Pontifícia Universidade Católica
do Rio de Janeiro

Francisco José Plácido da Cunha

JAT4BDI: Uma nova abordagem para testes de agentes deliberativos

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-graduação de Informática da PUC-Rio.

Orientador: Prof. Carlos José Pereira de Lucena

Rio de Janeiro

Dezembro de 2014

Pontifícia Universidade Católica
do Rio de Janeiro

Francisco José Plácido da Cunha

JAT4BDI: Uma nova abordagem para testes de agentes deliberativos

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Informática do Centro Técnico Científico da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

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

Prof. Alessandro Fabricio Garcia
Departamento de Informática - PUC-Rio

Prof. Andrew Diniz da Costa
Departamento de Informática - PUC-Rio

Prof. José Eugenio Leal
Coordenador (a) Setorial do Centro Técnico Científico - PUC-Rio

Rio de Janeiro, 17 de Dezembro de 2014.

Francisco José Plácido da Cunha

Graduou-se em Bacharel em Ciência da Computação pela Universidade Federal Fluminense em 2004.

Ficha Catalográfica

Cunha, Francisco José Plácido.

JAT4BDI: Uma nova abordagem para testes de agentes deliberativos / Francisco José Plácido da Cunha; orientador: Carlos José Pereira de Lucena – Rio de Janeiro PUC, Departamento de Informática, 2014.

v., 82 f,; il. ; 29,7 cm

1. Dissertação (mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Teses. 2. Engenharia de Software 3. Sistemas Multiagentes 4. Abordagens de teste de sistemas multiagentes. I. Lucena, Carlos José Pereira de. II. Pontifícia Universidade Católica do Rio de Janeiro. III. Departamento de Informática IV. Título.

CDD: 004

Agradecimentos

Agradeço a DEUS, pelas oportunidades e por sempre realizar em minha vida coisas muito maiores do que peço ou penso. A glória é tua, Senhor!

Agradeço a minha mãe, Lizete Plácido da Cunha, pelo esforço dispensado para meus estudos, pelo amor incondicional e, principalmente, pela vida de oração dedicada a mim. Obrigado mamãe!

Agradeço ao meu pai, Francisco Lemos da Cunha, pelo investimento, apoio e incentivo em todos os momentos. Obrigado pelo exemplo de luta e trabalho que sempre me ensinou. Vamos lá, vencer nossas batalhas. Muito obrigado, pai!

Agradeço a minha esposa, Carla Michele da Fonseca Soares da Cunha, por estar ao meu lado em todos os momentos, pelo apoio, compreensão e extrema paciência desde sempre. Agradeço a você e ao nosso branquela, João Felipe Soares da Cunha. Muito obrigado!

Agradeço ao Professor Carlos José Pereira de Lucena pelo constante incentivo, confiança e apoio durante o Mestrado. Obrigado professor por me ajudar a ser um aluno melhor.

Tenho muito a agradecer ao Gustavo Robichez de Carvalho e Soeli Teresinha Fiorini, pela oportunidade de ter trabalhado no Laboratório de Engenharia de Software, pelo convide de cursar o Mestrado e pelo apoio dado durante todo o período em que aí trabalhei. De todo coração, muito obrigado!

Não poderia deixar de agradecer a Leonardo Abreu de Barros desde aquela conversa mais séria até este momento. Obrigado pela oportunidade de trabalhar com uma equipe fantástica no Instituto TECGRAF e toda inspiração que isso traz. Muito obrigado, Leo!

Agradeço aos amigos do LES pelo apoio e à PUC-Rio que me deu a oportunidade de me tornar Mestre em Informática exigindo, para isso, apenas dedicação.

Resumo

Cunha, Francisco José Plácido; Lucena, Carlos José Pereira. **JAT4BDI: Uma nova abordagem de testes para agentes deliberativos.** Rio de Janeiro, 2014. 82p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O crescimento e a popularidade da web impulsionaram o desenvolvimento de softwares baseados em rede. O uso de sistemas multiagentes (SMAs) nesse contexto é considerado uma abordagem promissora em vem sendo aplicada em diferentes áreas tais como: segurança, missões ou cenários críticos de negócios, monitoramento avançado de ambientes e pessoas, etc., o que significa que analisar as escolhas que este tipo de software pode fazer torna-se crucial. Contudo, as metodologias propostas até o momento pela Engenharia de Software Orientada a Agentes (AOSE) concentraram seus esforços principalmente no desenvolvimento de abordagens disciplinadas para analisar, projetar e implementar um SMA e pouca atenção tem sido dada a forma como tais sistemas podem ser testados. Além disso, no que se refere a testes envolvendo agentes de software, algumas questões relacionadas à observabilidade e a controlabilidade dificultam a tarefa de verificação do comportamento, tais como: (i) a autonomia do agente em seu processo deliberativo; (ii) o fato das crenças e objetivos do agente estarem embutidos no próprio agente, dificultam a observação e controle do comportamento e; (iii) problemas associados à cobertura dos testes.

Neste trabalho é apresentada uma nova abordagem para testes unitários de agentes BDI escritos em BDI4JADE baseadas na combinação e adaptação das ideias suportadas pelo JAT Framework, um framework de testes para agentes escritos em JADE e no modelo de faltas proposto por Zhang.

Palavras-chave

Agentes BDI; Verificação de Sistemas Autônomos; Teste de Agentes.

Abstract

Cunha, Francisco José Plácido; Lucena, Carlos José Pereira (Advisor). **JAT4BDI: A NEW APPROACH TO TESTING DELIBERATIVE AGENTS.** Rio de Janeiro, 2014. 82p. MSc. Dissertation - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The growth and popularity of the Web has fueled the development of software-based network. The use of multi-agent systems (MAS) in this context is considered a promising approach has been applied in different areas such as security, or mission critical business scenarios, enhanced monitoring of environments and people, etc., which means analyzing the choices that this type of software can become crucial. However, the methodologies proposed so far by the Software Engineering Oriented Agents (AOSE) focused their efforts mainly on developing disciplined approach to analyze, design and implement an SMA and little attention has been given to how such systems can be tested. Furthermore, with regard to tests involving software agents, some issues related to the controllability and observability difficult the task of checking the behavior, such as: (i) the duration of the agent in its decision-making process; (ii) the fact of the agent's beliefs and goals are embedded in the agent itself, hampering the observation and control of behavior; (iii) problems associated with test coverage.

In this research a novel approach for unit testing of agents written in BDI4JADE BDI based on the combination and arrangement of ideas supported by JAT Framework, a framework for testing agents written in JADE and fault model proposed by Zhang is displayed.

Keywords

BDI Agent; Verification Autonomous System; Testing Agent.

Sumário

Lista de figuras

Lista de tabelas

1
Introdução

Sistemas Multiagentes (SMAs) são sociedades nas quais entidades autônomas (agentes), heterogêneas e projetadas individualmente, trabalham em função de objetivos que podem ser comuns ou diferentes (LÓPEZ, 2003). Jennings e Wooldridge definem agentes como entidades situadas em um ambiente e capazes de realizar comportamentos autônomos nesse ambiente para alcançar seus objetivos (JENNINGS e WOOLDRIDGE, 1996).

O crescimento e popularidade da web impulsionaram o desenvolvimento de softwares baseados em rede. Para Zambonelli, o uso de agentes para esses tipos de sistemas é considerado uma abordagem promissora (ZAMBONELLI, JENNINGS, *et al.*, 2001) e vem sendo aplicada em diferentes áreas, tais como: segurança, missões ou cenários críticos de negócios, monitoramento avançado de ambientes e pessoas, etc., o que significa que analisar as escolhas que este tipo de software pode fazer torna-se crucial (FISHER, DENNIS e WEBSTER, 2013).

1.1.
Motivação

Apesar do uso crescente de SMA em cenários críticos, as metodologias propostas até o momento pela Engenharia de Software Orientada a Agentes (AOSE) concentraram seus esforços, principalmente no desenvolvimento de abordagens disciplinadas para analisar, projetar e codificar um SMA e pouca atenção tem sido empregada a forma como tais sistemas poderiam ser testados (CAIRE, COSSENTINO, *et al.*, 2004).

Como mencionado anteriormente, os sistemas multiagentes são aqueles que decidem por si próprios àquilo que devem fazer e quando devem fazer para alcançar um objetivo. Tais sistemas variam no grau de autonomia empregada pelo agente e vão desde sistemas quase totalmente controlados com a ajuda de intervenção humana até aqueles quase totalmente automatizados, ou seja, com o mínimo de intervenção humana. Normalmente, o uso de níveis diferentes de autonomia se justifica por questões de precisão em relação à capacidade

humana e segurança na execução da operação como, por exemplo: (i) acesso a locais de difícil acesso; (ii) em ambientes perigosos; (iii) em atividades longas e repetitivas ou; (iv) que precisam de um baixo tempo de resposta. Tais atividades possuem maior risco quando desempenhadas por humanos devido a fatores como fadiga ou estresse. Contudo, quanto mais autonomia é empregada em um agente, mais rigorosa deve ser a verificação do comportamento desempenhado por este agente (FISHER, DENNIS e WEBSTER, 2013).

Em relação ao desenvolvimento de software baseado em agentes, uma arquitetura amplamente conhecida e recomendada para o projeto de agentes com altos níveis de autonomia é a arquitetura BDI (*belief – desire – intention*). Tal arquitetura foi proposta por Rao e Georgeff (RAO e GEORGEFF, 1995) sendo baseada no modelo filosófico de Bratman (BRATMAN, 1987). Nela, três atitudes mentais (belief, desire e intention) compõem a base do conhecimento e do mecanismo deliberativo dos agentes. Os agentes deliberam sobre o conhecimento que possuem sobre si e sobre o ambiente que estão situados a fim de atingirem seus objetivos.

1.2. Problema

De acordo com o padrão IEEE 610.12, "*a testabilidade do software é o grau com o qual um sistema ou componente facilita o estabelecimento de um critério de teste e sua execução a fim de determinar se tais critérios foram encontrados*" (IEEE 610.12, 1990). Ainda em relação à testabilidade do software, duas características básicas precisam ser consideradas:

Controlabilidade: capacidade de controlar a entrada e o estado interno do componente que está sendo testado.

Observabilidade: capacidade de observar o resultado produzido pelo componente que está sendo testado.

No que se refere a testes envolvendo agentes de software, algumas questões relacionadas à controlabilidade e à observabilidade dos agentes precisam ser cuidadosamente consideradas: (i) um agente é uma entidade autônoma e, consequentemente, pode ser difícil controlar seu comportamento através de uma ferramenta de testes; (ii) as crenças e objetivos do agente estão embutidos no próprio agente, assim, eles não podem ser facilmente observados e controlados por uma ferramenta – quase sempre uma ferramenta de teste pode somente observar um agente através de suas interações com outros

agentes e com o ambiente; (iii) sem a adoção de uma estratégia eficiente para cobertura dos testes, o mesmo certamente torna-se não escalável, dado o número de possibilidades a serem testadas (BINDER, 1999) e (VOAS e MILLER, 1995).

O trabalho de Winikoff e Cranfield apresenta uma análise quantitativa do esforço necessário para cobrir e, consequentemente verificar, todas as possíveis decisões que podem ser tomadas pelo agente. Sua conclusão é enfática ao afirmar que, testar um SMA através da verificação de cada caminho do "espaço comportamental" do agente, ou seja, de cada caminho do conjunto de todos os caminhos possíveis de serem realizados, é inviável (WINIKOFF e CRANFIELD, 2010).

