, em geral, é realizada mapeando-se os requisitos dos sistemas em especificações de propriedades dos agentes. Através de técnicas de verificação é possível checar se o projeto do sistema real está em conformidade com as propriedades especificadas. Os sistemas que atendem ou cumprem com suas especificações são classificados como sistemas corretos (Bordini et al., 2006).

Em um estudo recente, Bakar e Selamat (2018) relatam que trabalhos de pesquisa vêm sendo realizados na área de verificação de agentes e sistemas multiagentes nos últimos vinte e cinco anos (Bakar e Selamat, 2018). Tais números demonstram que a verificação desse tipo de sistema é uma área de pesquisa ativa e que vale a pena ser explorada e estudada. A Figura 2.2 apresenta como os trabalhos estão distribuídos por ano de publicação.

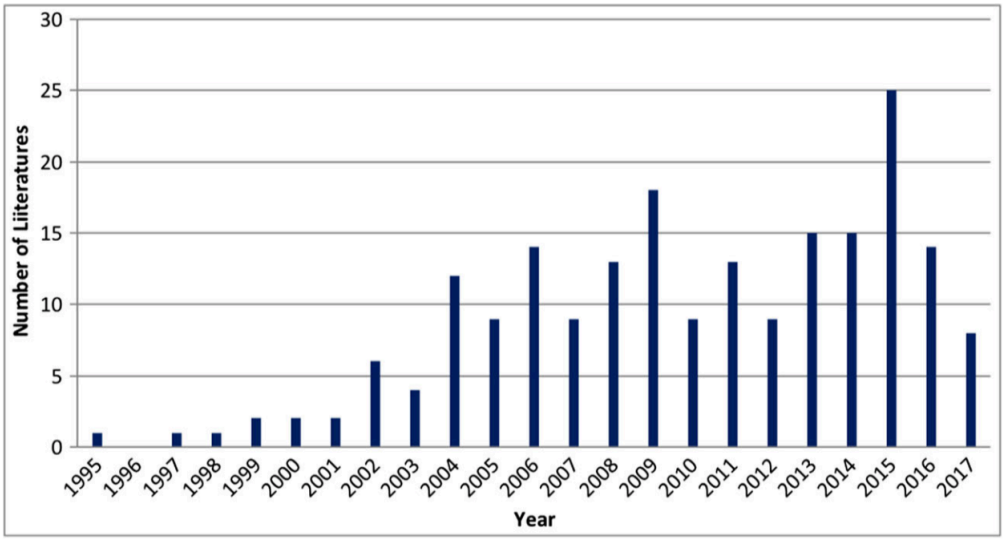


Figura 2.2: Publicações por Ano **(Bakar e Selamat, 2018)**.

Em seu estudo, Bakar e Selamat classificaram os trabalhos de acordo com a técnica de verificação utilizada (Bakar e Selamat, 2018), como apresentado na Figura 2.3.

Devido à complexidade dos agentes descrita acima, meios tradicionais de teste são inadequados para sistemas baseados em agentes que operam em

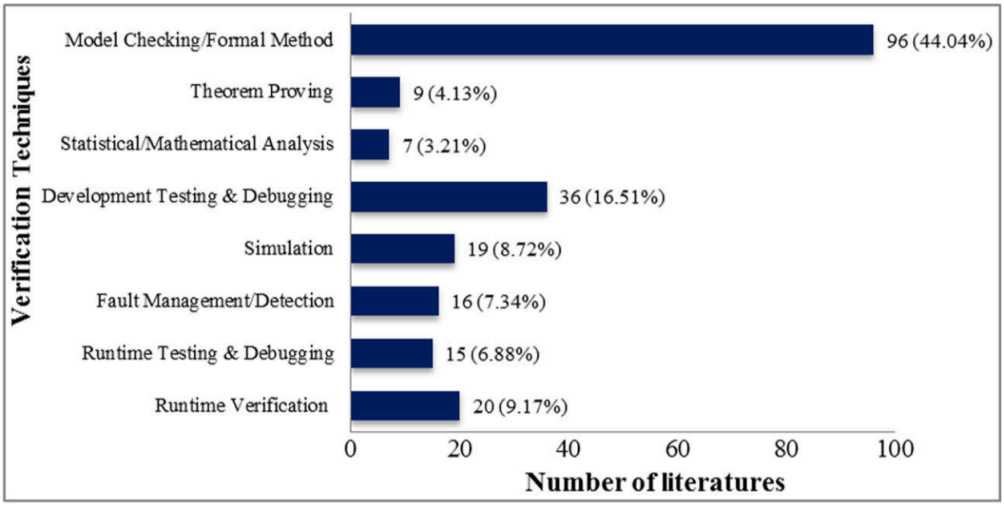


Figura 2.3: Publicações por Técnica de Verificação **(Bakar e Selamat, 2018)**.

ambientes onde há uma gama relativamente ampla de casos possíveis. Para algumas aplicações, pode ser possível usar a expertise do domínio para limitar as possíveis falhas a um subconjunto gerenciável que um ambiente arbitrário poderia apresentar. Para outros domínios, entretanto, isso simplesmente não é suficiente, e outras abordagens devem ser consideradas (Miles et al., 2010).

Algumas soluções adotam a verificação de especificação abstrata do software em todos ou em um subconjunto limitado de casos, por exemplo, usando a abordagem de *model checking*. Nesse caso, a verificação por si só não garante a exatidão da implementação, pois a implementação pode não seguir a especificação (Miles et al., 2010).

2.6 Linguagem de Modelagem ANA-ML

Esta seção apresenta uma visão geral sobre a linguagem de modelagem ANA-ML (*Adaptive Normative Agent – Modeling Language*), proposta com o objetivo de modelar *informações úteis* à criação de agentes capazes de adaptar o seu comportamento para lidar com normas (Viana et al., 2016). A linguagem ANA-ML é uma extensão da linguagem MAS-ML proposta por (Silva, 2004) que, por sua vez, trata-se de uma extensão da linguagem UML[3], focada nos diagramas de classes.

[3]Unified Modeling Language

Modelando Agentes

A Figura 2.4 mostra um exemplo do diagrama utilizado para modelar a classe do agente. A metaclasse *AgentClass* permite modelar a capacidade do agente raciocinar através da arquitetura BDI. Para representar as crenças, tem-se o estereótipo «*belief*». Para representar o estado mental do agente, estão presentes os estereótipos: (i) «*desire*» – para representar o conjunto de desejos e, (ii) «*intention*» – para definir o conjunto de intenções. A propriedade *Importance* representa quão importante é a realização de um comportamento para o agente. A importância é avaliada levando em consideração a motivação do agente em relação a um objetivo ou a satisfação em executar uma ação. O esteriótipo «*sensor*» representa a percepção do agente em relação ao ambiente.

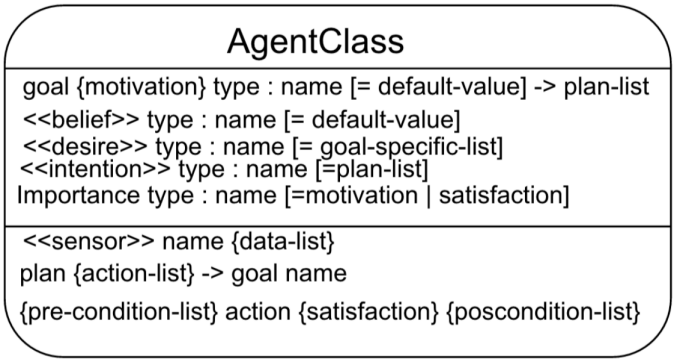


Figura 2.4: Modelando a classe Agente.

Modelando Organizações

A Figura 2.5 mostra um exemplo da representação da metaclasse *OrganizationClass*, a qual agrupa os agentes de um sistema multiagente em grupos e papéis, ambos definindo a estrutura dos diferentes grupos de agentes e subgrupos dentro da organização.

Foi feita uma modificação em relação ao conceito de MAS-ML para inserir a entidade norma, que será explicada em detalhes em seguida. Essa é uma característica estrutural, mas que modificará o comportamento dos agentes.

Modelando Normas

A Figura 2.6 mostra o exemplo de representação de uma norma em ANA-ML. O compartimento superior contém o nome da norma e deve ser

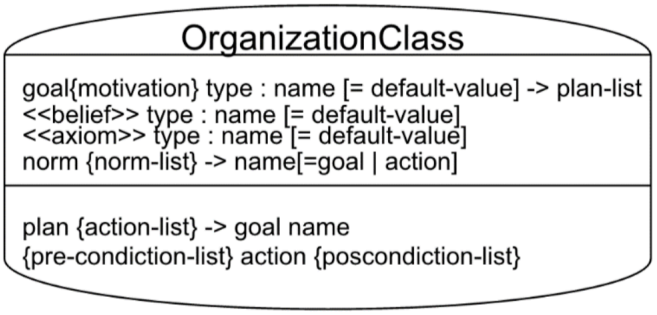


Figura 2.5: Modelando a Organização.

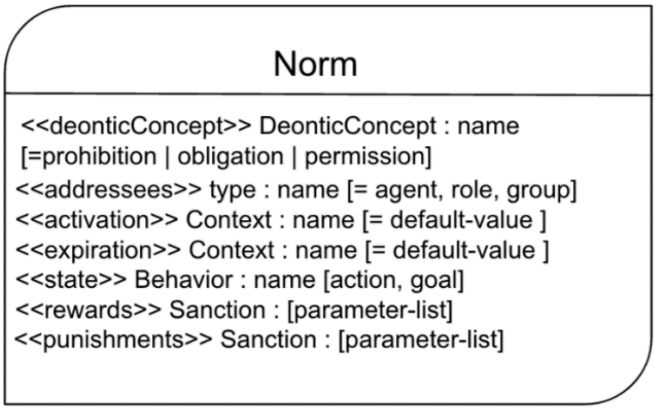


Figura 2.6: Modelando a Norma.

único no ambiente. No compartimento inferior são definidas suas características estruturais, sendo descritos os esteriótipos como segue:

ANA-ML modela os seguintes atributos da entidade *Norma*:

- *deonticConcept* é usado para indicar se a norma estabelece uma obrigação, uma permissão ou um proibição ao agente;
- *addressees* identifica os agentes, grupos ou papéis responsáveis pelo cumprimento da norma;
- activation identifica uma condição de ativação da norma;
- *expiration* identifica uma condição para que uma norma se torne inativa;
- *rewards* identifica um conjunto de recompensas conferidas ao agente caso ele cumpra com a norma;
- *punishments* identifica um conjunto de punições aplicadas ao agente caso viole com a norma e,
- *state* é usado para indicar um estado ou ação que está sendo regulada pela norma.

3
Trabalhos Relacionados

Este capítulo apresenta na Seção 3.1 os trabalhos existentes na literatura relacionados ao *projeto e desenvolvimento de agentes* capazes de lidar com normas, a Seção 3.2 discute trabalhos sobre o *teste de agentes e sistemas multiagentes* e a Seção 3.3 apresenta trabalhos que utilizam a *geração de casos de testes* para testar agentes. A Seção 3.4 apresenta uma *análise comparativa* dos trabalhos apresentados e, por fim, a Seção 3.5 descreve como alguns trabalhos *contribuíram* e *influenciaram* a solução proposta nesta tese.

3.1
Framework para Agentes Normativos

O trabalho de (Castelfranchi et al., 2000) apresenta um modelo arquitetural que permite projetar agentes orientados a objetivos que: (i) verificam se o agente é responsável por cumprir a norma; (ii) em quais situações a norma está ativada ou desativada, considerando as crenças do agente e, (iii) avaliam qual o impacto da norma sobre a geração e seleção de objetivos e planos. Entretanto, o trabalho apresenta apenas diretrizes em alto nível sobre como se dá a influência das normas sobre o raciocínio do agente, não discutindo como pode ser verificado o cumprimento ou violação de normas e a detecção e resolução de conflitos entre normas.

O trabalho de (Lopez, 2003) formaliza agentes que utilizam diferentes estratégias de decisão para o cumprimento ou violação da norma. Os agentes são classificados em: *Social* – sempre decidem por cumprir com as normas do sistema e *Rebellious* – sempre decidem por violar as normas do sistema. O trabalho apresenta experimentos que demonstram que o uso de agente *Social* podem levar a uma melhor ordem social do sistema. Nesse caso, entretanto, objetivos importantes do agente podem não ser atingidos em prol de cumprir com as normas. Enquanto isso, os agentes *Rebellious* tendem a declinar com a ordem social do sistema e sofrer fortes punições.

O trabalho de (Dastani e Torre, 2004) operacionaliza o modelo BOID (*Belief-Obligation-Intention-Desire*) proposto por (Broersen et al., 2001). Tal operacionalização é uma extensão do modelo BDI que considera a influência das crenças, obrigações, intenções e desejos do agente sobre a geração de

objetivos, aplicando a noção de tipos de agentes para guiar a geração de objetivos. Nesta abordagem, um agente é dito *egoísta*, se seus desejos se sobrepõem a suas obrigações, ou *social*, se suas obrigações se sobrepõem a seus desejos. Dessa forma, este trabalho adota uma abordagem estática para lidar com normas baseando-se no tipo de agentes, não fornecendo a flexibilidade necessária para lidar com a natureza dinâmica inerente aos sistemas normativos onde, a cada instante, novas normas podem ser adotadas, ativadas, desativadas, cumpridas ou violadas.

O trabalho de (Kollingbaum, 2005) apresenta uma infraestrutura para a construção de agentes normativos cujo principal objetivo é apenas o cumprimento de normas e não a realização de seus objetivos. Como mencionado no trabalho de (Lopez, 2003), tal estratégia pode levar o agente a não realizar muitos dos seus objetivos importantes.

O trabalho de (Meneguzzi e Luck, 2009) apresenta uma extensão da linguagem AgentSpeak(L) possibilitando o desenvolvimento de agentes capazes de se comportarem de acordo com as normas recém adotadas, na qual novos planos são criados para cumprir com obrigações e planos que violam normas proibitivas são suprimidos. Na prática, a abordagem de (Meneguzzi e Luck, 2009) cria, seleciona ou suprime planos em resposta a obrigações e proibições. Consequentemente, planos os quais o agente tem alto interesse em realizar podem não ser executados. Além disso, o conflito entre normas não é considerado nesta abordagem.

O trabalho de (Neto, 2012) apresenta uma extensão do *framework* Jason (Bordini et al., 2007) através da extensão da linguagem *AgentSpeak(L) Normativo*, que permite o desenvolvimento de agentes capazes de entender, seguir ou violar as normas contidas no ambiente. Entretanto, o trabalho apresenta algumas limitações em relação a representação de normas adotada, tais como: (i) as normas regulam objetivos e ações, entretanto, podem ser estendidas a fim de regular um conjunto de objetivos, ações ou planos; (ii) a propriedade *addressees* da norma não considera a especificação de organizações de agentes, e (iii) as normas adotadas no trabalho definem obrigações ou proibições, mas também podem ser adotadas as permissões como mecanismo de regular o comportamento de agentes.

3.2 Teste em Sistemas Multiagente

O trabalho apresentado por (Low et al., 1999) propõe diferentes critérios de cobertura para o teste de agentes BDI. Os autores derivam dois tipos de grafos de fluxo de controle: um com *nós*, representando os planos para

o agente BDI e outro com *arcos* representando mensagens ou outros eventos que iniciam um determinado plano. Os critérios de cobertura são definidos com base na cobertura de nó, arco e caminho e, também, alguns baseados no sucesso ou fracasso da execução planos. Nesta abordagem não são consideradas as interações entre agentes e nem agentes normativos.

No trabalho de (Núñez et al., 2005), agentes são especificados e testados através de um framework formal genérico. Os agentes são testados individualmente e seu comportamento no sistema multiagente é observado. A ideia central do trabalho é observar o comportamento do agente em relação aos dados de entrada informados e validar a conformidade com a especificação. Os autores utilizam a ideia de uma *máquina de estados finitos* que representam estados com o predicado que deve ser atingido e o conjunto de transições. A abordagem não se aplica a agentes normativos.

O trabalho de (Tiryaki et al., 2006) propõe uma abordagem de desenvolvimento dirigida por testes que apoia a construção sistemas multiagentes de forma iterativa e incremental. A abordagem é apoiada pelo framework de testes SUnit, que estende o framework JUnit. Características de agentes normativos não são discutidas nesse trabalho.

O trabalho de (Coelho et al., 2007) apresenta o JAT, um framework para construção e execução de cenários de teste de sistemas multiagentes baseado no uso da programação orientada a aspectos para monitorar a execução dos testes e controlar a interação entre os agentes. Entretanto, este trabalho não suporta o teste de agentes BDI, não lida com questões normativas e utiliza um modelo de faltas *limitado* para identificação de falhas.

O trabalho de (Zhang et al., 2007) apresenta uma abordagem para testes unitários de agentes. A ideia principal é testar as unidades internas dos agentes (crenças, planos, eventos e mensagens). A abordagem está baseada na análise do código fonte e a cobertura dos testes restringe-se a planos individuais. Esta abordagem não lida com agentes normativos e não fornece nenhum suporte de ferramenta de testes.

O trabalho de (Shaw et al., 2008) apresenta resultados teóricos do uso de árvores de metas e planos para representar os caminhos de decisão dos agentes. O foco do trabalho é raciocinar sobre os recursos reutilizáveis usando redes Petri, onde o melhor plano é selecionado com base nos recursos exigidos. A abordagem não trata com agentes normativos, não define métricas de cobertura e nem a utilização de um modelo de faltas.

O trabalho de (Zhou et al., 2008) apresenta uma abordagem que utiliza lógica proposicional para mostrar se um objetivo é alcançado ou não. Os autores introduzem implicações parciais fortes e fracas e estudaram sua semântica

e resultado. O trabalho não lida com agentes normativos.

O trabalho de (Miller et al., 2010) afirma que a interação entre os agentes é complexa e seu teste é importante. Os autores definem dois conjuntos de critérios de cobertura. O primeiro usa a especificação do protocolo de comunicação, enquanto o segundo considera os planos que enviam e recebem mensagens. Nenhum modelo de faltas e nenhuma ferramenta de apoio aos testes são apresentados. A abordagem lida com agentes BDI nem questões normativas.

O trabalho de (Winikoff e Cranefield, 2014), publicado inicialmente em 2010, foca na análise do tamanho do espaço de comportamentos para o agente BDI. Os autores concluem que o tratamento de falhas tem maior impacto no tamanho do espaço de comportamento do que o esperado. Eles identificaram os fatores que influenciam o tamanho do espaço comportamental e apresentam uma aplicação industrial para verificar a análise apresentada. Nenhum suporte de ferramenta é discutido. O autor discute diferentes aspectos de teste para os agentes BDI, em vez de fornecer uma abordagem de teste concreta.

O trabalho de (Thangarajah et al., 2014) apresenta um mecanismo para verificar o nível de completude das metas em agentes BDI. Os autores rastreiam os recursos consumidos pelas metas gerando algumas medidas que são usadas para quantificar a completude. A identificação de falhas com relação às medidas de cobertura não é abordada por essa abordagem.

O trabalho de (Cunha et al., 2015) apresenta uma abordagem para testar os elementos de um agente BDI (crenças, planos, objetivos, eventos e mensagens). A programação orientada a aspectos é utilizada para monitorar o ciclo de raciocínio do agente em teste. Uma ferramenta é disponibilizada para apoiar o uso de casos de teste. Entretanto, a abordagem não lida com o teste de agentes normativos.

O trabalho de (Rehman e Nadeem, 2015) propõe uma abordagem para testar sistemas multiagentes baseados em artefatos de design do Prometheus. Diferentes interações entre o agente e outros atores são consideradas no teste. O protótipo de uma ferramenta para gerar caminhos de teste é discutido. A abordagem não lida com agentes normativos e não discute o modelo de faltas utilizado.

O trabalho de (Abushark et al., 2015) apresenta uma abordagem para identificar falhas em agentes onde a estrutura dos planos é verificada em relação às especificações de requisitos em tempo de projeto. Os autores utilizam diagramas de design do Prometheus. Entretanto, defeitos inseridos diretamente no código da aplicação durante a implementação não puderam ser detectados.

3.3 Geração de Casos de Testes

O trabalho de (Zhang et al., 2007) apresenta uma abordagem de teste baseado em modelo para sistemas multiagentes. É apresentado um framework de teste que considera as diferentes sequências de execução dos agentes. Nenhuma estratégia para medir a cobertura dos testes foi tomada, além de não suportar o teste de agentes normativos.

O trabalho de (Nguyen et al., 2007) apresenta um framework de testes para a metodologia Tropos que possibilita o teste de unidade, integração e de sistema. Casos de teste são derivados dos requisitos dos agentes. Esta abordagem restringe-se ao teste dos objetivos do agente e dependências importantes, como crenças, mensagens e planos não são consideradas. A abordagem não lida com agentes normativos.

O trabalho de (Zhang et al., 2009) apresenta uma abordagem para testes unitários de sistemas multiagentes. O sistema em teste é avaliado em relação à conformidade com seu modelo de design. Os autores propõem um framework para testar individualmente os planos do agente. Os casos de teste são executados com dados gerados manualmente. Não foram definidos o modelo para identificação de falhas e nem quaisquer critérios de cobertura. A abordagem não lida com agentes normativos.

O trabalho de (Padgham et al., 2013) apresenta uma proposta para a criação de oráculos de teste a partir de artefatos do Prometheus. É proposto um modelo de faltas para as unidades elementares dos agentes. O teste depende que cada agente tenha seu código "aumentado" para possibilitar a configuração e execução dos casos de teste. Falhas relacionadas aos eventos, planos e crenças são considerados e os resultados dos casos de teste são avaliados com relação aos tipos de falhas e ao número de ocorrências. Entretanto, critérios de cobertura mais abrangentes, como por exemplo, cobertura de objetivos e planos estão ausentes. Além disso, falhas relacionadas aos objetivos e planos dos agentes também estão ausentes no modelo de faltas apresentado. A abordagem não lida com agentes normativos.

O trabalho de (Rehman et al., 2016) apresenta uma abordagem de teste baseada em modelo apoiada nos artefatos de design do Prometheus e por critérios de cobertura para os objetivos e planos do agente. O código do agente em teste necessita ser instrumentado para realização da análise de cobertura. A avaliação da abordagem é feita através da injeção de falhas no sistema em teste. A abordagem não lida com agentes normativos.

3.4 Análise Comparativa

Esta seção apresenta uma análise comparativa dos trabalhos apresentados anteriormente. Os critérios de comparação foram extraídos de revisões sistemáticas sobre teste de agentes e sistemas multiagentes, como por exemplo, os trabalhos de (Rehman e Nadeem, 2013) e (Houhamdi, 2011). A seguir, são descritos os critérios de comparação utilizados. A Figura 3.1 apresenta uma análise dos trabalhos relacionados selecionados para os critérios de comparação definidos.

Metodologia de Modelagem

Existem diferentes metodologias utilizadas para modelar um sistema multiagente. Esse parâmetro identifica qual modelo foi utilizado pela abordagem de teste.

Artefato Base para o Teste (código fonte/modelo)

Esse parâmetro identifica qual artefato é utilizado na verificação da abordagem de teste, por exemplo: especificação/modelo de diagrama ou código fonte. Particularmente, a pesquisa realizada nesta tese focou em abordagens de *teste baseado em modelo*. Entretanto, abordagens que utilizam outras técnicas, como por exemplo, a análise do código fonte, também mostraram-se importantes para o amplo entendimento do trabalho.

Nível de Teste

Esse parâmetro verifica se o teste verifica o comportamento individual do agente ou o comportamento social emergente, resultante da interação entre os agentes e o ambiente. Com base no nível de teste, são definidos detalhes adicionais da verificação como, por exemplo, os elementos internos do agente, a interação de agentes e fluxo de informações entre diferentes artefatos de design.

Artefato de Entrada

Esse parâmetro apresenta quais são os elementos do agente verificados pela abordagem de teste.

Geração de Dados de Teste

Dados de teste são usados na execução de casos de teste. Uma abordagem de teste pode usar dados prontos ou gerar seus próprios dados de teste. Esse parâmetro identifica se a abordagem de teste gera automaticamente seus dados de teste ou não.

Critério de Cobertura

Esse parâmetro identifica se a abordagem de testes apresenta alguma estratégia para lidar com a cobertura dos testes.

Suporte de Ferramentas

Este parâmetro identifica se abordagem possui o suporte de algum tipo de ferramenta ou não. Normalmente, tais ferramentas auxiliam na validação dos resultados obtidos e descrevem se o processo aplicado funciona ou não.

3.5 Influências na Solução Proposta

A abordagem proposta nesta tese verificou problemas, analisou soluções e considerou diversas ideias (mesmo teóricas) de diferentes trabalhos que, de alguma forma, influenciaram a solução apresentada neste trabalho. As principais influências, contribuições e restrições são analisadas a seguir.

Principais Influências

Durante o desenvolvimento desta pesquisa, diversos trabalhos influenciaram e contribuíram para a solução proposta, dos quais se pode recordar:

- A extensão do BDI4JADE para se tornar um framework para desenvolvimento de agentes normativos apoiou-em em ideias apresentadas nos trabalhos de (Neto et al., 2013) e (Neto et al., 2010);
- O framework de testes N-JAT4BDI foi estendido a partir do trabalho de (Cunha et al., 2015) que, por sua vez, foi inicialmente inspirado no trabalho de (Coelho et al., 2007);
- O modelo de faltas normativo seguiu as diretrizes e formatos utilizados por (Rehman, 2017);

	Critérios de Comparação						
Trabalhos	Metodologia de Modelagem	Base de Teste (Código / Modelo)	Nível de Teste	Artefato de Entrada	Geração de Dados de Teste	Critério de Cobertura	Suporte de Ferramenta
Low et al., 1999	No specific methodology (all following BDI architecture)	Code Based	Unit Testing	Plans and Nodes as statements	Yes	Yes	Yes
Nunez et al., 2005	Not specified	Model based	Unit Testing	Utility State Machine	No	No	No
Tiryaki et al. 2006	No specific methodology	Code Based	Unit Testing	Plans	No	No	Yes
Coelho et al., 2007	No specific methodology	Code Based	Integration Testing	Beliefs, Goals and Messages	No	No	Yes
Zhang et al., 2007	Prometheus	Model based	Unit Testing	Plans	Yes Manually	No	No
Zhang et al., 2007	Prometheus	Model based	Unit Testing	Plans	Yes Manually	No	No
Nguyen et al., 2007	Tropos	Model based	System Testing	Goals	No	No	Yes
Shaw et al. 2008	No specific methodology (BDI architecture)	Model based	Integration Testing	Goals and Plans	No	No	No
Zhou et al., 2008	No specific methodology (BDI architecture)	Code Based	Unit Testing	Goals	No	No	No
Zhang et al., 2009	Prometheus	Model based	Unit Testing	Plans	Yes Manually	No	Yes
Miller et al., 2010	Prometheus Methodology	Model based	Integration Testing	Protocol Diagram	No	Yes	No
Winikoff and Cranefield, 2010	No specific methodology	Code Based	Unit Testing	Plans	No	No	No
Padgham et al., 2013	Prometheus Methodology	Model based	Unit Testing	Agent and Capability Diagrams	Yes	Yes	No
Thangarajah et al. 2014	No specific methodology (BDI architecture)	Code based	Unit Testing	Goals	No	Yes	No
Abushark et al. 2015	Prometheus Methodology	Model based	Integration testing	Scenario, Role, Agent and goal diagrams	No	No	Yes
Cunha et al., 2015	No specific methodology (BDI architecture)	Code Based	Unit Testing	Beliefs, Plans, Goals and Messages	No	No	Yes
Rehman and Nadeem, 2015	Prometheus Methodology	Model based	Unit Testing	Plans	No	No	Yes
Rehman et al., 2016	Prometheus Methodology	Model based	Unit Testing	Goals and Plans	No	Yes	Yes

Figura 3.1: Análise Comparativa dos Trabalhos Selecionados.

- Os critérios de cobertura utilizados nos testes dos comportamentos de agentes normativos foram inspirados no trabalho de (Rehman et al., 2016).

Principais Diferenças

As principais diferenças entre a solução proposta e os trabalhos relacionados e destacados como influência são:

- Apresentação de um método bem definido e completo para teste de agentes normativos;
- Descrição de um modelo de faltas que apresenta os tipos de falhas e as condições em que tais falhas ocorrem, relacionando tais falhas com as características normativas que afetam o comportamento dos agentes;
- Geração de casos de teste que lidam com a variabilidade de comportamento dos agentes em decorrência do efeito de uma norma;
- Geração de casos teste semânticos com nomes correlatos ao tipo de verificação a ser realizada, permitindo: (i) que o desenvolvedor entenda o que o teste verifica, pelo seu nome, e (ii) que a possível manutenção dos testes seja mais fácil, pela facilidade de rastreamento disponibilizada pelo esquema de nomeação adotado;
- O uso de modelos desenvolvidos em ANA-ML;
- A existência de mecanismos que permitem entender o efeito das normas nas decisões do agente;

Principais Restrições e Limitações

As principais restrições da solução proposta são:

- O conflito entre normas é um fato conhecido no âmbito dos sistemas normativos. Apesar do framework NBDI4JADE suportar o desenvolvimento de agentes que lidam com questões conflitantes entre normas, o framework de teste N-JAT4BDI não fornece suporte identificação de falhas em agentes normativos afetados por normas conflitantes;
- As normas de um sistema não são isoladas uma das outras. Às vezes, a ativação, desativação, cumprimento ou violação de uma norma pode levar a ativação, desativação, cumprimento ou violação de outras normas (Lopez, 2003). Tais relacionamentos são conhecidos como *interlocking* e não são tratados na abordagem de teste proposta nesta tese;

- O desenvolvimento de sistemas baseados em agentes apoia-se na existência de diversas plataformas. A abordagem proposta restringe o teste a agentes normativos desenvolvidos em NBDI4JADE.

4
Um Modelo de Faltas para Agentes Normativos

Este Capítulo apresenta o *modelo de faltas* utilizado na identificação de falhas nos comportamentos de agentes BDI normativos. O *modelo de faltas normativo* é apresentado na Seção 4.1. A Seção 4.2 apresenta a classificação dos níveis de falhas. A Seção 4.3 apresenta os tipos de falhas e as condições em que tais falhas podem ocorrer. Finalmente, a Seção 4.4 apresenta uma breve discussão sobre falhas em sistemas distribuídos em geral.

4.1
Modelo de Faltas

Uma maneira eficaz de identificar os tipos de falhas de um componente é definir um modelo que especifica hipoteticamente as prováveis situações em que a falha pode ser encontrada (Binder, 2000) (Myers et al., 2011) (Burnstein, 2006). Cada suposição introduz a ocorrência de uma falha no software, que pode ser identificada como uma falha existente no sistema. Por exemplo, um sistema de gerenciamento de uma conferência define que um revisor poderá revisar, no máximo, três artigos da conferência. Portanto, uma suposição pode ser definida da seguinte maneira: *é considerada uma falha do sistema se, a um revisor da conferência, forem atribuídos mais de três artigos para revisão*. Neste caso, os testadores seguem esta suposição para verificar precisamente a ocorrência ou não da condição que configura a falha no sistema. Esta estratégia de teste de software é conhecida como *teste dirigido por falhas* (Binder, 2000).

O teste visa identificar as inconsistências entre o comportamento executado e o comportamento esperado para um componente. Tal divergência indica a presença de falhas, seja na implementação do componente ou em sua especificação. No caso dos agentes, se crenças, planos, objetivos ou outros estados internos estabelecidos previamente não forem alcançados durante a execução do agente, diferentes tipos de falhas poderão ser identificadas.

No teste dirigido por falhas é preciso conhecer o contexto no qual as falhas podem ser detectadas. Esse conhecimento é comumente chamado de *modelo de faltas* (Myers et al., 2011). Primeiramente, são exploradas e definidas quais características do componente devem ser verificadas, nesse caso, quais carac-

terísticas dos agentes normativos. Em seguida são analisados os possíveis contextos de falha associados ao comportamento dos agentes que sofrem o efeito das normas. Tais características são denominadas *características testáveis* e os possíveis contextos de falha são chamados de *tipos de falhas* (Binder, 2000). Os tipos de falhas são classificados em diferentes níveis, permitindo distinguir quais representam erros e quais são apenas possíveis fontes de erros. Com base no modelo de faltas, é possível localizar e identificar os tipos de falhas através da execução de casos de teste.

4.2 Classificação das Falhas

As *características testáveis* do agente normativo podem ser identificadas através da análise das normas e sua ação sobre a estrutura interna do agente. Por exemplo, se o agente decide por cumprir com a norma, ele deveria, obrigatoriamente, selecionar planos que resultem em comportamentos que estejam em conformidade com a norma. Esta análise é importante para garantir a abrangência das *características testáveis*.

Algumas falhas indicam, claramente, a presença de erros no agente em teste, enquanto outras falhas apenas indicam a possível existência de erros. Assim, para estabelecer essa diferença, as falhas foram classificadas em diferentes níveis. Outra vantagem da classificação de falhas é poder decidir sobre a continuidade do teste. Se uma falha é classificada como um erro, outros testes dependentes não deveriam ser executados. Por outro lado, se a falha não é classificada como um erro: (i) outros testes dependentes podem, ainda, ser executados ou, (ii) a execução dos testes é interrompida e o desenvolvedor investiga a falha. Portanto, a classificação de falhas auxilia de forma que as ações apropriadas possam ser aplicadas quando esta for detectada. As falhas são classificadas em três níveis, como segue:

- **Nível 1 (Exception)**: as *falhas de nível 1* são exceções lançadas pelo agente durante a execução do teste. Exceções não são eventos específicos dos agentes, mas sim a ocorrência de condições que alteram o fluxo normal de execução. Nosso framework de testes captura quaisquer exceções lançadas pelo agente em teste e automaticamente as classifica como uma *falha de nível 1*.
- **Nível 2 (Error)**: as *falhas de nível 2* são declaradas como erros. Por exemplo, se uma norma é *endereçada* ao agente em teste e este não tem seu comportamento afetado pela norma, tal situação é considerada uma falha e indica a existência de inconsistências entre a especificação,

o design e a implementação. Os ajustes para corrigir o erro podem estar em quaisquer destas partes.

- **Nível 3 (Warning)**: em geral, trata-se de um erro, embora existam situações em que possa não ser. Quando uma *falha de nível 3* é identificada, o desenvolvedor recebe uma mensagem de aviso, sendo sua a responsabilidade de avaliar se trata-se de um erro ou não.

4.3 Tipos de Falhas

Com base na discussão anterior, é proposto um modelo de faltas para apoiar a identificação de falhas em *agentes BDI normativos*, que se restringe à definição de normas utilizada neste trabalho (Neto, 2012) e aos diferentes contextos em que tais falhas podem ser encontradas.

Falha no Endereçamento

Como visto, *normas* são mecanismos que regulam o comportamento dos agentes de uma sociedade (Bogdanovych et al., 2009), sendo, portanto, criadas para serem cumpridas de forma que os agentes possam alcançar seus objetivos individuais, mantendo os padrões sociais definidos. A propriedade *addressee* da norma define os agentes responsáveis pelo cumprimento da norma. Para ter efeito, a norma deve, necessariamente, afetar todos os agentes a que se destina. Se o agente não reconhece a norma, seu comportamento não é afetado pela norma. As seguintes falhas relacionadas ao endereçamento das normas podem ocorrer:

A norma é reconhecida pelo agente a quem foi endereçada? As normas devem ser reconhecidas por todos os agentes destinatários. O não reconhecimento pode indicar uma falha potencial.

A norma é reconhecida pelo agente a quem não foi endereçada? As normas devem ser reconhecidas apenas pelos agentes aos quais foram explicitamente endereçadas; caso contrário, é uma indicação de falha potencial.