Dessa forma, diante da necessidade de verificar e compreender os comportamentos complexos executados pelo agente, avaliar sua eficácia e do desafio e implicações referentes à testabilidade do mecanismo deliberativo e comportamento emergente, este trabalho concentra-se na tarefa de apoiar o desenvolvedor na construção de casos de teste para agentes BDI. As seguintes questões surgem como motivação para pesquisa deste trabalho:

- *Q_1: Como podemos apoiar o desenvolvimento de sistemas multiagentes através da construção e manutenção de casos de testes para agentes BDI?*
- *Q_2: Como podemos controlar e observar as ações executadas durante o ciclo de raciocínio de agentes BDI através do uso de casos de teste?*

1.3. Limitações das abordagens existentes

Embora seja possível encontrar trabalhos na literatura que abordam e apresentam estratégias para o teste de agentes BDI (NGUYEN, PERINI, *et al.*, 2009) (NUNEZ, RODRIGUEZ e RUBIO, 2005) (NGUYEN, PERINI e TONELLA, 2008) (WINIKOFF e CRANEFIELD, 2010) (LOW, CHEN e RONNQUIST, 1999) (ZHANG, THANGARAJAH e PADGHAM, 2007) (ZHANG, THANGARAJAH e PADGHAM, 2009) nenhum destes trabalhos se preocupa em fornecer mecanismos que auxiliem na identificação de falhas de implementação e na observação do estado interno dos elementos do agente.

Apesar das consideráveis contribuições dos trabalhos mencionados, somente dois (NGUYEN, PERINI, *et al.*, 2009) (LOW, CHEN e RONNQUIST, 1999) tratam de abordagens que oferecem ferramentas para suporte ao teste de

agentes BDI. Mesmo assim, tais ferramentas possuem maior compromisso com estratégias e critérios de cobertura do que controle e observação do estado interno do agente.

1.4. Trabalho proposto e principais contribuições

Tendo em vista o problema existente e as limitações das abordagens atuais em relação às questões que motivaram nossa pesquisa, este trabalho propõe uma nova abordagem para testes unitários de agentes BDI baseados na combinação e adaptação de ideias suportadas por outros trabalhos da literatura de agentes.

Este trabalho apoiou-se no uso de agentes mock e aspectos para testes de sistemas multiagentes conforme proposto no trabalho de Coelho *et al.* (COELHO, KULESZA, *et al.*, 2006), nas ideias do JAT Framework (COELHO, CIRILO, *et al.*, 2007) e no modelo de faltas proposto por Zhang (ZHANG, 2011) e, assim, auxiliar o desenvolvedor de agentes BDI4JADE (NUNES, LUCENA e LUCK, 2011) na construção e manutenção de casos de testes e sua avaliação.

Para corroborar com o estudo realizado foi criada uma ferramenta para servir de prova de conceito da nova abordagem, que chamamos de JAT4BDI. Tal ferramenta tem como diretriz apoiar o desenvolvimento de agentes de software através da construção de casos de testes. Nesta atividade, o desenvolvedor pode contar com um conjunto de métodos que auxiliam na verificação das decisões tomadas e na observação do estado interno do agente durante a execução de seu ciclo de raciocínio.

1.5. Organização do trabalho

Além do capítulo de introdução, este documento apresenta mais cinco capítulos que estão organizados da seguinte forma. O Capítulo 2 apresenta a terminologia utilizada e os conceitos básicos necessários à compreensão do trabalho. O Capítulo 3 apresenta a solução proposta e detalhes referentes à implementação da ferramenta utilizada como prova de conceito da abordagem proposta. O Capítulo 4 apresenta exemplos de cenários de uso e sua construção na ferramenta JAT4BDI. O Capítulo 5 confronta os trabalhos relacionados com a abordagem proposta nesta dissertação. Finalmente o Capítulo 6 apresenta as

considerações finais a respeito do trabalho, suas contribuições e propostas futuras.

2
Conceitos básicos

Este capítulo apresenta os conceitos básicos necessários ao entendimento desta dissertação. Ele contempla uma relação com a terminologia utilizada no trabalho (Seção 2.1), uma visão geral sobre agentes de software e suas características (Seção 2.2), uma apresentação sobre de teste de software em geral (Seção 2.3) e sobre testes em SMA (Seção 2.4). São apresentados ainda os frameworks do JAT (Seção 2.5) e do BDI4JADE (Seção 2.7) utilizados na solução proposta deste trabalho.

2.1.
Terminologia utilizada

Esta seção apresenta os principais conceitos, termos e definições relacionados ao teste de software utilizados nesta dissertação, de acordo com o "Standard Glossary of Terms used in Software Testing v.2.4, July, 2014", documento de referência do International Software Testing Qualifications Board (ISTQB). Outros termos e conceitos importantes que, de alguma forma corroboram com o entendimento do trabalho, também são apresentados e definidos nessa seção.

Casos 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 (CARROLL, 1995).

Defeito: falha em um componente ou sistema que pode fazer com que tal componente ou sistema pare de executar a sua função. Um defeito, se

encontrado durante a execução, pode provocar o fracasso da operação desempenhada pelo sistema ou componente;

Erro: uma ação humana que produz um resultado incorreto;

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 execução de um teste sobre o componente ou sistema em teste, produzindo um resultado real;

Falha: Desacordo do componente ou sistema a partir da sua entrega esperada, serviço ou resultado;

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 a partir de, um defeito. A injeção de falhas destina-se a imitar as 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 do software em 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: Testes com base em uma análise da estrutura interna do componente ou sistema;

Teste de Caixa Preta: Testes, 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.

Em relação aos componentes dos agentes e termos relacionados, foram adotadas as definições utilizadas por (PADGHAM e WINIKOFF, 2004).

Belief (crença): É algum aspecto do conhecimento ou informação do agente sobre o ambiente, sobre si próprio ou sobre outros agentes;

Desire/Goal (desejo/objetivo): um desejo do agente / aquilo que se tem como objetivo a atingir;

Event (evento): é a ocorrência significativa de algo que o agente deve responder de alguma forma;

Intention (intenção): é um desejo com o qual o agente se comprometeu e se empenha em alcançar;

Plan (plano): é uma sequência de ações para alcançar um objetivo;

Base de conhecimento: é o conjunto de todas as informações que o agente possui sobre si mesmo, sobre o ambiente em que está situado ou sobre outros agentes;

Ciclo de raciocínio: conjunto de passos executados pelo agente que apoiam o mecanismo deliberativo na tentativa de alcançar um objetivo;

Mecanismo deliberativo: consiste no processo de como os agentes selecionam e descartam os planos que devem ser executados e em que ordem;

Além das definições já apresentadas, segue abaixo uma relação de outros termos e abreviaturas utilizados nesta dissertação:

AOSE: Agent-Oriented Software Engineering;

AUT: Agent Under Test – representa o agente que está sendo testado;

JAT: Jade Agent Testing Framework – framework para teste de agentes;

Mock Agent: agente que simula o comportamento de um agente real;

SMA: Sistema Multiagentes;

2.2. Agentes

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, tais como apresentados em (LUCENA, 1987) e (CHOREN e LUCENA, 2005). A tecnologia de agentes de software tem se mostrado mais adequada no objetivo de tratar melhor a complexidade inerente a tais sistemas (LUCENA, 1987) (CHOREN e LUCENA, 2005) (WEISS, 1999) (SILVA e LUCENA, 2007).

2.2.1. Visão geral

Ainda não existe um consenso universal sobre a definição de um agente embora, a apresentada por Wooldridge e Jennings seja cada vez mais adotada pelos pesquisadores da área (PADGHAM e WINIKOFF, 2004).

"*Um agente é um sistema de computador que está situado em um ambiente e que é capaz de realizar ações autônomas neste ambiente a fim de alcançar os objetivos projetados.*" (WOOLDRIDGE e JENNINGS, 1995). A Figura 1 ilustra uma representação da definição de agente.

Figura 1 – O agente atua e sofre influência do seu ambiente

Wooldridge e Jennings caracterizam as principais propriedades dos agentes, como segue (WOOLDRIDGE e JENNINGS, 1995):

Autonomia: agentes devem ser capazes de solucionar problemas delegados a eles sem a intervenção direta de humanos ou de outros agentes, ou seja, devem 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 de software, a fim de resolver seus problemas.

Reativo: agentes devem perceber as modificações ocorridas em seu ambiente e responder em tempo hábil a tais modificações.

Proatividade: agentes não devem simplesmente agir em resposta ao seu ambiente, eles devem ser capazes de visualizar oportunidades, manter um comportamento direcionado ao alcance de suas metas e tomar a iniciativa quando apropriado.

2.2.2. Agentes racionais

Wooldridge faz distinção entre um agente e um agente racional ou inteligente afirmando que este último precisa, necessariamente, ser ainda mais reativo, proativo e social (WOOLDRIDGE, 2002).

Neste trabalho, estamos interessados no teste de agentes racionais na busca de seus objetivos. Ser racional significa que um agente não deve fazer coisas "estúpidas", tais como comprometer-se, simultaneamente, com duas atividades conflitantes como, por exemplo, planejar gastar uma quantidade de dinheiro nas férias e, ao mesmo tempo, gastar esse dinheiro na compra de um carro (PADGHAM e WINIKOFF, 2004). Uma análise detalhada do significado de "racional" pode ser encontrada na obra de Bratman (BRATMAN, 1987) a qual constitui a base do modelo de raciocínio dos agentes proposto por Rao e Georgeff (RAO e GEORGEFF, 1991).

Na questão do desenvolvimento de agentes racionais, uma arquitetura amplamente conhecida e utilizada para projetar e implementar tais tipos de agentes é a arquitetura BDI (belief – desire – intention), proposta por Rao e Georgeff (RAO e GEORGEFF, 1995) e baseada no modelo filosófico de Bratman (BRATMAN, 1987). Nela, três atitudes mentais (*belief, desire e intention*) compõem a base do raciocínio e do mecanismo deliberativo dos agentes. Os agentes deliberam sobre o conhecimento que possuem sobre si e sobre o ambiente que estão situados a fim de atingirem seus objetivos.

2.2.3. Sistemas multiagentes

Geralmente, um agente de software não é encontrado sozinho em uma aplicação ou sistema, mas em conjunto com outros agentes, de tipos iguais ou diferentes, formando uma sociedade ou organização (WOOLDRIDGE, 2002). A esta sociedade dá-se o nome de Sistemas Multiagentes (SMA). Logo, um SMA consiste em uma sociedade de agentes capazes de interagir entre si e que, para interagirem com sucesso, são necessárias as habilidades de cooperação, coordenação e negociação entre eles, assim como na sociedade humana (WOOLDRIDGE, 2002).

2.3. Teste de Software

De acordo com Pezzè e Young (PEZZÈ e YOUNG, 2008), 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. Com a Engenharia de Software acontece o mesmo: 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 artefatos construídos de forma regular. Requisitos de qualidade de software usados em um ambiente podem ser muito diferentes e incompatíveis para outro ambiente ou domínio de aplicação, e à medida que o sistema cresce, sua estrutura evolui e, frequentemente, se deteriora (PEZZÈ e YOUNG, 2008).

A verificação do software é uma importante atividade que engloba todo o processo de desenvolvimento e manutenção (ADRION, BRANSTAD e CHERNIAVSKY, 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 erros em todas as etapas do processo de construção do software (SCHACH, 1996).

Contudo, o custo da verificação de 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, 2008).

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 pré-definidas, necessitando projetar uma solução personalizada, que seja adequada ao problema, aos requisitos e ao ambiente (PEZZÈ e YOUNG, 2008).