Falha na Condição de Ativação ou Desativação

A condição de ativação indica sob quais condições uma norma é aplicável para regular o comportamento do agente. A ausência de uma condição de contexto capaz de ativar a norma pode indicar que a norma nunca será aplicada. Se uma norma é endereçada ao agente, espera-se que, em algum

momento, ocorra sua ativação. Dessa forma, existem três contextos de possíveis falhas associados à ativação da norma, conforme descrito abaixo:

A condição de ativação é alcançada por uma condição de contexto? Se a condição de ativação nunca é alcançada por uma condição de contexto, tal situação pode indicar uma falha potencial. Entretanto, a aplicação das normas é temporária e, pode ocorrer de não existir uma condição de contexto que ative a norma. Nesse caso, a falha é classificada como *falha de nível 3*.

A norma é ativada quando ocorre a condição de ativação? Se a condição de ativação é alcançada e a norma não é ativada, tal situação pode indicar uma falha potencial.

A norma é ativada mesmo quando não ocorre a condição de ativação? Se a condição de ativação não é alcançada e mesmo assim a norma é ativada, tal situação pode indicar uma falha potencial.

A condição de desativação indica sob quais condições a norma não será mais aplicável para regular o comportamento do agente. A ausência de uma condição de contexto capaz de desativar a norma pode indicar que a norma estará sempre sendo aplicada. Se uma norma é endereçada ao agente, espera-se que, em algum momento, ocorra sua desativação. Dessa forma, existem três contextos de possíveis falhas associados à desativação da norma, conforme descrito abaixo:

A condição de desativação está associada a uma condição de contexto? Se condição de desativação nunca é alcançada por uma condição de contexto, tal situação pode indicar uma falha potencial. Entretanto, a aplicação das normas é temporária, podendo, inclusive, ser de longa duração no ambiente. Nesses casos, pode não ocorrer uma condição de contexto que desative a norma. Nesse caso, a falha é classificada como *falha de nível 3*.

A norma é desativada quando ocorre a condição de desativação? Se a condição de desativação é alcançada e a norma não é desativada, tal situação pode indicar uma falha potencial.

A norma é desativada mesmo quando não ocorre a condição de desativação? Se a condição de desativação não é alcançada e mesmo assim a norma é desativada, tal situação pode indicar uma falha potencial.

Falha na Motivação

Embora as normas sejam um mecanismo eficaz para regular o comportamento de agentes (Lopez, 2003), a natureza autônoma dos agentes permite que estes decidam por cumprir ou violar uma norma, de acordo com seu interesse individual. Para motivar o cumprimento das normas são usadas recompensas.

Por outro lado, para inibir a violação das normas, são usadas as punições. Algumas falhas nos comportamentos dos agentes podem decorrer de sua motivação em cumprir ou não com a norma, como segue:

O agente recebe a devida recompensa por cumprir com a norma? Se o cumprimento de uma norma determina que o agente deve receber uma recompensa e isso não ocorre, isso pode influenciar futuras decisões do agente. Tal situação pode indicar uma falha potencial.

O agente recebe a recompensa sem ter cumprido com a norma? Se o agente não cumpre com a norma, não deve receber a recompensa por tal. Se o agente recebe a recompensa mesmo não tendo cumprido com a norma, tal situação pode indicar uma falha potencial.

O agente recebe a devida punição por violar uma norma? Se o agente viola uma dada norma, ele deve receber a devida punição. Caso o agente não receba a punição, tal situação pode indicar uma falha potencial.

O agente recebe uma punição mesmo tendo cumprido com a norma? Se um agente recebe uma punição, mesmo tendo cumprido a norma, tal situação pode indicar uma falha potencial.

Falha no Conceito Deôntico

O *conceito deôntico* define uma permissão, obrigação ou proibição sobre o comportamento do agente. De forma prática, proibir ou obrigar o agente de executar um comportamento significa que mecanismo de seleção de planos aplicáveis deve considerar os efeitos da norma sobre os planos selecionados. Abaixo são descritos os pontos de falha associados a essa característica.

O agente não executa as ações obrigatórias corretamente? Se um agente não executa um comportamento ou ação obrigatória, tal situação pode indicar uma falha potencial.

O agente executa ações proibidas? Se o agente executa um comportamento ou ação que não deveria ser executado (proibido), tal situação pode indicar uma falha potencial.

Falha no Estado Interno

Ao cumprir com a norma, o agente pode ter um conjunto de estados internos ou ações regulados (crenças, planos, mensagens, eventos, etc.) pela norma. Dessa forma, espera-se que os estados sejam afetados, apropriadamente.

O estado interno do agente é apropriadamente afetado quando este decide por cumprir com a norma? Se o agente decide cumprir com a norma, seu estado

interno deve ser afetado em conformidade com as propriedades da norma. Se o estado interno do agente não é afetado adequadamente, tal situação pode indicar uma falha potencial.

4.4 Falhas Decorrentes de Execução Concorrente

Sistema Multiagente é um tipo de programa assíncrono no qual as entidades executam simultaneamente. Portanto, também podem ocorrer falhas decorrentes da execução concorrente dos agentes. A literatura possui um vasto estudo sobre falhas em sistemas concorrentes, como por exemplo: condições de *deadlocks* causadas pela disputa de um recurso; o *não-determinismo* resultante de chamadas assíncronas a eventos simultâneos, dentre outros (Sen e Agha, 2006).

O trabalho de Eytani *et al.* (2008) apresenta algumas estratégias e abordagens desenvolvidas para testar sistemas concorrentes (Eytani et al., 2008). No entanto, os agentes têm particularidades adicionais em relação a outros sistemas concorrentes, tais como: (i) *a definição dinâmica das metas que serão alcançados*; (ii) *a seleção apropriada dos planos aplicáveis para a realização das metas*; (iii) *o armazenamento das crenças no próprio agente e*, (iv) *a influência de normas sobre os itens mencionados*. Tais características levam os sistemas baseados em agentes a tipos especiais de falhas conforme apresentado nas seções anteriores deste capítulo.

Essa pesquisa limita-se à identificação de falhas em agentes cujos comportamentos são afetados pela ação de uma norma, e não considera problemas relacionados à concorrência. Além disso, o framework de teste proposto se restringe ao teste unitário dos componentes internos dos agentes normativos, e as interações entre os agentes limita-se a troca de mensagens entre estes. Para evitar falhas decorrentes da execução concorrente e manter a *controlabilidade* do teste, nossa abordagem utiliza o conceito de *mock agents* (Coelho et al., 2006).

Apesar de não abordarmos nesta pesquisa problemas relacionados à execução concorrente, o mencionado trabalho de Eytani *et al.* (2008) apresenta abordagens que podem ser usadas para detectar falhas em sistemas concorrentes (Eytani et al., 2008).

5
Desenvolvimento de Agentes Normativos

Neste capítulo é apresentado o framework N-JAT4BDI (Cunha et al., 2018), que permite a construção e execução de casos de teste e o NBDI4JADE (Cunha et al., 2018), um framework para o desenvolvimento de agentes BDI normativos. O NBDI4JADE é uma extensão do framework BDI4JADE (Nunes et al., 2011), que adiciona a capacidade de lidar com normas. A Seção 5.1 apresenta detalhes do N-JAT4BDI e a Seção 5.2 apresenta o NBDI4JADE.

5.1
N-JAT4BDI: Um Framework para Testar Agentes Normativos

Esta seção apresenta o N-JAT4BDI e seu suporte para a construção e execução de casos de teste para testar agentes desenvolvidos com o NBDI4JADE.

5.1.1
Visão Geral da Arquitetura do Framework

A Figura 5.1 apresenta as principais classes do N-JAT4BDI e suas dependências. A classe *NBDIAgent* representa o agente normativo que desejamos testar (AUT, do inglês, Agent Under Test). Internamente, a classe *NBDIAgent* estende a classe *Agent* do JADE e, portanto, utiliza toda infraestrutura para execução e comunicação de agentes provida pela plataforma JADE. A classe *NMockAgent* também estende a classe *Agent* e representa um agente simples cujo propósito é interagir com o AUT para simular a comunicação entre agentes no ambiente. Através da interface *NTestReporter* os agentes mock implementam o resultado da interação com o AUT. A classe *NTestCase* representa um caso de teste que pode, efetivamente, testar o comportamento do agente normativo. Para isso, a classe *NTestCase* estende a classe *TestCase* do framework JUnit, reaproveitando sua infraestrutura para executar os casos de teste e visualizar os resultados. A classe *Monitor* representa o aspecto que monitora o ciclo de raciocínio do AUT. A classe *Synchronizer* representa o aspecto que coordena a ordem das interações entre o AUT e os agentes mock, quando isto é necessário.

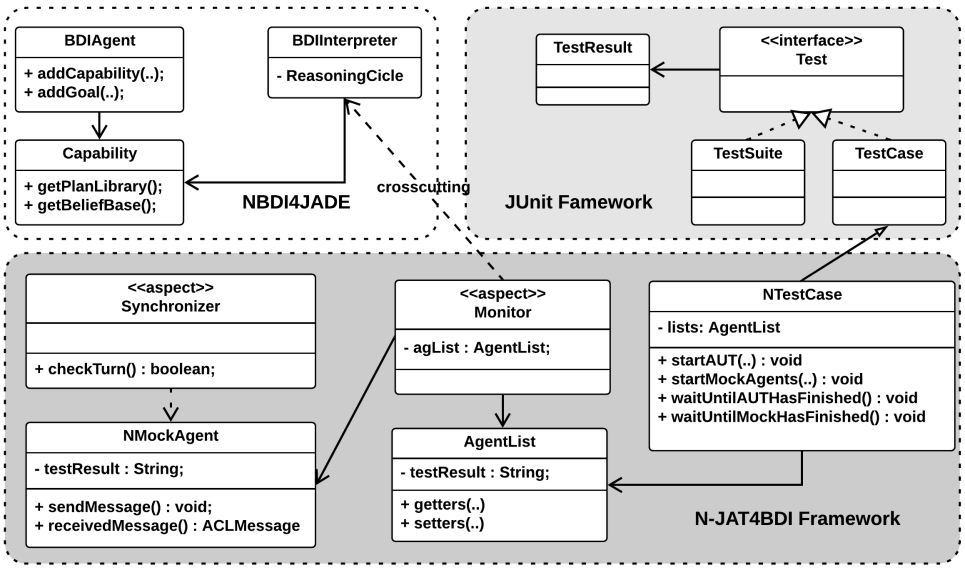


Figura 5.1: Arquitetura do N-JAT4BDI Framework.

5.1.2 Visão Geral do Funcionamento do Framework

São apresentados abaixo os componentes do framework e seu funcionamento. Também descrevemos como o uso de programação orientada à aspectos pode apoiar a testabilidade de agentes BDI normativos.

- *Agent Under Test* (AUT): agente cujo comportamento é testado;
- *Agente Mock*: trata-se de uma implementação falsa de um agente real que interage com o AUT durante a execução do caso de teste com a finalidade de simular a comunicação (as trocas de mensagens) do AUT;
- *Monitor*: componente responsável por monitorar o ciclo de raciocínio do AUT e as mensagens enviadas e recebidas por ele;
- *Synchronizer*: componente responsável por orquestrar a execução de um cenário de teste, definindo a ordem em que os agentes mock interagem com o AUT;
- *Cenário de Teste*: define um conjunto de casos de teste cuja intenção é verificar o comportamento de uma parte do sistema;
- *Caso de Teste*: define uma condição com a qual o AUT será exposto. Contém as pré-condições necessárias e verifica se o resultado obtido está em conformidade com a especificação para estas condições;

A Figura 5.2 apresenta o fluxo de execução dos componentes no funcionamento do framework. As pré-condições para o teste de agentes são definidas pelos cenários de teste e seus respectivos casos de teste. Em seguida, dá-se a

execução dos casos de testes envolvidos. Cada caso de teste configura o ambiente de execução do teste (define as pré-condições do teste) e, em seguida, cria o AUT e os agentes mock necessários (etapas 2 e 3). O número de agentes mock varia de acordo com o cenário de teste. Durante a execução do AUT o Monitor "monitora" o ciclo de raciocínio do AUT (etapa 4), coletando informações sobre a execução do AUT e suas trocas de mensagens com os agentes mock (etapa 1). As decisões e informações coletadas do AUT são armazenadas em estruturas internas do framework (etapa 5) permitindo, ao final do caso de teste, consultar tais informações e observar o comportamento executado pelo AUT. De acordo com o cenário de teste, o AUT pode interagir com outros agentes do ambiente e, nesses casos, tal iteração é simulada pelos agentes mock (etapas 7 e 8). Para orquestrar a ordem em que o AUT e os agentes mock interagem, temos o aspecto Synchronizer, que sinaliza o momento em que a comunicação deve ocorrer (etapa 6).

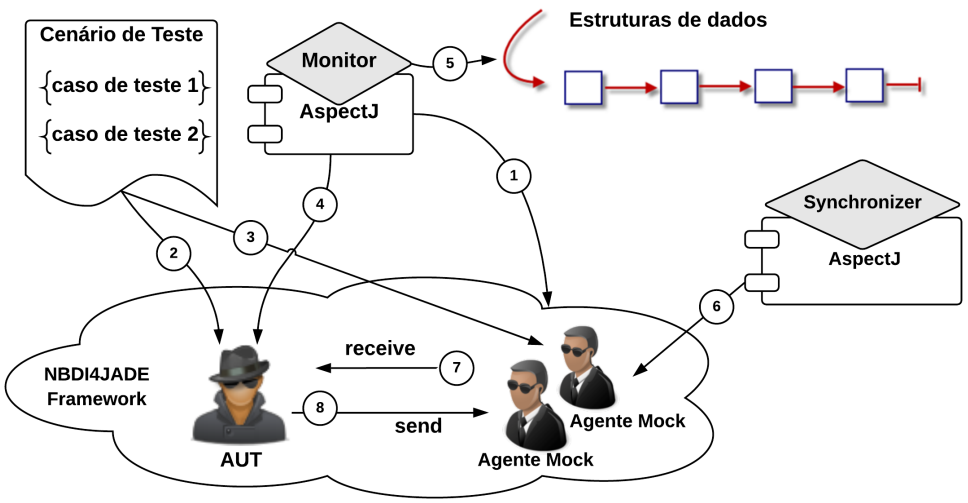


Figura 5.2: Interação entre os Componentes Participantes do Framework.

As seguintes estruturas de dados foram criadas para armazenar as informações coletadas pelo aspecto monitor durante a execução do caso de teste:

- *Normas Sensoreadas*: mantém as normas sensoreadas pelo AUT;
- *Planos Executados*: identifica quais planos foram selecionados e executados pelo AUT;
- *Planos Obrigatórios*: contém os planos classificados como obrigatórios;
- *Planos Permitidos*: contém os planos classificados como permitidos;
- *Planos Proibidos*: contém os planos classificados como proibidos;

- *Crenças*: contém as crenças do agente e seus valores;
- *Mensagens Recebidas*: contém as mensagens recebidas pelo AUT;
- *Mensagens Enviadas*: contém as mensagens enviadas pelo AUT;
- *Condições de Ativação*: mantém as condições de contexto quando ocorre a ativação de uma norma;
- *Condições de Desativação*: mantém as condições de contexto quando ocorre a desativação de uma norma;
- *Normas Ativadas*: contém as normas ativadas durante a execução do teste;
- *Normas Desativadas*: contém as normas desativadas durante a execução do teste;
- *WorkDoneList*: contém os identificadores dos agentes mock que completaram seu plano;

Quando o AUT é executado pelo caso de teste, o componente *Monitor* monitora o ciclo de raciocínio do AUT, coleta e armazena as informações nas estruturas de dados apropriadas. Por exemplo, na execução de um caso de teste, o AUT seleciona e executa o plano A, o Monitor observa que o plano A foi executado e armazena-o na lista de planos executados. Dessa forma, ao final da execução do agente, pode-se verificar quais planos foram executados pelo AUT. Neste caso, podemos verificar, explicitamente, se ele executou ou não o plano A.

Da mesma forma, quando um agente mock termina a execução de seu plano, o *Monitor* inclui o identificador do agente mock em *WorkDoneList* e notifica o caso de teste que a interação entre o agente mock e o AUT foi concluída (etapa 10). Sem essa notificação, o caso de teste não saberia quando as interações entre o AUT e um agente mock terminaram.

O componente *Synchronizer* é um elemento opcional, usado quando deseja-se testar um cenário no qual precisamos estabelecer uma ordem de interação entre os agentes mock e o AUT. O Synchronizer mantém uma lista com a sequência de interação com o AUT que é carregada no início do teste.

Por representarem dois interesses transversais do framework, os componentes *Monitor* e *Synchronizer* foram implementados como aspectos (Kiczales et al., 1997) (Kiczales et al., 2001), evitando, assim, o espalhamento de código responsável por monitorar e sincronizar a execução dos agentes, nos códigos dos agentes e da plataforma.

5.1.3 Projeto de Cenários de Teste

Uma consideração importante no teste de software é o projeto de cenários de teste eficazes (Myers et al., 2011). Por mais criativos e, aparentemente completos, os testes não podem garantir a ausência de falhas (Myers et al., 2011). Assim, um bom projeto de cenários de teste é importante para tentar criar testes tão completos quanto possíveis (Myers et al., 2011). Nesse caso, a principal questão torna-se: *Qual subconjunto dentre todos os cenários de teste possíveis tem a chance de detectar o maior número de falhas?*

Em geral, o método menos eficaz é escolher aleatoriamente um subconjunto qualquer de cenários de teste. A probabilidade de detectar a maioria das falhas utilizando uma coleção arbitrariamente selecionada de cenários de teste tem pouca chance de ser um subconjunto satisfatório (Myers et al., 2011). As abordagens propostas para o teste de sistemas baseados em agentes não definiram, até o momento, um método eficiente para seleção de cenários de teste. Para os exemplos discutidos nesta tese, adotamos a técnica de *error-guessing* (Myers et al., 2011) para o projeto de casos de teste, onde a ideia central é enumerar uma lista de cenários nos quais condições excepcionais podem ocorrer e, em seguida, escrever ou gerar os casos de testes com base na lista. A Figura 5.3 contém um modelo para a descrição do cenário de teste utilizado ao longo do trabalho.

Agente		*<Agente Under Test (AUT)>*
Cenário	**Input**	*<Descreve a entrada do cenário de teste que compreende as normas envolvidas, as mensagens que devem ser enviadas para o AUT e o estado do ambiente utilizado no cenário de teste>*
	Output	*<Descreve os comportamentos esperados para o AUT – se deve enviar uma mensagem específica ou deve afetar um recurso específico do ambiente>*

Figura 5.3: Template Utilizado nos Cenários de Teste.

5.1.4 Estrutura do Caso de Teste

Para executar um caso de teste, tarefas de configuração, inicialização e gerenciamento podem ser necessárias. Nossa abordagem utiliza uma estrutura para os casos de teste que permite a geração de casos de teste (apresentada no

Capítulo 6).A estrutura dos casos de testes no N-JAT4BDI framework suporta os seguintes recursos:

1. *Inicialização*: inclui tarefas de configuração das pré-condições do teste, como a definição de crenças, planos, condições de contexto e normas do ambiente;

2. *Atribuição dos dados de entrada*: informam ao framework de teste como atribuir os valores definidos na inicialização para configurar os testes;

3. *Interação externa*: fornece métodos apropriados que permite ao testador observar a interação do AUT com outros agentes do ambiente (agentes mock). Além disso, é fornecido um conjunto de assertivas de verificação que possibilitam checar os estados e as decisões do AUT ao longo da execução do caso de teste;

5.1.5 Métodos Verificadores

Com o objetivo de entender as decisões tomadas e observar as mudanças no estado interno do agente durante a execução do caso de teste, o componente *Monitor* coleta e preenche estruturas de dados específicas. Para consultar as informações armazenadas nessas estruturas de dados, definimos um conjunto de métodos verificadores (ou seja, assertivas de verificação, no estilo JUnit) que respondem se um agente se comportou ou não como esperado. O N-JAT4BDI fornece os seguintes métodos verificadores:

- *assertIsAddressed*: verifica se a norma foi endereçada ao agente;
- *assertIsActivated*: verifica se a norma está ativa;
- *assertNormFulfillment*: verifica se o agente cumpriu com a norma;
- *assertNormAffectGoal*: verifica se a norma afeta um objetivo do agente;
- *assertNormAffectPlan*: verifica se a norma afeta um plano do agente;
- *assertIsDeactivated*: verifica se a norma expirou durante a execução do teste;
- *assertReceivedReward*: verifica se o agente recebeu a recompensa por cumprir com a norma;
- *assertReceivedPunishment*: verifica se o agente recebeu a punição por violar com a norma;
- *assertIsPermission*: verifica a restrição normativa que afeta o agente é uma permissão;

- *assertIsProhibition*: verifica a restrição normativa que afeta o agente é uma proibição;
- *assertIsObligation*: verifica a restrição normativa que afeta o agente é uma obrigação;
- *assertNormState*: verifica o estado interno do elemento regulado pela norma;

5.1.6 Detalhando o Framework de Testes

Esta seção detalha os componentes apresentados na Figura 5.1.

Agentes Mock para Testar a Interação

Em um sistema real, os agentes interagem com outros agentes enviando e recebendo mensagens, observando e modificando o ambiente no qual estão situados. Para simular essa interação entre os agentes, o N-JAT4BDI utiliza implementações "falsas" de agentes reais, os *agentes mock* (Coelho et al., 2006). O agente mock é uma adaptação do conceito de *objetos mock* (Mackinnon et al., 2000), amplamente utilizado na orientação à objetos, para o contexto de sistemas multiagentes. Um agente mock interage com o AUT, enviando ou recebendo mensagens.

A Classe NMockAgent

A classe *NMockAgent* implementa o conceito de agentes mock no N-JAT4BDI e, como qualquer outro agente da plataforma, estende a classe *Agent*. Assim, um plano do *NMockAgent* representa um comportamento simples do JADE e define as mensagens enviadas pelo *NMockAgent* em sua interação com o AUT. Há muitas maneiras de relatar o resultado da interação com o AUT. Em nossa implementação, o *NMockAgent* implementa a interface *TestReporter* que disponibiliza um conjunto de métodos que devem ser implementados por um agente que deseja relatar o resultado da interação com o AUT. A Figura 5.4 apresenta o código parcial de um agente mock.

```
6  public class ChairMockAgent extends NMockAgent {
7
8      public void action() {
9
10         ACLMessage msg = receive();
11
12         if (msg != null) {
13             checkMessage(msg);
14         }
15
16         sendMessage(msg.getContent(), "reviewerAgent");
17     }
```

Figura 5.4: Código Parcial de um Agente Mock.

Monitorando o Ciclo de Raciocínio dos Agentes

O N-JAT4BDI define um aspecto para monitorar o ciclo de raciocínio do AUT e as trocas de mensagens com os agentes mock. Além disso, o aspecto *Monitor* tem a responsabilidade de coletar as informações do AUT e armazená-las nas estruturas de dados apropriadas.

O Aspecto Monitor

O aspecto *Monitor* define os pontos da plataforma e dos agentes que serão interceptados para monitorar as transições do estado interno do AUT em cada passo do ciclo de raciocínio. As informações monitoradas e coletadas são armazenadas em um conjunto de estruturas de dados internas e disponibilizas durante a execução do teste. A Figura 5.5 apresenta o código parcial do aspecto Monitor no qual uma norma é adicionada ao conjunto de normas ativas.

Sincronizando a Interação dos Agentes

O aspecto *Synchronizer* intercepta o código da classe *NMockAgent*, responsável pelo envio de mensagens para o AUT, e adiciona um comportamento antes do envio de uma mensagem para verificar se é a sua vez de enviá-la para AUT. A Figura 5.6 apresenta o código parcial do aspecto Synchronizer.

O Aspecto Synchronizer

As linhas 9 e 10 definem o ponto no código do *NMockAgent* responsável pelo envio de mensagens. O aspecto adiciona código extra para verificar se é sua

```
 9⊕ * Aspecto que monitora o ciclo de raciocínio do agente
13 public aspect AgentMonitor {
14
15   /** Estrutura de dados que armazena as decisões do agente */
16   private DataList dataList = getInstance();
17
19⊕   * Monitora quando uma norma é adicionada ao ambiente
23⊖  pointcut addNewNorm(Norm norm) :
24       execution(void BeliefBase.addNorm(..)) && args(norm);
25
27⊕   * Adiciona uma norma ao conjunto de normas ativas
31⊖  after(Norm norm) : addNewNorm(norm) {
32     dataList.addActiveNorm(norm);
33     dataList.addAddressee(norm);
34   }
35 }
```

Figura 5.5: Código Parcial do Aspecto Monitor.

vez de enviar uma mensagem para o AUT (linhas 12-21). A classe *OrderList* contém os identificadores dos agentes mock que devem enviar mensagens para o AUT, ordenado pela prioridade de interação (linha 13). Assim, se o método *orderList.checkTurn()* retornar verdadeiro, significa que o agente mock pode enviar uma mensagem para o AUT, caso contrário, deve aguardar um tempo e verificar novamente se chegou sua vez de enviar a mensagem (linhas 14-16).

```
 7 public aspect Synchronizer {
 8
 9⊖    pointcut MockSendMessage(Agent agent, ACLMessage message) :
10        execution(void NMockAgent.sendMessage(..)) && args(agent, message);
11
12⊖    before(Agent agent, ACLMessage message) : MockSendMessage (agent, message) {
13        OrderList orderList = OrderList.getInstance();
14        while (!orderList.checkTurn(agent.getAID())) {
15            try {
16                Thread.sleep(500);
17            } catch (InterruptedException e) {
18                e.printStackTrace();
19            }
20        }
21    }
22 }
```

Figura 5.6: Código Parcial do Aspecto Synchronizer.

Criando Casos de Teste

Como pode ser visto na Figura 5.1, o N-JAT4BDI permite a construção e execução de casos de teste ao estender o framework de testes JUnit. Esta estratégia tem a vantagem de aproveitar o aprendizado dos desenvolvedores,

se estes já estiverem familiarizados com o JUnit. Entretanto, para suportar o teste de agentes BDI normativos, a classe *NTestCase* fornece um conjunto de assertivas de verificação (semelhantes aos métodos assertivos do JUnit) e um conjunto métodos para configuração do ambiente, definindo as pré-condições necessárias para execução do caso de teste. A Figura 5.7 ilustra como o caso de teste é monitorado pelo framework. Durante a execução do caso de teste, o aspecto Monitor intercepta cada passo do ciclo de raciocínio e armazena nas estruturas de dados apropriadas as informações coletadas. Essas informações registram todas as decisões e estados alcançados pelo agente durante a execução do caso de teste e são consultadas pelos métodos assertivos.

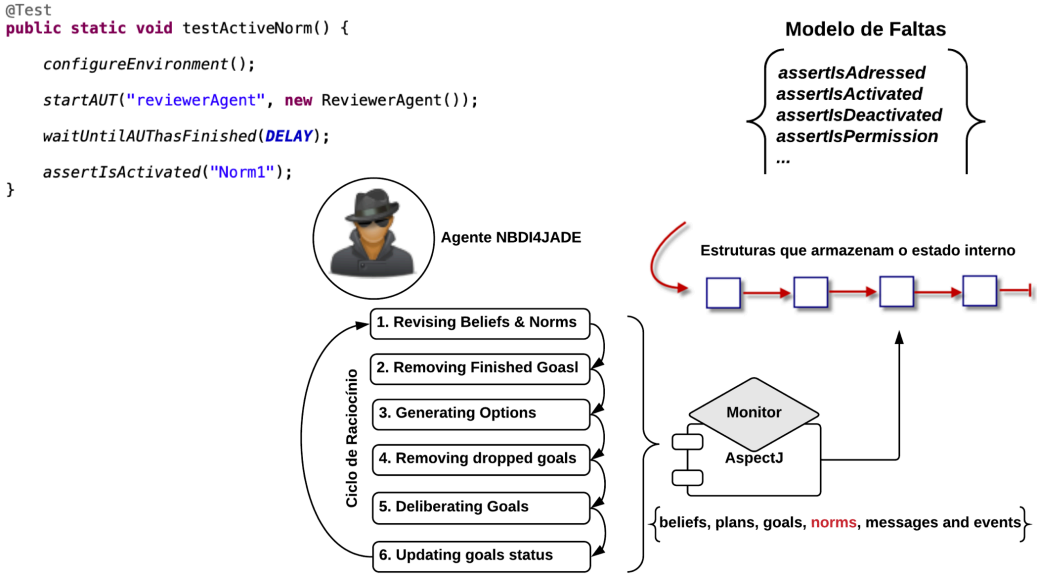


Figura 5.7: Visão Geral da Execução do Framework N-JAT4BDI.

A Classe NTestCase

A Figura 5.8 apresenta um exemplo de um caso de teste. A classe *NumberOfReviewerTestCase* estende a classe *NTestCase*. O método *testActiveNorm* testa se uma norma está ativa. Inicialmente, o caso de teste é configurado com as pré-condições necessárias ao teste (linha 15). Na sequência, é criado o AUT (linha 17). O método *waitUntilAUThasFinished* (linha 19) lida com questões de sincronização entre as threads do AUT e do teste. A linha 21 cria uma instância da norma *NumberOfReviewers*. Por fim, a linha 23 verifica se a norma está ativa.

```
8  public class NumberOfReviewerTestCase extends NTestCase {
9
10     private static final long DELAY = 3000l;
11
12     @Test
13     public static void testActiveNorm() {
14
15         configureEnvironment();
16
17         startAUT("reviewerAgent", new ReviewerAgent());
18
19         waitUntilAUThasFinished(DELAY);
20
21         Norm norm = new Norm("NumberOfReviewers");
22
23         assertNormActive(norm);
24     }
25 }
```

Figura 5.8: Exemplo de um Caso de Teste.

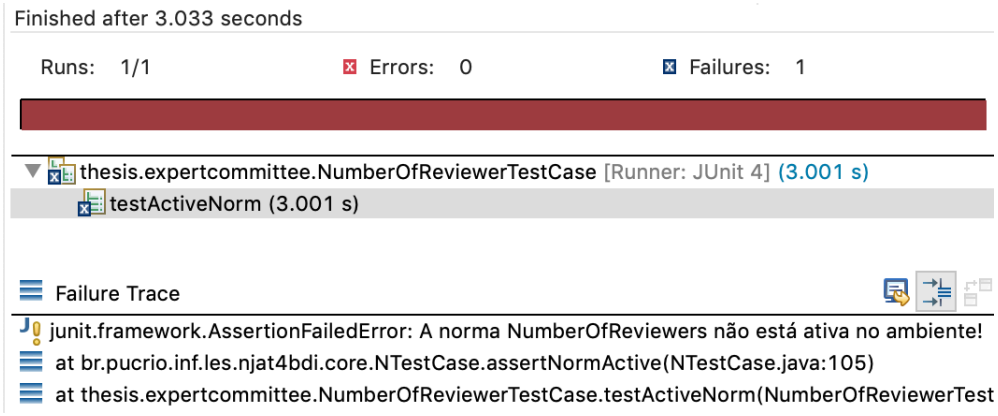


Figura 5.9: Resultado da Execução dos Casos de Teste.

Visualizando os Resultados

O N-JAT4BDI utiliza a infraestrutura do JUnit para executar e visualizar os resultados dos casos de teste. A Figura 5.9 apresenta o resultado da execução de um caso de teste. Podemos observar que o teste *testActiveNorm* falhou ao verificar se uma norma específica estava ativa no ambiente. O N-JAT4BDI exibe a barra vermelha, mesma indicação do JUnit para falhas na execução de casos de teste e uma mensagem apropriada no console do teste.

5.2 NBDI4JADE: Um Framework para Construção de Agentes Normativos

Esta seção apresenta o NBDI4JADE framework que possibilita a construção de agentes BDI normativos, ou seja, agentes que operam segundo seus interesses enquanto consideram as normas do ambiente. O NBDI4JADE é uma

versão normativa do framework BDI4JADE (Nunes et al., 2011), o qual segue a arquitetura BDI mas não lida com questões normativos. A Seção 5.2.1 apresenta uma visão geral do framework, a Seção 5.2.2 apresenta os componentes normativos do framework, a Seção 5.2.3 outros componentes importantes que compõem o framework e seu funcionamento e, por fim, a Seção 5.2.4 apresenta os pontos fixos e flexíveis do framework.

5.2.1 Visão Geral do Framework

A Figura 5.10 apresenta uma visão arquitetural do NBDI4JADE como extensão da arquitetura BDI de Rao e Georgeff (Rao et al., 1995), onde as partes em azul representam os pontos onde foram introduzidas modificações e extensões na arquitetura original.

Um agente NBDI4JADE funciona como segue: o agente percebe informações do ambiente através de seus sensores e atualiza suas crenças utilizando a função *BeliefNormReviewFunction* que, além de revisar as crenças dos agentes, é responsável por: (i) verificar se novas normas sensoreadas são endereçadas ao agente e adicionar tais normas ao conjunto de normas adotadas; (ii) atualizar o conjunto de normas ativadas e desativadas, considerando que algumas normas se tornam ativas e outras inativas devido às percepções, ativação, desativação, cumprimento, ou violação de outras normas, (iii) e verificar o cumprimento e violação das normas, considerando que algumas podem ter sido cumpridas e outras violadas durante o funcionamento do agente.

Após a revisão das crenças e normas outra extensão é fornecida pela função *NormSelectionFunction* que é responsável por: (i) verificar e resolver conflitos entre normas; (ii) selecionar as normas não conflitantes que o agente tem interesse em cumprir ou violar e, (iii) adicionar novos eventos a base de eventos do agente levando em consideração as decisões normativas.

A função *NormFilter* é responsável por: (i) selecionar a intenção de maior importância para ser executada, relacionando-a com as condições de invocação e contexto dos planos gerando um conjunto de planos relevantes; (ii) analisar o contexto de tais planos e, de acordo com as crenças e normas do agente, gerar um conjunto de planos aplicáveis; (iii) selecionar o plano aplicável de maior importância para ser executado e, atualizar as intenções do agente a partir do plano selecionado. Nas seções seguintes são descritas cada uma das funcionalidades normativas.

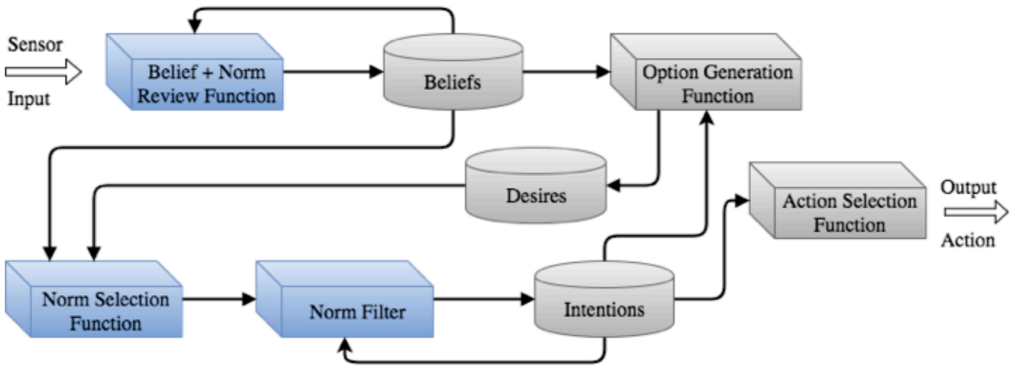


Figura 5.10: Arquitetura BDI + Arquitetura NBDI.

5.2.2 Componentes Normativos do Framework

Em seguida, são descritos cada uma das funcionalidades normativas da arquitetura.