O objetivo do teste e análise do software é, ou 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, 2008).

2.4. Testes em Sistemas Multiagentes

Como mencionado na seção 2.2.1, agentes são entidades que possuem as propriedades: autonomia, reatividade, proatividade, habilidade social, dentre outras. Um SMA é um sistema composto de múltiplos agentes autônomos que interagem uns com os outros em um ambiente a fim de alcançar seus objetivos individuais e do SMA como um todo. Usualmente, um SMA também é um sistema distribuído e descentralizado onde os agentes podem estar localizados fisicamente em diferentes computadores (ROUFF, 2002).

O teste de agentes de software está relacionado à capacidade de testar as propriedades particulares dos agentes. Algumas dessas propriedades, tais como autonomia e proatividade, tornam o teste de agentes de software uma tarefa desafiadora e guiam questões como (ROUFF, 2002):

Agentes são distribuídos e assíncronos: Agentes executam em paralelo e de forma assíncrona. Um agente pode ter de esperar por outros agentes para realizar seus objetivos pretendidos. Pode acontecer também, do agente funcionar corretamente quando se encontra sozinho e incorretamente ao ser colocado em comunidade, ou vice-versa. As ferramentas de testes devem ter uma visão global sobre a distribuição dos agentes, além do conhecimento local e individual de cada um, a fim de decidir se o sistema está funcionando de acordo com as especificações (CACCIARI e RAFIQ, 1999).

Agentes possuem autonomia: Agentes são entidades autônomas, o que significa que, a mesma entrada de um teste pode resultar em diferentes comportamentos, em diferentes execuções uma vez que os agentes podem atualizar sua base de conhecimento entre as duas execuções, ou podem aprender a partir de execuções anteriores resultando em diferentes decisões para situações semelhantes.

Agentes trocam mensagens: Agentes se comunicam através da troca de mensagens. Técnicas tradicionais de teste, envolvendo a invocação de métodos, não podem ser aplicadas diretamente, pois os agentes podem adotar estratégias próprias (como simplesmente não responder a uma mensagem recebida) para o envio e recebimento de mensagens.

Agentes sofrem a influência de fatores ambientais e normativos: O ambiente e as restrições (normas, regras, leis, etc.) são fatores importantes que influenciam e governam o comportamento dos agentes. Diferentes configurações no ambiente podem afetar o resultado dos testes. Logo, o ambiente e os fatores

restritivos são relevantes e devem ser considerados no projeto de testes dos agentes.

De acordo com Nguyen, Perini, e Tonella, o teste em agentes de software pode ser classificado em diferentes níveis: *teste de unidade*, *teste do agente*, *teste de integração*, *teste de sistema* e *teste de aceitação* (NGUYEN, PERINI, e TONELLA, 2007). Cada nível é descrito a seguir:

***Teste de Unidade**:* testa todas as partes e elementos que compõem um agente tais como: planos, base de conhecimento, eventos ocorridos, etc., certificando-se de que, individualmente, eles trabalham conforme projetado.

***Teste de Agente**:* testa a integração dos diferentes elementos do agente; verifica a capacidade do agente em cumprir seus objetivos ou perceber e alterar o ambiente.

***Teste de Integração**:* testa a interação entre os agentes, os protocolos de comunicação, a interação e integração dos agentes com o ambiente e com os recursos compartilhados; observa como são realizados os comportamentos coletivos, ou seja, o teste de integração se certifica de que um grupo de agentes e recursos funciona corretamente quando trabalham juntos.

***Teste de Sistema**: t*esta o SMA como um sistema em execução no ambiente de destino; verifica as propriedades emergentes e outras que são esperadas pelo sistema como um todo.

***Teste de Aceitação**:* testa o SMA no ambiente de execução do cliente e verifica se ele atende aos objetivos das partes interessadas.

Esta classificação é semelhante a aquelas utilizadas nos testes de sistemas tradicionais e destina-se tão somente a facilitar compreensão do trabalho (NGUYEN, PERINI, e TONELLA, 2007).

Este trabalho de dissertação se limita ao teste de unidade focando na verificação dos elementos do agente.

2.5. JAT Framework

O JAT (Jade Agent Testing Framework) (COELHO, CIRILO, *et al.*, 2007) é um framework para testes de sistemas multiagentes que se baseia no uso de agentes “mock”. Um agente mock é uma implementação “falsa” de um agente real, criada com o propósito restrito de testar agentes (COELHO, KULESZA, *et al.*, 2006).

2.5.1. Visão Geral do Framework

Um agente mock é responsável por enviar mensagens para o agente em teste (AUT), verificar as respostas deste e verificar se o ambiente foi afetado como esperado. O trabalho de controlar a interação entre os agentes mock e AUT é do elemento Synchronizer, o qual é responsável por definir a ordem em que os mocks interagem com o AUT. Outro elemento presente no framework é o Monitor que é responsável por observar o estado interno dos agentes e suas transições. A Figura 2 retrata todos os participantes que compõem o JAT Framework.

Agent Under Test (AUT): agente cujo comportamento é verificado;

Mock Agent: implementação "falsa" de um agente real que interage com o AUT;

Monitor: responsável por observar transição dos estados internos dos agentes;

Synchronizer: controla a ordem em que os *mocks* interagem com o AUT;

Test Scenario: conjunto de condições a qual o AUT será exposto, para verificar se o mesmo está de acordo com sua especificação nestas condições;

Test Suite: consiste em um conjunto de cenários de teste e um conjunto de operações realizadas para preparar o ambiente de teste antes de iniciar um cenário de teste.

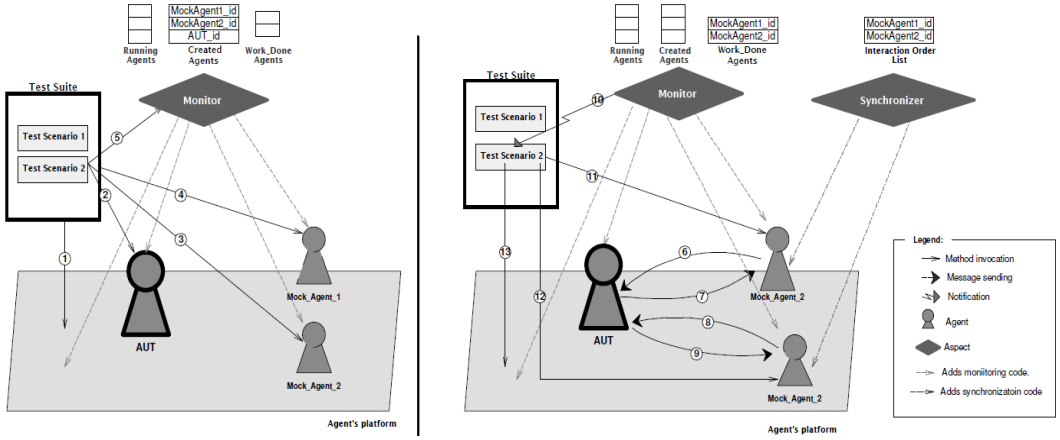

Figura 2 – Fluxo entre os participantes de um teste unitário no JAT.

Cada agente AUT segue o fluxo apresentado na Figura 2. No passo 1, o *test suite* cria os agentes da plataforma e demais elementos necessários para configurar o ambiente de teste. Depois disso, o cenário de teste é iniciado. Cada cenário de teste cria um ou mais agentes mock que interagem com o AUT

(passo 3 e 4) – o número de agentes mock varia de acordo com o cenário de teste que foi definido. Em seguida, é criado o AUT (passo 2). O agente Monitor será notificado quando a interação entre o AUT e os agentes mock terminarem (passo 5). Neste ponto, o AUT e os agentes mock iniciam sua interação. Os agentes mock enviam mensagens para o AUT, que respondem de volta ou vice-versa (passos 6 a 9). Os passos de 6 a 9 são repetidos tantas vezes quanto necessário para executar e concluir os planos dos agentes mock. Por exemplo, o agente mock 1 pode responder três mensagens antes de finalizar sua atividade no teste, e o agente mock 2 pode responder a somente uma mensagem do AUT antes do seu plano terminar. Durante este processo, o Monitor observa a interação dos agentes e mantem o registro das mudanças ocorridas no estado interno dos agentes. Para isso são usadas três listas como ilustrado na figura 2.

Created Agents List: mantém os identificadores dos agentes que foram criados, mas que ainda não estão executando;

Running Agents List: mantém os identificadores dos agentes que estão executando;

WorkDone Agents List: mantém os identificadores dos agentes mock que completaram seu plano;

Quando um agente mock conclui o seu plano, o agente Monitor inclui o identificador deste agente na lista *WorkDone* e então notifica o cenário de teste que a interação entre o agente mock e o AUT foi concluída (passo 10). Esta notificação desbloqueia a execução do cenário de teste que agora é capaz de: (i) perguntar ao agente mock se o AUT atuou ou não como o esperado (passos 11 e 12); e (ii) verificar se o ambiente foi afetado como esperado (passo 13). Sem tais notificações, o cenário de teste não seria capaz de saber quando as interações entre o AUT e os agentes mock terminaram, isto é, quando o cenário de teste chegou ao fim e a verificação (resultado final = resultado esperado) poderia ser feita em um estado intermediário resultando em um falso positivo ou um falso negativo do teste. Esta é a razão porque o Monitor é essencial nessa abordagem.

O Synchronizer é um elemento opcional. Ele só é usado quando o desenvolvedor do teste necessita estabelecer uma ordem na interação entre os agentes mock e o AUT. O Synchronizer mantém uma lista com a ordem da interação que é carregada no começo do cenário de teste. Esta lista contém os identificadores dos agentes mock que tem o direito de interagir com o AUT em um momento específico no cenário de teste.

Na figura 2, o agente mock 1 deve enviar uma mensagem para o AUT antes do agente mock 2. Assim, o cenário de teste é parcialmente implementado pelo plano de cada agente mock, e o Synchronizer é o elemento responsável por compor o cenário de teste. Logo, o Synchronizer é o elemento responsável por definir o momento em que cada agente mock deve executar em um cenário de teste.

Os elementos Monitor e o Synchronizer representam dois interesses transversais da abordagem. Sendo assim, sua implementação está entrelaçada e espalhada por diversas partes do código dos agentes mock, do AUT e da plataforma. Portanto, a estratégia adotada para implementação foi utilizar um aspecto para cada um desses elementos (COELHO, KULESZA e *et al.*, 2006) (GRISWOLD, SHONLE, *et al.*, 2006) (MEYER, 1997).

2.5.2. O modelo de faltas do JAT

Os agentes encapsulam uma estrutura interna complexa composta de planos, objetivos e crenças. Um plano é representado por uma sequência de ações, como por exemplo, o envio de mensagens ou a execução de um procedimento interno, que é executado para alcançar um objetivo específico. Objetivos, assim como as crenças, podem ser expressos como atributos do agente e serem caracterizados por um tipo, um nome e um valor default que pode ser modificado durante a execução do agente (SILVA, CHOREN e LUCENA, 2004). Essas abstrações associadas aos agentes são origens de novas classes de faltas. Além das falhas que podem existir em um sistema OO (decorrentes da implementação em uma linguagem OO), podem existir falhas em: um plano ou crença do agente, falhas nas interações entre os agentes, no comportamento emergente do SMA, falhas nas restrições que regulam um SMA, para mencionar algumas. Por razões práticas, uma abordagem de teste deveria focar em um modelo de faltas específico (BINDER, 1999). O modelo de faltas define um subconjunto de faltas consideradas pela abordagem de teste, que compreende sua capacidade de detecção de falhas e, por isso, delimita o tipo de falta visa detectar (BINDER, 1999).

A abordagem do JAT define como um candidato inicial a modelo de faltas um conjunto de faltas específicas do agente que podem ser manifestadas como uma falha na execução dos planos e, consequentemente, prejudicar o alcance de um objetivo, são eles: (i) falta na ordenação das mensagens; (ii) falta no

conteúdo das mensagens; (iii) falta que aumentam o tempo de resposta da mensagem; (iv) falta nas crenças dos agentes – semelhante as faltas nos atributos da OO; (v) falta nas procedures internas dos agentes – semelhante as faltas nos métodos da OO.

2.6. Modelo de Faltas proposto por Zhang

As novas abstrações introduzidas pelo paradigma de agentes de software definem novas classes de falhas. Uma maneira eficaz para se revelar as falhas do sistema ou componente submetido ao teste é definir para estes um modelo de faltas, que especifica as situações em que, supostamente, é susceptível desta ser encontrada (BINDER, 1999) (MYERS, SANDLER, *et al.*, 2004) (BURNSTEIN, 2002). Uma abordagem de testes, por praticidade, deve focar em um modelo de faltas específico, definindo um subconjunto dos tipos de faltas considerados mais relevantes pela abordagem de teste, delimitando, portanto, os tipos de faltas que se pretende revelar (BINDER, 1999). Neste trabalho, utilizamos uma abordagem de teste dirigida por falhas, onde é utilizada a construção de casos de testes com o objetivo de revelar e identificar falhas decorrentes de implementação.

Adotamos em nossa abordagem o modelo de faltas proposto por Zhang (ZHANG, 2011). Em seu trabalho, Zhang utiliza este modelo para apoiar a elaboração de cenários e a geração automatizada de casos de testes para agentes BDI através da ferramenta Prometheus Design Tool (PDT) (ZHANG, 2011).

Com base no modelo de faltas de Zhang, são definidas as características testáveis, as condições de falha para cada elemento do agente e as falhas que revelam (ZHANG, 2011).

2.6.1. Faltas em planos

Um plano, em linhas gerais, consiste de um fato gerador para o evento que dispara o plano, uma condição de contexto e o corpo do plano. O fato gerador de um plano indica a relevância do plano para o evento. A condição contexto de um plano determina a aplicabilidade do plano em relação às crenças do agente. O corpo do plano possui uma sequência de ações executadas para alcançar um

objetivo específico. Ao considerar um plano como um elemento relevante para o teste do agente, se pretende revelar falhas como:

- *Um plano é, de fato, considerado por um evento?*

Um plano deve ser considerado como aplicável tão logo seja relevante para o evento. Isto requer que o evento seja do tipo correto e também que todos os atributos necessários (fato gerador e condição de contexto) estejam presentes. Se isso não ocorrer, uma falha devido ao fato do plano não considerado aplicável será identificada.

- *A condição de contexto do plano é avaliada corretamente na seleção do plano?*

A condição de contexto de um plano indica em que situação o plano é aplicável. A ausência de uma condição de contexto denota que o plano é sempre aplicável em quaisquer situações. Se o desenvolvedor especifica uma condição de contexto então, se espera que essa condição de contexto seja avaliada como verdadeira em alguma situação e como falsa em outras.

- *O plano dispara somente os eventos que foram especificados para serem disparados?*

Há dois possíveis pontos de falha: um evento que esperávamos que fosse disparado pelo plano e que não acontece; um evento disparado em algum teste que não foi projetado para ocorrer.

- *O plano completa sua execução?*

Durante a execução normal do programa, pode haver algumas razões que levem ao fracasso do plano que está sendo testado como, por exemplo, mudanças ocorridas no ambiente depois que o plano é selecionado. No entanto, em um ambiente de teste controlado, todos os planos que foram selecionados para execução devem finalizar. Assim, se o plano em teste não finalizar devemos considerar uma falha em sua implementação (ZHANG, 2011).

2.6.2. Faltas em planos cíclicos

Na especificação do projeto precisamos considerar a hierarquia dos planos. A interação de um plano com seus subplanos pode formar um ciclo. Um plano cíclico é tratado como um tipo especial de unidade do agente e é testado como se fosse uma entidade única. Alguns critérios precisam ser verificados no teste de planos cíclicos para o agente, tais como:

- *O ciclo ocorre em tempo de execução?*

A especificação de um plano cíclico no projeto implica que pode ocorrer um ciclo durante a execução. Uma falha é revelada quando um ciclo esperado não é formado em tempo de execução. Outra possibilidade de falha acontece na situação oposta, ou seja, é identificada uma execução cíclica mesmo quando nenhum plano cíclico foi especificado no projeto.

- *A execução do plano cíclico termina?*

A execução de um ciclo pode continuar indefinidamente cabendo ao projetista do agente definir as condições de parada da execução. Contudo, é possível que um ciclo seja executado infinitamente por causa de algum erro de implementação. Para esta verificação, uma alternativa é introduzir um limite máximo pré-definido para o número de iterações que ocorrem. Se a execução cíclica excede esse limite, uma falha será identificada (ZHANG, 2011).

2.6.3. Faltas em eventos

As características que guiam o teste de um evento tem relação com o fato de sempre haver um único plano aplicável (cobertura completa) para responder ao evento ou mais de um plano aplicável (sobreposição) para responder a este evento. Diz-se, ainda, que o evento que está sendo testado tem uma cobertura incompleta se não houver um plano aplicável a este em alguma situação.

A ocorrência de sobreposição de planos ou cobertura incompleta para um evento pode ter sido definida pelo projetista e, consequentemente, deve ser permitida pela aplicação. No entanto, tais permissões são geralmente descritas em linguagem natural e podem passar despercebidas no desenvolvimento do agente, merecendo neste caso, a verificação de sua ocorrência, mesmo quando especificadas.

Dessa forma, duas características devem ser consideradas ao testar um evento:

- *Sempre existe um plano que se aplica ao evento?*

Se não, a especificação do projeto precisa ser verificada para garantir que a cobertura incompleta tenha sido prevista e permitido para o evento. Caso contrário, uma falha será identificada.

- *Existe mais de um plano aplicável para o evento?*

No caso de existir um evento que é tratado por vários planos é preciso verificar se o projetista definiu, de fato, a sobreposição para o evento. Não sendo assim, uma falha foi identificada.

- *Existe um plano que se aplica ao evento e que nunca é executado?*

Nessa condição, o mais provável é que um erro de codificação tenha ocasionado a situação de falha (ZHANG, 2011).

2.6.4. Faltas em crenças

O teste de uma crença visa verificar basicamente dois aspectos da mesma: (i) se a estrutura para o armazenamento dos dados foi projetada conforme especificado; (ii) se a atualização de uma crença dispara corretamente os eventos adequados, quando especificados dessa forma.

A primeira verificação ocorre no nível da aplicação, pois uma crença pode ser estruturada e implementada de diferentes maneiras dependendo da plataforma utilizada. Contudo, deve respeitar a estrutura prevista no projeto do agente. A segunda se destina a verificação dos eventos disparados pela alteração do conteúdo da crença, sua remoção da base de conhecimento do agente ou a inclusão de uma nova crença.

- *A estrutura da crença foi implementada conforme o que foi especificado?*

Esta falta pode ocorrer em duas situações: A primeira acontece quando o desenvolvedor negligencia a codificação de algum elemento da estrutura da crença e a segunda verifica se ocorreu uma falha na implementação de uma crença, apesar desta estar estruturalmente correta.

- *O evento apropriado é disparado na manipulação da crença?*

O teste da crença deve considerar se, ao sofrer uma atualização, a mesma provoca corretamente os efeitos esperados. Assim, se uma crença, ao ter seu valor atualizado dispara um evento, esta situação deve ser verificada pelo teste (ZHANG, 2011).

2.7. BDI4JADE

Embora muitas plataformas de agentes sejam baseadas em uma linguagem de propósito geral como, por exemplo, a linguagem JAVA, muitas dessas plataformas utilizam, na implementação dos agentes, uma linguagem de programação que é própria da plataforma ou processadas em tempo de execução pela plataforma do agente (como uma DSL ou XML). Como consequência, a adoção dessa abordagem restringe o uso de muitos recursos avançados da linguagem base da plataforma, tais como a reflexão e as

anotações, não permitindo sua utilização pelos desenvolvedores. Outro aspecto negativo é que o uso de uma nova linguagem de programação pode dificultar a integração, do ponto de vista da implementação, entre o SMA e tecnologias existentes (NUNES, LUCENA e LUCK, 2011).

Essas limitações motivaram a criação da plataforma de agentes BDI, BDI4JADE a qual se utiliza da infraestrutura robusta e madura da plataforma JADE que não adota a arquitetura BDI (BELLIFEMINE, CAIRE e GREENWOOD, 2007). Dessa forma, foi construído sobre a plataforma JADE, um mecanismo de raciocínio baseado na arquitetura BDI, conforme Figura 3, que permite escrever agentes usando somente a linguagem Java e, eliminando os problemas acima mencionados.

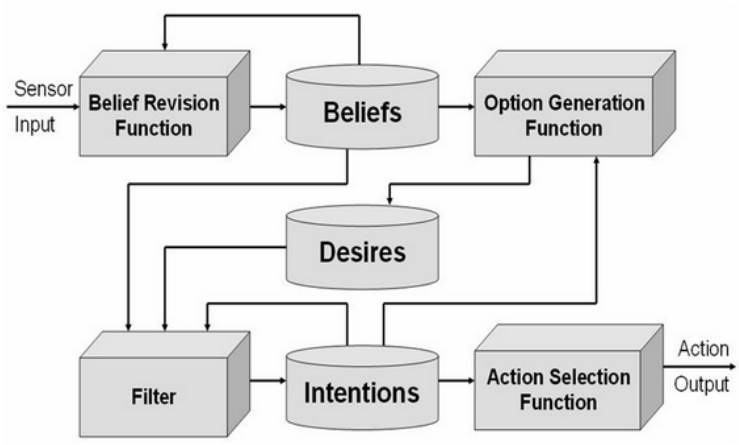

Figura 3 – Uma representação da arquitetura BDI

Alguns conceitos e definições importantes utilizados pelo BDI4JADE são:

BDIAgent: representa um agente da plataforma que segue a arquitetura BDI;

Capability: representa uma capacidade do agente. As capacidades foram introduzidas nos SMAs para apoiar a modularidade e reusabilidade de parte dos comportamentos dos agentes. Nas capacidades está contida a base de crenças e a biblioteca de planos do agente.

Estratégias: São pontos do ciclo de raciocínio do agente que podem ser personalizados.

Goal: representa um desejo que o agente quer alcançar;

Intenção: representa um desejo que o agente está comprometido em alcançar, ou seja, quando um agente tem uma intenção ele irá selecionar planos para tentar alcançar essa intenção.

Belief Base e Belief: representam as características do ambiente ou do próprio agente. As crenças podem ser vistas como o componente informativo do

sistema. A base de crença é um conjunto de crenças, cada uma possuindo um nome e um valor;

Plan Library e Plan: O BDI4JADE fornece uma infraestrutura que permite selecionar um plano de uma biblioteca de planos com a finalidade de alcançar um objetivo. Um plano possui um conjunto de ações para alcançar esse objetivo;

Eventos: O BDI4JADE fornece meios para criar observadores para as crenças e os goals, a fim de que sejam notificados quando forem atualizados;

Cada componente citado é utilizado pela plataforma no ciclo de raciocínio do agente. O ciclo de raciocínio está baseado no algoritmo do interpretador BDI apresentado por Rao e Georgeff, descrito em seis passos (RAO e GEORGEFF, 1995):

Passo 1 - Revising beliefs: neste passo acontece a revisão das crenças do agente.