A Função *BeliefNormReviewFunction*

A função *BeliefNormReviewFunction* executa três tarefas relacionadas as normas: *adopting norms*, que analisa o conjunto de normas adotadas; *reviewing (de)activated norms*, que revisa o conjunto de normas ativadas e desativadas e, *verifying fulfillment norms*, que verifica as normas cumpridas ou violadas.

A tarefa *adopting norms* recebe como entrada as novas normas e atualiza o conjunto de normas adotadas, verificando: (i) se a nova norma não existe na base de normas adotadas, e (ii) se o agente é o destinatário da norma; O Algoritmo 1 apresenta a operacionalização desta tarefa. A operação têm início a partir do conjunto de normas sensoreadas pelo agente e, para cada norma sensoreada, as seguintes verificações são realizadas: (i) se a norma sensoreada ainda não existe na base de normas adotadas (linha 2), e (ii) se a norma sensoreada é endereçada ao agente (linha 3). Se ambas as condições são satisfeitas, a norma é adicionada ao conjunto de normas adotadas (linha 4). A verificação se uma norma sensoreada já existe na base de normas adotadas é realizada utilizando a função *exist* que recebe uma norma e um conjunto de normas, e retorna *true* se a norma já existe no conjunto de normas. Já a verificação se a norma sensoreada é endereçada ao agente é realizada pela função *isAddressees* que recebe como entrada o agente, os papéis assumidos por ele e os grupos que o mesmo pertence e o campo Addressees da norma.

Após analisar crenças e normas, é executada a tarefa *Reviewing (De)Activated Norms* para verificar e atualizar o conjunto de normas ativa-

Algorithm 1: Tarefa *Adopting Norms*

Data: Normas identificadas, representadas por *sensednorms*
Data: Normas adotadas, representadas por *adoptednorms*
Data: Nome do agente, representado por *name*
Data: Papéis assumidos pelo agente, representado por *rules*
Data: Grupos que o agente faz parte, representado por *groups*
Result: Conjunto de normas adotadas atualizado

```
1 foreach Norm sn in sensednorms do
2     if !exist(sn, sensednorms) then
3         if isAddressees(name, roles, groups, sn.addressees) then
4             adoptednorms.add(sn);
5         end
6     end
7 end
```

das ou desativadas. O Algoritmo 2 descreve a operacionalização da tarefa.

Esta tarefa é executada em dois passos. O primeiro passo *verifica as normas que se tornaram ativas*, isto é, o contexto de ativação de cada norma adotada pelo agente é verificado através do método *verifyContext* (linha 2), se o contexto de ativação é satisfeito, a norma é adicionada ao conjunto de normas ativadas (linha 3). O segundo passo é a *verificação das normas desativadas*, isto é, o contexto de desativação de cada norma ativada é verificado utilizando a função *verifyContext* (linha 7), se o contexto de desativação é satisfeito, a norma desativada é removido do conjunto de normas ativadas (linha 8) e adicionada ao conjunto de normas desativadas (linha 9).

Após revisar as normas ativadas e desativadas, a tarefa *VerifyFulfillment-Norms* verifica as normas cumpridas ou violadas como descrito no Algoritmo 3.

A tarefa é realiza em dois passos. O primeiro passo *verifica as normas ativadas que foram cumpridas ou violadas* como segue: (i) se a norma é uma obrigação e o comportamento regulado pela mesma foi realizado, tal norma foi cumprida e a relação `normfulfillment` da norma é atualizada para FULFILLED (linha 2); e, (ii) se a norma é uma proibição e o comportamento regulado pela mesma foi realizado, tal norma foi violada e a relação `normfulfillment` da norma é atualizada para VIOLATED (linha 3). O segundo passo *verifica as normas desativadas que foram cumpridas ou violadas* como segue: (i) se a norma é uma obrigação e o comportamento regulado pela mesma não foi realizado, tal norma foi violada e a relação `normfulfillment` da norma é atualizada para VIOLATED (linhas 13 e 14) e, (ii) se a norma é uma proibição e o comportamento regulado pela mesma não foi realizado, tal norma foi cumprida e a relação `normfulfillment` da norma é atualizada para FULFILLED (linhas 16 e 17).

Algorithm 2: Tarefa *Reviewing (De)Activated Norms*

```
Data: Base de crenças do agente, representada por bels
Data: Normas adotadas, representadas por adoptednorms
Data: Normas ativadas, representadas por acnorms
Data: Normas desativadas, representadas por deacnorms
Result: Conjunto de normas ativadas e desativadas atualizado
1  foreach Norm ad in adoptednorms do
2      if verifyContext(ad.activation, bels, acnorms, deacnorms) then
3          acnorms.add(ad);
4      end
5  end
6  foreach Norm deac in acnorms do
7      if verifyContext(deac.deactivation, bels, acnorms, deacnorms)
       then
8          acnorms.remove(deac);
9          deacnorms.remove(deac);
10     end
11 end
```


A Função *NormSelectionFunction*

Os objetivos da função *NormSelectionFunction* são: (i) gerar objetivos de acordo com as crenças do agente; (ii) selecionar as normas ativadas que o agente tem intenção de cumprir ou violar e, (iii) atualizar o conjunto de objetivos, levando em consideração as normas selecionadas. Dessa forma, são executadas as tarefas *Detecting and Overcoming Conflicts*, *Choosing Norms* e *Compliance with Norms*.

A fim de selecionar as normas que devem ser cumpridas ou violadas, o agente precisa, primeiramente, avaliá-las, isto é, medir os ganhos e perdas pelo cumprimento ou violação das normas do sistema a partir dos componentes das normas. Para isso, são avaliados sua *influência deôntica* (ou seja, obrigação ou proibição) sobre o comportamento regulado e a *importância* para o agente em receber as recompensas ou punições da norma.

A *influência deôntica* de uma norma avalia se a norma é uma obrigação, ou seja, se a norma influencia o agente a realizar o comportamento regulado por ela. Neste caso, a *influência deôntica* é igual a importância do comportamento ser realizado. Se a norma é uma proibição, ela impede que o agente realize um determinado comportamento, então a *influência deôntica* é igual ao negativo da importância do comportamento ser realizado.

Para avaliar a *importância* por receber as recompensas ou punições (ou seja, as sanções) basta avaliar as sanções associadas. A *importância* por receber

Algorithm 3: Tarefa *Verifying Fulfillment Norms*

```
Data: Normas ativadas, representadas por acnorms
Data: Normas desativadas, representadas por deacnorms
Data: Um mapa (estrutura de dados) indicando se uma norma foi
      cumprida ou violada
Data: Um mapa indicando se o comportamento regulado pela
      norma foi realizado
Result: Conjunto de normas ativadas e desativadas atualizado
1  foreach Norm ac in activatednorms do
2      if behaviorrealization.get(ac) == realized then
3          if ac.deonticconcept = OBLIGATION then
4              normfulfillment.put(ac, FULFILLED)
5          end
6          if ac.deonticconcept = PROHIBITION then
7              normfulfillment.put(ac, VIOLATED)
8          end
9      end
10 end
11 foreach Norm deac in deactivatednorms do
12     if behaviorrealization.get(deac) == not-realized then
13         if deac.deonticconcept = OBLIGATION then
14             normfulfillment.put(deac, FULFILLED)
15         end
16         if deac.deonticconcept = PROHIBITION then
17             normfulfillment.put(deac, VIOLATED)
18         end
19     end
20 end
```

um conjunto de sanções é igual a soma da importância por receber cada sanção. Assim, como resultado é verificado se é mais importante cumprir com a norma e receber as recompensas ou violá-la e receber as punições.

Assim, avaliar a norma considera se a soma da *influência deôntica* da norma e da *importância* das recompensas é maior que do que a *importância* das punições da norma.

A tarefa *Detecting and Overcoming Conflicts*, é responsável por detectar e resolver conflitos entre normas. Se duas normas diferentes estão ativadas, sendo uma delas uma obrigação e a outra uma proibição e, regulam o mesmo comportamento então, elas estão em conflito e o agente não é capaz de cumprir ou violar ambas as normas, ao mesmo tempo. O Algoritmo 4 apresenta a operacionalização da tarefa.

Se a *influência deôntica* da primeira norma mais a *importância* por cumprir a primeira norma e a importância das punições por violar a segunda norma é maior do que a importância por receber as recompensas por cumprir

Algorithm 4: Tarefa *Detecting and Overcoming Conflicts*

```
   Data: Normas ativadas, representadas por activatednorms
   Data: Um Map representando a norma selecionada, representada
         por normselection
   Result: As normas a serem cumpridas ou violadas
 1 foreach Norm ac1 in activatednorms do
 2     foreach Norm ac2 in activatednorms and ac1 diferente ac2 do
 3         if detectedConflict(ac1, ac2) then
 4             if deonticInfluence(ac1) + rewardsImportance(ac1) +
               punishmentsImportance(ac2) >= rewardsImportance(ac2)
               + punishmentsImportance(ac1) then
 5                 normselection.put(ac1,TO_BE_FULFILLED);
 6                 normselection.put(ac2,TO_BE_VIOLATED);
 7             else
 8                 normselection.put(ac1,TO_BE_VIOLATED);
 9                 normselection.put(ac2,TO_BE_FULFILLED);
10             end
11         end
12     end
13 end
```

a segunda norma mais a importância por receber as punições por violar a primeira norma (linha 4). Se tal condição é satisfeita, a primeira norma é selecionada para ser cumprida e o seu status de seleção é atualizado para TO_BE_FULFILLED, a segunda norma é selecionada para ser violada, sendo seu status atualizado para TO_BE_VIOLATED. Caso contrário, a primeira norma é selecionada para ser violada e a segunda norma para ser cumprida.

A tarefa *Choosing Norms* é responsável por selecionar quais normas não conflitantes serão cumpridas ou violadas e sua operacionalização é apresentada no Algoritmo 5. Assim como na superação de conflitos, normas não conflitantes são selecionadas para serem violadas ou cumpridas levando em conta o que é mais vantajoso para o agentes.

Se o agente decide cumprir uma obrigação, receberá as recompensas e não receberá as punições. Caso contrário, se o agente decide violar uma obrigação, receberá as punições e não receberá as recompensas. No caso da proibição, se o agente decide cumprir uma proibição, receberá as recompensas e não receberá as punições. Caso contrário, se o agente decide violar uma proibição, receberá as punições e não receberá as recompensas.

Assim, o agente decide por cumprir uma norma se a soma de sua *influência deôntica* mais a *importância* por receber suas recompensas é maior ou igual a *importância* por receber as punições da norma. Caso contrário, o agente decide por violar a norma.

Algorithm 5: Tarefa *Choosing Norms*

```
   Data: Normas ativadas, representadas por activatednorms
   Data: Um Map representando a norma selecionada, representada
         por normselection
   Result: As normas a serem cumpridas ou violadas
1  foreach Norm ac in activatednorms do
2      if nonConflicting(ac) then
3          if deonticInfluence(ac) + rewardImportance(ac) >=
           punishimentImportance(ac) then
4              normselection.put(ac, TO_BE_FULFILLED);
5          else
6              normselection.put(ac, TO_BE_VIOLATED);
7          end
8      end
9  end
```

Após selecionar as normas a serem cumpridas ou violadas, a tarefa *Compliance with Norms* é responsável por rever os objetivos do agente para tornar o agente ciente das decisões tomadas. A realização desta tarefa é operacionalizada pelo Algoritmo 6 que verifica se existe um comportamento que efetive a decisão normativa (linha 2). Se não existe e uma norma de obrigação foi selecionada para ser cumprida ou uma norma de proibição foi selecionada para ser violada, então um novo evento, representando a adição do comportamento regulado pela norma é gerado e adicionado.

Algorithm 6: Tarefa *Compliance with Norms*

```
   Data: Normas ativadas, representadas por activatednorms
   Data: Um Map representando a norma selecionada, representada
         por normselection
1  foreach Norm ac in activatednorms do
2      if !existBehavior(ac) then
3          if (ac.deonticconcept = OBLIGATION and
           normselection.get(ac) = TO_BE_FULFILLED) or
           (ac.deonticconcept = PROHIBITION and
           normselection.get(ac) = TO_BE_VIOLATED) then
4              normselection.put(ac, TO_BE_FULFILLED);
5          end
6      end
7  end
```

A Função *NormFilter*

Os principais objetivos da função *NormFilter* são selecionar um objetivo para ser alcançado, recuperar os planos relevantes para atingir tal objetivo, verificar os planos aplicáveis, selecionar o plano de maior importância e, rever o conjunto de intenções. Para tanto, esta função executa as seguintes tarefas:

A tarefa *SelectingGoal* é responsável pelo processo de seleção de objetivos, onde cada objetivo possui uma prioridade associada que é definida levando em consideração a motivação do agente em atingir tal objetivo. A prioridade de um objetivo é definida como a soma da motivação do agente para atingir o objetivo e a influência normativa das normas sobre tal objetivo. A influência normativa avalia, para comportamentos obrigatórios, a importância para o agente realizar o comportamento, receber as recompensas e não receber as punições e, para comportamentos proibidos, a importância para o agente realizar o comportamento, receber as punições e não receber as recompensas. O Algoritmo 7 calcula e retorna a prioridade do objetivo (linha 2).

Algorithm 7: priorityGoal(goal, activatednorms)

Data: Objetivo do agente, representado pelo tipo *goal*
Data: Conjunto de normas ativadas, representado por activatednorms
Result: Prioridade do objetivo

```
1 priority = 0;
2 priority = motivation(goal) + normativeInfluence(goal,
  activatednorms);
3 return priority;
```

Após definir a prioridade do objetivo, o Algoritmo 8 é executado a fim de selecionar o objetivo de maior prioridade. Primeiro, é estabelecido que o primeiro objetivo é o de maior prioridade (linha 1). Na sequência, é verificada a prioridade de cada objetivo e o de maior prioridade é retornado (ver linhas de 2 a 7).

Após selecionar o objetivo de maior prioridade o próximo passo é encontrar os planos relevantes. Esta etapa não sofreu modificações e funciona como no BDI4JADE. Após encontrar os planos relevantes, é necessário verificar quais deles são aplicáveis. Para tanto, a tarefa *CheckContext* foi criada a fim de permitir a verificação explicita de condições normativas e funciona como apresentado no Algoritmo 9, que executa os seguintes passos:

1. Verifica se o contexto está relacionado somente a condições que devem ser satisfeitas a partir das crenças do agente (linhas 1 e 2). Neste caso, o plano deve fazer parte dos planos aplicáveis;

Algorithm 8: selectGoal(goals, activatednorms)

```
Data: Conjunto de objetivos do agente, representadas por goals
Data: Conjunto de normas ativadas, representadas por
      activatednorms
Result: Objetivo com maior prioridade
1 selectGoal = goals.getGoal(0);
2 foreach Goal goal in goals do
3 |   if priorityGoal(goal, activatednorms) > priorityGoal(selectGoal,
  |   activatednorms) then
4 |   |   selectGoal = goal;
5 |   end
6 end
7 return selectGoal;
```

2. Verifica se o contexto está relacionado a condições normativas (linha 4), em caso afirmativo, o método *normativeConsequence* é utilizado para verificar se o contexto é satisfeito a partir do estado atual das normas do agente (linha 5);

Algorithm 9: checkContext(context, beliefs, acnorms,deacnorms)

```
Data: Um contexto, representado por context
Data: Um conjunto de crenças, representadas por bels
Data: Um conjunto de normas ativadas, representadas por acnorms
Data: Um conjunto de normas ativadas, representadas por
      deacnorms
Result: Se a condição de contexto é satisfeita
1 if context type beliefcontext then
2 |   return verifyContext(context, bels);
3 end
4 if context type normactiveconditioncontext then
5 |   return normativeConsequence(context, acnorms, deacnorms);
6 end
```

Após encontrar os planos relevantes e aplicáveis é necessário escolher um deles para ser executado. A fim de capacitar o agente com a habilidade necessária para tomar tal decisão, modificamos a função *selectPlan*, onde o plano de maior importância é selecionado e a importância de cada plano é avaliada considerando a prioridade da condição de invocação do plano, a importância de realizar os comportamentos que compõem o corpo do plano e a influência normativa sobre o plano. O plano com maior importância é retornado como descrito no Algoritmo 10.

A importância do plano, operacionalizada no Algoritmo 11, é definida pelas seguintes funções:

Algorithm 10: selectPlan(plans, acnorms)

```
Data: Lista de planos, representadas por plans
Data: O conjunto de normas ativadas, representadas por acnorms
Result: O plano com maior importância
1 selectedPlan = plans.get(0);
2 foreach Plan plan in plans do
3 |   if planImportance(plan, acnorms) >
  |   planImportance(selectedPlan, acnorms) then
4 |   |   selectedPlan = plan;
5 |   end
6 end
7 return selectedPlan;
```

(i) *planPriority*, descrita no Algoritmo 11, avalia a prioridade da condição de invocação do plano;

(ii) *mainImportance*, descrita no Algoritmo 12, avalia a importância de atingir os objetivos e ações que compõem o plano e;

(iii) *normativeInfluence*, descrita no Algoritmo 13, avalia a influência das normas ativas sobre o plano.

Algorithm 11: planImportance(plan, acnorms)

```
Data: Plano, representadas por plan
Data: Um conjunto de normas ativadas, representadas por acnorms
Result: A importância do plano
1 return planPriority(plan, acnorms) + mainImportance(plan) +
  normativeInfluence(plan, acnorms);
```

Algorithm 12: mainImportance(plan)

```
Data: Corpo do Plano, representado por plan
Data: Um conjunto de normas ativadas, representadas por acnorms
Result: A motivação de executar as ações do plano
1 foreach NormativeCondition nc in acnorms do
2 |   bmi = bmi + motivation(plan);
3 end
4 return bmi;
```

Algorithm 13: bodyNormativeInfluence(body, acnorms)

Data: Plano, representadas por *plan*
Data: O conjunto de normas ativadas, representadas por *acnorms*
Result: A influência normativa sobre o plano
1 **foreach** *NormativeCondition nc in acnorms* **do**
2 | bni = bni + normativeInfluence(plan, acnorms);
3 **end**
4 return bni;

5.2.3 Outros Componentes do NBDI4JADE

A seção anterior forneceu uma visão dos componentes da arquitetura BDI que foram modificados para lidar com as questões normativas. Nesta seção, serão apresentados outros componentes do framework.

Agente NBDI. Um agente NBDI representa um agente normativo que segue a arquitetura BDI. Ele agrega um ciclo de raciocínio, responsável pelo mecanismo deliberativo o qual direciona o comportamento, as estratégias e os recursos do agente.

Capacidade. Um agente NBDI não inclui diretamente uma base de crenças e uma biblioteca de planos, mas elas fazem parte da capacidade. A capacidade é uma parte estrutural do agente constituída de (i) um conjunto de planos e (ii) um subconjunto da base de conhecimento que é manipulado por esses planos. Esse recurso é um mecanismo de design para dar suporte à modularidade e à capacidade de reutilização.

Estratégias. Um agente NBDI está associado a diferentes estratégias, que são pontos para personalizar o ciclo de raciocínio, permitindo modificar o comportamento padrão dos agentes. Estratégias egoístas ou sociais podem ser personalizadas à cada agente através de diferentes estratégias.

Goal. Os objetivos representam o estado motivacional do sistema. É uma entidade que representa um desejo que o agente quer alcançar.

Intenção. A intenção captura o componente deliberativo do sistema e representa um objetivo com o qual o agente está comprometido em alcançar, ou seja, quando um agente tem uma intenção, ele seleciona planos para tentar alcançar essa intenção.

Base de Crenças e **Crença**. As crenças representam características do agente e do ambiente que são atualizadas de acordo com a percepção das mudanças ocorridas neste ambiente. As crenças podem ser vistas como o componente informativo do sistema e a base de crenças é um conjunto dessas informações.

Biblioteca de Planos e **Plano**. O NBDI4JADE fornece uma infraestrutura para selecionar e executar planos a partir de uma biblioteca de planos existente. Os planos contêm as ações que são executadas para atingir um objetivo específico.

Eventos. O NBDI4JADE fornece meios para criar *observadores* de crenças, objetivos e normas, a fim de notificá-los quando esses componentes são atualizados. Qualquer componente que se registre como um observador é notificado quando crenças são criadas, atualizadas ou removidas, por exemplo.

Esses componentes são usados no ciclo de raciocínio do agente, que é baseado no interpretador BDI apresentado em (Rao et al., 1995). Este ciclo é implementado em seis etapas e considera as questões normativas mencionadas anteriormente:

1. *Revisar de crenças*: Este primeiro passo do ciclo consiste em revisar as crenças dos agentes. Todo tratamento para percepção, ativação e desativação de uma norma também é realizado neste passo.

2. *Remover os objetivos encerrados*. Antes do ciclo de raciocínio ser executado, alguns objetivos podem já ter sido "encerrados", isto é, já foram alcançados ou, não são mais desejadas ou ainda, são consideradas inatingíveis. Assim, tais objetivos são removidos do conjunto de objetivos.

3. *Gerar opções*. Nesta etapa, os objetivos disponíveis para o agente são determinados. Pode-se também, gerar novos objetivos, determinar que objetivos existentes não são mais desejados ou manter objetivos.

4. *Remover objetivos cancelados*. Quando um objetivo é determinado como não mais desejado no passo anterior, ele é removido do conjunto de objetivos do agente.

5. *Deliberar objetivos*. Nesta etapa, os objetivos do agente são particionados em dois grupos: (i) objetivos à serem atingidos, as intenções e, (ii) objetivos que não serão tentados a alcançar. Este último permanece como um desejo do agente, mas o agente não está comprometido em alcançá-lo no momento.

6. *Atualizar o status dos objetivos*. Com base na partição realizada na etapa anterior, o status dos objetivos é atualizado.

NBDI4JADE Core

Um agente BDI normativo deve estender a classe *NBDIAgent*, que por sua vez é uma extensão da classe *Agent* do JADE. Um agente NBDIAgent (que a partir de agora será referido apenas como agente) é composto por um conjunto de intenções e capacidades.

Quando um objetivo é adicionado ao agente, uma intenção é criada e associada ao objetivo. As intenções possuem status, que são: (i) *achieved* – o objetivo foi alcançado; (ii) *no longer desired* – o objetivo não é mais desejado; (iii) *plan failed* – o último plano executado falhou; (iv) *trying to achieve* – o agente está executando um plano para alcançá-lo; (v) *unachievable* – todos os planos disponíveis foram executados, mas nenhum deles obteve sucesso; e (vi) *waiting* – o agente não está tentando alcançar o objetivo.

Conforme mencionado, crenças e planos não fazem parte diretamente de um agente, conforme proposto na arquitetura BDI, mas parte de capacidades. As plataformas de agentes JACK (Winikoff, 2005) e Jadex (Pokahr et al., 2005) também implementam esses conceitos dessa forma. Como uma capacidade está associada a um conjunto de planos, e estes aos objetivos que podem alcançar, esses objetivos definem os objetivos que a capacidade pode alcançar.

NBDI4JADE Goal

Um objetivo é qualquer objeto Java que implemente a interface *Goal*. É fornecido um conjunto de *goals* predefinidos: (i) *BeliefGoal* – a entrada desse goal é o nome de uma crença. O goal é alcançado quando uma crença com o nome informado é parte das crenças do agente; (ii) *BeliefSetValueGoal<T>* – a entrada desse goal é o nome e o valor de uma crença. O goal é alcançado quando uma crença com o nome e o valor fornecidos é parte das crenças do agente; (iii) *CompositeGoal* – representa um goal que é formado por outros subobjetivos. Possui duas subclasses, que indicam se o objetivo deve ser alcançado de forma paralela ou sequencial; (iv) *ParallelGoal* – representa um goal que visa atingir todos os subobjetivos que a compõem de forma paralela; (v) *SequentialGoal* – representa um goal que visa atingir todos os subobjetivos que a compõem de forma sequencial e, (vi) *MessageGoal* – é alcançado quando uma mensagem específica é recebida pelo agente.

NBDI4JADE Belief

A classe *BeliefBase* oferece métodos para adicionar, remover e atualizar crenças. Uma crença tem duas propriedades principais: um nome e um valor. O nome da crença deve ser único na base de crenças. A classe *Belief<T>* é uma classe abstrata e define os métodos que devem ser implementados por subclasses para a definição e recuperação de uma crença.

NBDI4JADE Plan

A representação dos planos está associada a três classes principais: (i) *Plano* – não indica diretamente as ações a serem executadas, mas possui algumas informações, tais como: (i) identificação do plano; (ii) a biblioteca de planos a qual pertence; (iii) os goals que é capaz de alcançar; e (iv) os modelos de mensagens que podem ser processados. Além disso, define métodos importantes que devem ser implementados pelas subclasses:

- *createPlanBody()*: retorna uma instância de um comportamento do JADE, que corresponde ao corpo a ser executado para atingir o objetivo. Essa instância de comportamento também deve implementar a interface *PlanBody* (verificada em tempo de execução).
- *initGoals()*: método que inicia os goals que o plano pode alcançar;
- *initMessageTemplates()*: método que inicia os modelos de mensagens (do JADE) que oplano pode processar;
- *matchesContext(Goal goal)*: método que verifica um contexto para determinar se o plano é capaz de alcançar o goal;

A classe *PlanInstance* representa a instância de um plano, criada para alcançar um objetivo específico, de acordo com a especificação de um plano. Internamente, a instância de um plano possui referência para: o comportamento a ser executado; a intenção com a qual o agente está comprometido a alcançar; o plano associado e, o estado interno da instância do plano.

A classe *PlanBody* representa as ações que devem ser executadas pelo plano. Dois métodos devem ser implementados pelo corpo do plano: (i) *EndState getEndState()* – retorna o estado final do corpo do plano. Se este ainda não terminou, é retornado *null*. O NBDI4JADE entende que o plano executou com sucesso quando seu corpo termina com um estado final SUCCESSFUL e, (ii) *init(PlanInstance planInstance)* – método invocado quando o corpo do plano é instanciado.

NBDI4JADE Message

As mensagens são recebidas e enviadas no NBDI4JADE utilizando a infraestrutura oferecida pelo JADE. As interações são feitas enviando mensagens e usando o método *receive* (MessageTemplate) para receber uma resposta.

NBDI4JADE Event

O NBDI4JADE implementa os eventos através do padrão de projeto *observer* (Gamma, 1995) para possibilitar a observação de eventos que ocorram em um agente. Atualmente, existem três tipos de eventos: eventos de crença, objetivo e norma. Os ouvintes de crenças podem ser associados a uma base de crenças e, quando uma crença é adicionada, removida ou alterada, o ouvinte será notificado. Os ouvintes de objetivos, por sua vez, estão associados a uma intenção e não notificados quando ocorre uma mudança no status do objetivo. Ouvintes das normas são notificados quando uma nova norma é sensoreada, ativada, desativada, cumprida ou violada.

5.2.4 Pontos Fixos e Flexíveis

Frameworks são geradores de aplicações que estão diretamente relacionados a um domínio específico. Para garantir flexibilidade às aplicações geradas, é necessário estabelecer pontos no framework que permitam a resolução de problemas específicos (Markiewicz e Lucena, 2001).

Os recursos presentes em todos os aplicativos constituem o núcleo do framework e são os pontos fixos do framework, os *frozen-spots*. Os pontos flexíveis que fornecem a capacidade de extensão do framework são chamados de *hot-spots*.

Os *frozen-spots* apresentados pelo NBDI4JADE são:

- mecanismo para criar agentes normativos;
- mecanismo para configurar as capacidades e objetivos do agente;
- ambiente para execução e gerenciamento de agentes estendidos do JADE;
- mecanismo para criar e gerenciar as trocas de mensagens baseado na ACL Language e nas especificações FIPA;

Os *hot-spots* especificamente fornecidos pelo NBDI4JADE estão relacionados com as estratégias de deliberação do agente, levando em consideração como estes vão lidar com as normas e, pode-se listar:

- mecanismo para personalizar a estratégia de atualização das crenças dos agentes;
- mecanismo para personalizar a política de seleção de objetivos do agente, considerando uma postura social ou egoísta para o alcance dos objetivos;
- mecanismo que permite estabelecer a estratégia para seleção dos planos planos aplicáveis ao alcance de um objetivo;

6
Um Método de Teste Baseado em Modelos para Agentes Normativos

Esta seção descreve o método proposto para geração de casos de teste. Os modelos são, originalmente, representados através da linguagem de modelagem ANA-ML (Viana et al., 2016) e especificam os objetivos, planos e crenças dos agentes, além de definir as normas e como estas afetam os agentes.

O método sistematiza os passos de extração das informações dos modelos, a geração dos caminhos de testes, a geração e execução dos casos de teste gerados e a visualização dos resultados.

6.1
Visão Geral do Método

A Figura 6.1 descreve as etapas do método e as elipses azuis representam as etapas onde ocorre o processamento de informações. O método inicia com a *extração de informações* das especificações ANA-ML. Os modelos representam os objetivos, planos, crenças e as relações entre os agentes. Entretanto, não validamos a corretude dos modelos utilizados e, assumimos como pré-requisito que os modelos estão em conformidade com os requisitos da aplicação. Como resultado, a *extração de informações* produz uma estrutura com os objetivos e planos (*Goal-Plan Tree*, ver Figura 6.2) dos agentes. Nesta estrutura, cada nó armazena *metadados* sobre o nó, por exemplo: (i) um nó do tipo plano armazena informações sobre os goals alcançados por este plano, a condição de ativação e contexto do plano, as normas que afetam o plano e como afetam. Além disso, adicionamos manualmente informações sobre os cenários e capacidades relacionados ao nó, e (ii) um nó do tipo goal armazena informações sobre os planos que são processados pelo goal, os cenários e capacidades que o envolvem e as normas que o afetam. A próxima etapa é a *geração dos caminhos de teste* a partir da *Goal-Plan Tree* e dos critérios de cobertura. Um critério de cobertura define os caminhos que devem ser percorridos na *Goal-Plan Tree* e representam, na prática, possíveis decisões do agente. Como resultado, esta etapa produz uma lista de caminhos de testes, usada como entrada para a próxima etapa, a *geração dos casos de teste*, na qual ocorre a geração das pré-condições, a definição do resultado esperado para cada teste, a criação

dos scripts de teste e a geração dos casos de testes propriamente. Finalmente, a etapa de *execução do caso de teste* executa os casos de teste utilizando o N-JAT4BDI. A avaliação dos resultados dos testes compara se os resultados obtidos são aqueles esperados para o teste e, no caso de falha, mensagens de erros são exibidas no console utilizando a infraestrutura de visualização fornecida pelo JUnit.

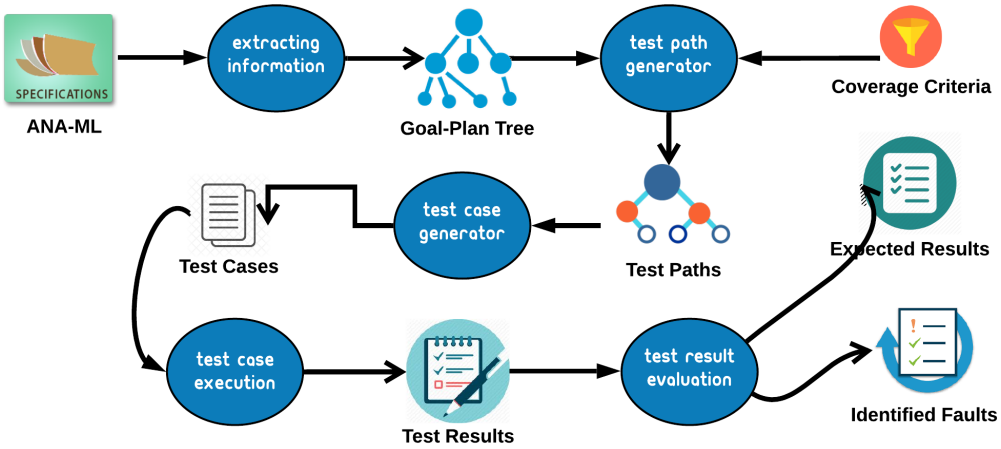


Figura 6.1: Uma Visão Geral do Método de Teste.

6.2 Extração das Informações do Modelo

O Algoritmo 14 operacionaliza os passos para extração das informações dos modelos e para a geração da árvore de objetivos e planos (Goal-Plan Tree – GPT, definida abaixo).

Goal-Plan Tree: De acordo com Winikoff e Cranefield, *goals* e *planos* podem ser visualizados como uma árvore onde cada goal tem como filhos as instâncias de planos que são aplicáveis a ele, e cada instância de plano tem como filhos os subgoals que eles chamam. Cada goal é percebido por uma de suas instâncias de plano e cada instância de plano precisa que todos os seus subgoals sejam alcançados (Winikoff e Cranefield, 2014).

Primeiramente, são extraídos os *goals*, *planos* e *crenças* do diagrama de classe do agente para suas respectivas listas. Em seguida, é criado o nó *root* da GPT que representa o objetivo principal do agente, recuperado das listas criadas pela função *getMainGoal*. São gerados os metadados para o objetivo principal através da função *metadata* e atribuídos ao objetivo principal que, por fim, é adicionado ao nó root da GPT. Cada *goal* da lista é adicionado hierarquicamente à GPT juntamente com seus metadados. Para cada goal adicionado, são recuperados os planos aplicáveis ao goal e gerados os metadados relacionados aos planos e estes, adicionados à GPT.

Algorithm 14: Passos para Extração das Informações do Modelo

Data: Diagrama do Ambiente, representado por *DE*
Data: Diagrama de Organização, representado por *DO*
Data: Diagrama de Classe do Agente Revisor, representado por *DA*
Data: Diagrama de Normas, representado por *norms*
Result: Retorna a *Goal-Plan Tree* (GPT)
// GL = goals list; PL = plans list; BL = belief list
1 Extrai os goals do agente: GL← DA.goals;
2 Extrai os planos: PL← DA.plans;
3 Extrai as crenças do agente: BL← DA.beliefs;
4 Cria o nó ROOT da GPT;
5 mainGoal ← getMainGoal();
6 Adiciona os metadados ao mainGoal: mainGoal ← metadata(mainGoal, GL, PL, BL, norms);
7 Adiciona o *main goal* ao nó ROOT: ROOT ← mainGoal;
8 **foreach** *GOAL g in GL* **do**
9 $g \leftarrow$ metadata(g, GL, PL, BL, norms);
10 GPT $\leftarrow g$;
11 **foreach** *PLAN p in PL.getApplicablePlan(g)* **do**
12 $p \leftarrow$ metadata(p, GL, PL, BL, norms);
13 GPT $\leftarrow p$;
14 **end**
15 **end**
16 return **GPT**;

A Figura 6.2 apresenta a *Goal-Plan Tree* do agente *Reviewer* gerada a partir dos modelos ANA-ML do Expert Committee, onde *G* representa o *main goal* do agente; *P* representa o plano aplicável a *G*; *SG1*, *SG2*, *SG3* e *SG4* são subgoals de *P* e *{P11}*, *{P21, P22, P23}*, *{P31, P32, P33, P34}* e *{P41, P42, P4}* são os planos aplicáveis a cada subgoal, respectivamente.

6.3 Critérios de Cobertura

Um problema essencial nos testes é como cobrir as possibilidades a se testar onde, mesmo programas pequenos, podem ter um número enorme de possibilidades. Não importa se os testes são unitários, de integração ou de sistema, não é viável testar com todas as entradas (Ammann e Offutt, 2016). Assim, um objetivo importante em um projeto de testes é encontrar o menor número de testes capaz de encontrar a maior quantidade de falhas possível.