Passo 2 – Removing Finished Goals: os objetivos que já tenham sido "finalizados", ou seja, que já foram alcançados, os que não são mais desejados ou aqueles considerados como inalcançáveis são removidos nesse passo.

Passo 3 – Generating options: neste passo, são determinados os objetivos atuais do agente (seus desejos). Novos objetivos podem ser criados, objetivos existentes podem ser considerados como não mais desejados e outros ainda podem ser mantidos.

Passo 4 – Removing dropped goals: quando um objetivo, ou um conjunto de objetivos, é determinado como não mais desejado no passo anterior, este é removido do conjunto de objetivos do agente e os observadores são notificados sobre a ocorrência desse evento.

Passo 5 – Deliberating goals: neste passo os objetivos são divididos em dois grupos: (i) aqueles objetivos que o agente tentará alcançar, isto é, suas intenções; e, (ii) aqueles que o agente não tentará alcançar no momento. Estes últimos permanecem sendo um objetivo, mas o agente não está interessado em alcançá-lo no momento.

Passo 6 – Updating goals status: baseado na partição dos objetivos do passo anterior, o status dos mesmos são atualizados. Os objetivos que foram selecionados têm o status atualizado para *"trying to achieve"* enquanto os que não foram selecionados tem o status atualizado para *"waiting"*. Quando um objetivo tem o status *"trying to achieve"*, o agente selecionará os planos adequados para alcançar esse objetivo.

A adoção do BDI4JADE para este trabalho se baseia nas mesmas motivações descritas para sua construção e pela inexistência de quaisquer mecanismos de testes para esta plataforma.

2.8. Sumário

Esse capítulo apresentou uma síntese com a terminologia utilizada pelo texto da dissertação, apresentou ainda uma visão sobre os agentes de software, o teste de software e o teste em agentes de softwares. Foi apresentado ainda, o JAT Framework que serviu de inspiração para o desenvolvimento da ferramenta JAT4BDI. Apresentamos também o modelo de faltas proposto por Zhang e, finalmente, o BDI4JADE.

3
Uma nova abordagem para testes de agentes deliberativos

Neste capítulo, é apresentada uma abordagem para testes unitários de agentes BDI escritos em BDI4JADE, em resposta as limitações dos trabalhos analisados (Seção 1.3 e Seção 5), e que individualmente não atendem a motivação definida neste trabalho. Apresentamos também o JAT4BDI, uma ferramenta construída para servir de prova de conceito da abordagem proposta. A Seção 3.1 descreve as restrições e decisões de projetos adotadas, a Seção 3.2 apresenta uma visão geral da abordagem, a Seção 3.3 apresenta as estruturas envolvidas, a Seção 3.4 apresenta como as partes funcionam em conjunto e, por fim, a Seção 3.5 apresenta em detalhes a ferramenta desenvolvida.

3.1.
Restrições e decisões de projeto

A elaboração de uma abordagem de testes para agentes deliberativos, que neste trabalho está sendo considerado como sinônimo de agentes BDI, não é uma tarefa trivial e algumas restrições foram tomadas para limitar o escopo do trabalho. Nessa seção são descritas as restrições adotadas durante o desenvolvimento da abordagem proposta e a implementação da ferramenta de testes unitários para apoio a construção de casos de testes de agentes BDI.

A primeira decisão de projeto foi a adoção do BDI4JADE como plataforma de desenvolvimento de agentes BDI. As rações para essa decisão residem no fato de que, em BDI4JADE, os agentes BDI são escritos totalmente na linguagem Java, não sendo necessária a utilização de uma linguagem própria da plataforma para o desenvolvimento dos agentes ou arquivos de configuração adicionais.

Restringimos também o escopo dos testes ao nível de unidade. Alguns pesquisadores acreditam que, testar isoladamente cada agente que compõem um SMA não é uma tarefa relevante, pois não garante o funcionamento do sistema quando estes são colocados juntos (WINIKOFF e CRANEFIELD, 2010). Nossa abordagem não vai de encontro a esse pensamento, mas assim como

outros pesquisadores, entendemos que o funcionamento correto e individual do agente é indispensável para o funcionamento do SMA como um todo (ZHANG, THANGARAJAH e PADGHAM, 2007). As abordagens que tratam dos testes de integração e testes de sistema continuam necessárias e essenciais à qualidade final do SMA.

Adotamos para este trabalho o modelo de faltas proposto por Zhang em (ZHANG, 2011) como um guia para a construção dos métodos assertivos que são disponibilizados pela ferramenta. Este modelo de faltas aborda e descreve as situações de falhas dos elementos que compõe o agente.

3.2. Visão geral da abordagem

Nesta seção será apresentada uma visão geral das ideias suportadas pela abordagem proposta. Algumas dessas ideias já foram apresentadas na Seção 2.5, e estão associadas ao JAT Framework. Como já foi mencionado, este trabalho adaptou as ideias utilizadas pelo JAT Framework para possibilitar o teste de agentes BDI escritos em BDI4JADE.

Conforme ilustrado na Figura 4, segue abaixo os principais elementos utilizados pela solução proposta:

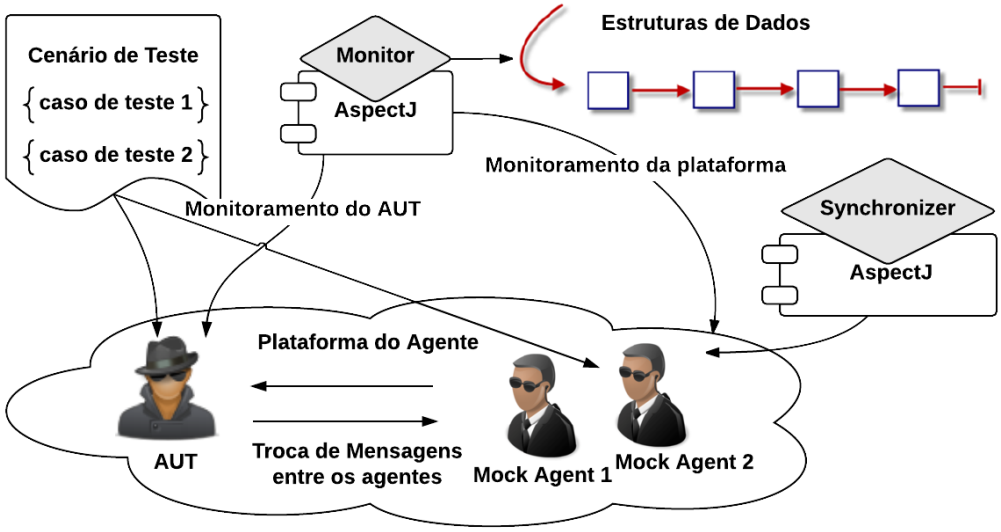

Figura 4 – Participantes do fluxo do teste unitário.

i. **AUT (Agent Under Test)**: representa o agente cujo comportamento será verificado pela execução de um cenário de teste;
ii. **Mock Agent**: elemento criado estritamente para interagir com o AUT, representa uma implementação “falsa” de um agente real;

iii. **Monitor**: elemento responsável por monitorar o ciclo de raciocínio do agente e preencher as estruturas de dados com as informações extraídas durante a execução do caso de teste;

iv. **Synchronizer**: elemento que organiza a ordem de execução dos mocks agents em um cenário de teste e;

v. **Caso de teste**: que define uma condição com a qual o AUT será exposto, isto é, cria o(s) agente(s) Mock que irá(o) interagir com ele e verificar se o AUT obedece a sua especificação nessas condições;

Todos os elementos apresentados já foram vistos na Seção 2.5, quando da apresentação do JAT Framework. Algumas adaptações foram feitas para contemplar o teste de agentes BDI. São elas:

i. **AUT**: enquanto no JAT o AUT representa um agente escrito em JADE, em nossa abordagem o AUT representa um agente BDI escrito em BDI4JADE;

ii. **Monitor**: no JAT, é responsável por monitorar o estado interno dos agentes enquanto que, em nossa abordagem, o monitoramento é realizado no ciclo de raciocínio dos agentes;

iii. **Casos de teste**: em nossa abordagem, o desenvolvedor se utiliza dos métodos assertivos para testar e verificar o comportamento dos agentes;

3.3. As estruturas de dados envolvidas

Como já foi dito, uma das principais características de um agente é a autonomia. Cada agente possui sua própria “thread” de execução e seu comportamento é determinado por um conjunto de inferências feitas mediante suas crenças, objetivos e planos (GARCIA, LUCENA e COWAN, 2004). Com o objetivo de testar efetivamente o agente – considerando que os testes são baseados em alguma forma de comparação do resultado esperado com o resultado produzido – é necessário saber como cada um dos componentes do agente se comportou em cada etapa do ciclo de raciocínio durante a execução. Portanto, a informação sobre o estado das crenças, planos e eventos é útil para verificar se o agente executou o comportamento conforme esperado.

A ideia de criar e manter estruturas contendo as transições dos estados do agente adotada pelo JAT Framework motivou a construção de um novo conjunto

de estruturas de dados capaz de armazenar as informações sobre os componentes internos dos agentes, visando identificar a ocorrência de possíveis falhas de acordo com o modelo de faltas apresentado na Seção 2.6. Para obter esse resultado, devemos adicionar código (nos agentes e plataformas envolvidas, que no nosso caso é o BDI4JADE e o próprio JADE). Tais códigos devem ser adicionados nos pontos onde ocorrem alterações no estado de um componente e nos locais onde são tomadas decisões importantes do mecanismo deliberativo.

Fazendo assim, no entanto, o código adicionado estaria espalhado em muitos pontos e por muitos módulos da plataforma. Percebemos então que, assim como ocorre no JAT, monitorar o ciclo de raciocínio do agente é, naturalmente, um interesse transversal; nesses casos, uma solução, amplamente adotada no que se refere ao monitoramento de interesses transversais, é definir um aspecto para apontar diretamente os locais de execução no agente, isto é, no seu ciclo de raciocínio, e no código da plataforma que representam as transições no estado dos componentes (BRIAND, LABICHE e LEDUC, 2005) (COELHO, DANTAS, *et al.*, 2006). Utilizamos a estratégia de monitorar o ciclo de raciocínio do agente e plataforma através de um aspecto, implementado na linguagem ASPECTJ.

Para armazenar o estado interno dos componentes do agente em teste (AUT) durante a execução do caso de teste, foram criadas as seguintes estruturas de dados conforme Tabela 1.

Tabela 1 – Estruturas de dados preenchidas na execução do AUT.

Estruturas de dados com as informações do AUT	
Conjunto de Crenças	Armazena as crenças do agente.
Conjunto de Planos	Armazena os planos do agente.
Conjunto de Capacidades	Armazena as capacidades do agente.
Conjunto de Eventos	Armazena os eventos disparados.
Conjunto de Mensagens	Armazena as mensagens recebidas e enviadas pelo agente.
Conjunto de Goals	Armazena os objetivos do agente.