Critérios de cobertura são regras para ajudar a determinar se uma suíte de testes testou adequadamente um programa (Memon et al., 2001). A partir de uma perspectiva de engenharia, um dos benefícios mais fortes dos critérios de cobertura é que eles fornecem uma "condição de parada" para os testes,

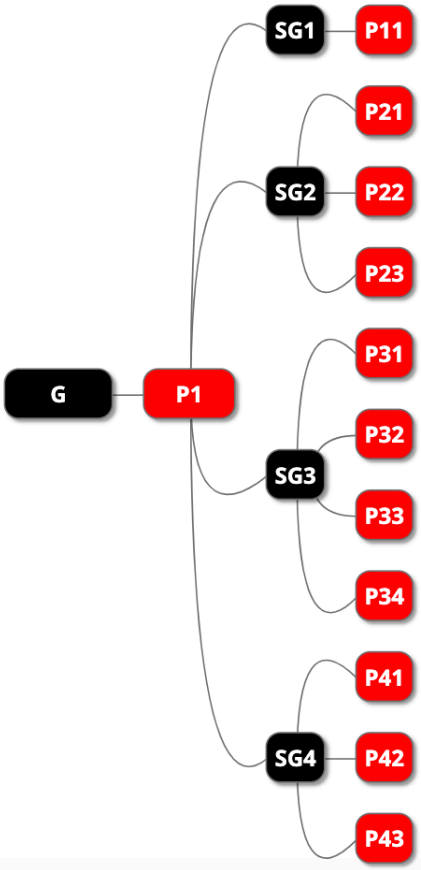


Figura 6.2: A *Goal-Plan Tree* Gerada dos Modelos ANA-ML.

ou seja, é possível saber com antecedência, aproximadamente, quantos testes são necessários. Os critérios de cobertura mais conhecidos lidam com questões estruturais, tais como *statement coverage*, *branch coverage*, *path coverage* e *data flow coverage* e exigem que cada *statement*, *branch*, *path* e *data flow* sejam executados pelo programa no conjunto de testes (Frankl e Weiss, 1993) (Frankl e Weyuker, 1993).

Entretanto, de acordo com Low *et al.*, esses critérios de cobertura não podem ser utilizados diretamente em sistemas multiagentes, pois as noções de *statement*, *branch*, *path* e *data flow* não são equivalentes às de outros paradigmas de programação, como segue: (i) *statement coverage* – é insuficiente em sistemas multiagentes pois um *statement* pode ter êxito ou falhar; (ii) *branch coverage* – os branches são determinísticos em outros paradigmas de programação, enquanto que, em sistemas multiagentes, geralmente não são. Para garantir que um branch específico será executado, os branches alternativos deverão ser levadas em conta ao gerar um caso de teste. Consequentemente, a geração de casos de teste é mais complicada e mais difícil; (iii) *path coverage* – um path define um ponto de início e fim em um programa de programação tradicional. Em sistemas multiagentes, um *path* pode encerrar prematuramente porque as

instruções (planos) podem falhar; (iv) *data flow coverage* – nos paradigmas tradicionais envolve ações executadas sobre os dados. No entanto, os sistemas multiagentes têm seu foco nas capacidades dos agentes. Portanto, *data flow coverage* não é aplicável para sistemas multiagentes (Low et al., 1999).

É possível que, ao testar o sistema, algumas partes permaneçam não testadas, o que pode causar falhas na operação do agente. Assim, para atender as características dos sistemas multiagentes, novos critérios de cobertura foram definidos para garantir a cobertura dos goals e planos, como apresentado abaixo:

Definição de Path: Rehman *et al.* define *path* como um caminho completo que inicia em um nó i e termina em um nó f (Rehman et al., 2016).

- *All Goals Coverage*. Um conjunto de *paths* (P) satisfaz o critério *all goals coverage* para a Goal-Plan Tree (T) se cada nó do tipo goal (G) da árvore (T) está incluído em, pelo menos, um path P $\in T$.
- *Scenario Coverage*. Um conjunto de *paths* (P) satisfaz o critério *scenario coverage* para a Goal-Plan Tree (T) se cada cenário (S) da árvore (T) está incluído em, pelo menos, um path P $\in T$.
- *Capability Coverage*. Um conjunto de *paths* (P) satisfaz o critério *capability coverage* para a Goal-Plan Tree (T) se cada capacidade (C) da árvore (T) está incluído em, pelo menos, um path P $\in T$.
- *All Plans Coverage*. Um conjunto de *paths* (P) satisfaz o critério *all plans coverage* para a Goal-Plan Tree (T) se nó do tipo plano (P) da árvore (T) está incluído em, pelo menos, um path P $\in T$.
- *Goal-Plan Coverage*. Um conjunto de *paths* (P) satisfaz o critério *goal-plan coverage* para a Goal-Plan Tree (T) se cada branch da árvore (T) está incluído em, pelo menos, um path P $\in T$.

6.4 Geração dos Caminhos de Teste

O próximo passo do método é a geração dos caminhos de teste a partir da *Goal-Plan Tree* e dos critérios de cobertura, produzindo como resultado, uma lista de caminhos de teste (*test paths*), como representado na Figura 6.3.

Nosso gerador de caminhos de teste possui duas atividades principais: o *PathReader*, que lê a *Goal-Plan Tree* e separa cada ramo da árvore em listas de ramos e o *PathFinder*, que procura pelos ramos que satisfazem ao critério de cobertura informado. O Algoritmo 15 operacionaliza a geração dos caminhos de teste no método.

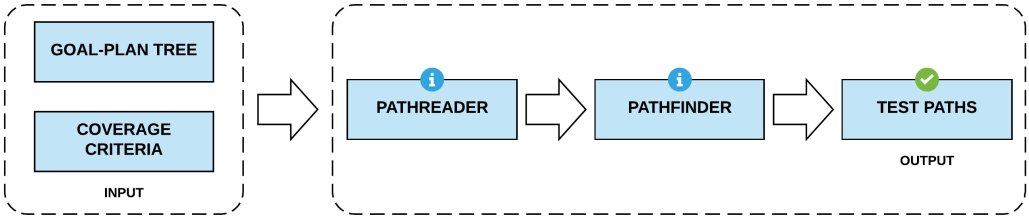


Figura 6.3: Processo para Geração de Caminhos de Teste.

Algorithm 15: Algoritmo para Geração dos Caminhos de Teste

Data: Goal-Plan Tree, representado por *GPT*
Data: Coverage Criteria, representado por *criteria*
Result: Lista de caminhos de teste, representado por *testPaths*

```
1  if criteria = ALL_GOALS then
2  |  paths = findPathAllGoals(GPT);
3  |  foreach path in paths do
4  |  |  testPaths.add(path, ALL_GOALS);
5  |  end
6  end
7  if criteria = SCENARIO then
8  |  paths = findPathScenario(GPT);
9  |  foreach path in paths do
10 |  |  testPaths.add(path, SCENARIO);
11 |  end
12 end
13 if criteria = CAPABILITY then
14 |  paths = findPathCapability(GPT);
15 |  foreach path in paths do
16 |  |  Step 3: testPaths.add(path, CAPABILITY);
17 |  end
18 end
19 if criteria = PLANS then
20 |  paths = findPathPlan(GPT);
21 |  foreach path in paths do
22 |  |  testPaths.add(path);
23 |  end
24 end
25 if criteria = GOAL_PLAN then
26 |  paths = findPathGoalPlan(GPT);
27 |  foreach path in paths do
28 |  |  testPaths.add(path);
29 |  end
30 end
31 return testPaths;
```

Para cada critério de cobertura, é recuperado o conjunto de *paths* com o menor número de ramos. Para o critério ALL_GOALS (linha 1), a

função *findPathAllGoals* (linha 2) recupera os caminhos de teste percorrendo cada nó da *Goal-Plan Tree* e marcando todos os nós do tipo goal como visitados até que todos tenham sido visitados. O caminho que possui um nó marcado é adicionado à lista de caminhos de teste. De forma semelhante, o critério SCENARIO utiliza a função *findPathScenario* (linha 8) que visita cada nó e marca todos os nós que têm esse cenário como parte de seus metadados. O critério CAPABILITY utiliza a função *findPathCapability* (linha 14) que visita cada nó e marca todos os nós que possuem a capacidade como parte de seus metadados. Para o critério ALL_PLANS é utilizada a função *findPathPlan* que visita cada nó e marca todos os nós do tipo plano. Por fim, o critério GOAL_PLAN utiliza a função *findPathGoalPlan* que marca todas as ramificações da árvore.

Goals e *planos* são tipos básicos de nós da árvore e estão diretamente relacionados aos critérios *all goals coverage*, *all plans coverage* e *goal-plan coverage*. Os critérios *scenario coverage* e *capability coverage* são considerados cobertura de metadados, e depende dos metadados de cada nó. A Figura 6.4 apresenta um exemplo dos caminhos de testes gerados a partir da *Goal-Plan Tree* para os critérios *All Goals* e *Capability*.

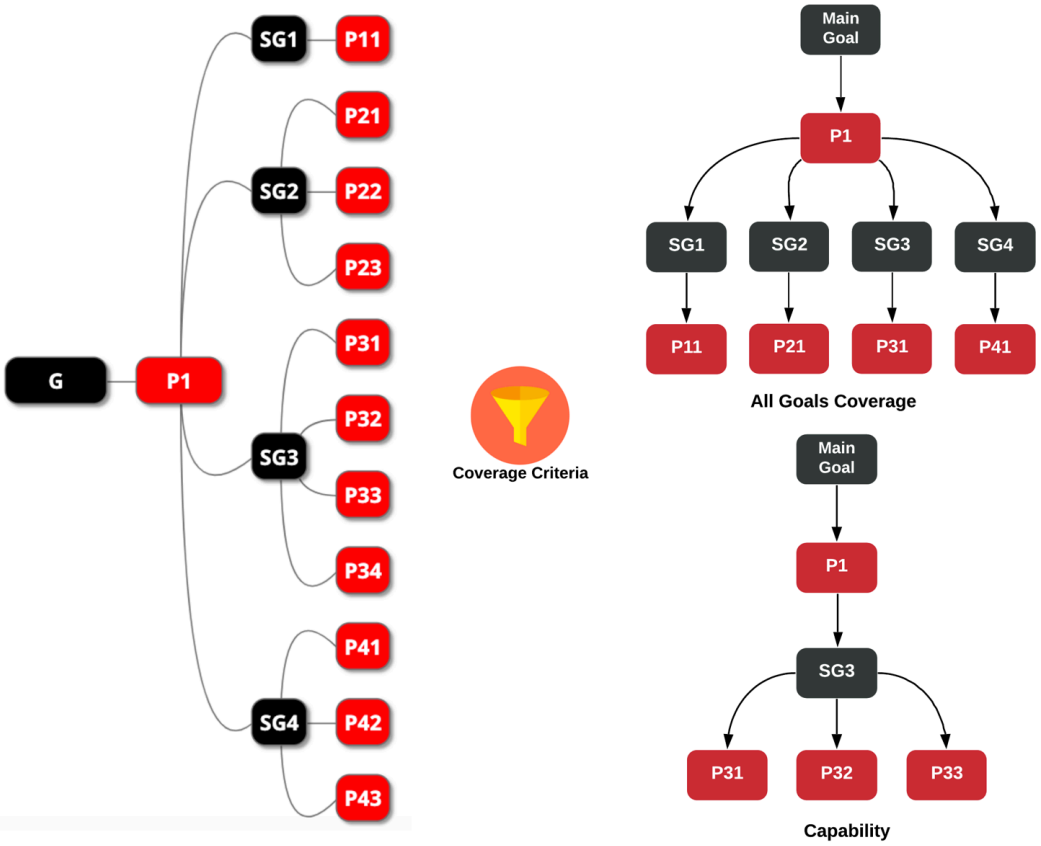


Figura 6.4: Caminhos de Teste a partir da *Goal-Plan Tree*.

A tabela 6.1 apresenta os caminhos de teste gerados pelo Algoritmo15

para o agente *Reviewer* do cenário Expert Committee.

6.5 Geração dos Casos de Teste

A próxima etapa do método é a geração dos casos de teste que é dividida em duas tarefas: (i) a geração dos scritps de teste, e (ii) a geração dos casos de teste propriamente ditos. A tarefa de gerar os scritps de teste é operacionalizada pelo Algoritmo 16.

Algorithm 16: Algoritmo de Geração dos Scripts de Teste.

```
   Data: Lista de caminhos de teste, representados por testpaths
   Data: Nome do agente em teste, representados por agent
   Data: Norma sendo verificada, representados por norm
   Data: Critério de cobertura, representado por coveragecriterion
   Result: Os scripts de teste são gerados
 1 filteredPaths = filterByCriterion(testpaths, coveragecriterion);
 2 List<TestCaseStructure> testCases = new ArrayList();
 3 foreach typeOfFault do
 4     if typeOfFault != STATE then
 5         result = null;
 6         testCases.add(createTestCaseStructure(agent, norm,
           typeOfFault, result));
 7     else
 8         foreach path in filteredPaths do
 9             if path.action = message then
10                 result = configureMessage();
11             end
12             if path.action = plan then
13                 result = configurePlan();
14             end
15             inputData = configureInputData(path);
16             testCases.add(createTestCaseStructure(agent, norm,
               typeOfFault, inputData, result));
17         end
18     end
19     foreach testCase in testCases do
20         generateTestFile(testCase);
21     end
22 end
```

Assim, para cada norma endereçada ao agente, é gerado um conjunto de scripts. Ao gerar o script são verificados os tipos de falhas e, para falhas dos tipos *endereçamento*, *condição*, *motivação* e *conceito deôntico* (linha 4), utilizamos a função *createTestCaseStructure* que cria uma estrutura de dados para representar um caso de teste para essas falhas. Esta função permite con-

Tabela 6.1: Test Paths por Critério de Cobertura

#	Critério	Test Paths
1	All goals	ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → PlagiarismCheckerGoal (SG_1) → PlagiarismPlan (P_{11})
1		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → PresentationQualityGoal (SG_2) → FormatPlan (P_{21})
1		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ArtifactEvaluationGoal (SG_3) → DocumentationPlan (P_{31})
1		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ContentAnalysisGoal (SG_4) → SoundnessPlan (P_{41})
2	Scenario	ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → PlagiarismCheckerGoal (SG_1) → PlagiarismPlan (P_{11})
2		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ArtifactEvaluationGoal (SG_3) → DocumentationPlan (P_{33})
3	Capability	ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ArtifactEvaluationGoal (SG_3) → ConsistencyPlan (P_{31})
3		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ArtifactEvaluationGoal (SG_3) → CompletenessPlan (P_{32})
3		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ArtifactEvaluationGoal (SG_3) → DocumentationPlan (P_{33})
3		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ArtifactEvaluationGoal (SG_3) → ReusePlan (P_{34})
4	Plan	ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → PlagiarismCheckerGoal (SG_1) → PlagiarismPlan (P_{11})
4		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → PresentationQualityGoal (SG_2) → FormatPlan (P_{21})
4		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → PresentationQualityGoal (SG_2) → LanguagePlan(P_{22})
4		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → PresentationQualityGoal (SG_2) → FigureTablePlan(P_{23})
4		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ArtifactEvaluationGoal (SG_3) → ConsistencyPlan (P_{31})
4		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ArtifactEvaluationGoal (SG_3) → CompletenessPlan (P_{32})
4		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ArtifactEvaluationGoal (SG_3) → DocumentationPlan (P_{33})
4		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ArtifactEvaluationGoal (SG_3) → ReusePlan (P_{34})
4		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ContentAnalysisGoal (SG_4) → SoundnessPlan (P_{41})
4		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ContentAnalysisGoal (SG_4) → SignificancePlan (P_{42})
4		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ContentAnalysisGoal (SG_4) → VerifiabilityPlan (P_{43})
5	Goal-Plan	ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → PlagiarismCheckerGoal (SG_1) → PlagiarismPlan (P_{11})
5		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → PresentationQualityGoal (SG_2) → FormatPlan (P_{21})
5		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → PresentationQualityGoal (SG_2) → LanguagePlan(P_{22})
5		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → PresentationQualityGoal (SG_2) → FigureTablePlan(P_{23})
5		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ArtifactEvaluationGoal (SG_3) → ConsistencyPlan (P_{31})
5		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ArtifactEvaluationGoal (SG_3) → CompletenessPlan (P_{32})
5		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ArtifactEvaluationGoal (SG_3) → DocumentationPlan (P_{33})
5		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ArtifactEvaluationGoal (SG_3) → ReusePlan (P_{34})
5		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ContentAnalysisGoal (SG_4) → SoundnessPlan (P_{41})
5		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ContentAnalysisGoal (SG_4) → SignificancePlan (P_{42})
5		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ContentAnalysisGoal (SG_4) → VerifiabilityPlan (P_{43})
5		ReviewPaperGoal (G) → ReviewPaperPlan (P_1) → ContentAnalysisGoal (SG_4) → ExpertisePlan (P_{44})

figurar o nome do teste, a norma do ambiente, o tipo de método verificador (assertiva de verificação) que deve ser utilizado e o agente que será testado, como apresentado na Figura 6.5. Além disso, para tais falhas, não é necessário a utilização dos caminhos de teste. Para falhas do tipo *estado interno* (linha 7), são selecionados os caminhos de teste de acordo com o critério de cobertura desejado. Para cada caminho de teste, são extraídos dos metadados as informações utilizadas para configurar a entrada de dados do teste (linha 15) e o resultado esperado (linha 10 ou 13, dependendo se a norma regula um plano ou mensagem). A estrutura do caso de teste é, então, adicionada a uma lista de scripts de teste (linha 16). As informações utilizadas como entrada de dados dos testes são as crenças e as condições de contexto dos planos contidos no caminho de teste.

A segunda tarefa da geração dos casos de teste é a geração dos casos executáveis a partir dos scritps de teste. Essa tarefa é operacionalizada pela função *generateTestFile* (linhas 19-21).

```
79    private void createTestCaseStructure(String agentUnderTest,
80            String norm, String typeOfFault,
81            ExpectedResult expectedResult) {
82
83        TestCaseStructure testCase = new TestCaseStructure();
84        testCase.setAgentUnderTest(agentUnderTest);
85        testCase.setNorm(norm);
86        testCase.setAssertive(getAssertive(typeOfFault));
87        testCase.setNameTestCase("test" + this.norm +
88                getTypeOfFaultShortName(typeOfFault));
89
90        testCases.add(testCase);
91
92    }
```

Figura 6.5: Criação da Estrutura de um Caso de Teste.

O Algoritmo 17 operacionaliza a geração de casos de teste executáveis (função *generateTestFile*, linha 20). O gerador de casos de teste recebe o script de teste, gera o nome da classe que será o nome da classe do caso de teste (linha 1). As linhas 2-4 compilam o script de teste em um caso de teste executável.

A Figura 6.6 ilustra os passos descritos acima para a etapa de geração de casos de teste.

6.6 Execução dos Casos de Teste

Esta etapa do método é responsável pela execução dos casos de teste gerados na etapa anterior. Após serem gerados, os casos de teste são manualmente executados. Durante a criação dos scripts de teste, são gerados os dados

Algorithm 17: Algoritmo de Geração de Casos de Teste.

Data: Conjunto de scripts de teste, representados por *scripts*
Result: Os casos de teste são gerados

```
1 className = script.getClassName();
2 Class aClass =
  CompilerUtils.CACHED_COMPILER.loadFromJava(className,
  script);
3 Runnable runner = (Runnable) aClass.newInstance();
4 runner.run();
```

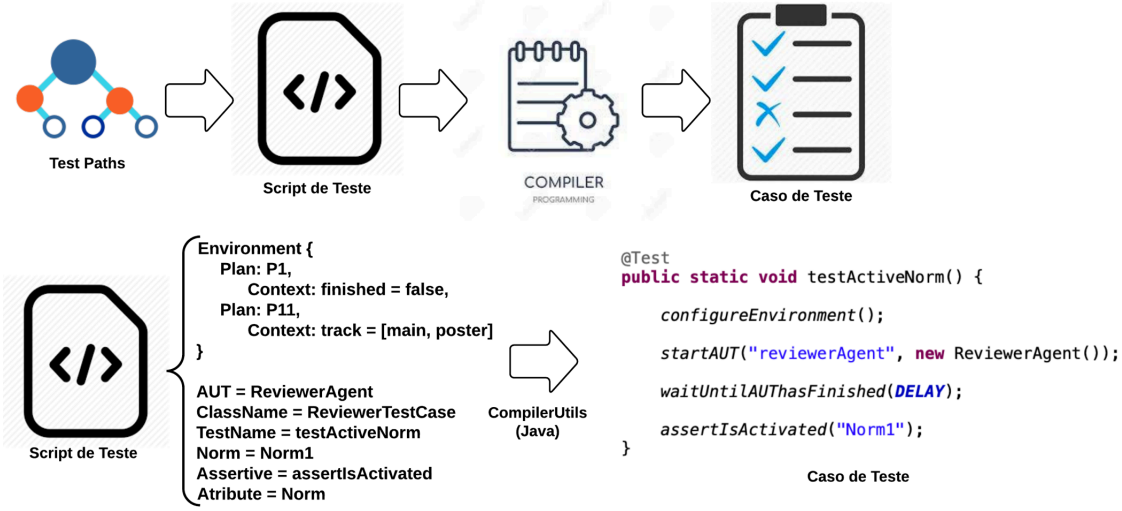


Figura 6.6: Passos da Etapa de Geração de Casos de Teste.

de entrada do teste a partir dos metadados dos nós dos caminhos de teste. Informações sobre as crenças e a condição de contexto dos planos de são encapsulados em uma estrutura de dados e usadas na configuração dos dados de entrada de cada caso de teste. Tal configuração é feita injetando-se os metadados na base de conhecimento do agente. Dessa forma, é possível *induzir* o agente nas decisões tomadas durante a execução do caso de teste para seguir o caminho de teste selecionado. Novamente utilizamos a programação orientada a aspectos na configuração dos dados de entrada dos testes. A Figura 6.7 apresenta o código parcial do injetor de dados de entrada.

```
6  public aspect InputDataInjector {
7
8      private DataInput dataInput = DataInput.getInstance();
9
10     pointcut addBelief(Belief<?> belief) :
11           execution(void BeliefBase.addBelief(..)) && args(belief);
12 }
```

Figura 6.7: Injetor de Dados de Entrada dos Casos de Teste.

6.7 Resultado da Execução dos Casos de Teste

Por fim, a última etapa do método é a avaliação dos resultados obtidos a partir da execução dos casos de teste. A *avaliação dos testes* é o processo de analisar os resultados obtidos dos testes e reportá-lo aos desenvolvedores. Apesar de, aparentemente simples, avaliar os resultados requer conhecimentos sobre o domínio, sobre os cenários testados, sobre a cobertura dos testes, sobre os resultados obtidos, etc. Se os testes forem automatizados, a avaliação poderá (e deverá) ser incorporada nos scripts de teste. No entanto, quando a automatização está incompleta ou quando a saída correta não pode ser codificada nas asserções, essa tarefa fica mais complicada (Rehman et al., 2016).

Como dito, o N-JAT4BDI utiliza a visualização fornecida pelo framework JUnit e, dessa forma, a avaliação consiste em analisar os casos de teste que "passaram" e aqueles que "quebraram" durante a execução. A Figura 6.8 apresenta um exemplo do resultado da execução da suíte de teste que verifica o impacto da norma *Norm 1* sobre o comportamento do agente *Reviwer*.

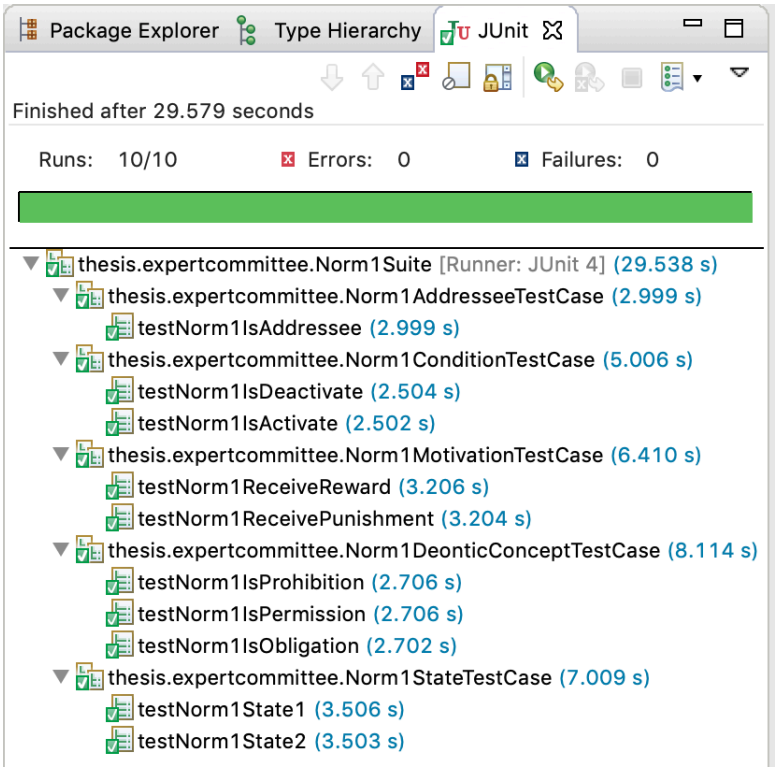


Figura 6.8: Resultado da Execução dos Casos de Teste.

7
Cenários de Uso

Este capítulo apresenta os cenários usados na demonstração da aplicabilidade da solução proposta. O primeiro é apresentado na Seção 7.2 e descreve uma aplicação para apoiar o gerenciamento de submissões e o processo de revisão de artigos em conferências (Zambonelli et al., 2000). O segundo é apresentado na Seção 7.3 e descreve uma aplicação no domínio de mercados virtuais para uma agência de viagens e eventos turísticos (Halatsis et al., 1994).

7.1
Metodologia Aplicada

Esta seção descreve como os cenários são modelados, ajudando o desenvolvedor a trabalhar de forma conjunta com ANA-ML e NBDI4JADE. São descritas as especificações necessárias para representar um projeto ANA-ML e detalhes dos diagramas usados no método de geração de casos de testes.

7.1.1
Diagramas Estruturais

O uso de ANA-ML começa com a identificação das classes de *ambiente* e da *organização principal*. Descrever a classe da organização principal engloba definir seus objetivos, crenças, planos e normas. A classe de ambiente deve ser modelada como um agente, com seus objetivos, crenças, planos e ações.

Após a identificação das classes de ambiente e da organização principal, são identificados os agentes e os papéis exercidos por eles na organização principal. No cenário do Expert Committee, cada agente desempenha um papel único e bem definido, que sofre a influência das normas da organização. Nem todas as classes são modeladas em diagramas de papel e organização. As classes que não exercem papéis e nem estão relacionadas a papéis devem ser modeladas em diagramas de classes. Todas as classes modeladas no diagrama de organização estão relacionadas à mesma classe de ambiente e as entidades que residem em outro ambiente não podem exercer papéis nessa organização.

7.1.2 Diagramas Dinâmicos

São utilizados diagramas de atividades para modelar os fluxos de execução internas ao agente. Este diagrama modela a ordem de execução do raciocínio do agente com relação às normas e as possíveis ações a serem realizadas entre (i) agentes que estão exercendo papéis e monitorando as normas do ambiente, (ii) normas ativas no ambiente endereçadas aos agentes, (iii) tomadas de decisão do agente para raciocinar sobre uma norma no ambiente. Além disso, em ANA-ML, um diagrama de atividades também modela os planos dos agentes, organizações e ambientes.

7.2 Cenário 1: Expert Committee System

O *Expert Committee* (EC) é um sistema multiagente que auxilia o gerenciamento de submissões e o processo de revisão dos artigos em uma conferência e foi escolhido por ser um *benchmark* da literatura de agentes (Zambonelli et al., 2000), além de permitir explorar apropriadamente o uso de normas em um contexto prático.

O sistema define quatro tipos de agentes onde cada agente fornece diferentes serviços e participam do sistema de maneira interativa, autônoma e proativa, desempenhando uma das seguintes funções:

- **Author**: representa o autor do artigo no EC, encaminhando o artigo submetido para um evento específico;
- **Chair**: distribui os artigos entre os revisores, organiza as apresentações e sessões, solicita ao *Coordinator* novos revisores quando não há revisores suficientes, notifica os autores da aceitação ou não dos artigos e gerencia as premiações;
- **Reviewer**: avalia as propostas do *Chair*. Os revisores podem aceitar ou rejeitar uma proposta de revisão. Depois de aceitarem, eles devem seguir as diretrizes da conferência e reportar suas revisões e avaliações dos artigos ao *Chair* antes do prazo final;
- **Coordinator**: auxilia o *Chair*, convidando revisores adicionais para auxiliar no processo de revisão de artigos em situações específicas;

As diretrizes da conferência são representadas no EC através de normas, trazendo maior flexibilidade ao sistema. Como cada conferência tem seu próprio conjunto de diretrizes, espera-se que o comportamento dos agentes varie entre diferentes conferências. Para ilustrar, considere uma *Conferência A*

que observa rígidos critérios de formatação do texto como um dos critérios de aceitação dos artigos submetidos, enquanto que uma *Conferência B* apenas recomenda o uso de estilos básicos de formatação. Ao estabelecer diferentes normas relacionadas a formatação do texto, espera-se que os revisores atuem, para cada conferência, em conformidade com a norma de formatação apropriada ao avaliar esse critério dos artigos submetidos.

Neste cenário de uso, exploramos normas que afetam os comportamentos dos agentes revisores. A *importância* de realizar um comportamento é avaliada levando-se em consideração, também, a *motivação* do agente em atingir um objetivo, ou sua *satisfação* em realizar uma ação.

A Figura 7.1 ilustra as principais interações no ambiente do sistema *Expert Committee*, os repositórios de informações acessados por cada agente e exemplos de normas, representando diretrizes de diferentes conferências.

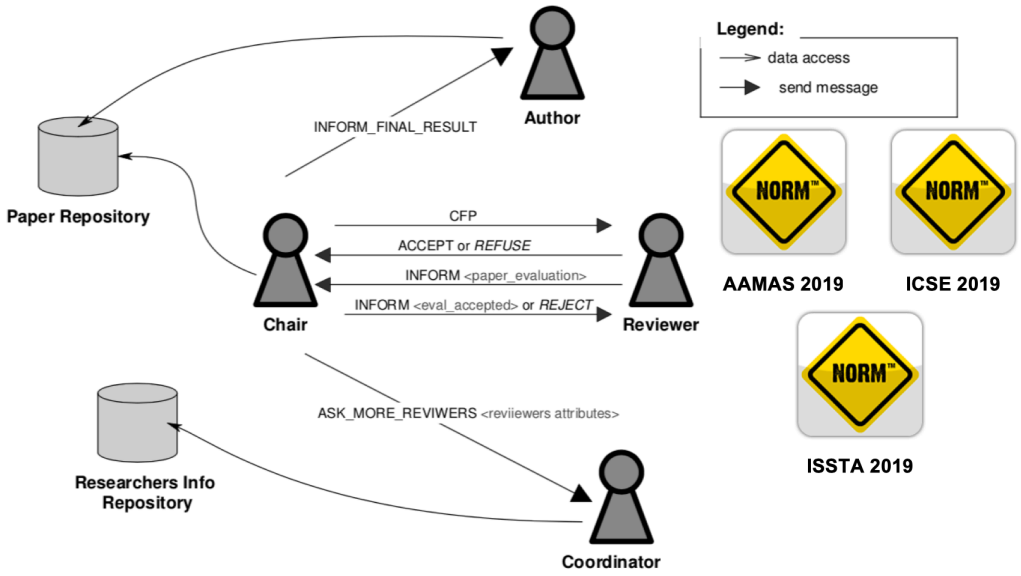


Figura 7.1: Interações entre os agentes no sistema Expert Committee.

Normas

As normas endereçadas aos agentes revisores são definidas da seguinte forma:

Norma 1: A verificação de plágio do artigo é crucial na tarefa de revisão. Se um artigo é considerado plágio, então tal artigo deve ser rejeitado por seu revisor.

Recompensa: Aumento da reputação do revisor.

Punição 1: Diminuição da reputação do revisor.

Punição 2: Diminuição da reputação da conferência.

Norma 2: O revisor deve rejeitar os artigos que não seguem todos os padrões de qualidade.

Recompensa: Aumento da reputação do revisor.

Punição 1: Diminuição da reputação do revisor.

Punição 2: O revisor não recebe mais artigos para revisão.

Norma 3: Artigos da trilha de *avaliação de artefatos* devem ser aceitos se seus artefatos são avaliados e classificados como reutilizáveis.

Recompensa 1: Premiação do artigo associado.

Recompensa 2: Aumento da reputação do revisor.

Punição 1: Reputação do revisor é diminuída.

Punição 2: Revisor não realiza mais avaliação de artefatos.

7.2.1 Modelagem a partir de ANA-ML

Nesta seção são apresentados os diagramas ANA-ML.

Diagramas Estruturais

A Figura 7.2 apresenta o diagrama da *organização principal* usando uma representação simplificada (isto é, omitindo seus atributos e operações), mas suficiente para o entendimento das relações existentes no cenário abordado.

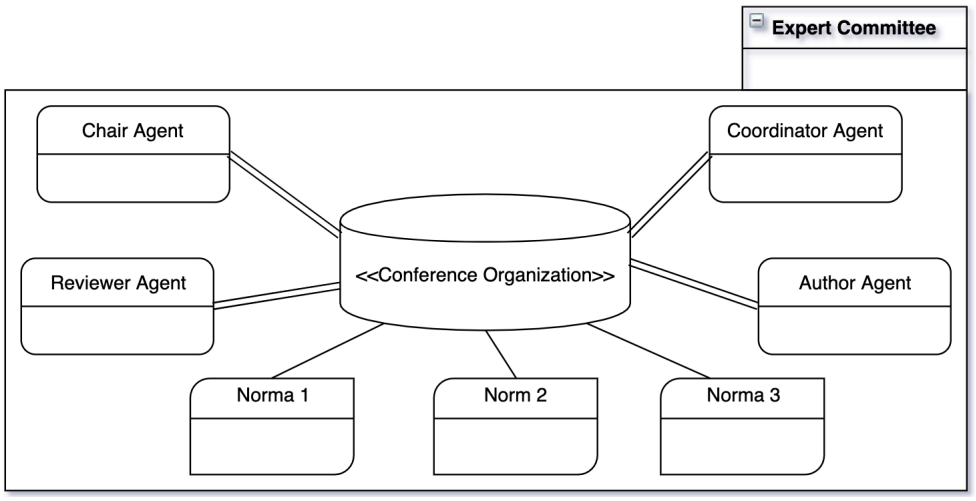


Figura 7.2: Diagrama de Organização da Conferência.

O diagrama apresenta a classe de ambiente *Expert Committee*. Além disso, para este cenário foram modelados quatro tipos de agentes: *Author*,

Reviewer, *Coordinator* e *Chair*. Foram definidas, também, as normas ***Norma 1***, ***Norma 2*** e ***Norma 3***, endereçadas ao agente revisor.

Descrição Parcial da Organização e Ambiente

Ao analisar a descrição do cenário de uso, identificou-se a organização principal *Conference Organization*, localizada no ambiente *Expert Committee*. O ambiente é modelado como um ambiente ativo, onde informações são adicionadas ao longo do tempo para que atributos relacionados às submissões e grupo de revisores possam ser introduzidas ou atualizadas no sistema, como pode ser visto parcialmente na Figura 7.3. Esta é uma visualização parcial e simplificada do diagrama, não modelando aspectos relacionados ao gerenciamento das normas com o ambiente. Fizemos assim, pois estamos interessados apenas nos aspectos normativos que influenciam o comportamento dos agentes.