As estruturas apresentadas anteriormente são utilizadas para classificar os elementos internos do agente da seguinte forma:

Crenças inseridas: são as crenças inseridas na base de crenças do agente durante a execução;

Crenças removidas: são as crenças removidas da base de crenças dos agentes durante sua execução;

Crenças atualizadas: são as crenças que tiveram seu conteúdo alterado;

Planos executados: são os planos que foram executados pelo agente;

Planos não executados: são os planos da biblioteca de planos do agente que não foram executados;

Objetivos alcançados: são aqueles objetivos que o agente conseguiu atingir;

Objetivos não alcançados: são os objetivos que o agente não conseguiu atingir durante sua execução;

Intenções: são os objetivos que, em algum momento da execução do agente, foram perseguidos pelo agente;

Mensagens enviadas: mensagens enviadas pelo agente (com todas as informações da mensagem);

Mensagens recebidas: mensagens recebidas pelo agente (com todas as informações da mensagem);

Eventos disparados: são os eventos que ocorrem durante a execução do agente;

As estruturas de dados apresentadas acima foram utilizadas pelos métodos assertivos que, ao consultarem seu conteúdo, conseguem verificar se durante a execução do agente, um componente interno do agente alcançou ou não um estado esperado.

3.4. Sincronizando as partes envolvidas

Para sincronizar as partes apresentadas na Seção 3.2 um cenário de teste é escolhido e parcialmente implementado pelo plano executado por cada agente mock que participa deste cenário. Algumas vezes, dependendo do cenário de teste, é necessário definir em que ordem os mocks vão interagir com o AUT. Esta sincronização é, tradicionalmente, tratada através de construções sincronizadas dos trechos de código e das estruturas de dados, nos lugares que precisam ser sincronizados. Semelhante ao que ocorre com as estruturas de dados, o código para sincronização pode estar espalhado pelo código dos agentes mock e pelas principais funcionalidades, impedindo o reuso dos agentes mock em diferentes cenários de teste. Então, de maneira análoga ao

monitoramento do ciclo de raciocínio do agente, a sincronização é também um interesse transversal e pode ser, efetivamente, implementado como um aspecto.

Nossa abordagem, assim como no JAT, define um aspecto para implementar esse interesse. O aspecto Syncronizer é o elemento responsável pela composição do cenário de teste que está parcialmente implementado nos planos de cada agente mock. Tal elemento é responsável por definir o momento em que cada agente mock deve entrar em ação em um cenário de teste. No entanto, o Syncronizer define somente a ordem em que cada agente mock irá interagir com um AUT. Ele não define as ações executadas pelos agentes mock, as quais deverão estar implementadas no comportamento de cada agente mock envolvido no cenário do AUT.

Representar a sincronização dos agentes mock como um aspecto permite o reuso desse agente mock em diferentes cenários de teste sem a necessidade de alteração no código do mesmo.

3.5. JAT4BDI: projeto e implementação

Esta seção apresenta, em detalhes, a ferramenta *JAT4BDI*, desenvolvida para servir como prova de conceito para a abordagem de testes proposta. Serão apresentados os componentes, as estruturas e a utilização da ferramenta em seu propósito de apoiar o desenvolvimento de casos de testes para agentes escritos em BDI4JADE. A Figura 5 apresenta as abordagens e tecnologias utilizadas na proposta de solução do JAT4BDI e suas interdependências.

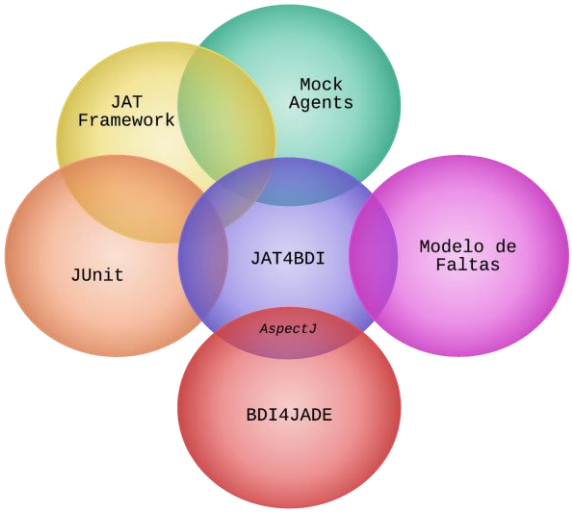

Figura 5 – Dependências entre o JAT4BDI e as abordagens utilizadas.

3.5.1. Detalhes do JAT4BDI

A ferramenta está organizada internamente em quatro pacotes: (i) *o pacote* ***core***; (ii) *o pacote* ***annotations***; (iii) *o pacote* ***aspects*** *e*; (iv) *o pacote* ***faultinjection***, conforme apresentado na Figura 6. O pacote ***core*** contém as classes responsáveis pelo suporte à criação e execução dos casos de testes e pela verificação das informações coletadas do ciclo de raciocínio e do estado interno dos agentes durante a execução do agente em teste (AUT). O pacote ***annotations*** contém as classes (anotações JAVA) associadas a cada falta do modelo de faltas e utilizadas para anotar partes do código onde se deseja injetar uma determinada falha para verificação. O pacote ***aspects*** contém as classes (aspectos desenvolvidos na linguagem AspectJ) responsáveis pelo monitoramento do ciclo de raciocínio dos agentes e pela sincronização na execução dos casos de teste entre o AUT e os agentes mock (vide Seção 3.4). Por fim, o pacote ***faultinjection*** contém uma classe (também se trata de um aspecto implementado na linguagem AspectJ) responsável por injetar falhas no comportamento do agente em teste.

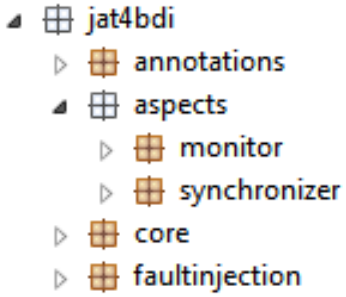

Figura 6 – Estrutura de pacotes utilizados pela ferramenta

As classes contidas no pacote ***core*** fornecem os mecanismos para o apoio a construção e manutenção dos casos de testes. A principal classe do pacote é a *JAT4BDITestCase*. Esta classe estende a classe *TestCase* do JUnit, a qual fornece toda a infraestrutura necessária para a execução de testes unitários automatizados. A classe *JAT4BDITestCase* oferece ainda, métodos para executar os agentes mock e um conjunto de métodos assertivos para verificação do estado do agente, conforme a Figura 7.

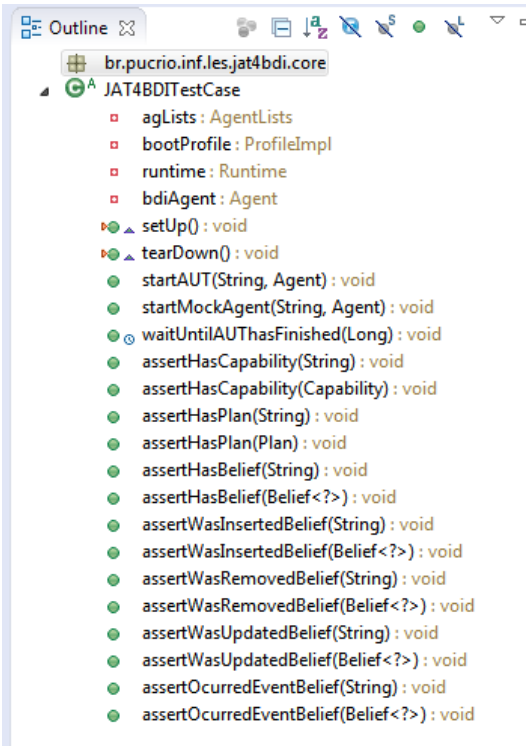

Figura 7 – Atributos e operações fornecidos pela classe *JAT4BDITestCase*

Outra classe importante do pacote ***core*** é a classe *JAT4BDIMockAgent* que implementa o conceito de agentes mock na ferramenta JAT4BDI. Diferentemente dos agentes que estão sendo testados (AUT), os mocks são agentes JADE simples e estendem a classe Agent. Esta decisão baseia-se na simplicidade do comportamento que os agentes mock devem representar. Logo, o plano de um agente mock (*JAT4BDIMockAgent*), é representado por um comportamento do JADE. Outro ponto importante é que os agentes mock devem reportar o resultado da interação com o AUT. Para isso, foi definida uma interface que contém um conjunto de métodos que precisa ser implementado pelo agente que deseja reportar o resultado da interação com o AUT. Portanto, para reportar os resultados, a classe *JAT4BDIMockAgent* implementa a interface *JAT4BDITestReporter*. Outra maneira de verificar o resultado da interação entre o agente mock e o AUT é utilizar os métodos assertivos do AUT para verificar as informações trocadas (conteúdo, performativa, etc.) com os agentes mock.

O pacote ***aspects*** contém as classes responsáveis pelo monitoramento dos agentes e o armazenamento das informações para análise e verificação. As principais classes contidas nesse pacote são: *AgentLists*, *ReasoningCycle* e *Synchronizer*. A classe *AgentLists* mantém o conjunto de estruturas de dados capazes de armazenar as informações do ciclo de raciocínio e do estado interno dos agentes. Essas estruturas são preenchidas durante a execução do teste do

agente através do aspecto *ReasoningCycle*, responsável pelo monitoramento do agente.

A classe *ReasoningCycle* define o aspecto responsável por monitorar e coletar as informações sobre o ciclo de raciocínio e o estado interno do AUT e o preenchimento das estruturas de dados apresentadas anteriormente. Para isso, são interceptados os pontos da plataforma ou do agente para coleta das informações sobre seu funcionamento, conforme figura 8 onde é apresentado o fragmento de código responsável por este monitoramento.

```
12 public aspect ReasoningCycle {
13
14     private AgentLists agLists = AgentLists.getInstance();
15
16     pointcut capabilitySet(Capability capability) : execution(void BDIAgent.addCapability(..)) && args(capability);
17
18     after(Capability capability) : capabilitySet(capability) {
19         agLists.addCapability(capability);
20     }
21
22     pointcut planSet(Plan plan) : execution(void PlanLibrary.addPlan(..)) && args(plan);
23
24     after(Plan plan) : planSet(plan) {
25         agLists.addPlan(plan);
26     }
27
28     pointcut beliefMap(Belief<?> belief) : execution(void BeliefBase.addBelief(..)) && args(belief);
29
30     after(Belief<?> belief) : beliefMap(belief) {
31         agLists.addMapOfBelief(belief);
32         agLists.addInsertedBeliefSet(belief);
33     }
34
35     pointcut removedBeliefSet() : execution(* BeliefBase.removeBelief(..));
36
37     after() returning (Belief<?> belief) : removedBeliefSet() {
38         agLists.addRemovedBeliefSet(belief);
39     }
40
```

Figura 8 – O aspecto ReasoningCycle preencher as estruturas de dados.

Outra classe do pacote ***aspects*** é a *Synchronizer.* Trata-se de um aspecto responsável por “orquestrar” a execução dos agentes mock nos casos de testes. Em seu funcionamento o aspecto adiciona um fragmento de código antes do código de envio de mensagens (linhas 6-15), fazendo com que os mocks verifiquem se é a sua vez de enviar a mensagem para AUT. A classe *OrderList* contém os identificadores dos agentes mock que devem enviar uma mensagem para AUT em um cenário de teste específico, ordenado pela prioridade da interação (linha 7). Assim, se o retorno do método *orderList.checkTurn()* for verdadeiro, o agente pode enviar a mensagem para o AUT, caso contrário, o agente aguarda alguns segundos e verifica novamente se chegou a sua vez de enviar a mensagem conforme o código apresentado nas linhas (linhas 10-12) da Figura 9.

```
1  public aspect Synchronizer {
2
3      pointcut MockSendMessage(...):
4          call(... pucrio.inf.les.jat.core.JAT4BDISynchronizedMockAgent+.send(..)) ...;
5
6      before(...) : MockSendMessage(agent, message) {
7          OrderList orderList = OrderList.getInstance();
8          // O agente somente envia a mensagem se é
9          // sua vez senão ele dorme
10         while(!orderList.checkTurn(agent.getAID())){
11             ...
12             Thread.sleep(500);
13             ...
14         }
15
16     }
17 }
```

Figura 9 – Código parcial do aspecto Synchronizer.