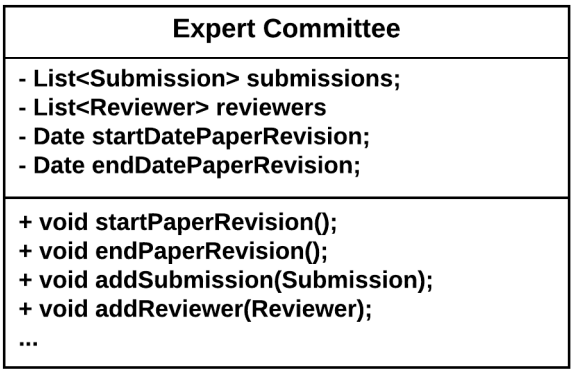


Figura 7.3: Representação Parcial da Classe do Ambiente.

A organização principal *Conference Organization* (parcialmente ilustrada na Figura 7.4), representa a conferência. Todos as informações sobre as atividades, sessões, eventos e prazos de uma conferência são modelados na organização principal do sistema, sendo que apenas uma instância dessa classe pode ser criada por ambiente e não exercendo nenhum papel.

O objetivo da organização principal é representar o funcionamento da conferência e todas as atividades adjacentes realizadas, tais como: o gerenciamento do orçamento, o gerenciamento das premiações dos participantes, distribuição de artigos aos revisores (motivação máxima), seleção dos palestrantes, a organização dos eventos, workshops e sessões e a revisão dos artigos submetidos (motivação máxima). Para alcançá-lo, a organização principal define normas específicas para cada participante (autores, revisores e coordenadores). As crenças da organização principal representam as informações compartilha-

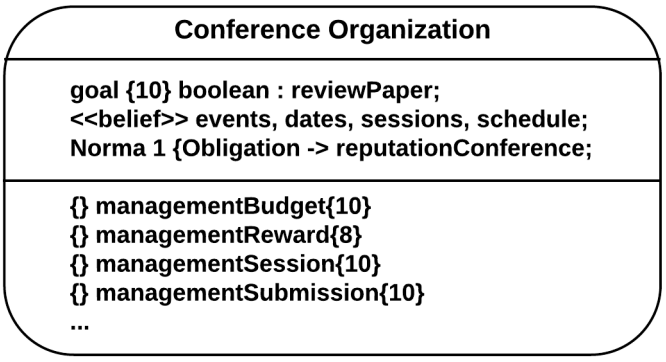


Figura 7.4: Representação Parcial da Classe da Organização Principal.

das pelas entidades do sistema e são relativas ao controle do orçamento, à organização dos eventos e seus locais, prazos, submissões, informações sobre revisores e coordenadores, diretrizes e reputação da conferência.

Modelando Normas

A seguir, a Figura 7.5 apresenta as normas ativas modeladas para o cenário apresentado. Como pode ser observado, *Norma 1* está endereçada aos agentes revisores; sua ativação ocorre ao iniciar a conferência e expira no término da conferência; é definido como recompensa para o agente que cumpre com a norma, o aumento de sua reputação e, como punição, a diminuição da reputação do revisor, a diminuição da reputação da conferência e o revisor ainda é penalizado não recebendo mais artigos para revisar; a norma define uma obrigação para a execução do plano de verificação de plágio nos artigos.

Norma 2 está endereça aos agentes revisores; sua ativação ocorre no início da conferência e termina ao fim da conferência; se o agente cumpre com a norma, sua recompensa é o aumento de sua reputação, caso contrário, recebe uma punição da diminuição da reputação; a norma estabelece uma obrigação para execução do plano que checa a formatação dos artigos submetidos.

Norma 3 também está endereçada aos agentes revisores; sua ativação se dá no início da conferência e sua desativação no final da conferência; ao cumprir com a norma, o agente recebe como recompensa, o aumento de sua reputação e, caso não cumpra, a diminuição de sua reputação e não recebe mais artefatos para avaliar; a norma estabelece uma obrigação de execução do plano que verifica o nível de reúso do artefato.

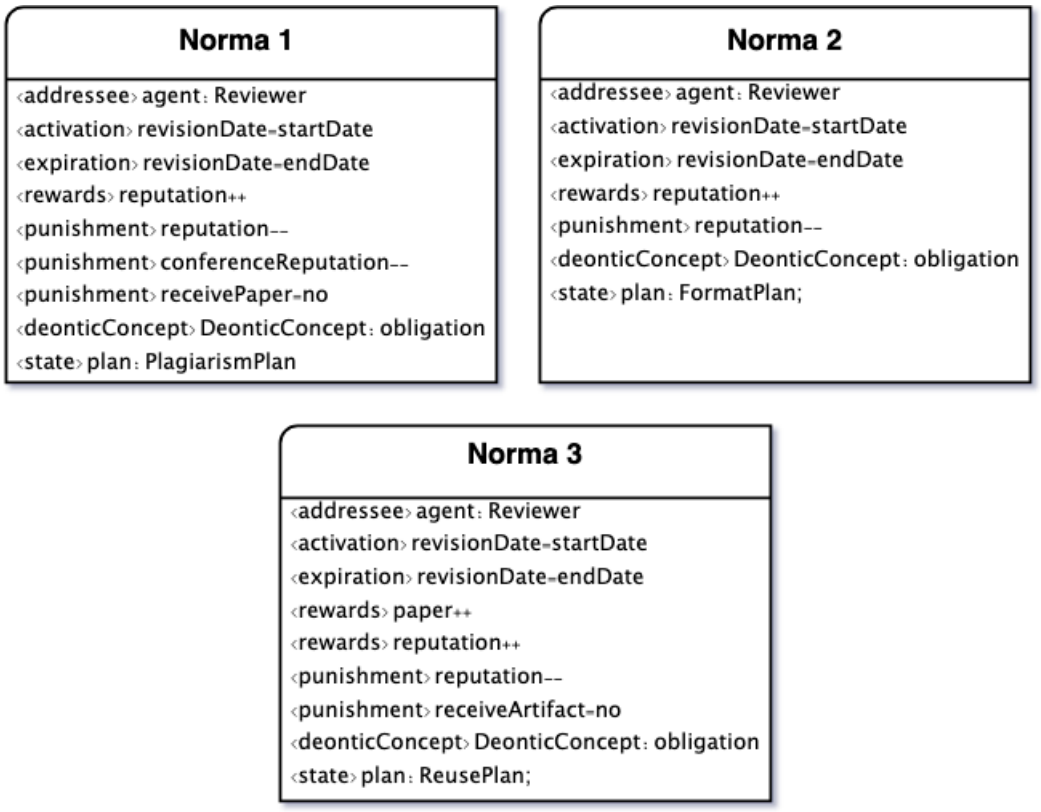


Figura 7.5: Normas Ativas para o Expert Committee.

Modelando Agentes

A Figura 7.6 ilustra a classe *Reviewer Agent* descrevendo (i) os objetivos "*reviewPaper*" e "*increasedReputation*", (ii) suas crenças de como lidar com as informações dos artigos e sua reputação, (iii) o desejo de aumentar sua reputação e revisar artigos e (iv) sua lista de intenções.

A parte inferior contêm as normas sensoreadas pelo agente e a lista de planos que o agente executa para alcançar seu objetivo e subobjetivos de revisar os artigos.

7.3 Cenário 2: Sistema Multiagente para Agência de Turismo

Este cenário de uso apresenta o *TourAgent*, um sistema multiagente para consultoria turística cujo objetivo é facilitar o trabalho realizado nas agências de viagens, possibilitando a seleção de passeios turísticos personalizados e a venda de pacotes turísticos pré-definidos, lidando com informações turísticas subjacentes. Além disso, o sistema segue um conjunto de normas visando entregar a melhor experiência a seus clientes, satisfazendo suas necessidades e

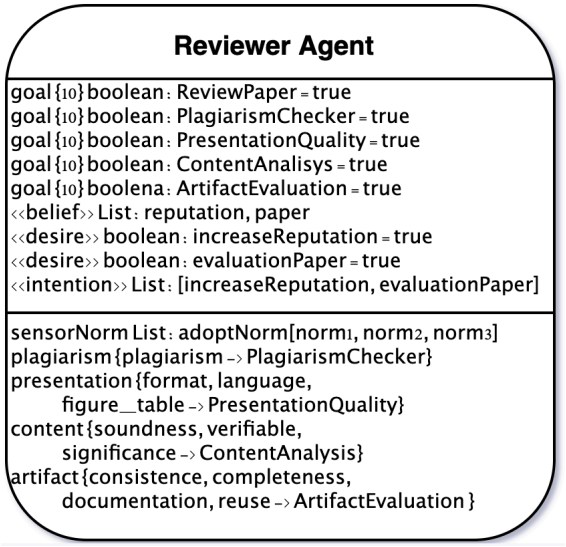


Figura 7.6: A Classe *Reviewer Agent* (parcial).

garantindo sua segurança.

O *TourAgent* é formado por dois tipos de agentes que refletem os procedimentos envolvidos no ambiente de uma agência de turismo, onde cada agente possui um papel bem definido, um objetivo específico e o conhecimento para alcançá-lo, de forma a aumentar a reputação da agência. Os agentes para esta aplicação são: (i) *employee*, agente que representa um funcionário da agência, com as competências para elaborar os pacotes de viagens, realizar cotações, vender pacotes e eventos e gerenciar toda a parte administrativa da agência, e (ii) *user*, agente que representa o cliente que interage com o funcionário da agência, solicitando um serviço ou informação. Para especializar as atividades da agência, o agente *employee* pode ser encontrado desempenhando dois papéis: *seller*, responsável pelo atendimento, negociação e venda de pacotes turísticos e eventos, e *manager*, responsável por lidar com as questões administrativas.

Ao solicitar um serviço, o cliente fornece suas informações pessoais, e informações sobre o tipo de pacote ou evento que deseja. Tais informações são utilizadas pelo funcionário (*seller*) que verifica se há restrições etárias ou de transporte envolvidas. Dessa forma, o agente *seller* procura as melhores alternativas, priorizando o preço mais baixo. Ao final do atendimento, o cliente informa uma nota referente ao atendimento recebido pela agência, contribuindo para o aumento ou diminuição da reputação da agência nesse segmento de trabalho.

Normas

Para este cenário, serão consideradas, para fins de estudo, normas que regulam o comportamento do agente *seller*, definidas da seguinte forma:

Norma 1: Se um evento possui restrições etárias, tal evento não pode ser oferecido a um cliente que não atenda a tais restrições.

Punição 1: A reputação da agência é diminuída.

Norma 2: Se um passeio envolve a utilização de transporte, então o transporte deve prover lugares apropriados para todos os participantes.

Punição 1: A reputação da agência é diminuída.

Punição 2: O passeio é cancelado.

Norma 3: O preço do pacote turístico só pode ser reajustado se não estiver em negociação com um cliente.

Punição 1: A reputação da agência é diminuída.

Punição 2: O cliente cancela a negociação.

7.3.1 Modelagem a partir de ANA-ML

Nesta seção são apresentados os diagramas ANA-ML criados de acordo com as informações definidas anteriormente. São apresentados os diagramas de classes estáticos definidos para o sistema.

Diagramas Estruturais

O diagrama da organização é apresentado na Figura 7.7 e ilustra a representação simplificada da organização principal. Foram representadas as classes da organização principal, do ambiente, dos agentes, os papéis e as normas.

No diagrama é apresentada a classe da organização principal *Agency Organization* e dois tipos de agentes: *employee* e *user*. Além disso, o diagrama ilustra os papéis *manager* e *seller* associados ao agente *employee* e as normas *Norm 1*, *Norm 2* e *Norm 3*.

Descrição Parcial da Organização e Ambiente

De acordo com a descrição do cenário, é possível identificar a organização principal *Agency Organization* localizada no ambiente *Tourism Environment*.

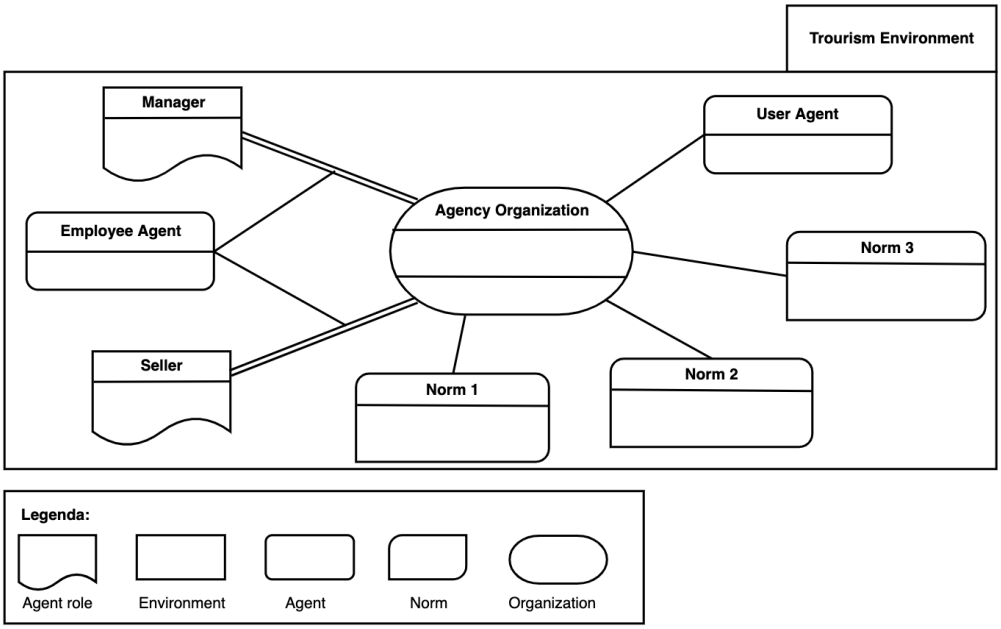


Figura 7.7: Diagrama da Organização Principal.

O ambiente é modelado como um ambiente ativo, onde informações são adicionadas ao longo do tempo para que atributos relacionados aos pacotes turísticos, funcionários e clientes possam ser introduzidas ou atualizadas no sistema. A classe *Tourism Environment* está ilustrada na Figura 7.8. Foram implementados os métodos (i) *addPercept* para inserir novas características do ambiente, (ii) *executeAction* para quais as ações o ambiente irá executar e (iii) *stop* para finalizar o sistema.

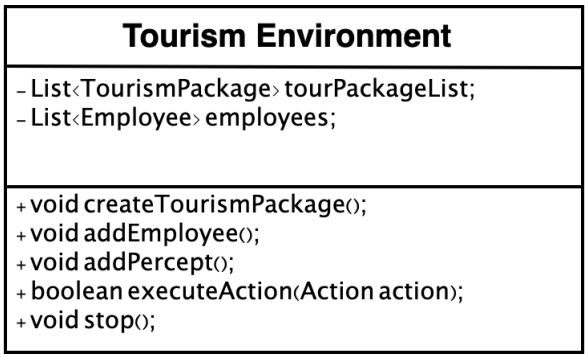


Figura 7.8: Classe do Ambiente do Tourism Environment.

A classe da organização principal *Agency Organization* é ilustrada parcialmente na Figura 7.9 e não exerce nenhum papel. Apenas uma instância dessa classe pode ser criada por ambiente. A fim de atender aos clientes que desejam solicitar informações e comprar pacotes turísticos e lidar com as informações administrativas da agência, a organização principal define os papéis *seller* e

manager do agente *employee*.

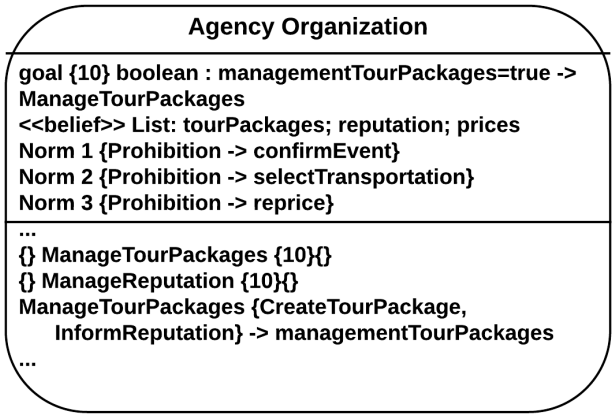


Figura 7.9: Classe da Organização Principal (parcial).

Os objetivos da organização principal são a venda serviços turísticos (pacotes e eventos) e o aumento da reputação da agência. Para alcançá-los, a organização principal define planos para: (i) criar novos pacotes turísticos; (ii) atualizar o ambiente para informar dos novos pacotes criados ou aqueles que não existem mais, e (iii) avaliar a reputação da agência. As crenças da organização principal estão relacionadas as informações relativas aos pacotes turísticos, os preços estabelecidos para os pacotes e passeios e a reputação da agência.

Descrição das Normas

A Figura 7.10 apresenta as normas definidas para o cenário TourAgent. **Norm 1** afeta o ambiente da seguinte forma: (i) não permite a seleção de eventos que possuem restrições etárias à clientes que não atendam as restrições; (ii) possui conceito deôntico de *proibição*; (iii) é endereçada ao agente *seller*; (iv) está ativa quando o agente *seller* inicia a negociação com o agente *user*; (v) expira quando a negociação se encerra e, (vi) sua punição é ter a reputação da agência diminuída.

Norm 2 afeta o ambiente da seguinte forma: (i) obriga o uso de transporte apropriado para eventos e pacotes turísticos, caso necessitem; (ii) possui conceito deôntico *obrigatório*; (iii) é endereçada ao agente *seller*; (iv) está ativa quando o agente *seller* inicia a negociação com o agente *user*; (v) expira quando a negociação se encerra e, (vi) suas punições são ter a reputação da agência diminuída e o passeio cancelado.

Norm 3 afeta o ambiente da seguinte forma: (i) a agência não pode reajustar o preço de um pacote turístico que está em negociação; (ii) possui

conceito deôntico *proibição*; (iii) é endereçada ao agente *seller* e *manager*; (iv) está ativa quando o agente *seller* inicia a negociação com o agente *user*; (v) expira quando a negociação se encerra e, (vi) suas punições são ter a reputação da agência diminuída e o passeio cancelado.

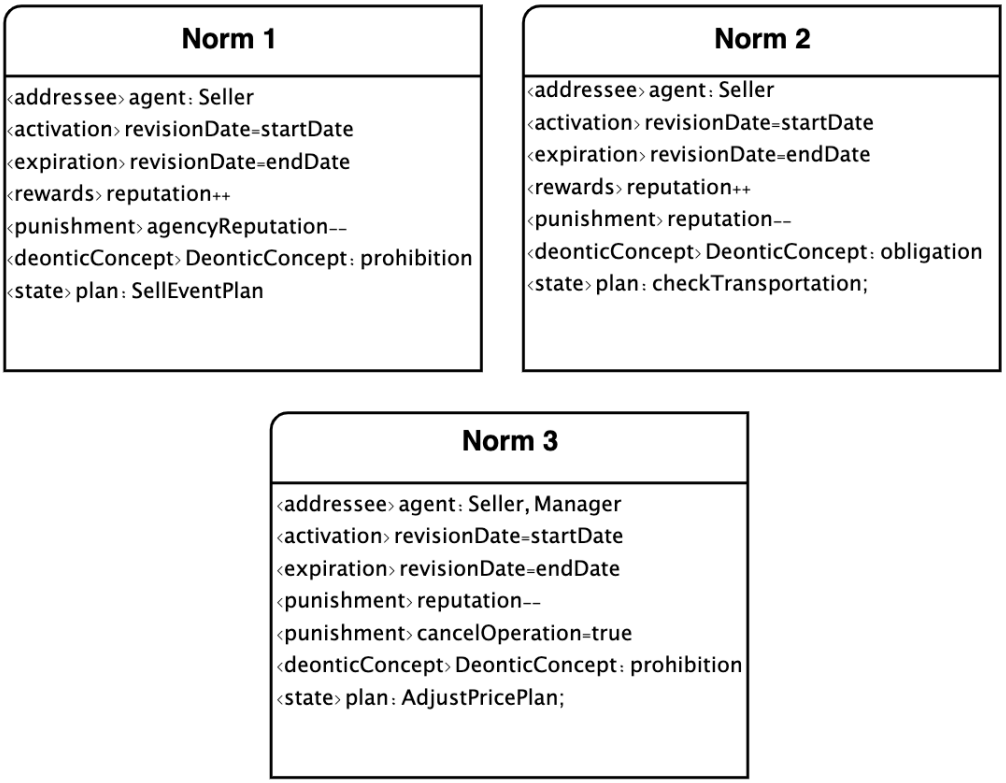


Figura 7.10: Normas Vigentes no Sistema de Turismo.

Modelando Agentes

A Figura 7.11 ilustra a classe do papel *Seller* descrevendo: (i) os objetivos *selectPackage* e *selectEvent*; (ii) suas crenças de com lidar com as informações sobre os pacotes e solicitações dos clientes; (iii) o desejo de aumentar a reputação da agência, e (iv): sua lista de intenções a serem realizadas.

A parte inferior do diagrama contém a lista de normas sensoreadas pelo agente e a lista de planos que o agente executa para alcançar seus objetivos e subobjetivos.

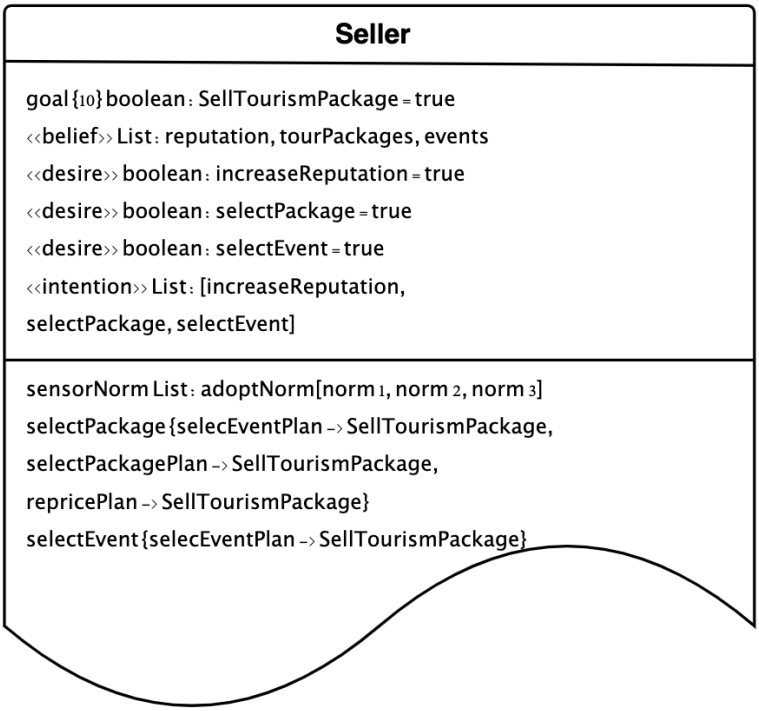


Figura 7.11: A Classe do Papel Seller (parcial).

8
Avaliações: Análises, Resultados e Discussões

De acordo com (Hsueh et al., 1997), a avaliação da confiabilidade do software envolve a investigação de suas falhas. A natureza destrutiva da falha e a latência em sua ocorrência dificultam a identificação de sua causa no ambiente operacional. No caso de sistemas baseados em agentes, pode ser difícil recriar o cenário de falha, o que dificulta ainda mais sua investigação (Hsueh et al., 1997).

Neste capítulo, são apresentados dois estudos: o primeiro com o objetivo de avaliar o framework de testes N-JAT4BDI (Seção 8.1) e o segundo para avaliar os casos de teste gerados pelo método proposto (Seção 8.2).

8.1
Estudo 1 – Avaliação do Framework de Teste

Para avaliar o N-JAT4BDI, serão medidas tanto a sua eficácia quanto a sua eficiência em relação à inspeção manual do código na identificação de falhas em agentes normativos. Definimos como *eficácia* o número de falhas precisamente identificadas em relação ao número total de falhas. Espera-se que os desenvolvedores identifiquem um número maior de falhas utilizando o N-JAT4BDI do que os desenvolvedores usando a inspeção manual do código. Definimos como *eficiência* o número de falhas precisamente identificadas dividido pela quantidade de tempo gasto neste objetivo. Espera-se também, que desenvolvedores utilizando o N-JAT4BDI sejam mais eficientes.

8.1.1
Objetivos Específicos

Especificamente, o objetivo deste estudo é avaliar o uso do N-JAT4BDI considerando que: (i) o N-JAT4BDI verifica a conformidade dos comportamentos de agentes BDI com as normas do ambiente; (ii) o N-JAT4BDI fornece mecanismos para controlar os dados de entrada dos testes e, (iii) o N-JAT4BDI fornece mecanismos para observar as decisões do agente e identificar possíveis falhas. Desta forma, por meio deste estudo pretendemos responder a seguinte questão de pesquisa.

Questão de Pesquisa (QP_1): *Como podemos apoiar o teste de sistemas multiagentes normativos através da construção e manutenção de casos de testes para agentes BDI?*

(i) ***Verificando*** *se os agentes agem em conformidade com as normas endereçadas a eles.*

(ii) ***Controlando*** *as ações executadas durante o ciclo de raciocínio na execução do caso de teste.*

(iii) ***Observando*** *as decisões tomadas pelo agente durante a execução do caso de teste.*

Associadas à questão de pesquisa acima apresentamos as seguintes hipóteses sobre o N-JAT4BDI:

- H_{10}: *A taxa de acerto na identificação de falhas em agentes normativos não depende do uso do N-JAT4BDI ou da inspeção manual do código.*
- H_{11}: *O N-JAT4BDI aumenta a taxa de acerto na identificação de falhas em agentes normativos se comparado à inspeção manual do código.*
- H_{20}: *O tempo de identificação de falhas em agentes normativos não depende do uso do N-JAT4BDI ou da inspeção manual do código.*
- H_{21}: *O N-JAT4BDI diminui o tempo de identificação de falhas em agentes normativos se comparado à inspeção manual do código.*
- H_{30}: *A dificuldade na identificação de falhas em agentes normativos não depende do uso do N-JAT4BDI ou da inspeção manual do código.*
- H_{31}: *O N-JAT4BDI diminui a dificuldade na identificação de falhas em agentes normativos se comparado à inspeção manual do código.*

Para analisar as hipóteses H_{10}, H_{11}, H_{20}, H_{21}, H_{30} e H_{31} foi realizado um experimento com 10 participantes com diferentes níveis de conhecimentos e experiências sobre o desenvolvimento de sistemas multiagentes, o uso de casos de teste e a inspeção manual de falhas em código.

8.1.2 Design do Experimento

Tanto o framework N-JAT4BDI (teste) quanto o framework NBDI4JADE (agente normativo) são tecnologias pouco divulgadas até momento. Portanto, o estudo foi projetado de forma a mitigar o impacto da falta de experiência dessas tecnologias no resultado final do experimento.

O projeto experimental deste estudo consiste de cinco passos: (i) seleção do grupo de participantes; (ii) caracterização do participante; (iii) treinamento do participante; (iv) execução das tarefas do experimento; (v) aplicação do formulário de feedback.

Inicialmente procedeu-se o recrutamento e seleção dos participantes, passo 1 (ver Apêndice A.1). No passo 2, foram coletadas informações através da aplicação de um questionário com perguntas referentes à formação e o nível de conhecimento sobre o desenvolvimento com agentes, o uso de casos de teste e a inspeção de código fonte (ver Apêndice A.2). O passo 3 envolveu um breve treinamento do participante com o objetivo de fornecer o conhecimento básico necessário ao experimento. O passo 4 se divide em duas partes, cada parte com duas tarefas de identificação de falhas (ver Apêndice A.3). Para executar as tarefas, dividiu-se os participantes em dois grupos, A e B. Na primeira etapa, os participantes do grupo A realizaram duas tarefas de identificação de falhas utilizando a inspeção manual do código fonte. Na segunda etapa, os participantes do grupo A realizaram as duas tarefas restantes com o auxílio do N-JAT4BDI. De forma inversa, os participantes do grupo B realizaram as duas primeiras tarefas utilizando o N-JAT4BDI e, na segunda etapa, os participantes do grupo B realizaram as duas tarefas restantes utilizando a inspeção manual do código. Com isso, projetou-se um *Quadrado Latino* (Freeman, 1979), podendo aplicar métricas para avaliar a eficácia e eficiência do N-JAT4BDI na identificação de falhas por grupos, por etapa e por atividades. Por fim, no passo 5, foi aplicado um formulário para obter *feedback* do participante com relação a sua experiência no experimento, no uso do N-JAT4BDI, auxiliando, inclusive, em futuras melhorias do framework (ver Apêndice A.4).

O experimento contou com a participação de 10 alunos do curso de pós-graduação da PUC-Rio. Sua execução foi previamente e individualmente agendada com cada participante, de forma que o mesmo tivesse total atenção, foco e tempo para realizar as tarefas. Todos os participantes já tinham cursado, em algum momento, a disciplina de *Seminário em Sistemas Multiagentes*. A Figura 8.1 ilustra a configuração do *Quadrado Latino*.

	Grupo A	Grupo B
Primeira Etapa	Inspeção Manual	Usando N-JAT4BDI
Segunda Etapa	Usando N-JAT4BDI	Inspeção Manual

Figura 8.1: *Quadrado Latino* projetado para o experimento.

8.1.3 Métricas

Para avaliar o desempenho dos participantes na execução das tarefas, utilizou-se três métricas: tempo de execução (medido em minutos), taxa de acerto e nível de dificuldade percebida.

Para calcular a métrica *tempo*, o participante registrou a hora e minuto no início e no fim da execução de cada tarefa. Desta forma, é possível calcular o tempo total (em minutos) gasto em uma tarefa específica.

Para calcular a *taxa de acerto*, calculou-se quão próxima a reposta estava da resposta correta (em porcentagem). Ao considerar tarefas parcialmente executadas, é possível avaliar e computar acertos intermediários, além de identificar os pontos que trouxeram dificuldades aos participantes.

Para calcular a *dificuldade*, o participante registrou, ao final de cada tarefa, sua percepção sobre a dificuldade percebida na realização da tarefa, em uma escala de 1 a 5, onde 1 representa a menor percepção de dificuldade e 5 representa a maior percepção de dificuldade.

8.1.4 Análise de Dados e Resultados

A análise dos dados coletados no experimento e a apresentação dos resultados foram divididas em três subseções. Na primeira subseção é analisado o perfil dos participantes. Na segunda subseção é analisado o resultado de cada tarefa com o N-JAT4BDI e com a inspeção manual de código. Na terceira subseção é analisado o *Quadrado Latino* projetado, de forma horizontal, por etapas, e de forma vertical, por grupo. Para isso, será seguida uma abordagem quantitativa e qualitativa.

8.1.4.1 Perfil dos Participantes

Neste passo do experimento, o objetivo foi coletar informações sobre a experiência dos participantes no desenvolvimento de agentes, no uso de casos de teste e na inspeção de código. O estudo contou com a participação de 10 alunos do curso de pós-graduação do departamento de informática da PUC-Rio, dos quais 7 estavam cursando o mestrado e 3 o doutorado.

O gráfico da Figura 8.2 ilustra a distribuição da experiência de cada participante por área de interesse. O perfil dos participantes foi classificado após aplicação do formulário de caracterização (ver apêndice A.2). Utilizando uma escala de 1 a 5, onde 1 representa nenhum conhecimento e 5 representa total familiaridade no assunto, observa-se no gráfico que, em relação ao desenvolvimento com agentes, 1 participante classificou seu conhecimento como 1, 3 participantes como 2, 2 participantes como 3, 3 participantes como 4 e 1 participante classificou seu conhecimento como 5. Sobre o uso de casos de teste, nenhum participante classificou seu conhecimento como 1, 2 participantes se classificaram como 2, 4 participantes como 3, 3 participantes como 4 e 1 participantes como 5. Sobre a habilidade em realizar a inspeção de código fonte, nenhum participante classificou seu conhecimento como 1 ou 2, 3 participantes classificaram-se como 3, 5 participantes como 4 e 2 participantes classificaram seu conhecimento como 5.

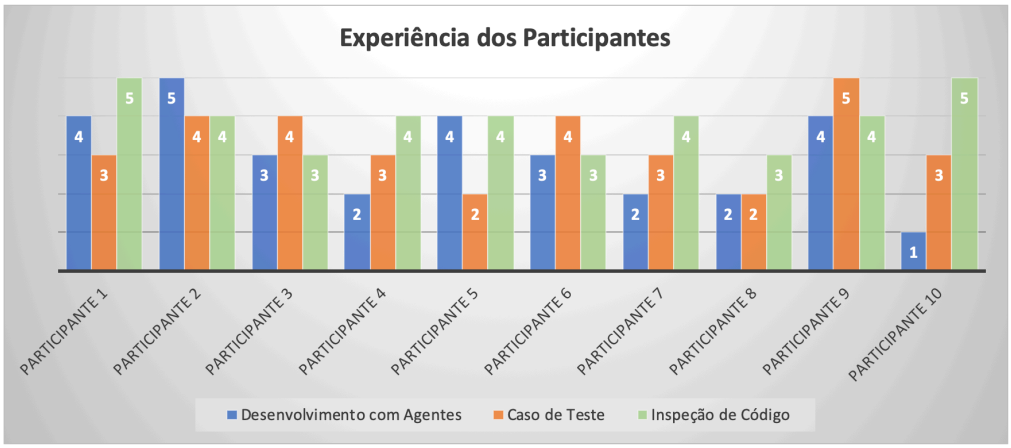


Figura 8.2: Experiência nas Áreas por Participantes.

8.1.4.2 Análise e Resultado das Tarefas

Neste experimento, os participantes foram divididos aleatoriamente em dois grupos, A e B e as tarefas foram aplicadas em duas etapas. Para cada grupo foram aplicadas as mesmas tarefas. Entretanto, em cada etapa, foi invertida

a ordem das técnicas utilizadas na investigação das falhas, proporcionando a comparação dos resultados de uma tarefa na mesma etapa, com e sem o uso do N-JAT4BDI. Abaixo, são analisados e apresentados os resultados de cada tarefas.

Para calcular a métrica tempo, o participante registra a hora de inicio e fim da tarefa. Desta forma, é calculado o tempo, em minutos, gasto na execução da tarefa. Além disto, ao final, o participante registra o nível de dificuldade escolhendo um valor em uma escala de 1 a 5. A métrica taxa de acerto é representada em porcentagem e varia de acordo com os passos corretos que o participante conseguiu realizar. Um acerto ocorre quando é possível obter as informações solicitadas e uma atribuição errônea, ocorre quando não é possível obter as informações.

Tarefa 1

Na tarefa 1, o participante deve verificar informações sobre uma norma. Ao executar a tarefa com inspeção manual, o participante deve localizar a classe, o método e o ponto no código onde acredita ser possível verificar as informações solicitadas. Utilizando o N-JAT4BDI, o participante deve configurar, iniciar o agente em teste e utilizar corretamente as assertivas de verificação.

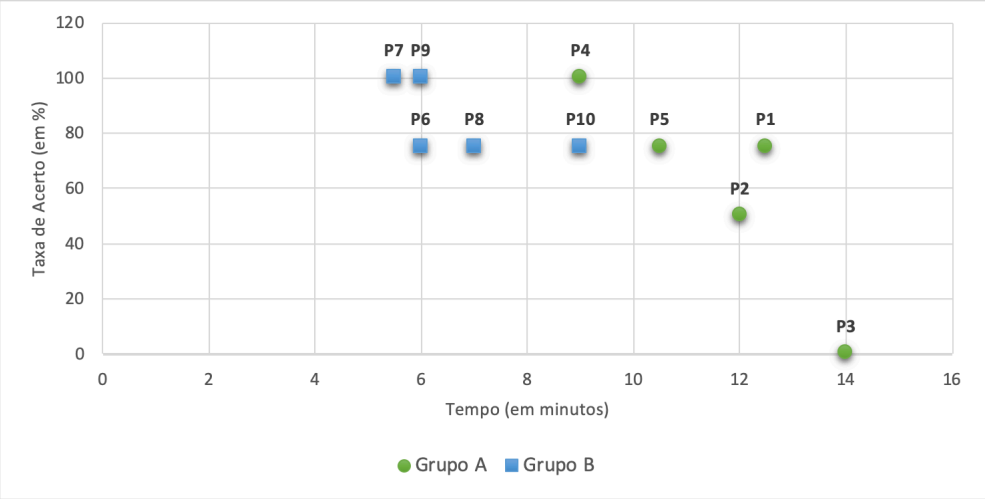


Figura 8.3: Taxa de Acerto por Tempo da Tarefa 1.