Tem-se ainda o pacote ***annotations*** contendo as anotações JAVA. Essas anotações são responsáveis por configurar as falhas que serão injetadas nos agentes.

Por último, o pacote ***faultinjection*** contém a classe responsável por injetar as falhas no AUT verificando a eficácia das assertivas fornecidas. Trata-se de um aspecto que, durante a execução do caso de teste, injeta no AUT uma falha específica que se deseja verificar.

A Figura 10 apresenta o diagrama de classes com as principais classes da ferramenta. Assim, é possível ter uma visão das principais classes e suas dependências.

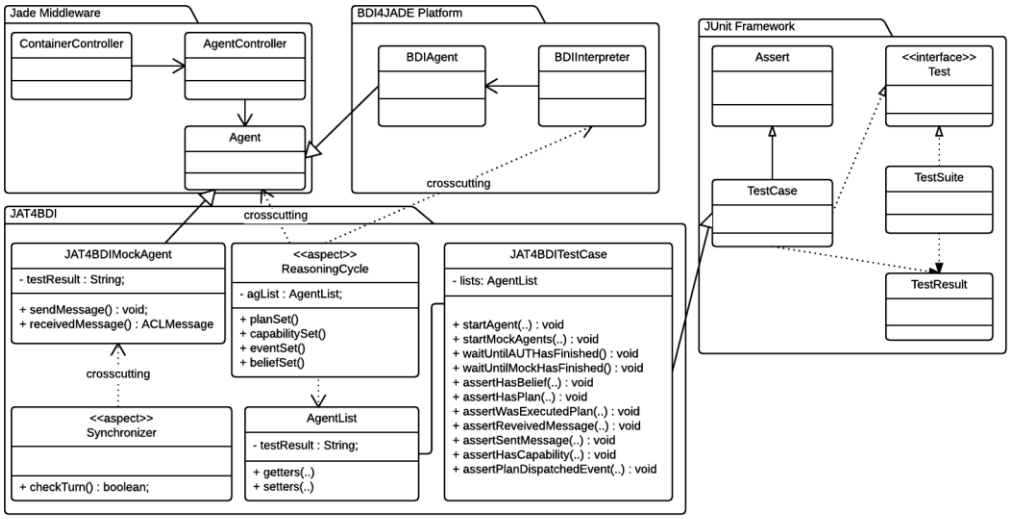

Figura 10 – Diagrama de classe com as principais classes da ferramenta

3.5.2. Assertivas

Para auxiliar o desenvolvimento dos casos de teste assim como a verificação e análise do comportamento, do estado interno e do ciclo de raciocínio do agente em teste, a ferramenta disponibiliza um conjunto de métodos assertivos no estilo JUnit com o propósito de identificar possíveis falhas na implementação do agente considerando, para isso, as condições de falhas apresentadas no modelo de faltas utilizado. Não se trata de assertivas JAVA que são inseridas no código dos agentes e garantem um determinado comportamento ou estado do AUT, mas sim métodos oferecidos pela própria ferramenta para apoiar a identificação das falhas ocorridas ou comportamentos indesejados.

Segue abaixo a lista com os métodos fornecidos pelo JAT4BDI para apoiar o desenvolvimento de casos de testes:

Métodos para configuração do ambiente:

- **`setUp`**: método executado antes do caso de teste, utilizado para configurações;
- **`tearDown`**: executado ao final da execução do caso de teste;
- **`startAUT`**: inicia a execução do AUT;
- **`startMockAgent`**: inicia a execução de um agente mock na plataforma;
- **`waitUntilAUThasFinished`**: aguarda o fim da execução do AUT;
- **`waitUntilMockHasFinished`**: aguarda o fim da execução do agente mock;

Métodos que apoiam a identificação de falhas associadas às crenças:

- **`assertHasBelief`**: verifica se o agente possui uma crença em sua base de crenças;
- **`assertWasInsertedBelief`**: verifica se uma crença foi inserida na base de crenças durante a execução do agente;
- **`assertWasRemovedBelief`**: verifica se uma crença foi removida da base de crenças durante a execução do agente;
- **`assertWasUpdatedBelief`**: verifica se uma crença teve seu valor alterado durante a execução do agente.

Métodos que apoiam a identificação de falhas associadas aos planos:

- **assertHasPlan**: verifica se um plano faz parte da biblioteca de planos do agente.
- **assertWasExecutedPlan**: verifica se um plano foi executado.
- **assertHasAssociatedGoal**: verifica se um plano está associado a um goal (objetivo do agente).

- **assertHasCyclePlan**: verifica se ocorreu um ciclo durante a execução do agente.
- **assertPlanDispatchedEvent**: verifica se o plano disparou um evento.

Métodos que apoiam a identificação de falhas associadas as capacidades:

- **assertHasCapability**: verifica se um agente possui uma capacidade.
- **assertHasInCapability**: verifica se uma capacidade possui um componente que pode ser: uma crença ou um plano.

Métodos que apoiam a identificação de falhas associadas às trocas de mensagens:

- **assertReceivedMessageFrom**: verifica se o agente recebeu uma mensagem de outro agente.
- **assertSentMessageTo**: verifica se o agente enviou uma mensagem para outro agente.
- **assertContentReceivedMessageEquals**: verifica o conteúdo da mensagem recebida pelo AUT de outro agente.
- **assertContentSentMessageEquals**: verifica o conteúdo da mensagem enviada pelo AUT para um outro agente.
- **assertPerformativeReceivedMessageEquals**: verifica a performativa da mensagem recebida pelo agente.
- **assertPerformativeSentMessageEquals**: verifica a performativa da mensagem enviada pelo AUT.

Método que apoia a identificação de falhas associadas aos eventos:

- **assertDispatchedEvent**: verifica se um evento foi disparado por um plano ou por uma crença.

3.5.3. Passos para execução

A construção de casos de teste é realizada através da classe *JAT4BDITestCase.* O desenvolvedor deve construir os casos de testes no estilo JUnit de modo a contemplar toda a especificação do cenário de teste especificada. Para isso, são utilizados os métodos de verificação apresentados na seção anterior. A execução de um cenário de teste segue o fluxo apresentado na Figura 11.

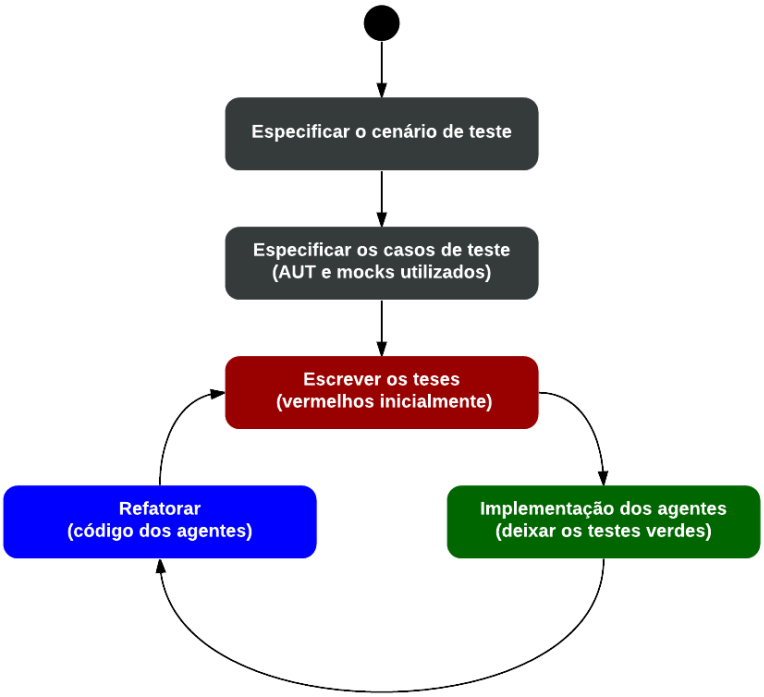

Figura 11 – Workflow para execução dos testes unitários no JAT4BDI.

O procedimento para utilização da ferramenta se inicia com a especificação do cenário de teste a qual desejamos verificar o comportamento. Uma vez especificado o cenário de teste, o próximo passo é especificar os casos de testes que cobrem o cenário especificado. A Tabela 2 apresenta um modelo para o cenário de teste.

Tabela 2 – Modelo para descrição de um cenário de teste.

Agente	**<AUT – Agent Under Test>**
Cenário de Teste (input)	*<Descreve a entrada do cenário de teste que compreende as entradas para o AUT neste cenário e o estado inicial do agente e as outras*

	variáveis do ambiente>
Resultado Esperado	*<Descreve o comportamento esperado para o AUT – se ele deveria enviar uma mensagem ou alterar o ambiente>*

O foco do trabalho proposto está na construção de casos de teste e, dessa forma, o desenvolvedor do caso de teste deve criar uma classe estendendo a classe *JAT4BDITestCase*. Após criar a classe que representa o caso de teste o desenvolvedor poderá implementar os testes utilizando os métodos fornecidos pela ferramenta para criação dos agentes mock, do AUT e verificar o resultado após a execução do agente. A Figura 12 apresenta um exemplo simples para a verificação de uma crença do AUT através do caso de teste *testVerificaExistenciaDeUmaCrenca*, que verifica se o agente possui a crença "*msg*" em sua base de crenças.

```
 8 public class BeliefHelloWorldTestCase extends JAT4BDITestCase {
 9
10     static final long DELAY = 5000l;
11
12     public void testVerificaExistenciaDeUmaCrenca() {
13
14         startAUT("HelloWorld", new BDIAgentHelloWorld());
15
16         waitUntilAUThasFinished(DELAY);
17
18         assertHasBelief("msg");
19
20     }
```

Figura 12 – Um exemplo de execução de um caso de teste no JAT4BDI.

O resultado da execução do teste acima está representado pela figura 13.

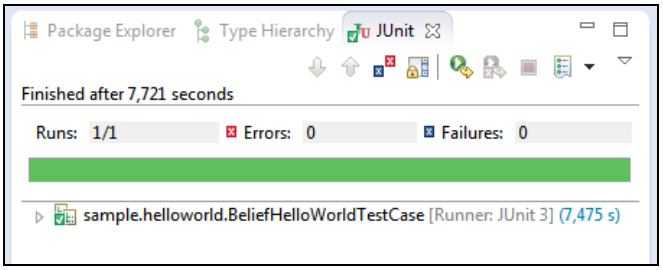

Figura 13 – Resultado da execução de um teste do caso de teste.

3.6. Sumário

Neste capítulo foram apresentadas as restrições e decisões de projeto adotadas assim como uma visão geral da abordagem da solução proposta. Foi apresentado como o uso de agentes mock foi empregado nessa solução. Ainda,

foram apresentadas as estruturas de dados utilizadas para e o mecanismo de sincronismo da abordagem de teste e dos agentes. Finalmente, foi apresentada a ferramenta JAT4BDI, suas classes, o conjunto de assertivas oferecido e como utilizá-las.