A Figura 8.3 ilustra o gráfico da tarefa 1 que relaciona a taxa de acerto com o tempo. O grupo A representa os participantes que realizaram a tarefa através da inspeção manual, exibidos no gráfico por meio de bolas verdes. O grupo B representa os participantes que utilizaram o N-JAT4BDI, exibidos no gráfico por meio de quadrados azuis. Os participantes do grupo

A gastaram entre 10,5 a 14 minutos para realizar a tarefa, uma média de 11,6 minutos. Observa-se também que, apenas um participante obteve taxa de acerto máxima (100%), dois participantes apresentaram taxa de acerto de 75%, um participante obteve 50% e um participante não concluiu a tarefa, obtendo taxa de acerto de 0%, resultando em uma média de 60%. Por outro lado, os participantes do grupo B gastaram entre 5,5 e 9 minutos para executar a tarefa, uma média de 6,7 minutos. Observa-se, também, que dois participantes obtiveram 100% de taxa de acerto e outros três participantes obtiveram 75% de taxa de acerto, uma média de 85%. Dessa forma, supõe-se que, para a tarefa 1, o Grupo B, que utilizou o N-JAT4BDI, obteve melhor desempenho se comparado ao Grupo A, que utilizou a inspeção manual do código.

Tarefa 2

Na tarefa 2, o participante deve identificar as condições que podem estar provocando o comportamento inesperado do agente. Executando a tarefa através da inspeção manual do código, espera-se que o participante localize no código fonte, a classe, o método e o ponto no qual acredita ter problemas. Utilizando o N-JAT4BDI, espera-se que o participante configure, inicialize o agente em teste e utilize corretamente as assertivas de verificação apropriadas.

A Figura 8.4 ilustra o gráfico da tarefa 2 que relaciona a taxa de acerto com o tempo. O grupo A representa os participantes que realizaram a tarefa utilizando a inspeção manual do código, exibidos no gráfico por meio de bolas verdes. O grupo B representa os participantes que realizaram a tarefa com o N-JAT4BDI, exibidos no gráfico por meio dos quadrados azuis. Os participantes do grupo A gastaram de 9,5 a 14 minutos para executar a tarefa, com uma média de 11,5 minutos. Observamos que um participante não concluiu a tarefa e obteve taxa de acerto de 0%), um participante obteve taxa de acerto de 50% e três participantes obtiveram taxa de acerto de 75%, com uma média de 55%. Por outro lado, os participantes do grupo B, gastaram entre 6,5 e 10 minutos para executar a tarefa, com uma média de 8,4 minutos. Observamos que dois participantes alcançaram 100% de taxa de acerto, dois participantes obtiveram taxa de acerto de 75% e um participante teve taxa de acerto de 50%, com uma média de 80%. Assim, supomos que, para a tarefa 2, o Grupo B, que utilizou o N-JAT4BDI, obteve melhor desempenho se comparado ao Grupo A, que realizou a tarefa através da inspeção manual.

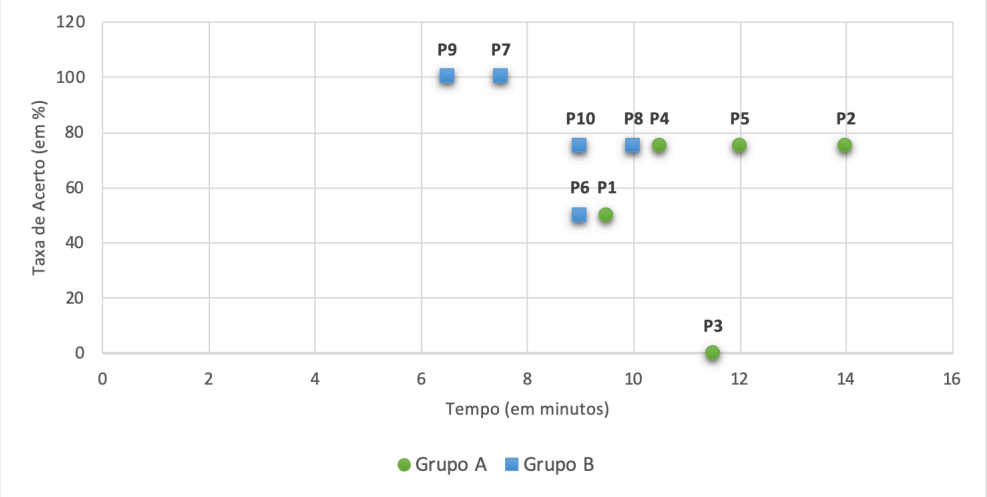


Figura 8.4: Taxa de Acerto por Tempo da Tarefa 2.

Tarefa 3

Na tarefa 3, o participante deve verificar o valor de algumas propriedades da norma. Utilizando a inspeção manual do código, espera-se o participante localize a classe, o método e o ponto onde é possível verificar o valor das propriedades solicitadas. Utilizando o N-JAT4BDI, espera-se que o participante configure, inicie o agente em teste e utilize as assertivas de verificação apropriadas para a tarefa.

A Figura 8.5 ilustra o gráfico da tarefa 3 que relaciona a taxa de acerto com o tempo. O grupo A representa os participantes que utilizaram o N-JAT4BDI, exibidos no gráfico por meio de bolas verdes. O grupo B representa os participantes que utilizaram a inspeção manual, exibidos no gráfico por meio de quadrados azuis. Os participantes do grupo A, gastaram entre 6 e 8 minutos para executar a tarefa, com uma media de 7 minutos. Além disso, observamos que 3 participantes obtiveram taxas de acerto máxima (100%) e outros dois participantes apresentaram taxa de acertos de 75%, com uma média de 90%. Por outro lado, os participantes do grupo B gastaram entre 9,5 e 14 minutos para executar a tarefa, com uma média de 11,5 minutos. Observamos que, dois participantes não concluíram a tarefa e obtiveram taxa de acerto de 0%, dois participante obtiveram taxa de acerto de 50% e um participante obteve taxa de acerto de 75%, com uma média de 35%. Dessa forma, supomos que, para a tarefa 3, o Grupo A, que utilizou o N-JAT4BDI, obteve melhor desempenho se comparado ao Grupo B, que realizou a inspeção manual do código.

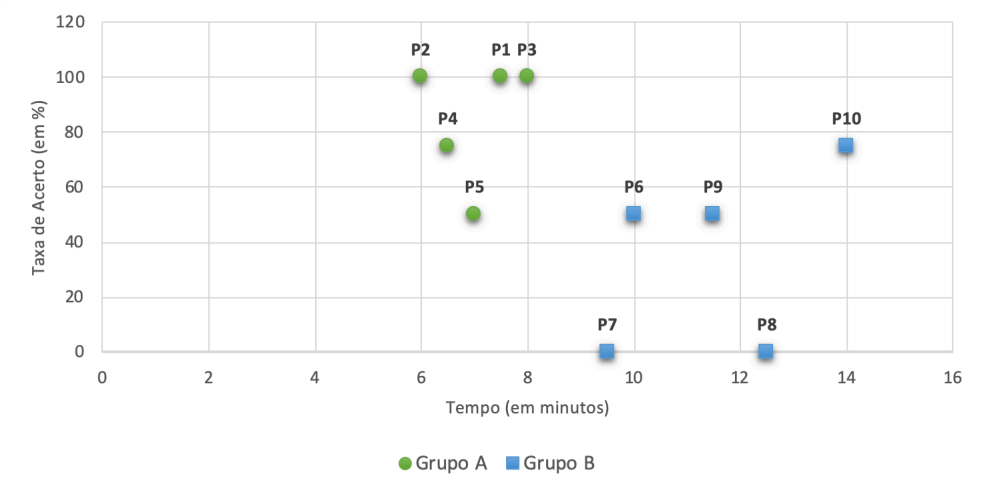


Figura 8.5: Taxa de Acerto por Tempo da Tarefa 3.

Tarefa 4

Na tarefa 4, o participante deve identificar possíveis falhas que causam o comportamento incorreto do agente. Executando a tarefa através da inspeção manual do código, o participante deve localizar a classe, o método e a linha do código onde é possível verificar as informações solicitadas. Utilizando o N-JAT4BDI, o participante deve configurar, inicializar e utilizar as assertivas de verificação apropriadas para a tarefa.

A Figura 8.6 ilustra o gráfico da tarefa 4 que relaciona a taxa de acerto com o tempo. O grupo A representa os participantes que utilizaram o N-JAT4BDI na execução da tarefa, exibidos no gráfico por meio de bolas verdes. O grupo B representa os participantes que utilizaram a inspeção manual do código, exibidos no gráfico por meio de quadrados azuis. Os participantes do grupo A gastaram um tempo entre 6,5 e 11,5 minutos, com média de 8,4 minutos. Além disto, três participantes obtiveram 100% de taxa de acerto, um participante obteve taxa de acerto de 75% e um participante obteve taxa de acerto de 50%, com média de 85%. Os participantes do grupo B gastaram entre 11 e 15 minutos, com uma média de 12,3 minutos. Observamos que, um participante obteve taxa de acerto máxima (100%), um participante obteve taxa de acerto de 75%, um participante obteve taxa de 50% e dois participantes não concluíram a tarefa, apresentando taxa de acerto de 0%, com média de 45%. Dessa forma, supomos que, para a tarefa 4, o Grupo A, que utilizou o N-JAT4BDI, obteve melhor desempenho se comparado ao Grupo B, que realizou a inspeção manual do código.

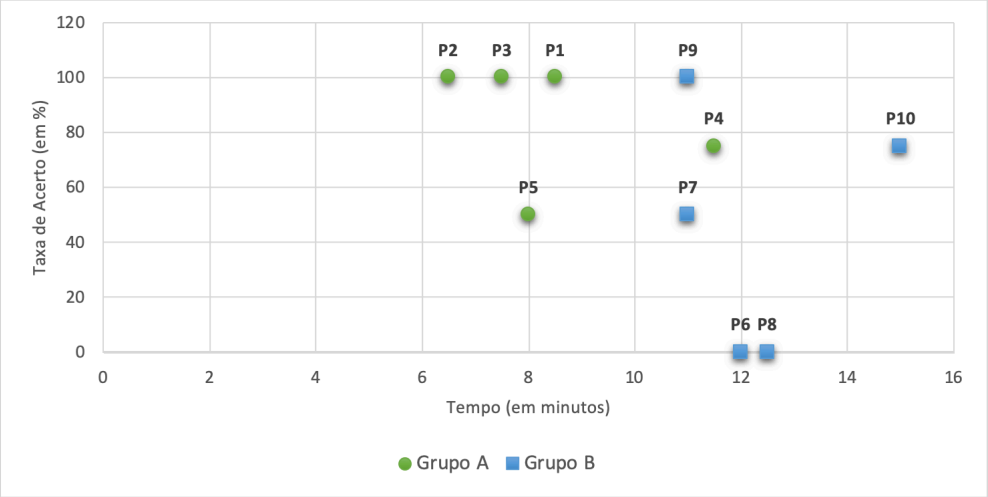


Figura 8.6: Taxa de Acerto por Tempo da Tarefa 4.

8.1.4.2 Média das Métricas

Para auxiliar na avaliação estatística utilizamos a linguagem de programação R. Inicialmente, calculamos as médias das métricas, ou seja, a média da taxa de acerto, a média do tempo e a média da dificuldade, como ilustrado na Figura 8.7. Na sequência, analisamos as variáveis envolvidas e descrevemos qual teste de hipóteses aplicar. Como geralmente é utilizado, definimos o valor de intervalo de confiança de 95%.

	Grupo A			Grupo B		
	Acerto	Tempo	Dificuldade	Acerto	Tempo	Dificuldade
Primeira Etapa	57,5	11,6	3,9	82,5	7,6	1,8
Segunda Etapa	87,5	7,7	1,6	40,0	11,9	4,0

Figura 8.7: Média das Métricas por Grupo e Etapa.

8.1.4.2 Variáveis e Análises

A variável independente em nosso experimento é a *ferramenta* utilizada pelos participantes. Como variáveis dependentes temos: (i) a *taxa de acerto* do participante ao executar uma tarefa; (ii) o *tempo* gasto na execução da tarefa e, (iii) a *dificuldade* encontrada na execução da tarefa. Além disto, a amostra de dados utilizada vêm de dois grupos independentes.

Para testar as hipóteses do estudo, primeiramente verificou-se, com teste Shapiro-Wilk, se a amostra segue uma distribuição normal e com o teste de Levene sua homocedasticidade. Se ambos os testes passam, utilizaremos o Teste T para avaliar as hipóteses nulas, ou seja, se há evidências de diferença entre as médias das populações. Caso contrário, será utilizado o teste não paramétrico de Mann-Whitney. Como dito, para as variáveis de *taxa de acerto* e *tempo*, mantemos um nível de confiança de 95% (alfa = 0,05).

8.1.4.2 Teste de Hipóteses

Começamos testando a hipótese nula H_{10}, que afirma que a taxa de acerto na identificação de falhas em agentes normativos não depende do uso do N-JAT4BDI ou da inspeção manual do código. O teste de Shapiro-Wilk não teve êxito, retornando um *p-value* = 0,002402, o que significa que Levene não pode ser usado. Consequentemente, utilizamos o teste não paramétrico de Mann-Whitney e obtivemos um *p-value* = 0,004196, indicando que há diferença estatisticamente significante entre as técnicas investigadas em relação a taxa de acerto. Sendo assim, a hipótese nula H_{10} foi rejeitada em favor da hipótese alternativa H_{11}, *o N-JAT4BDI aumenta a taxa de acerto na identificação de falhas em agentes normativos se comparado à inspeção manual do código.*

Em seguida, testamos a hipótese nula H_{20}, que afirma que o tempo gasto na identificação de falhas em agentes normativos não depende do uso do N-JAT4BDI ou da inspeção manual do código. O teste de Shapiro-Wilk retornou um *p-value* = 0,457394, indicando a normalidade da amostra. Aplicamos o teste de Levene para verificar a homocedasticidade e obtemos um *p-value* = 0,257725. Dessa forma, utilizamos o teste T e obtivemos um *p-value* = 0,000001. Sendo assim, nossa hipótese nula H_{20} foi rejeitada em favor da hipótese alternativa H_{21}, *o N-JAT4BDI diminui o tempo na identificação de falhas em agentes normativos se comparado à inspeção manual do código.*

Por fim, testamos a hipótese nula H_{30}, que afirma que a dificuldade percebida na identificação de falhas em agentes normativos não depende do uso do N-JAT4BDI ou da inspeção manual do código. O teste de Shapiro-Wilk retornou um *p-value* = 0,076269, indicando a normalidade da amostra. Na sequência, aplicamos o teste de Levene e obtivemos *p-value* = 0,7544561. Consequentemente, podemos utilizar o teste paramétrico T, obtendo um *p-value* = 0,00001 e um *p-variance* = 0,932776. Sendo assim, a hipótese nula H_{30} foi rejeitada em favor da hipótese alternativa H_{31}, *o N-JAT4BDI diminui o tempo na identificação de falhas em agentes normativos se comparado à inspeção manual do código.*

Análise das Suposições Derivadas do *Quadrado Latino*

A partir das análises horizontal e vertical do *Quadrado Latino* obtemos indícios para algumas suposições. A Figura 8.8 apresenta as suposições para a métrica *taxa de acerto* na análise horizontal. Observamos que: (i) a suposição na linha 1 é refutada pela suposição da linha 3 e, (ii) as suposições das linhas 2 e 4 são equivalentes, ou seja, o N-JAT4BDI auxilia no teste de agentes normativos com maior taxa de acerto.

Primeira Etapa	1	(i) o grupo A executa com uma taxa de acerto maior que o grupo B;
	2	(ii) o N-JAT4BDI apoia o teste de agentes normativos com uma taxa maior de acerto;
Segunda Etapa	3	(i) o grupo A executa com uma taxa de acerto menor que o grupo B;
	4	(ii) o N-JAT4BDI apoia o teste de agentes normativos com uma taxa maior de acerto;

Figura 8.8: Suposições de Taxa de Acerto – Horizontal.

A Figura 8.9 apresenta as suposições para a métrica *tempo* na análise horizontal. Observamos que: (i) a suposição da linha 1 é refutada pela suposição da linha 3 e, (ii) as suposições das linhas 2 e 4 são equivalentes, ou seja, o N-JAT4BDI ajuda os participantes no teste de agentes normativos com um tempo menor.

Primeira Etapa	1	(i) o grupo A executa em um tempo maior do que o grupo B;
	2	(ii) o N-JAT4BDI ajuda os participantes a executar a tarefa em um tempo menor;
Segunda Etapa	3	(i) o grupo A executa em um tempo menor do que o grupo B;
	4	(ii) o N-JAT4BDI ajuda os participantes a executar a tarefa em um tempo menor;

Figura 8.9: Suposição de Tempo – Horizontal.

A Figura 8.10 apresenta as suposições para a métrica *dificuldade* na análise horizontal. Podemos observar que: (i) a suposição da linha 1 é refutada pela suposição da linha 3 e, (ii) as suposições das linhas 2 e 4 são equivalentes, isto é, o uso do N-JAT4BDI diminui o nível de dificuldade em realizar as tarefas.

Primeira Etapa	1	(i) o grupo A teve um nível de dificuldade maior do que o grupo B;
	2	(ii) o N-JAT4BDI diminui o nível de dificuldade em realizar as tarefas;
Segunda Etapa	3	(i) o grupo A teve um nível de dificuldade menor do que o grupo B;
	4	(ii) o N-JAT4BDI diminui o nível de dificuldade em realizar as tarefas;

Figura 8.10: Suposição de Nível de Dificuldade – Horizontal.

Foram analisadas até o momento de forma horizontal as suposições restantes, mitigando o viés dos grupos, taxa de acerto, tempo de execução e nível de dificuldade. Resta verificar as suposições geradas na análise vertical. Serão analisadas agora, as suposições geradas de forma vertical no *Quadrado Latino*.

A Figura 8.11 apresenta as suposições para a métrica *taxa de acerto* na análise vertical. Dessa forma, podemos observar que: (i) a suposição da linha 1 é refutada pela suposição da linha 3 e, (ii) as suposições das linhas 2 e 4 são equivalentes, ou seja, o N-JAT4BDI apoia o teste de agentes normativos com uma taxa maior de acerto.

Primeira Etapa	1	(i) os participantes possuem uma taxa de acerto maior na primeira etapa do que na segunda;
	2	(ii) o N-JAT4BDI apoia o teste de agentes normativos com uma taxa maior de acerto;
Segunda Etapa	3	(i) os participantes possuem uma taxa de acerto menor na primeira etapa do que na segunda;
	4	(ii) o N-JAT4BDI apoia o teste de agentes normativos com uma taxa maior de acerto;

Figura 8.11: Suposição de Taxa de Acerto – Vertical.

A Figura 8.12 apresenta as suposições para a métrica *tempo* na análise vertical. Com isto, podemos observar que: (i) a suposição da linha 1 é refutada pela suposição da linha 3 e, (ii) as suposições das linhas 2 e 4 são equivalentes, ou seja, o uso do N-JAT4BDI ajuda no teste de agentes normativos com um tempo menor.

Por sua vez, a Figura 8.13 apresenta as suposições para a métrica *dificuldade* na análise vertical. Assim, podemos observar que: (i) a suposição da linha 1 é refutada pela suposição da linha 3 e, (ii) as suposições das linhas 2

Grupo A	1	(i) os participantes executam a tarefa em um tempo maior na primeira etapa em relação a segunda etapa;
	2	(ii) o N-JAT4BDI ajuda o participante a executar a tarefa em um tempo menor;
Grupo B	3	(i) os participantes executam a tarefa em um tempo menor na primeira etapa em relação a segunda etapa;
	4	(ii) o N-JAT4BDI ajuda o participante a executar a tarefa em um tempo menor;

Figura 8.12: Suposição de Tempo – Vertical.

e 4 são equivalentes, isto é, o N-JAT4BDI diminui a percepção de dificuldade no teste de agentes normativos.

Grupo A	1	(i) os participantes percebem um nível de dificuldade maior na primeira etapa em relação a segunda etapa;
	2	(ii) o N-JAT4BDI diminui o nível de dificuldade em realizar as tarefas;
Grupo B	3	(i) os participantes percebem um nível de dificuldade menor na primeira etapa em relação a segunda etapa;
	4	(ii) o N-JAT4BDI diminui o nível de dificuldade em realizar as tarefas;

Figura 8.13: Suposição de Dificuldade – Vertical.

Após a análise das suposições do *Quadrado Latino* de forma horizontal e vertical, com relação a métrica *taxa de acerto*, restaram na Figura 8.8 as suposições das linhas 2 e 4 e na Figura 8.11 as suposições das linhas 2 e 4. Pode-se observar que tais suposições são equivalentes. Logo, nossa primeira conclusão é que temos fortes indícios que o N-JAT4BDI apoia o teste de agentes normativos com uma taxa de acerto maior. Para a métrica *tempo*, restaram na Figura 8.9 as suposições da linha 2 e 4 e, na Figura 8.12 as suposições da linha 2 e 4. Pode-se observar que tais suposições são equivalentes. Logo, a segunda conclusão é que temos fortes indícios de que o N-JAT4BDI ajuda no teste de agentes normativos com um tempo menor. Por fim, para a métrica *dificuldade*, restaram na Figura 8.10 as suposições da linha 2 e 4 e na Figura 8.13 as suposições da linha 2 e 4. Pode-se observar que tais suposições são equivalentes. Logo, a terceira conclusão é que temos fortes indícios de que o N-JAT4BDI diminui a dificuldade no teste de agentes normativos.

A Figura 8.14 apresenta os erros encontrados nas respostas dos participantes do estudo ao utilizar o N-JAT4BDI e a inspeção manual na execução das tarefas.

Uso da Inspeção Manual	Uso do N-JAT4BDI
Falta de conhecimento do framework NBDI4JADE, no que se refere aos atributos normativos	Escolha da assertiva de verificação errada para a identificação de uma falha
Erro na identificação da classe responsável por uma informação	Configuração incorreta das pré-condições do caso de teste
Erro na identificação do método apropriado para obter uma informação	Redundância nos testes, ou seja, diferentes casos de teste verificando as mesmas falhas
Verificação em pontos do código que não fornecem a informação solicitada	

Figura 8.14: Tipos de Erros Identificados nas Avaliações Aplicadas.

De forma geral, os participantes se mostraram mais confortáveis em realizar as tarefas com o N-JAT4BDI do que a inspeção manual, ambas as técnicas com o objetivo de verificar o comportamento do agente. A principal razão parece ser o nível de abstração oferecido pelo N-JAT4BDI que, encapsula muitas das tarefas de inspeção do código em assertivas de verificação (no formato JUnit). Mesmo os participantes com maior experiência no desenvolvimento de agentes relatam que encontrar falhas investigando diretamente o código não é uma tarefa simples. A seguir, apresentamos alguns depoimentos extraídos do formulário de feedback após a execução das tarefas.

"*Para usar a inspeção manual, precisa conhecer bem o código do agente, saber onde procurar! Isso é complicado para quem não usa agentes. A ferramenta de testes já entrega as coisas mastigadas, sem dúvidas que ajuda! Acho que não tem como comparar!*" – *Participante 9*

"*Foi um pouco confuso configurar os casos de teste, mas se passar por isso fica fácil testar!*" – *Participante 8*

"*Gostei da ideia da ferramenta de testes mas fiquei na dúvida de quantos testes devo criar para resolver o problema.*" – *Participante 1*

8.1.5 Possíveis Ameaças

Esta seção discute os fatores que representam possíveis ameaças à validade do estudo realizado e as medidas tomadas para contorná-las.

Validade da Conclusão. Entendemos como maiores ameaças à validade da conclusão: (i) o engajamento dos participantes, tendo em vista que a

duração do experimento levou em torno de 1,5 horas. Para mitigar essa ameaça, propusemos um número pequeno de tarefas simples. Aplicamos as tarefas de forma alternada para mitigar o viés da técnica utilizada; (ii) outra possível ameaça seria o conhecimento prévio sobre agentes e sobre o framework de testes sendo avaliado. Contornamos a heterogeneidade dos participantes, selecionando alunos que já cursaram a disciplina de Seminário de Sistemas Multiagentes e oferecendo um treinamento no uso do N-JAT4BDI; (iii) outra possível ameaça é o número pequeno de participantes da amostra, que não permite cobrir perfis representativos da população, e (iv) a efetividade do ambiente proposto para o estudo, uma vez que, para executar as tarefas com a inspeção do código, o desenvolvedor não precisou programar um ambiente artificial para testar o agente. Essa é uma ameaça a uma conclusão justa da eficácia e eficiência entre as duas técnicas exploradas.

Validade da Construção. Foram identificadas as seguintes ameaças à validade de construção: (i) treinamento insuficiente e tarefas não realistas. Para mitigar o efeito do pouco treinamento, respondemos às perguntas dos participantes à medida que estes necessitavam. Para evitar a polarização dos resultados em função do auxílio oferecido, limitamos nossas explicações a detalhes da arquitetura e hierarquia de classes dos agentes (nas tarefas de inspeção manual) e a explicações da estrutura dos casos de testes (para tarefas com o N-JAT4BDI), nos casos onde tais esclarecimentos eram absolutamente necessários.

Validade Interna. As possíveis ameaças à validade interna se relacionam ao conjunto de tarefas propostas e aos participantes. Tentamos mitigar essas ameaças propondo tarefas simples e que exigissem do participantes somente o necessário para termos a percepção o objeto investigado. Por outro lado, essa estratégia pode não refletir o contexto dos testes de uma aplicação real. Em relação aos participantes, ao misturarmos alunos que pesquisam sobre agentes com aqueles que conhecem agentes, podemos não ter uma amostra representativamente justa.

Validade Externa. O maior risco externo está relacionado ao framework de agentes utilizado. Em nosso experimento, inspecionamos códigos de agentes NBDI4JADE. Entretanto, não levamos em consideração as dificuldade e facilidades ao inspecionar o código em outros frameworks tais como: Jason, Jadex ou Jack. Para uma generalização dos resultados, são necessárias repetições do estudo com outras linguagens de desenvolvimento de agentes e cenários, para determinar se os achados deste capítulo podem ser, de certa forma, generalizados.

8.2 Estudo 2 - Avaliação dos Casos de Teste Gerados

Neste estudo, pretende-se avaliar o método proposto com relação à eficácia dos casos de teste gerados e, para tal, foi realizado um experimento onde o método foi aplicado aos cenários de uso apresentados no Capítulo 7. Em seguida, apresentamos, analisamos e discutimos os resultados obtidos.

8.2.1 Objetivos Específicos

Especificamente, o objetivo deste estudo é avaliar a eficácia dos casos de teste gerados, considerando que: (i) os casos de teste lidam com a variabilidade de comportamentos regulados pelas normas; (ii) os casos de teste cobrem os diferentes tipos de falhas e, (iii) os casos de teste executam cenários que representam as decisões dos agentes. Desta forma, pretendemos responder a seguinte questão de pesquisa.

- **Questão de Pesquisa (QP_2)**: *Como podemos utilizar o teste baseado em modelos para apoiar o teste de agentes BDI normativos?*
  - (i) ***Lidando** com a variabilidade dos comportamentos regulados pelas normas.*
  - (ii) ***Cobrindo** a identificação dos tipos de falhas nos comportamentos.*
  - (iii) ***Executando cenários** que representam as decisões tomadas pelo agente.*

Associadas à questão de pesquisa acima apresentamos as seguintes hipóteses:

- H_{10}: *Os casos de teste gerados não são eficazes na identificação de falhas em agentes BDI normativos.*
- H_{11}: *Os casos de teste gerados são eficazes na identificação de falhas em agentes BDI normativos.*

Para analisar as hipóteses H_{10} e H_{11} foi realizado um estudo onde: (i) aplicamos o método aos cenários de uso; (ii) injetamos falhas nos agentes para avaliar a eficácia dos casos de teste na identificação dessas falhas, e (iii) analisamos os resultados obtidos.

8.2.2 Design do Experimento

O projeto experimental deste estudo consiste em três etapas, como segue:

1. *Configuração.* Primeiramente, os cenários de uso foram selecionados, modelados utilizando ANA-ML e implementados em NBDI4JADE. As informações contidas nos diagramas ANA-ML foram manualmente extraídas para arquivos no formato XML, de forma a refletir integralmente as informações descritas nos modelos e facilitar a manipulação das informações pela ferramenta de geração dos casos de teste.

2. *Execução.* Foram gerados e executados os casos de teste de acordo com as etapas descritas no método. A execução dos casos de teste teve o suporte do N-JAT4BDI.

3. *Análise das falhas.* Após a execução dos casos de teste, as falhas não detectadas foram analisadas e relacionadas a suas causas. Falhas podem ser causadas por problemas na especificação, na implementação ou estar associada a uma situação específica, inclusive, podendo se tratar de um falso positivo.

8.2.3 Métricas

Nesta avaliação, utilizou-se a métrica: *número de falhas detectadas*. Para calcular a métrica *número de falhas detectadas* adotamos a técnica de *injeção de falhas* (Voas e McGraw, 1997). Assim, espera-se que os casos de teste gerados possam identificar as falhas injetadas. A *eficácia* dos casos de teste é dada pelo número de falhas precisamente identificadas em relação ao total de falhas injetadas.

A *injeção de falhas* é uma técnica muito útil para avaliar a eficácia das abordagens de teste e sua ideia central é inserir falhas específicas durante a execução do sistema para verificar se o teste é capaz de detectar a falha injetada com precisão (Voas e McGraw, 1997).

8.2.4 Análise de Dados e Resultados

A análise dos dados e a discussão dos resultados foram divididas em quatro subseções. Na primeira subseção são apresentados os tipos de falhas injetadas, na segunda subseção as falhas são categorizadas, na terceira subseção é apresentada a distribuição das falhas e na quarta subseção são discutidos os resultados obtidos.

8.2.4.1 Tipos de Falhas Injetadas

De acordo com o modelo de faltas apresentado no Capítulo 4, foram injetadas falhas de todos os tipos para avaliar a capacidade de detecção dos casos de teste.

O modelo de faltas define cinco tipos de falhas: (i) falhas no endereçamento; (ii) falhas na condição de ativação/desativação; (iii) falhas na motivação; (iv) falhas no conceito deôntico e, (v) falhas no estado interno. Para operacionalizar a injeção de falhas, foi implementada uma ferramenta baseada no uso de anotações Java e na programação orientada a aspectos, onde o *aspecto injetor de falhas* intercepta o comportamento dos agentes em teste e introduz uma falha específica descrita no modelo.

O Injetor de Falhas

O *injetor de falhas* foi implementado como um aspecto na linguagem AspectJ. Ele intercepta o código do agente em teste e adiciona um "comportamento defeituoso". Definimos uma anotação para cada tipo de falha do modelo, como segue:

- *@ActivateNorm*: força a ativação de uma norma no ambiente. O atributo dessa anotação é a norma que desejamos ativar. Essas falhas simulam a situação onde a norma é ativada mesmo que a condição de ativação não tenha sido alcançada;
- *@DeactivateNorm*: força a desativação de uma norma no ambiente. O atributo dessa anotação é a norma que desejamos desativar. Essas falhas simulam a situação onde a norma é desativada mesmo que a condição de desativação não tenha sido alcançada;
- *@IncreaseReward*: força a recompensa do agente. O atributo dessa anotação é o valor da recompensa. Essas falhas simulam a recompensa de um agente que cumpriu com a norma;
- *@IncreasePunishment*: força a punição do agente. O atributo dessa anotação é o valor da punição. Essas falhas simulam a punição de um agente que não cumpriu com a norma;
- *@ChangeAddressee*: força o endereçamento da norma para agente. O atributo dessa anotação é a norma e a lista de agentes endereçados pela norma. Essas falhas permitem verificar se uma norma é sensoreada por um agente o qual não foi endereçada;

- *@ChangeBelief*: força a alteração do valor de uma crença. O atributo dessa anotação é o nome da crença e o valor. Essas falhas simulam se a norma regula a crença do agente;
- *@ChangeMessage*: força o envio e o recebimento de mensagens. O atributo dessa anotação é o nome do agente destinatário e a mensagem. Essas falhas simulam se a norma regula o envio de mensagens pelo agente;

Para injetar a falha no código do agente, o desenvolvedor precisa, apenas, incluir as anotações (e especificar os valores de seus atributos) em seus métodos. A Figura 8.15 ilustra o código parcial no qual uma falha é injetada por uma das anotações especificadas acima.

```
38    public static class RedCarPlan extends Behaviour implements PlanBody {
39
41        public EndState getEndState() {
44
46        public void init(PlanInstance planInstance) {
48
49        @IncreaseReward(reward = 10)
50        @Override
51        public void action() {
52            System.out.println("Comportamento do agente RedCar");
53        }
54
56        public boolean done() {
59
60    }
```

Figura 8.15: Código Parcial do Método Anotado.

A Figura 8.16 ilustra o código parcial do aspecto injetor de falhas. Este aspecto define o *pointcut agentRevision* (linhas 13-14) que intercepta o método que revisa as crenças do agente. Este pointcut é usado em combinação com o pointcut *annotedWithIncreaseReward* (linha 11), que detalha a falha a ser injetada e interceptam o ponto onde as anotações foram definidas, criando ou substituindo a crença *reward* (padrão no NBDI4JADE para lidar com recompensas) pelo valor recebido no atributo da anotação (linhas 19-21). Em seguida, o aspecto procede com a execução do agente (linha 22).

8.2.4.2 Categorização das Falhas

Os trabalhos de Boehm *et al.* (2005) e Ramberger *et al.* (2004) apresentam a importância da categorização de falhas no escopo dos testes unitários (Boehm et al., 2005) (Ramberger et al., 2004). De acordo com Boehm *et al.*, são definidas duas categorias de falhas: (i) falhas que indicam que algo foi desenvolvido incorretamente (*commission faults*) e, (ii) falhas que indicam que algo foi esquecido no projeto ou na implementação (*omission faults*).

```
 9 public aspect FaultInjector {
10
11     pointcut annotedWithIncreaseReward(IncreaseReward cp) : @withincode(cp);
12
13     pointcut agentRevision(BDIAgent bdiAgent) :
14         call(* BeliefRevisionStrategy.reviewBeliefsAndNorms(..)) && args(bdiAgent);
15
16     void around(BDIAgent bdiAgent, IncreaseReward cp) : agentRevision(bdiAgent)
17                                     && annotedWithIncreaseReward(cp) {
18
19         int score = cp.reward();
20         Belief<?> reward = new TransientBelief<>("reward", score);
21         bdiAgent.getAllBeliefs().add(reward);
22         proceed(bdiAgent, cp);
23     }
24
25 }
```

Figura 8.16: Código Parcial do Aspecto Injetor de Falhas.

Para este estudo, as falhas são categorizadas da seguinte forma:

1. *implementação incompleta* é uma *omission fault* de codificação
   Uma falha desta categoria indica que um *recurso* especificado no design não foi implementado.

2. *implementação incorreta* é uma *commission fault* de codificação
   A lógica interna do componente não está implementada corretamente.

3. *falsos positivos*
   Algumas falhas detectadas não são problemas reais quando investigadas.

4. *falhas redundantes*
   Uma falha é considerada redundante se ela foi causada por problemas identificados anteriormente por outra falha.

8.2.4.3 Resultados das Falhas Detectadas pelos Casos de Teste

Se um caso de teste "passou", então o resultado real corresponde ao resultado esperado. Por outro lado, um caso de teste falha se não atravessar o caminho definido para a entrada ou se não alcançar os estados internos previamente definidos. Esse é o efeito esperado ao injetarmos uma falha, que o teste "quebre", indicando que identificou a falha injetada. Neste experimento, foram injetadas 217 falhas no total, sendo 153 na aplicação do EC e 66 na aplicação do TourAgent, abrangendo todos os tipos de falhas do modelo de faltas.

A Figura 8.17 apresenta o número de casos de testes gerados para cada critério de cobertura e o total de falhas injetadas e detectadas para cada tipo de falha do cenário Expert Committee.

Tipos de Falhas Injetadas/Detectadas por Critério de Cobertura						
		Addressee	Condition	Motivation	Deontic Concept	States
Critério	# Casos	Injected/Detected	Injected/Detected	Injected/Detected	Injected/Detected	Injected/Detected
All Goals	53	12/12	24/12	4/4	12/12	5/5
Scenario	17	3/3	6/3	2/2	3/3	3/3
Capability	36	9/9	18/9	0/0	9/9	0/0
All Plans	153	33/33	66/33	10/10	33/33	11/11
Goal-Plan	153	33/33	66/33	10/10	33/33	11/11

Figura 8.17: Falhas Injetadas e Detectadas no Expert Committee.

Podemos observar que, exceto para falhas do tipo *condition*, os casos de testes gerados detectaram precisamente todas as falhas injetadas, ou seja, falhas dos tipos *addressee*, *motivation*, *deontic concept* e *state* foram 100% detectadas. Para falhas do tipo *condition*, apenas 50% foi precisamente detectada. Analisamos as falhas não detectadas e identificamos que todas estavam relacionadas com a desativação da norma, ou seja, o agente em teste não reconheceu que a norma foi intencionamente ativada. Identificamos duas razões para esse comportamento: (i) a condição de ativação da norma estava sendo anulada após a injeção da falha devido a um erro de implementação, e (ii) a condição de desativação estava fixada, *hard coded*.

A Figura 8.18 apresenta o número de casos de testes gerados para cada critério de cobertura e o total de falhas injetadas e detectadas para cada tipo de falha do cenário TourAgent.

Tipos de Falhas Injetadas/Detectadas por Critério de Cobertura						
		Addressee	Condition	Motivation	Deontic Concept	States
Critério	# Casos	Injected/Detected	Injected/Detected	Injected/Detected	Injected/Detected	Injected/Detected
All Goals	32	6/6	12/12	4/4	6/6	4/4
Scenario	16	3/3	6/6	2/2	3/3	2/2
Capability	32	6/6	12/12	4/4	6/6	4/4
All Plans	48	9/9	18/18	6/6	9/9	6/6
Goal-Plan	64	12/12	24/24	8/8	12/12	8/8

Figura 8.18: Falhas Injetadas e Detectadas no TourAgent

Podemos observar que, para o cenário do TourAgent, todos os tipos de falhas foram precisamente detectados, ou seja, falhas dos tipos *addressee*, *condition*, *motivation*, *deontic concept* e *state* foram 100% detectadas. Creditamos essa precisão aos seguintes fatores: (i) a aplicação do TourAgent é de menor

complexidade e tamanho; (ii) os casos de teste deste cenário foram gerados após a análise dos problemas encontrados no cenário do EC; (iii) foram feitos ajustes no código da aplicação do TourAgent para corrigir os defeitos de implementação observados com os casos executados para o EC.

Diante dos resultados obtidos, verificamos que os casos de testes identificaram 78,43% das falhas injetadas no EC. Se considerarmos o total de falhas injetadas, os casos de teste identificaram 84,80% das falhas injetadas. As Figuras 8.19 e 8.20 apresentam em detalhes as falhas injetadas.

8.2.5 Possíveis Ameaças

Esta seção apresenta as possíveis ameaças a validade do estudo e ações tomadas para mitigar tais ameaças.

Validade Interna

As ameaças internas residem na qualidade da implementação dos sistemas podendo gerar falhas decorrentes de uma implementação ausente ou incorreta. Para avaliar melhor se os casos pegariam as falhas injetadas, identificamos essa ameaça em alguns casos no cenário do Expert Committee e ajustamos para o cenário do TourAgent, obtendo um resultado ainda melhor.

Validade Externa

Entendemos que o maior risco externo está relacionado com a linguagem de modelagem escolhida para especificar os agentes e o framework utilizado no desenvolvimento dos agentes testados. ANA-ML mostrou-se interessante à modelagem dos aspectos adaptativos dos agentes regulados por normas (Viana et al., 2016). Entretanto, a linguagem não oferece abstrações para lidar com o teste de agentes normativos. Apesar dos resultados interessantes apresentados por (Viana et al., 2016) sobre ANA-ML, sua discussão e utilização estão restritas à poucos trabalhos de pesquisa. De forma semelhante, o uso de agentes NBDI4JADE representa outra ameaça a validade externa deste estudo. O NBDI4JADE framework (Cunha et al., 2018) foi desenvolvido como parte de nossa infraestrutura de teste e não está disponível à comunidade.

Validade de Construção

Identificamos como possíveis ameaças à validade de construção os seguintes itens: (i) o uso de cenários simples e que não refletem a complexidade de aplicações reais. Entretanto, mesmo não contemplando o tamanho e complexidade de aplicações reais, para mitigar tais ameaças, propusemos em nossos cenários contextos que possibilitam exercitar os mesmos tipos de testes necessários em aplicações maiores, e (ii) nossas verificações se baseiam no uso de métodos assertivos simples e, muitas vezes, o uso de verificações mais completas se faz necessário. Contudo, nossa abordagem monitora e coleta informações sobre o agente em teste, registrando suas decisões e estados. Assim, o desenvolvimento de assertivas capazes de verificações mais completas é totalmente possível.

Validade de Conclusão

Uma possível ameaça à validade de conclusão está na métrica escolhida. Escolhemos como métrica para este estudo, o número de falhas detectadas. Entretanto outras métricas importantes devem ser consideradas para a avaliação de um método de geração de casos de teste como, por exemplo, o número de casos de testes gerados, o tempo de execução, nível de cobertura, dentre outras.

		Número de Casos de Teste Gerados																																	
		Addressee				Activation				Deactivation				Punishment				Reward				Deontic Concept				State					
Coverage Criteria	Test Path	N1	N2	N3	T	N1	N2	N3	T	N1	N2	N3	T	N1	N2	N3	T	N1	N2	N3	T	N1	N2	N3	T	N1	N2	N3	T	Total	Total Por Critério
All Goals	G->P1->SG1->P11	1	1	1	3	1	1	1	3	1	1	1	3	1	0	0	1	1	0	0	1	1	1	1	3	3	0	0	3	17	57
	G->P1->SG2->P21	1	1	1	3	1	1	1	3	1	1	1	3	0	1	0	1	0	1	0	1	1	1	1	3	0	2	0	2	16	
	G->P1->SG3->P31	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
	G->P1->SG4->P41	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
Scenario	G->P1->SG1->P11	1	1	1	3	1	1	1	3	1	1	1	3	1	0	0	1	1	0	0	1	1	1	1	3	3	0	0	3	17	17
Capability	G->P1->SG4->P41	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	36
	G->P1->SG4->P42	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
	G->P1->SG4->P43	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
All Plans	G->P1->SG1->P11	1	1	1	3	1	1	1	3	1	1	1	3	1	0	0	1	1	0	0	1	1	1	1	3	3	0	0	3	17	153
	G->P1->SG2->P21	1	1	1	3	1	1	1	3	1	1	1	3	0	1	0	1	0	1	0	1	1	1	1	3	0	2	0	2	16	
	G->P1->SG2->P22	1	1	1	3	1	1	1	3	1	1	1	3	0	1	0	1	0	1	0	1	1	1	1	3	0	2	0	2	16	
	G->P1->SG2->P23	1	1	1	3	1	1	1	3	1	1	1	3	0	1	0	1	0	1	0	1	1	1	1	3	0	2	0	2	16	
	G->P1->SG3->P31	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
	G->P1->SG3->P32	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
	G->P1->SG3->P33	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
	G->P1->SG3->P34	1	1	1	3	1	1	1	3	1	1	1	3	0	0	1	1	0	0	1	1	1	1	1	3	0	0	2	2	16	
	G->P1->SG4->P41	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
	G->P1->SG4->P42	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
	G->P1->SG4->P43	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
Goal-Plan	G->P1->SG1->P11	1	1	1	3	1	1	1	3	1	1	1	3	1	0	0	1	1	0	0	1	1	1	1	3	3	0	0	3	17	153
	G->P1->SG2->P21	1	1	1	3	1	1	1	3	1	1	1	3	0	1	0	1	0	1	0	1	1	1	1	3	0	2	0	2	16	
	G->P1->SG2->P22	1	1	1	3	1	1	1	3	1	1	1	3	0	1	0	1	0	1	0	1	1	1	1	3	0	2	0	2	16	
	G->P1->SG2->P23	1	1	1	3	1	1	1	3	1	1	1	3	0	1	0	1	0	1	0	1	1	1	1	3	0	2	0	2	16	
	G->P1->SG3->P31	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
	G->P1->SG3->P32	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
	G->P1->SG3->P33	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
	G->P1->SG3->P34	1	1	1	3	1	1	1	3	1	1	1	3	0	0	1	1	0	0	1	1	1	1	1	3	0	0	2	2	16	
	G->P1->SG4->P41	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
	G->P1->SG4->P42	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	
	G->P1->SG4->P43	1	1	1	3	1	1	1	3	1	1	1	3	0	0	0	0	0	0	0	0	1	1	1	3	0	0	0	0	12	

Figura 8.19: Detalhamento dos Casos de Teste Gerados para o EC.

Coverage Criteria	Test Path	Número de Casos de Teste Gerados																																				Total	Total Por Critério
		Addressee				Activation				Deactivation				Punishment				Reward				Deontic Concept				State													
		N1	N2	N3	T	N1	N2	N3	T	N1	N2	N3	T	N1	N2	N3	T	N1	N2	N3	T	N1	N2	N3	T	N1	N2	N3	T										
All Goals	G->P1->SG1->P11	1	1	1	**3**	1	1	1	**3**	1	1	1	**3**	1	0	0	**1**	1	0	0	**1**	1	1	1	**3**	2	0	0	**2**									**16**	**32**
	G->P1->SG2->P21	1	1	1	**3**	1	1	1	**3**	1	1	1	**3**	1	0	0	**1**	1	0	0	**1**	1	1	1	**3**	2	0	0	**2**									**16**	
Scenario	G->P1->SG2->P21	1	1	1	**3**	1	1	1	**3**	1	1	1	**3**	1	0	0	**1**	1	0	0	**1**	1	1	1	**3**	2	0	0	**2**									**16**	**16**
Capability	G->P1->SG1->P11	1	1	1	**3**	1	1	1	**3**	1	1	1	**3**	1	0	0	**1**	1	0	0	**1**	1	1	1	**3**	2	0	0	**2**									**16**	**32**
	G->P1->SG2->P21	1	1	1	**3**	1	1	1	**3**	1	1	1	**3**	1	0	0	**1**	1	0	0	**1**	1	1	1	**3**	2	0	0	**2**									**16**	
All Plans	G->P1->SG1->P11	1	1	1	**3**	1	1	1	**3**	1	1	1	**3**	1	0	0	**1**	1	0	0	**1**	1	1	1	**3**	2	0	0	**2**									**16**	**48**
	G->P1->SG1->P12	1	1	1	**3**	1	1	1	**3**	1	1	1	**3**	0	1	0	**1**	0	1	0	**1**	1	1	1	**3**	0	2	0	**2**									**16**	
	G->P1->SG1->P13	1	1	1	**3**	1	1	1	**3**	1	1	1	**3**	0	0	1	**1**	0	0	1	**1**	1	1	1	**3**	0	0	2	**2**									**16**	
Goal-Plan	G->P1->SG1->P11	1	1	1	**3**	1	1	1	**3**	1	1	1	**3**	1	0	0	**1**	1	0	0	**1**	1	1	1	**3**	2	0	0	**2**									**16**	**64**
	G->P1->SG1->P12	1	1	1	**3**	1	1	1	**3**	1	1	1	**3**	0	1	0	**1**	0	1	0	**1**	1	1	1	**3**	0	2	0	**2**									**16**	
	G->P1->SG1->P13	1	1	1	**3**	1	1	1	**3**	1	1	1	**3**	0	0	1	**1**	0	0	1	**1**	1	1	1	**3**	0	0	2	**2**									**16**	
	G->P1->SG2->P21	1	1	1	**3**	1	1	1	**3**	1	1	1	**3**	1	0	0	**1**	1	0	0	**1**	1	1	1	**3**	2	0	0	**2**									**16**	

Figura 8.20: Detalhamento dos Casos de Teste Gerados para o TourAgent.

9
Conclusão e Trabalhos Futuros

Sistemas Multiagentes são adequados à construção de software grande e complexo (Weiss, 1999) (Nwana, 1996). Sua natureza possibilita novas formas de decompor, analisar, projetar e implementar tais sistemas, baseando-se nas características centrais dos agentes: autonomia, habilidade social, comportamento orientado à objetivos, dentre outras (Wooldridge, 2009). Entretanto, lidar com a autonomia dos agentes e sua diversidade de interesses é crucial para garantir uma ordem social desejável (Balke et al., 2013) e o uso de *normas* tem sido comumente adotado como um meio eficiente para regular o comportamento dos agentes (Balke et al., 2013) (Lopez, 2003).

Apesar do desenvolvimento de *sistemas baseados em agentes* representar um mercado em amplo crescimento (Helle et al., 2016), da perspectiva da qualidade do software, ainda existe uma lacuna no que se refere a como tais sistemas podem ser verificados e torna-se indispensável a realização de testes sistematizados de modo a aumentar a confiança e a produtividade, e diminuir os custos do software produzido (Delamaro et al., 2017). Entretanto, as características intrínsecas aos agentes tornam seu teste uma tarefa complexa e desafiadora (Miles et al., 2010).

Abordagens existentes na literatura, tais como: (Abushark et al., 2017) (Abushark et al., 2015) (Abushark et al., 2014) (Thangarajah et al., 2014) esforçam-se em apresentar soluções para aumentar a *controlabilidade* e a *observabilidade* dos testes de agentes e sistemas multiagentes. Contudo, tais abordagens ainda desconsideram vários fatores importantes no que se refere ao teste de agentes BDI normativos.

Diante disso, foi proposta uma infraestrutura para suportar o desenvolvimento e o teste de agentes BDI normativos. Inicialmente propomos o NBDI4JADE, um framework que estende a arquitetura *belief-desire-intention* (Rao et al., 1995), incluindo funções para lidar com o raciocínio normativo, capaz de: (i) perceber novas normas ou a extinção de normas do ambiente; (ii) gerar objetivos em conformidade com as normas adotadas; (iii) deliberar sobre cumprir ou violar normas e, (iv) selecionar comportamentos em conformidade com as normas e objetivos. Para testar os agentes desenvolvidos em NBDI4JADE, propomos um framework o qual chamamos de N-JAT4BDI, ca-

paz de: (i) monitorar o ciclo de raciocínio do agente em teste; (ii) controlar os dados de entrada do teste; (iii) observar os resultados obtidos; (iv) apoiar o desenvolvedor na identificação de falhas e, (v) construir e executar casos de testes. Por fim, para lidar com a variabilidade de comportamentos e sua cobertura, propomos um método para gerar casos de teste, lidando, ainda que moderadamente, com tais problemas. Avaliamos, respectivamente, a eficácia e a eficiência do N-JAT4BDI, assim como a eficácia dos casos de teste gerados pelo método proposto, através de dois estudos apresentados no Capítulo 8.

Como resultado, os estudos realizados mostraram indícios de que, tanto o N-JAT4BDI quanto o método para gerar casos de teste, auxiliam, efetivamente, no desenvolvimento de sistemas baseados em agentes de melhor qualidade e mais confiáveis.

9.1 Limitações do Trabalho

A seguir, são apresentadas as principais limitações da abordagem proposta neste trabalho:

- *Nível de teste*: a abordagem apresentada nesta tese focou no teste unitário de agentes, apesar de o framework proposto permitir a realização de níveis mais abrangentes de teste. O teste unitário, apesar de essencial, não é suficiente para garantir que o sistema funciona, integradamente, como esperado.
- *Teste envolvendo normas conflitantes*: uma situação comum em sistemas normativos é a ocorrência de conflito entre normas. Apesar do NBDI4JADE suportar a construção de agentes capazes de lidar com tais conflitos, o N-JAT4BDI não fornece suporte para testar o comportamento dos agentes nessas condições.
- *Teste envolvendo interlocking entre normas*: nossa abordagem testa o feito de uma única norma sobre o agente. Entretanto, as normas de um sistema não são isoladas umas das outras. Por vezes, a ativação, desativação, cumprimento ou violação de uma norma pode levar a ativação, desativação, cumprimento ou violação de outras normas (Lopez, 2003). Tais situações não são exploradas em nossa abordagem proposta.
- *Modelo de faltas restrito às propriedades normativas*: a existência de um modelo de faltas para auxiliar na definição dos tipos relevantes de falhas e situações em que ocorrerem, colabora para a efetividade do teste. Entretanto, nosso modelo de faltas define falhas relacionadas às propriedades da norma. Outras situações nas quais comumente podem

ocorrer problemas em sistemas normativos como o *conflito* de normas e o *interlocking* entre normas não são tratadas em nosso modelo.

- *Uso de modelos sem propósitos de teste*: O teste baseado em modelos pressupõe o uso de modelos capazes de representar os sistemas nas condições necessárias ao teste. O propósito de ANA-ML é modelar as questões adaptativas do agente normativo. Em nossa abordagem, complementamos as informações dos modelos para permitir lidar com alguns critérios de cobertura.
- *Simplicidade dos métodos verificadores*: a simplicidade dos métodos verificadores propostos para testar o comportamento dos agentes, em alguns casos, não testa totalmente o efeito da norma sobre o comportamento do agente. Por exemplo, se um agente recebe uma recompensa que aumenta em 10 a sua reputação, o método *assertReceiveReward* verifica somente se o agente recebeu a recompensa, não fazendo nenhuma verificação sobre o valor atribuído, o que pode levar a uma falha no comportamento do agente.
- *O teste não considera as ações executadas nos planos*: nossos testes verificam se um plano foi ou não executado em decorrência de uma norma. No entanto, não verificamos se as ações executadas pelo plano estão corretas.
- *Redundância em dados de teste*: Os casos de teste são gerados a partir dos caminhos de teste. Comumente, partes dos caminhos podem ser repetidas em vários caminhos de teste. Uma solução interessante seria considerar a possibilidade de reutilizar os caminhos comuns para melhorar o processo de geração de casos de teste. Nossa abordagem eventualmente gera casos de teste para caminhos inteiros, sem qualquer otimização.
- *Teste de agentes sociais*: Nossa abordagem lida com agentes que priorizam os interesses sociais do sistema, isto é, agentes sociais que cumprem com todas as normas.

9.2 Trabalhos Futuros

Nesta seção, apresentamos alguns trabalhos futuros. No geral, esses trabalhos são ações para corrigir e mitigar as limitações identificadas na presente abordagem, como segue abaixo:

- Expansão do modelo de faltas para contemplar outras condições que podem ser origem de falhas em sistemas multiagentes normativos, por exemplo: o *conflito* e o *interlocking* entre normas;

- Um mecanismo para permitir a customização dos métodos verificadores (assertivas de verificação), possibilitando ao testador personalizar como cada falha deve ser verificada.
- Analisar a viabilidade da utilização de informações históricas de falhas detectadas para otimizar a geração de dados de entrada e dos casos de teste.
- Expandir a abordagem para contemplar o teste de agentes desenvolvidos em outras plataformas.
- Otimização do algoritmo de geração de casos de teste para minimizar o número de casos gerados.
- Considerar o teste das ações dos agentes (dentro dos planos) e não apenas o impacto das normas sobre os comportamentos;
- Utilização de cenários mais complexos, realistas e de diferentes domínios.
- Propor uma ferramenta para visualizar os detalhes das decisões do agente frente a influência de normas que atuam concorrentemente no ambiente.

Referências bibliográficas

[Abushark et al., 2017]ABUSHARK, Y. et al. A framework for automatically ensuring the conformance of agent designs. **Journal of Systems and Software**, Elsevier, v. 131, p. 266–310, 2017.

[Abushark et al., 2014]ABUSHARK, Y. et al. Checking consistency of agent designs against interaction protocols for early-phase defect location. In: **Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems**. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2014. (AAMAS '14), p. 933–940. ISBN 978-1-4503-2738-1. Disponível em: <http://dl.acm.org/citation.cfm?id=2617388.2617395>.

[Abushark et al., 2015]ABUSHARK, Y. et al. Early detection of design faults relative to requirement specifications in agent-based models. In: **Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems**. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2015. (AAMAS '15), p. 1071–1079. ISBN 978-1-4503-3413-6. Disponível em: <http://dl.acm.org/citation.cfm?id=2772879.2773287>.

[Adrion et al., 1982]ADRION, W. R.; BRANSTAD, M. A.; CHERNIAVSKY, J. C. Validation, verification, and testing of computer software. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 14, n. 2, p. 159–192, jun. 1982. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/356876.356879>.

[Ahmad, 2012]AHMAD, A. **An agent-based framework incorporating rules, norms and emotions (OP-RND-E)**. Tese (Doutorado) — Universiti Tenaga Nasional, 2012.

[Alberti et al., 2011]ALBERTI, M. et al. Normative systems represented as hybrid knowledge bases. In: **Computational Logic in Multi-Agent Systems**. Berlin, Heidelberg: Springer, 2011. p. 330–346.

[Ammann e Offutt, 2016]AMMANN, P.; OFFUTT, J. **Introduction to software testing**. [S.l.]: Cambridge University Press, 2016.

[Autefage et al., 2015]AUTEFAGE, V.; CHAUMETTE, S.; MAGONI, D. Comparison of time synchronization techniques in a distributed collaborative swarm sys-

tem. In: IEEE. **Networks and Communications (EuCNC), 2015 European Conference on**. [S.l.], 2015. p. 455–459.

[Bakar e Selamat, 2018]BAKAR, N. A.; SELAMAT, A. Agent systems verification: systematic literature review and mapping. **Applied Intelligence**, Springer, v. 48, n. 5, p. 1251–1274, 2018.

[Balke et al., 2013]BALKE, T. et al. Norms in MAS: Definitions and Related Concepts. In: ANDRIGHETTO, G. et al. (Ed.). **Normative Multi-Agent Systems**. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, (Dagstuhl Follow-Ups, v. 4). p. 1–31. ISBN 978-3-939897-51-4. Disponível em: <http://drops.dagstuhl.de/opus/volltexte/2013/3998>.

[Bergenti et al., 2006]BERGENTI, F.; GLEIZES, M.-P.; ZAMBONELLI, F. **Methodologies and software engineering for agent systems: the agent-oriented software engineering handbook**. [S.l.]: Springer Science & Business Media, 2006.

[Binder, 2000]BINDER, R. **Testing object-oriented systems: models, patterns, and tools**. [S.l.]: Addison-Wesley Professional, 2000.

[Boehm et al., 2005]BOEHM, B.; ROMBACH, H. D.; ZELKOWITZ, M. V. **Foundations of empirical software engineering: the legacy of Victor R. Basili**. [S.l.]: Springer Science & Business Media, 2005.

[Boella et al., 2006]BOELLA, G.; TORRE, L. V. D.; VERHAGEN, H. Introduction to normative multiagent systems. **Computational & Mathematical Organization Theory**, Springer, v. 12, n. 2-3, p. 71–79, 2006.

[Boella e Torre, 2008]BOELLA, G.; TORRE, L. van D. Substantive and procedural norms in normative multiagent systems. **Journal of Applied Logic**, Elsevier, v. 6, n. 2, p. 152–171, 2008.

[Boella e Torre, 2004]BOELLA, G.; TORRE, L. van der. Regulative and constitutive norms in normative multiagent systems. In: **Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning**. AAAI Press, 2004. (KR'04), p. 255–265. ISBN 1-57735-199-1. Disponível em: <http://dl.acm.org/citation.cfm?id=3029848.3029882>.

[Boella e Torre, 2006]BOELLA, G.; TORRE, L. van der. An architecture of a normative system: counts-as conditionals, obligations and permissions. In: ACM. **Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems**. New York, NY, USA: ACM, 2006. p. 229–231.

[Bogdanovych et al., 2009]BOGDANOVYCH, A. et al. Developing virtual heritage applications as normative multiagent systems. In: SPRINGER. **International Workshop on Agent-Oriented Software Engineering**. Berlin, Heidelberg: Springer, 2009. p. 140–154.

[Bordini et al., 2006]BORDINI, R. H.; DASTANI, M.; WINIKOFF, M. Current issues in multi-agent systems development. In: SPRINGER. **International Workshop on Engineering Societies in the Agents World**. Berlin, Heidelberg: Springer, 2006. p. 38–61.

[Bordini et al., 2007]BORDINI, R. H.; HÜBNER, J. F.; WOOLDRIDGE, M. **Programming multi-agent systems in AgentSpeak using Jason**. [S.l.]: John Wiley & Sons, 2007.

[Bordini e Moreira, 2004]BORDINI, R. H.; MOREIRA, A. F. Proving bdi properties of agent-oriented programming languages: The asymmetry thesis principles in agentspeak (l). **Annals of Mathematics and Artificial Intelligence**, Springer, v. 42, n. 1-3, p. 197–226, 2004. Disponível em: <https://doi.org/10.1023/B:AMAI.0000034527.45635.e5>.

[Brat e Jonsson, 2005]BRAT, G.; JONSSON, A. Challenges in verification and validation of autonomous systems for space exploration. In: IEEE. **Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.** Montreal, Que., Canada: IEEE, 2005. v. 5, p. 2909–2914.

[Bratman, 1987]BRATMAN, M. **Intention, plans, and practical reason**. [S.l.]: Harvard University Press Cambridge, MA, 1987.

[Bratman, 1990]BRATMAN, M. E. What is intention? **Intentions in communication**, p. 15–31, 1990.

[Broersen et al., 2001]BROERSEN, J. M. et al. The boid architecture; conflicts between beliefs, obligations, intentions and desires. ACM Press, 2001.

[Bulling e Dastani, 2011]BULLING, N.; DASTANI, M. Verifying normative behaviour via normative mechanism design. In: **Twenty-Second International Joint Conference on Artificial Intelligence**. [S.l.: s.n.], 2011.

[Burnstein, 2006]BURNSTEIN, I. **Practical software testing: a process-oriented approach**. [S.l.]: Springer Science & Business Media, 2006.

[Caire et al., 2004]CAIRE, G. et al. **Multi-agent systems implementation and testing**. [S.l.]: na, 2004.

[Caire, 2007]CAIRE, P. A normative multi-agent systems approach to the use of conviviality for digital cities. In: SPRINGER. **International Workshop on Coordination, Organizations, Institutions, and Norms in Agent Systems**. Berlin, Heidelberg: Springer, 2007. p. 245–260.

[Castelfranchi et al., 2000]CASTELFRANCHI, C. et al. Deliberative normative agents: Principles and architecture. In: **Intelligent Agents VI. Agent Theories, Architectures, and Languages**. [S.l.]: Springer Berlin Heidelberg, 2000. p. 364–378.

[Cheng et al., 2009]CHENG, B. H. C. et al. Software engineering for self-adaptive systems: A research roadmap. In: **Software Engineering for Self-Adaptive Systems**. [S.l.]: Springer Berlin Heidelberg, 2009. p. 1–26.

[Choren e Lucena, 2005]CHOREN, R.; LUCENA, C. Modeling multi-agent systems with anote. **Software & Systems Modeling**, Springer, Berlin, Heidelberg, v. 4, n. 2, p. 199–208, 2005.

[Cialdini e Trost, 1998]CIALDINI, R. B.; TROST, M. R. Social influence: Social norms, conformity and compliance. McGraw-Hill, 1998.

[Clapper et al., 2007]CLAPPER, J. et al. Unmanned systems roadmap 2007-2032. **Office of the Secretary of Defense**, v. 188, 2007. Disponível em: <https://www.hsdl.org/?view&did=481851>.

[Coelho et al., 2007]COELHO, R. et al. Jat: A test automation framework for multi-agent systems. In: IEEE. **2007 IEEE International Conference on Software Maintenance**. Paris, France: IEEE, 2007. p. 425–434.

[Coelho et al., 2006]COELHO, R. et al. Unit testing in multi-agent systems using mock agents and aspects. In: **Proceedings of the 2006 International Workshop on Software Engineering for Large-scale Multi-agent Systems**. New York, NY, USA: ACM, 2006. (SELMAS '06), p. 83–90. ISBN 1-59593-395-6. Disponível em: <http://doi.acm.org/10.1145/1138063.1138079>.

[Conrad et al., 2002]CONRAD, M. et al. Graph transformations for model-based testing. In: **Modellierung**. [S.l.: s.n.], 2002. v. 12, p. 39–50.

[Criado et al., 2011]CRIADO, N.; ARGENTE, E.; BOTTI, V. Open issues for normative multi-agent systems. **AI communications**, IOS Press, v. 24, n. 3, p. 233–264, 2011.

[Cukic, 2001]CUKIC, B. The need for verification and validation techniques for adaptive control system. In: IEEE. **Proceedings 5th International Symposium**

on Autonomous Decentralized Systems. Dallas, TX, USA: IEEE, 2001. p. 297–298.

[Cunha et al., 2015]CUNHA, F. et al. Jat4bdi: An aspect-based approach for testing bdi agents. In: IEEE. **Web Intelligence and Intelligent Agent Technology (WI-IAT), 2015 IEEE/WIC/ACM International Conference on**. Singapore, Singapore: IEEE, 2015. v. 2, p. 186–189.

[Cunha et al., 2018]CUNHA, F. et al. Understanding normative bdi agents behavior. In: **Proceedings of the 30th International Conference on Software Engineering & Knowledge Engineering**. Pittsburgh, PA 15238 USA: KSI Research Inc. and Knowledge Systems Institute Graduate School, 2018. p. 244–247.

[Cunha et al., 2018]CUNHA, F. J. P. da et al. Extending bdi multiagent systems with agent norms. **International Journal of Computer, Electrical, Automation, Control and Information Engineering**, World Academy of Science, Engineering and Technology, v. 12, n. 5, p. 302–309, 2018. ISSN eISSN:1307-6892. Disponível em: <http://waset.org/Publications?p=137>.

[Dastani e Torre, 2004]DASTANI, M.; TORRE, L. van der. Programming boid-plan agents deliberating about conflicts among defeasible mental attitudes and plans. In: IEEE. **Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004.** New York, NY, USA: IEEE, 2004. p. 706–713.

[Delamaro et al., 2017]DELAMARO, M.; JINO, M.; MALDONADO, J. **Introdução ao teste de software**. [S.l.]: Elsevier Brasil, 2017.

[Dignum et al., 2002]DIGNUM, F.; KINNY, D.; SONENBERG, L. From desires, obligations and norms to goals. **Cognitive science quarterly**, Hermes Science Publications, v. 2, n. 3-4, p. 407–430, 2002.

[Erickson e Carroll, 1995]ERICKSON, T.; CARROLL, J. **Scenario-Based Design: Envisioning Work and Technology in System Development**. [S.l.]: New York: Wiley & Sons, 1995.

[Eytani et al., 2008]EYTANI, Y.; TZOREF, R.; UR, S. Experience with a concurrency bugs benchmark. In: IEEE. **2008 IEEE International Conference on Software Testing Verification and Validation Workshop**. Lillehammer, Norway: IEEE, 2008. p. 379–384.

[Fisher et al., 2013]FISHER, M.; DENNIS, L. A.; WEBSTER, M. P. Verifying autonomous systems. **Commun. ACM**, v. 56, n. 9, p. 84–93, 2013.

[Frankl e Weiss, 1993]FRANKL, P. G.; WEISS, S. N. An experimental comparison of the effectiveness of branch testing and data flow testing. **IEEE Transactions on Software Engineering**, IEEE, v. 19, n. 8, p. 774–787, 1993. Disponível em: <http://doi.ieeecomputersociety.org/10.1109/32.238581>.

[Frankl e Weyuker, 1993]FRANKL, P. G.; WEYUKER, E. J. An analytical comparison of the fault-detecting ability of data flow testing techniques. In: IEEE. **Software Engineering, 1993. Proceedings., 15th International Conference on**. Baltimore, MD, USA: IEEE, 1993. p. 415–424.

[Freeman, 1979]FREEMAN, G. Complete latin squares and related experimental designs. **Journal of the Royal Statistical Society: Series B (Methodological)**, Wiley Online Library, v. 41, n. 2, p. 253–262, 1979.

[Gamma, 1995]GAMMA, E. **Design patterns: elements of reusable object-oriented software**. [S.l.]: Pearson Education India, 1995.

[Halatsis et al., 1994]HALATSIS, C. et al. Matoura: Multi-agent tourist advisor. In: **Information and Communications Technologies in Tourism**. [S.l.]: Springer, 1994. p. 140–147.

[Haynes et al., 2017]HAYNES, C. et al. Engineering the emergence of norms: a review. **The Knowledge Engineering Review**, Cambridge University Press, v. 32, 2017.

[Helle et al., 2016]HELLE, P.; SCHAMAI, W.; STROBEL, C. Testing of autonomous systems–challenges and current state-of-the-art. In: WILEY ONLINE LIBRARY. **INCOSE International Symposium**. [S.l.], 2016. v. 26, n. 1, p. 571–584.

[Houhamdi, 2011]HOUHAMDI, Z. Multi-agent system testing: A survey. **International Journal of Advanced Computer**, Citeseer, 2011.

[Hsueh et al., 1997]HSUEH, M.-C.; TSAI, T. K.; IYER, R. K. Fault injection techniques and tools. **Computer**, IEEE, v. 30, n. 4, p. 75–82, 1997.

[Jennings e Wooldridge, 1996]JENNINGS, N.; WOOLDRIDGE, M. Software agents. **IEE review**, IET, v. 42, n. 1, p. 17–20, 1996.

[Kiczales et al., 2001]KICZALES, G. et al. Getting started with aspectj. **Commun. ACM**, ACM, New York, NY, USA, v. 44, n. 10, p. 59–65, out. 2001. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/383845.383858>.

[Kiczales et al., 1997]KICZALES, G. et al. Aspect-oriented programming. In: SPRINGER. **European conference on object-oriented programming**. Berlin, Heidelberg: Springer, 1997. p. 220–242. Disponível em: <https://doi.org/10.1007/BFb0053381>.

[Knobbout et al., 2016]KNOBBOUT, M.; DASTANI, M.; MEYER, J.-J. C. Formal frameworks for verifying normative multi-agent systems. In: **Theory and Practice of Formal Methods**. [S.l.]: Springer, 2016. p. 294–308.

[Kollingbaum, 2005]KOLLINGBAUM, M. J. **Norm-governed practical reasoning agents**. Tese (Doutorado) — University of Aberdeen Aberdeen, 2005.

[Kurd, 2005]KURD, Z. **Artificial neural networks in safety-critical applications**. Tese (Doutorado) — University of York, 2005.

[Lopez, 2003]LOPEZ, F. L. **Social Power and Norms: Impact on Agent Behaviour**. Tese (Doutorado) — University of Southampton, Southampton, United Kingdom, UK, 2003.

[Low et al., 1999]LOW, C. K.; CHEN, T. Y.; RÓNNQUIST, R. Automated test case generation for bdi agents. **Autonomous Agents and Multi-Agent Systems**, Springer, v. 2, n. 4, p. 311–332, 1999.

[Lucena et al., 2004]LUCENA, C. et al. **Software engineering for multi-agent systems II: research issues and practical applications**. Heidelberg, Berlin: Springer, 2004.

[Lucena, 1987]LUCENA, C. J. **Inteligência artificial e engenharia de software**. [S.l.]: Jorge Zahar, 1987.

[Luck et al., 2002]LUCK, M.; D'INVERNO, M. et al. Constraining autonomy through norms. In: ACM. **Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2**. Bologna, Italy, 2002. p. 674–681.

[Mackinnon et al., 2000]MACKINNON, T.; FREEMAN, S.; CRAIG, P. Endo-testing: unit testing with mock objects. **Extreme programming examined**, p. 287–301, 2000.

[Mahmoud et al., 2014]MAHMOUD, M. A. et al. A review of norms and normative multiagent systems. **The Scientific World Journal**, Hindawi Publishing Corporation, v. 2014, 2014. Disponível em: <http://dx.doi.org/10.1155/2014/684587>.

[Manyika et al., 2013]MANYIKA, J. et al. **Disruptive technologies: Advances that will transform life, business, and the global economy**. [S.l.]: McKinsey Global Institute San Francisco, CA, 2013.

[Markiewicz e Lucena, 2001]MARKIEWICZ, M. E.; LUCENA, C. J. d. Object oriented framework development. **Crossroads**, Citeseer, v. 7, n. 4, p. 3–9, 2001.

[Memon et al., 2001]MEMON, A. M.; SOFFA, M. L.; POLLACK, M. E. Coverage criteria for gui testing. **ACM SIGSOFT Software Engineering Notes**, ACM, New York, NY, USA, v. 26, n. 5, p. 256–267, 2001. Disponível em: <https://dl.acm.org/citation.cfm?doid=503209.503244>.

[Meneguzzi e Luck, 2009]MENEGUZZI, F.; LUCK, M. Norm-based behaviour modification in bdi agents. In: **Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 1**. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2009. (AAMAS '09), p. 177–184. ISBN 978-0-9817381-6-1. Disponível em: <http://dl.acm.org/citation.cfm?id=1558013.1558037>.

[Menzies e Pecheur, 2005]MENZIES, T.; PECHEUR, C. Verification and validation and artificial intelligence. **Advances in computers**, Elsevier, v. 65, p. 153–201, 2005.

[Micskei et al., 2012]MICSKEI, Z. et al. A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In: SPRINGER. **KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications**. Berlin, Heidelberg: Springer, 2012. p. 504–513.

[Mifflin, 2000]MIFFLIN, H. The american heritage dictionary of the english language. **New York**, 2000.

[Mikaelian, 2010]MIKAELIAN, T. A real options approach to testing. **MIT**, 2010.

[Miles et al., 2010]MILES, S. et al. Why testing autonomous agents is hard and what can be done about it. In: **URL http://www. pa. icar. cnr. it/cossentino/AOSETF10/docs/miles. pdf. AOSE Technical Forum**. [S.l.: s.n.], 2010.

[Miller et al., 2010]MILLER, T.; PADGHAM, L.; THANGARAJAH, J. Test coverage criteria for agent interaction testing. In: SPRINGER. **International Workshop on Agent-Oriented Software Engineering**. Berlin, Heidelberg: Springer, 2010. p. 91–105.

[Myers et al., 2011]MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing**. Hoboken, New Jersey, USA: John Wiley & Sons, 2011.

[Neto et al., 2010]NETO, B. F. dos S.; SILVA, V. T. D.; LUCENA, C. J. P. de. Using jason to develop normative agents. In: SPRINGER. **Brazilian Symposium on Artificial Intelligence**. Berlin, Heidelberg: Springer, 2010. p. 143–152.

[Neto et al., 2013]NETO, B. F. dos S.; SILVA, V. T. da; LUCENA, C. J. P. de. Developing goal-oriented normative agents: The NBDI architecture. In: **Communications in Computer and Information Science**. [S.l.]: Springer Berlin Heidelberg, 2013. p. 176–191.

[Neto, 2012]NETO, B. S. **Uma abordagem deontica para o desenvolvimento de agentes normativos autônomos**. Tese (Doutorado) — Tese de doutorado. Rio de Janeiro: PUC, Departamento de Informática, 2012.

[Nguyen et al., 2012]NGUYEN, C. D. et al. Evolutionary testing of autonomous software agents. **Autonomous Agents and Multi-Agent Systems**, Springer, v. 25, n. 2, p. 260–283, 2012.

[Nguyen et al., 2007]NGUYEN, D. C.; PERINI, A.; TONELLA, P. A goal-oriented software testing methodology. In: SPRINGER. **International Workshop on Agent-Oriented Software Engineering**. Berlin, Heidelberg: Springer, 2007. p. 58–72.

[Nunes et al., 2011]NUNES, I.; LUCENA, C.; LUCK, M. Bdi4jade: a bdi layer on top of jade. In: **Proc. of the Workshop on Programming Multiagent Systems**. [S.l.: s.n.], 2011. p. 88–103.

[Núñez et al., 2005]NÚÑEZ, M.; RODRÍGUEZ, I.; RUBIO, F. Specification and testing of autonomous agents in e-commerce systems. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 15, n. 4, p. 211–233, 2005.

[Nwana, 1996]NWANA, H. S. Software agents: An overview. **The knowledge engineering review**, Cambridge University Press, v. 11, n. 3, p. 205–244, 1996.

[Oates et al., 1997]OATES, T.; PRASAD, M. N.; LESSER, V. R. Cooperative information-gathering: a distributed problem-solving approach. **IEE Proceedings-Software**, IET, v. 144, n. 1, p. 72–88, 1997.

[Padgham e Winikoff, 2005]PADGHAM, L.; WINIKOFF, M. **Developing intelligent agent systems: A practical guide**. West Sussex PO19 8SQ, England: John Wiley & Sons, 2005.

[Padgham et al., 2013]PADGHAM, L. et al. Model-based test oracle generation for automated unit testing of agent systems. **IEEE Transactions on Software Engineering**, IEEE, v. 39, n. 9, p. 1230–1244, 2013.

[Pěchouček e Mařík, 2008]PĚCHOUČEK, M.; MAŘÍK, V. Industrial deployment of multi-agent technologies: review and selected case studies. **Autonomous agents and multi-agent systems**, Springer, v. 17, n. 3, p. 397–431, 2008.

[Pezzè e Young, 2009]PEZZÈ, M.; YOUNG, M. **Teste e análise de software: processos, princípios e técnicas**. [S.l.]: Bookman Editora, 2009.

[Pokahr et al., 2005]POKAHR, A.; BRAUBACH, L.; LAMERSDORF, W. Jadex: A bdi reasoning engine. In: **Multi-agent programming**. [S.l.]: Springer, 2005. p. 149–174.

[Pouly e Jouanneau, 2012]POULY, J.; JOUANNEAU, S. Model-based specification of the flight software of an autonomous satellite. **Embedded Real Time Software Systems (ERTS 2012)**, 2012.

[Ramberger et al., 2004]RAMBERGER, S.; GRUBER, T.; HERZNER, W. Experience report: Error distribution in safety-critical software and software risk analysis based on unit tests. In: **GI Jahrestagung (1)**. [S.l.: s.n.], 2004. p. 72–76.

[Rao e Georgeff, 1991]RAO, A. S.; GEORGEFF, M. P. Modeling rational agents within a bdi-architecture. **KR**, v. 91, p. 473–484, 1991.

[Rao et al., 1995]RAO, A. S.; GEORGEFF, M. P. et al. Bdi agents: from theory to practice. In: **ICMAS**. Cambridge, MA, EUA: MIT Press, 1995. v. 95, p. 312–319. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.7970>.

[Rehman, 2017]REHMAN, S. U. **An Automated Approach to Model Based Testing of Multi-agent Systems**. Tese (Doutorado) — Capital University, 2017.

[Rehman e Nadeem, 2013]REHMAN, S. U.; NADEEM, A. Testing of autonomous agents: A critical analysis. In: IEEE. **2013 Saudi International Electronics, Communications and Photonics Conference**. Fira, Greece: IEEE, 2013. p. 1–5.

[Rehman e Nadeem, 2015]REHMAN, S. U.; NADEEM, A. An approach to model based testing of multiagent systems. **The Scientific World Journal**, Hindawi Limited, v. 2015, p. 1–12, 2015. Disponível em: <https://doi.org/10.1155/2015/925206>.

[Rehman et al., 2016]REHMAN, S. U.; NADEEM, A.; SINDHU, M. Towards automated testing of multi-agent systems using prometheus design models. In: **The International Arab Journal of Information Technology**. Jordan: Zarqa University, 2016.

[Rouff, 2002]ROUFF, C. A test agent for testing agents and their communities. In: IEEE. **Proceedings, IEEE Aerospace Conference**. Big Sky, MT, USA: IEEE, 2002. v. 5, p. 5–2638.

[Rubino et al., 2005]RUBINO, R.; OMICINI, A.; DENTI, E. Computational institutions for modelling norm-regulated mas: An approach based on coordination artifacts. In: SPRINGER. **International Conference on Autonomous Agents and Multiagent Systems**. Berlin, Heidelberg: Springer, 2005. p. 127–141.

[Schach, 1996]SCHACH, S. R. Testing: principles and practice. **ACM Computing Surveys (CSUR)**, ACM, v. 28, n. 1, p. 277–279, 1996.

[Schumann e Visser, 2006]SCHUMANN, J.; VISSER, W. Autonomy software: V& v challenges and characteristics. In: IEEE. **2006 IEEE Aerospace Conference**. Big Sky, MT, USA: IEEE, 2006. p. 1–6.

[Sen e Agha, 2006]SEN, K.; AGHA, G. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In: SPRINGER. **Haifa verification conference**. Berlin, Heidelberg: Springer, 2006. p. 166–182.

[Sen e Airiau, 2007]SEN, S.; AIRIAU, S. Emergence of norms through social learning. In: **IJCAI**. [S.l.: s.n.], 2007. v. 1507, p. 1512.

[Shaw et al., 2008]SHAW, P. H.; FARWER, B.; BORDINI, R. H. Theoretical and experimental results on the goal-plan tree problem. In: **Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3**. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2008. (AAMAS '08), p. 1379–1382. ISBN 978-0-9817381-2-3. Disponível em: <http://dl.acm.org/citation.cfm?id=1402821.1402877>.

[Silva, 2004]SILVA, V. d. Uma linguagem de modelagem para sistemas multiagentes baseada em um framework conceitual para agentes e objetos. **Title in English: From a conceptual framework for agents and objects to a multi-agent system modeling language]. Ph. D. Thesis. Port. Presentation**, 2004.

[Silva, 2008]SILVA, V. T. da. From the specification to the implementation of norms: an automatic approach to generate rules from norms to govern the behavior of agents. **Autonomous Agents and Multi-Agent Systems**, Springer, v. 17, n. 1, p. 113–155, 2008.

[Silva e Lucena, 2007]SILVA, V. T. da; LUCENA, C. J. de. Modeling multi-agent systems. **Communications of the ACM**, ACM, v. 50, n. 5, p. 103–108, 2007.

[Thangarajah et al., 2014]THANGARAJAH, J. et al. Quantifying the completeness of goals in bdi agent systems. In: IOS PRESS. **Proceedings of the Twenty-first European Conference on Artificial Intelligence**. [S.l.], 2014. p. 879–884.

[Thangarajah et al., 2014]THANGARAJAH, J. et al. Towards quantifying the completeness of bdi goals. In: **Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems**. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2014. (AAMAS '14), p. 1369–1370. ISBN 978-1-4503-2738-1. Disponível em: <http://dl.acm.org/citation.cfm?id=2617388.2617477>.

[Tiryaki et al., 2006]TIRYAKI, A. M. et al. Sunit: A unit testing framework for test driven development of multi-agent systems. In: SPRINGER. **International Workshop on Agent-Oriented Software Engineering**. Berlin, Heidelberg: Springer, 2006. p. 156–173.

[Utting e Legeard, 2010]UTTING, M.; LEGEARD, B. **Practical model-based testing: a tools approach**. [S.l.]: Elsevier, 2010.

[Vasconcelos et al., 2009]VASCONCELOS, W. W.; KOLLINGBAUM, M. J.; NORMAN, T. J. Normative conflict resolution in multi-agent systems. **Autonomous agents and multi-agent systems**, Springer, v. 19, n. 2, p. 124–152, 2009.

[Viana et al., 2016]VIANA, M.; ALENCAR, P.; LUCENA, C. A modeling language for adaptive normative agents. In: **Multi-Agent Systems and Agreement Technologies**. [S.l.]: Springer, 2016. p. 40–48.

[Voas e McGraw, 1997]VOAS, J. M.; MCGRAW, G. **Software fault injection: inoculating programs against errors**. New York, NY, USA: John Wiley & Sons, Inc., 1997. ISBN 0-471-18381-4.

[Voas e Miller, 1995]VOAS, J. M.; MILLER, K. W. Software testability: The new verification. **IEEE software**, IEEE, v. 12, n. 3, p. 17–28, 1995.

[Weiss, 1999]WEISS, G. **Multiagent systems: a modern approach to distributed artificial intelligence**. Cambridge, MA, EUA: MIT press, 1999. ISBN 978-0262731317.

[Winikoff, 2005]WINIKOFF, M. Jack intelligent agents: an industrial strength platform. In: **Multi-Agent Programming**. [S.l.]: Springer, 2005. p. 175–193.

[Winikoff e Cranefield, 2014]WINIKOFF, M.; CRANEFIELD, S. On the testability of bdi agent systems. **Journal of Artificial Intelligence Research**, v. 51, p. 71–131, 2014. Disponível em: <https://doi.org/10.1613/jair.4458>.

[Wooldridge, 2009]WOOLDRIDGE, M. **An introduction to multiagent systems**. Hoboken, Nova Jersey, EUA: John Wiley & Sons, 2009. ISBN 978-0470519462.

[Wooldridge e Jennings, 1995]WOOLDRIDGE, M.; JENNINGS, N. R. Intelligent agents: Theory and practice. **The knowledge engineering review**, Cambridge University Press, v. 10, n. 2, p. 115–152, 1995.

[Young, 2007]YOUNG, H. P. Social norms. Department of Economics (University of Oxford), 2007.

[Zambonelli et al., 2000]ZAMBONELLI, F.; JENNINGS, N. R.; WOOLDRIDGE, M. Organisational abstractions for the analysis and design of multi-agent systems. In: SPRINGER. **International Workshop on Agent-Oriented Software Engineering**. Heidelberg, Berlin: Springer, 2000. p. 235–251.

[Zhang et al., 2007]ZHANG, Z.; THANGARAJAH, J.; PADGHAM, L. Automated unit testing for agent systems. **ENASE**, Citeseer, v. 7, p. 10–18, 2007.

[Zhang et al., 2007]ZHANG, Z.; THANGARAJAH, J.; PADGHAM, L. Model based testing for agent systems. In: **Software and Data Technologies**. Berlin, Heidelberg: Springer, 2007. p. 399–413.

[Zhang et al., 2009]ZHANG, Z.; THANGARAJAH, J.; PADGHAM, L. Automated testing for intelligent agent systems. In: SPRINGER. **International Workshop on Agent-Oriented Software Engineering**. Berlin, Heidelberg: Springer, 2009. p. 66–79.

[Zheng e Alagar, 2005]ZHENG, M.; ALAGAR, V. S. Conformance testing of bdi properties in agent-based software. In: IEEE. **Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific**. Taipei, Taiwan: IEEE, 2005. p. 8–pp.

[Zhou et al., 2008]ZHOU, Y.; TORRE, L. V. D.; ZHANG, Y. Partial goal satisfaction and goal change: weak and strong partial implication, logical properties, complexity. In: INTERNATIONAL FOUNDATION FOR AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS. **Proceedings of the 7th international joint**

conference on Autonomous agents and multiagent systems-Volume 1. [S.l.], 2008. p. 413–420.

A
Artefatos do Experimento

A.1
Formulário de Participação

TERMO DE CONSENTIMENTO DE PARTICIPAÇÃO

Responsável: Francisco José Plácido da Cunha

Este é um convite para você participar voluntariamente do estudo: **N-JAT4BDI: UM ESTUDO EXPERIMENTAL SOBRE O FRAMEWORK DE TESTES**. Por favor, leia atentamente as informações abaixo antes de dar seu consentimento para participar do estudo. Qualquer dúvida será prontamente esclarecida pelo responsável (e-mail: `fcunha@inf.puc-rio.br`).

OBJETIVO E BENEFÍCIOS DO ESTUDO

O objetivo deste estudo é coletar informações para atestar ou não a eficácia e eficiência do uso do N-JAT4BDI para apoiar a identificação de falhas em agentes normativos NBDI4JADE. Os resultados obtidos permitirão direcionar a evolução do framework e avançar no estado da arte sobre testes de sistemas baseados em agentes.

PROCEDIMENTOS

Aplicamos inicialmente um questionário para identificar o perfil do participante. Em seguida, faremos um breve treinamento sobre o uso do N-JAT4BDI e a apresentação dos conceitos necessários para execução das tarefas. Na sequência, serão propostas 4 tarefas relacionadas ao objetivo do estudo e, por último, será aplicado um questionário para obter o "feedback" e impressões do participante. Estimamos que a duração do estudo será de, no máximo, 90 minutos, sendo 30 minutos gastos para preenchimento dos questionários e o treinamento e 60 minutos gastos na execução das tarefas.

DESPESAS/ RESSARCIMENTO DE DESPESAS DO VOLUNTÁRIO

Todos os participantes envolvidos nesta pesquisa são isentos de quaisquer custos.

PARTICIPAÇÃO VOLUNTÁRIA

A participação neste estudo é voluntária tendo plena e total liberdade para abandonar o estudo a qualquer momento, sem quaisquer prejuízos para o participante.

GARANTIA DE SIGILO E PRIVACIDADE

As informações relacionadas ao estudo são confidenciais e qualquer informação divulgada em relatório ou publicação futura será feita de forma codificada, para que a confidencialidade seja mantida. O responsável pelo estudo garante que, sob hipótese alguma, as informações do participante serão divulgadas.

Diante do exposto acima eu, _____, declaro que fui esclarecido sobre os objetivos, procedimentos e benefícios do presente estudo e participo de livre e espontânea vontade do estudo em questão. Foi-me assegurado o direito de abandonar o estudo a qualquer momento, se eu assim o desejar. Declaro também não possuir nenhum grau de dependência profissional ou educacional com o pesquisador responsável (ou seja, o pesquisador responsável não pode me prejudicar de modo algum no trabalho ou nos estudos).

Rio de Janeiro, _____ de _____ de 2018.

Participante: _____

Assinatura: _____

A.2
Formulário de Caracterização do Participante

FORMULÁRIO DE CARACTERIZAÇÃO DO PARTICIPANTE

Responsável: Francisco José Plácido da Cunha

Este formulário tem por objetivo caracterizar sua experiência acadêmica e/ou profissional no desenvolvimento de software baseado em agente, no uso casos de teste e na habilidade de inspecionar códigos. Por favor, responda todas as questões o mais fielmente possível. Toda informação é confidencial.

DADOS DO PARTICIPANTE

Participante: _____

Maior titulação: ( ) Graduação ( ) Especialização ( ) Mestrado ( ) Doutorado

EXPERIÊNCIA NO DESENVOLVIMENTO COM AGENTES

1 = Eu não tenho familiaridade com este assunto. Eu nunca fiz isto.
2 = Eu já li, estudei ou tive aulas sobre isto. Conheço os conceitos e/ou técnicas de forma teórica.
3 = Eu já utilizei isto em exemplos simples na academia.
4 = Eu utilizo isto na prática em alguns projetos pessoais ou na indústria, mas não sou especialista.
5 = Eu sou muito familiar com este assunto. Eu me sentiria confortável fazendo isto.

De acordo com a definição da escala anterior, como você classifica seu conhecimento em relação ao paradigma de agentes? _____

Em relação às seguintes plataformas de agentes de software, informe como você classificaria o seu nível de compreensão de códigos escritos, de acordo com a definição da escala anterior:

- JADE _____
- JADEX _____
- JASON _____
- JACK _____

- BDI4JADE _____
- Outra plataforma (qual plataforma e nível de experiência): _____

EXPERIÊNCIA COM DESENVOLVIMENTO DE CASOS DE TESTES

1 = Eu não tenho familiaridade com este assunto. Eu nunca fiz isto.
2 = Conheço a teoria, mas nunca fiz na prática.
3 = Eu tenho pouca familiaridade. Já utilizei na academia em exemplos simples.
4 = Eu tenho boa familiaridade com este assunto.
5 = Eu sou muito familiar com este assunto. Eu me sentiria confortável fazendo isto.

De acordo com a definição da escala anterior, como você classifica a sua experiência no desenvolvimento de casos de teste para a construção do software? _____

EXPERIÊNCIA NA INSPEÇÃO DE CÓDIGO FONTE

1 = Eu não tenho familiaridade com este assunto. Eu nunca fiz isto.
2 = Conheço a teoria, mas nunca fiz na prática.
3 = Eu tenho pouca familiaridade. Já utilizei na academia em exemplos simples.
4 = Eu tenho boa familiaridade com este assunto.
5 = Eu sou muito familiar com este assunto. Eu me sentiria confortável fazendo isto.

De acordo com a definição da escala anterior, como você classifica a sua experiência na inspeção e investigação de código fonte, localizando e corrigindo *bugs* no desenvolvimento de software? _____

A.3
Tarefas do Experimento

TAREFAS DO EXPERIMENTO CONTROLADO

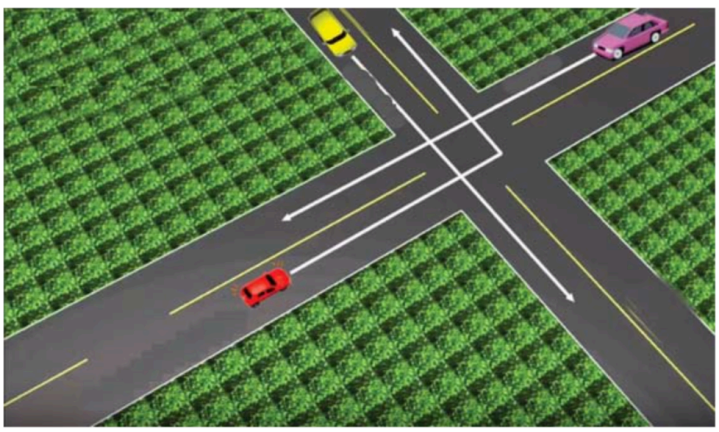
Responsável: Francisco José Plácido da Cunha

Este documento descreve as tarefas que serão executadas no experimento. É fundamental que o participante registre a hora de início e de término da atividade (no formato HH:MM, isto é, hora e minuto). Caso ultrapasse 15 minutos de duração, interrompa a execução da tarefa, registre o tempo final e passe para a próxima tarefa. Tarefas incompletas serão analisadas e terão todos os passos executados considerados.

CENÁRIO DE USO

Considere um cenário onde três carros autônomos chegam simultaneamente a um cruzamento não sinalizado e cujo objetivo é chegar ao seu destino, sem acidentes. Para isso, o carro PINK deve prosseguir em frente após o cruzamento, o carro YELLOW deve prosseguir em frente após o cruzamento e o carro RED deve virar à esquerda no cruzamento. Como não há sinalizações, os carros devem decidir suas ações, considerando as normas de trânsito vigentes.

O **artigo 29** do Código de Trânsito Brasileiro estabelece as seguintes normas: (i) *Norma 1*, os veículos que se deslocam nas vias principais têm a preferência; (ii) *Norma 2*, os veículos que circulam em uma rotatória têm a preferência, e (iii) *Norma 3*, em todos os outros casos, os veículos que vem da direita têm a preferência. Além disso, o **artigo 38**, afirma que antes de mudar de via, o condutor deve parar, ceder a passagem aos pedestres, ciclistas e veículos, respeitando as normas descritas no **artigo 29**.



TAREFA 1

INSPEÇÃO: De acordo com o cenário de uso apresentado e as informações sobre os códigos do agente, identifique a classe, o método e a linha onde devem ser colocadas instruções `System.out`, de forma a possibilitar a verificação do comportamento do agente.

N-JAT4BDI: De acordo com o cenário de uso apresentado, construa um caso de teste para verificar as informações solicitadas (consulte o guia de treinamento).

- Verificar se a Norma 3 está endereçada ao carro vermelho
  `System.out.println("Norma endereçada!");`
- Verificar se a Norma 3 está ativa
  `System.out.println("Norma ativa!");`
- Verificar se a Norma 3 é obrigatória
  `System.out.println("Norma obrigatória!");`

A classe *RedCarAgent* é a classe que implementa o carro vermelho e a classe *RedCarCapability* implementa sua capacidade. Os planos com as ações de virar a esquerda e ficar parado são respectivamente: *LeftPlan* e *StopPlan*. A norma que regula o direito de passagem dos veículos no cruzamento é a *Norma 3*.

HORA DE INÍCIO (HH:MM): ____:____

HORA DE TÉMINO (HH:MM): ____:____

TAREFA 2

INSPEÇÃO: Sabendo que o carro vermelho deveria ser o primeiro a se mover no cruzamento e isso não ocorre, identifique 3 condições que poderiam estar causando tal problema. Considere a especificação do caso de teste e localize no código fonte a classe, o método e a linha onde podemos verificar os problemas.

N-JAT4BDI: Sabendo que o carro vermelho deveria ser o primeiro a se mover no cruzamento e isso não ocorre, construa um caso de teste para testar 3 condições para verificar as informações da especificação do caso de teste.

A classe *RedCarAgent* é a classe que implementa o carro vermelho e a classe *RedCarCapability* implementa sua capacidade. Os planos com as ações de virar a esquerda e ficar parado são respectivamente: *LeftPlan* e *StopPlan*. A norma que regula o direito de passagem dos veículos no cruzamento é a *Norma 3*.

Nº	Caso de Teste	Resultado Esperado
1	O agente RedCarAgent é regulado pela *Norma 3* . O agente RedCarAgent chega em um cruzamento não sinalizado simultâneamente com outros 2 carros.	O agente RedCarAgent deve ser o primeiro carro a se mover, virando a esquerda no cruzamento.

HORA DE INÍCIO (HH:MM): ____:____

HORA DE TÉMINO (HH:MM): ____:____

TAREFA 3

INSPEÇÃO: De acordo com o cenário de uso apresentado, identifique no código do agente a classe, o método e a linha onde devem ser colocadas instruções `System.out`, para verificar as informações solicitadas.

N-JAT4BDI: De acordo com o cenário de uso apresentado, construa um caso de teste capaz de verificar as informações solicitadas (consulte o guia de treinamento).

- Verificar se a Norma 3 está endereçada ao carro rosa
  `System.out.println("Norma endereçada!");`
- Verificar se o plano MoveOnPlan é obrigado a ser executado
  `System.out.println("Plano obrigatório!");`
- Verificar se o plano StopPlan foi executado
  `System.out.println("Plano executado!");`

A classe *PinkCarAgent* é a classe que implementa o carro rosa e a classe *PinkCarCapability* implementa sua capacidade. Os planos com as ações de seguir em frente e ficar parado são respectivamente: *MoveOnPlan* e *StopPlan*. A norma que regula o direito de passagem dos veículos no cruzamento é a *Norma 3*.

HORA DE INÍCIO (HH:MM): ____:____

HORA DE TÉMINO (HH:MM): ____:____

TAREFA 4

Inspeção Manual: De acordo com o cenário de uso apresentado e a especificação de caso de teste abaixo, identifique 3 possíveis problemas que podem estar causando o comportamento inesperado do agente. Localize no código a classe, o método e a linha onde podemos verificar as informações. Sabe-se que o carro amarelo segue em frente ao chegar no cruzamento.

N-JAT4BDI: De acordo com o cenário de uso apresentado e a especificação de caso de teste abaixo, identifique 3 possíveis problemas que podem estar causando o comportamento inesperado do agente. Construa um caso de teste capaz de verificar as informações. Sabe-se que o carro amarelo segue em frente ao chegar no cruzamento

Nº	Caso de Teste	Resultado Esperado
1	O agente YellowCarAgent é regulado pela *Norma 3* . O agente YellowCarAgent chega em um cruzamento não sinalizado simultâneamente com outros 2 carros.	O agente YellowCarAgent deve parar no cruzamento e esperar que o agente RedCarAgent se mova para, em seguida, seguir em frente.

A classe *YellowCarAgent* é a classe que implementa o carro amarelo e a classe *YellowCarCapability* implementa a capacidade do carro amarelo. Os planos com as ações de seguir em frente e ficar parado são respectivamente: *MoveOnPlan* e *StopPlan*. A norma que regula o direito de passagem dos veículos no cruzamento é a *Norma 3*.

HORA DE INÍCIO (HH:MM): ____:____

HORA DE TÉMINO (HH:MM): ____:____

A.4
Formulário de Feedback

FORMULÁRIO DE FEEDBACK DO PARTICIPANTE

Responsável: Francisco José Plácido da Cunha

Este formulário tem por objetivo obter o feedback do participante em relação ao experimento executado. Solicitamos que o participante responda cada pergunta com o máximo de detalhes possível.
Por favor, responda **todas** as questões o mais fielmente possível. Toda informação é confidencial, sendo que seu nome ou quaisquer outros meios de identificação não serão divulgados em nenhuma hipótese.

EXPERIÊNCIA DO PARTICIPANTE

1 Descreva sua experiência ao realizar as tarefas que envolveram a inspeção e investigação do código fonte. Descreva as dificuldades e facilidades ao executar essas tarefas?

2 Descreva sua experiência ao realizar as tarefas que envolveram o uso do N-JAT4BDI. Quais foram as dificuldades e facilidades?

3 Você acha que o uso das assertivas de verificação fornecidos pelo N-JAT4BDI ajudaram a identificar mais rapidamente as falhas no código do agente?

4 Você acha que o uso das assertivas de verificação fornecidos pelo N-JAT4BDI ajudaram a identificar um número maior de falhas no código?

5 Quais as contribuições que o N-JAT4BDI traz para o desenvolvimento de agentes e quais são suas limitações?

A
Produção Científica

Este apêndice apresenta a *produção científica* realizada durante o desenvolvimento da pesquisa desta tese de doutorado. São apresentadas as publicações diretamente relacionadas à abordagem proposta nesta tese, os trabalhos produzidos em colaboração com outros pesquisadores e, por fim, os trabalhos publicados internamente no departamento.

Artigos Relacionados à Tese

1. FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Marx Leles Viana, Tassio Ferenzini Martins Sirqueira, Marcio Ricardo Rosemberg, Carlos José Pereira de Lucena. *Understanding normative bdi agents behavior*. **International Conference on Software Engineering & Knowledge Engineering**, 2018. (Qualis B1)

2. FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Tassio Ferenzini Martins Sirqueira, Marx Leles Viana, Carlos José Pereira de Lucena. *Extending bdi multiagent systems with agent norms*. **International Journal of Computer, Electrical, Automation, Control and Information Engineering – World Academy of Science, Engineering and Technology**, 2018.

3. FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Andrew Diniz da Costa, Marx Leles Viana, Carlos José Pereira de Lucena. *Jat4bdi: An aspect-based approach for testing bdi agents.*In: **IEEE. Web Intelligence and Intelligent Agent Technology (WI-IAT)**, 2015. (Qualis B1)

Publicações em Parceria com Outros Pesquisadores

1. Tassio Ferenzini Martins Sirqueira, Marx Leles Viana, FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Ingrid Nunes, Carlos José Pereira de Lucena. *Data provenance in multi-agent systems: relevance, benefits and research opportunities*. **International Journal of Metadata,**

Semantics and Ontologies (PRINT), v. 13, p. 9-19, 2018. (Qualis B1)

2. Marx Leles Viana, Lauro Caetano, FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Paulo Alencar, Lucena, Carlos José Pereira de Lucena. *Governance in Adaptive Normative Multiagent Systems for the Internet of Smart Things: Challenges and Future Directions*. In: **IEEE Big Data Conference Proceedings**, 2018, Seattle. First International Workshop on the Internet of Things Data Analytics (IoTDA), 2018. p. 1-4.

3. Marx Leles Viana, Paulo Alencar, Everton Guimarães, FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Donald Cowan, Carlos José Pereira de Lucena. *JSAN: A Framework to Implement Normative Agents*. In: **27th International Conference on Software Engineering & Knowledge Engineering**, 2015, Pittsburgh. p. 660-665. (Qualis B1)

4. Marx Leles Viana, Paulo Alencar, Donald Cowan, Everton Guimarães, FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Carlos José Pereira de Lucena. *The Development of Normative Autonomous Agents: an Approach*. In: **IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology**, 2015, Cingapura. (Qualis B1)

5. FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Ricardo de Almeida Venieris, Marx Leles Viana, Carlos José Pereira de Lucena. *Aprendizado de Máquina na Classificação de retinopatia diabética em fundoscopia*. In: **XV CBIS – Congresso Brasileiro de Informática em Saúde**, 2016, Goiânia.

6. Marx Leles Viana, FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Baldoino Neto, Paulo Alencar, Carlos José Pereira de Lucena. *A Framework for Supporting Simulation with Normative Agents*. In: **WESAAC – Workshop-Escola de Sistemas de Agentes, seus Ambientes e Aplicações**, 2015. Niterói – UFF, RJ.

7. FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Marx Leles Viana, Marcio Ricardo Rosemberg, Carlos José Pereira de Lucena. *Verifying the behavior of agents in BDI4JADE with AspectJ*. In: **WESAAC – Workshop-Escola de Sistemas de Agentes, seus Ambientes e apliCações**, 2015. Niterói – UFF, RJ.

8. Ariel Escobar Endara, Marx Leles Viana, FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Carlos José Pereira de Lucena. *Uma abordagem*

baseada em Sistemas Multiagentes para suporte a Telemedicina. In: **WESAAC – Workshop-Escola de Sistemas de Agentes, seus Ambientes e apliCações**, 2015. Niterói – UFF, RJ.

Publicações no Departamento

1. FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Marx Leles Viana, Tassio Ferenzini Martins Sirqueira, Marcio Ricardo Rosemberg, Carlos José Pereira de Lucena. **Understanding normative bdi agents behavior**, 2018.

2. FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Tassio Ferenzini Martins Sirqueira, Marx Leles Viana, Carlos José Pereira de Lucena. **Extending bdi multiagent systems with agent norms**, 2018.

3. FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Carlos José Pereira de Lucena. **Checking the behavior of BDI4JADE agents using an aspect-based approach**, 2017.

4. Marcio Ricardo Rosemberg, FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Roxana Portugal, Joana Pivatelli, Larissa Torres, Ana Maria Moura, Ricardo Venieris, Erica Riello, Bruno Olivieri, Paulo Henrique Alves, Marília Gutierrez Ferreira, Julio Cesar Sampaio Pereira Leite. **Explorando o conceito de software consciente: um estudo ancorado em revisão colaborativa**, 2017.

5. Marcio Ricardo Rosemberg, FRANCISCO JOSÉ PLÁCIDO DA CUNHA, Carlos José Pereira de Lucena, Daniel Schwabe, Marcos Poggi. **An architecture to mitigate buffer overflow attacks, using multiagent system concepts**, 2016.