4
Cenários de Uso

Este capítulo demonstra a aplicabilidade da abordagem proposta através da ferramenta JAT4BDI. Para isso, foram utilizados alguns cenários de uso.

4.1.
“Toy Problems”: exemplos exploratórios

Esta seção apresenta alguns exemplos de cenários de uso simples cuja finalidade é exercitar a abordagem proposta através da ferramenta JAT4BDI e explorar os recursos oferecidos pela mesma.

4.1.1.
Descrição do cenário de uso – GoHome

Este cenário de uso descreve um agente cujo objetivo é “ir para casa”. Para alcançar seu objetivo o agente pode escolher entre ir de ônibus ou ir de bicicleta dependendo da condição do tempo. Em caso de chuva, o agente deve ir de ônibus e em caso contrário, pode optar também por ir de bicicleta. A decisão de ir de bicicleta implica em um novo objetivo para o agente que é encontrar uma bicicleta disponível. Uma representação para a hierarquia de planos e objetivos do agente é apresentada na Figura 14.

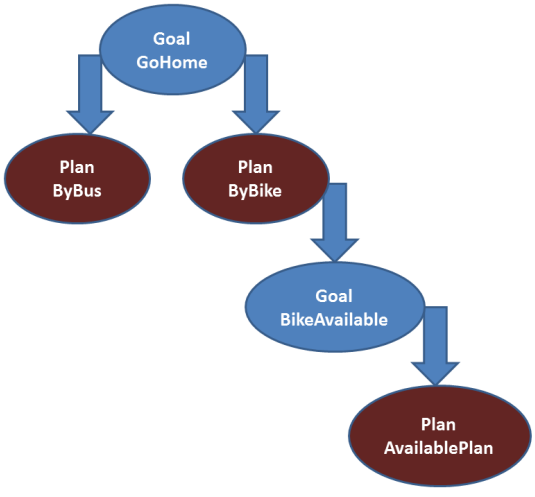

Figura 14 – Hierarquia de objetivos e planos do agente GoHome.

O primeiro passo da tarefa de verificação do comportamento do agente é especificar os cenários de teste que serão automatizados. Dada à simplicidade do exemplo e seu caráter demonstrativo, foi adotada a técnica “error-guessing” para o projeto de casos de teste (MEYER, 1997). Esta técnica utiliza a experiência e a intuição do testador para determinar quais situações podem provocar falhas e assim, projetar casos de testes específicos para essas situações onde podem ocorrer problemas (BEER e RAMLER, 2008). Para esta demostração, foi eleito o cenário de teste apresentado na Tabela 3.

Tabela 3 – Cenário de teste proposto para o agente GoHome.

Agent Under Test	**GoHome**
Cenário de Teste	O agente GoHome sabe que o tempo está chuvoso.
Resultado Esperado	O agente escolhe o ônibus como meio para ir para casa.

Para apoiar a verificação do comportamento do agente no caso de teste descrito na Tabela 3, é proposto uma sequência de casos de teste para serem implementados e executados na ferramenta visando identificar possíveis falhas existentes na implementação do agente conforme Tabela 4.

Tabela 4 – Casos de testes para o agente GoHome.

Item a ser testado	**Objetivo do teste**
Crença para condição do tempo	Verificar se foi criada uma crença para a condição do tempo.
Valor da crença	Verificar se quando o valor da crença é “chuva” o agente escolhe o plano ByBus.
Planos ByBus e ByBike	Verificar se os planos ByBus e ByBike foram adicionados na biblioteca de planos do agente.
Plano ByBus	Verificar se o plano ByBus é executado quando a crença possui o valor “chuva”.
Plano ByBike	Verificar se o plano ByBike é executado quando o valor da crença é diferente de “chuva”.
Plano AvailablePlan	Verificar se o plano AvailablePlan é executado quando o plano ByBike é executado.

Após a definição dos casos de testes o próximo passo é codificar os mesmos na ferramenta e observar os resultados obtidos.

4.1.2. Casos de testes – GoHome

Nesta seção será demonstrada a construção dos casos de testes definidos e descritos na seção 4.1.1.

O teste apresentado na Figura 15 verifica se uma determinada crença está presente na base de conhecimento do agente. Essa verificação apesar de simples é importante, pois é o primeiro passo do ciclo de raciocínio do agente ao selecionar os planos que podem levar o agente ao seu objetivo. Através da chamada `startAUT("GoHomeAgent", new BDIAgentGoHome())` o agente cujo comportamento se deseja observar é iniciado (linha 3). O método `waitUntilAUThasFinished(DELAY)` (linha 5) faz com que a thread de execução do teste aguarde até que execução do agente termine. Por último, o método `assertHasBelief("WEATHER")`(linha 7) verifica se existe uma crença chamada “WEATHER” na base de crenças do agente.

```
1  public void testVerificaSeExisteUmaCrencaComUmNome() {
2
3      startAUT("GoHomeAgent", new BDIAgentGoHome());
4
5      waitUntilAUThasFinished(DELAY);
6
7      assertHasBelief("WEATHER");
8
9  }
10
```

Figura 15 – Verifica a existência de uma crença na base de conhecimento.

O método apresentado na Figura 16 além de verificar a existência da crença “WEATHER”, verifica também seu valor, que nesse caso é “CHUVA”. A linha 7 cria o objeto com o valor esperado e a linha 9 verifica se este objeto está presente na base de conhecimento do agente.

```
1  public void testVerificaSeExisteUmaCrenca() {
2
3      startAUT("GoHomeAgent", new BDIAgentGoHome());
4
5      waitUntilAUThasFinished(DELAY);
6
7      Belief<string> belief = new TransientBelief<string>("WEATHER", "CHUVA");
8
9      assertHasBelief(belief);
10
11 }
```

Figura 16 – Verifica se uma crença possui um valor determinado.

O teste apresentado na Figura 17 verifica se os planos `ByBikePlan` e `ByBusPlan` foram adicionados a biblioteca de planos do agente respectivamente através dos métodos `assertHasPlan("ByBikePlan")` (linha 7) e `assertHasPlan("ByBusPlan")` (linha 8).

```
1  public void testVerificaSeAgentePossuiUmPlanoComUmNome() {
2
3      startAUT("GoHomeAgent", new BDIAgentGoHome());
4
5      waitUntilAUThasFinished(DELAY);
6
7      assertHasPlan("ByBikePlan");
8      assertHasPlan("ByBusPlan");
9
10 }
11
```

Figura 17 – Verifica a existência de planos na biblioteca de planos do agente.

O teste apresentado na Figura 18 verifica se o plano `ByBusPlan` foi executado pelo agente. O método responsável por essa verificação é `assertWasExecutedPlan(plan)` (linha 9).

```
1  public void testVerificaSeUmPlanoFoiExecutado() {
2
3      startAUT("GoHomeAgent", new BDIAgentGoHome());
4
5      waitUntilAUThasFinished(DELAY);
6
7      Plan plan = new SimplePlan("ByBusPlan", ByBusPlan.class);
8
9      assertWasExecutedPlan(plan);
10
11 }
12
```

Figura 18 – Verifica se o plano ByBusPlan foi executado.

De maneira análoga, o teste apresentado na Figura 19 verifica se o plano ByBikePlan foi executado pelo agente (linha 9). Este teste deve falhar de acordo com o caso de teste inicialmente definido para o agente.

```
1  public void testVerificaSeUmPlanoFoiExecutado() {
2
3      startAUT("GoHomeAgent", new BDIAgentGoHome());
4
5      waitUntilAUThasFinished(DELAY);
6
7      Plan plan = new SimplePlan("ByBikePlan", ByBikePlan.class);
8
9      assertWasExecutedPlan(plan);
10
11 }
12
```

Figura 19 – Verifica se o plano ByBikePlan foi executado.

Ainda avaliando se um plano foi executado, o teste da Figura 20 verifica se o plano AvailableBikePlan foi executado pelo agente (linha 9). Este teste também deve falhar.

```
1  public void testVerificaSeUmPlanoFoiExecutado() {
2
3      startAUT("GoHomeAgent", new BDIAgentGoHome());
4
5      waitUntilAUThasFinished(DELAY);
6
7      Plan plan = new SimplePlan("AvaliableBikePlan", AvaliableBikePlan.class);
8
9      assertWasExecutedPlan(plan);
10
11 }
12
```

Figura 20 – Verifica se o plano AvailableBikePlan foi executado.

4.1.3. Execução dos casos de testes – GoHome

A execução dos casos de testes apresentados anteriormente permite ao desenvolvedor observar o comportamento do agente de acordo com o que foi codificado. Dessa forma, é possível identificar se alguma ação não esperada ou projetada pelo desenvolvedor foi executada permitindo que este possa corrigir a implementação do agente e refazer os testes.

A Figura 21 apresenta a execução dos testes e apresenta falhas na execução de dois testes. Esse é, de fato, o comportamento esperado uma vez que o agente deve ir para casa de ônibus. Logo, o teste para verificação dos planos associados a ir para casa de bicicleta devem realmente falhar.

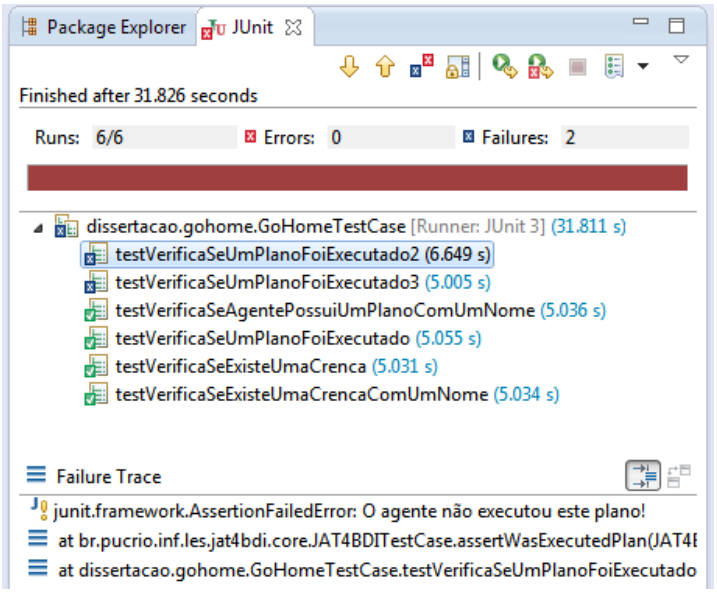

Figura 21 – Resultado da execução dos casos de testes.

A Figura 22 apresenta o log de execução do BDI4JADE informando que o agente GoHome atingiu seu objetivo (ir para casa) e, nesse caso, de ônibus conforme definido no caso de teste do agente.

```
************************************************************
TESTE CASE INICIADO
************************************************************
Agent container Main-Container@JADE-IMTP://arpoador is ready.
------------------------------------------
09:40:15,763 DEBUG (BDIAgent$BDIInterpreter:183) - Beginning BDI-interpreter cycle.
09:40:15,764 DEBUG (BDIAgent$BDIInterpreter:185) - Reviewing beliefs.
09:40:15,764 DEBUG (BDIAgent$BDIInterpreter:234) - Selected goals to be intentions: 1
09:40:15,764 DEBUG (BDIAgent$BDIInterpreter:251) - BDI-interpreter cycle finished.
09:40:15,765 DEBUG (BDIAgent$BDIInterpreter:183) - Beginning BDI-interpreter cycle.
09:40:15,765 DEBUG (BDIAgent$BDIInterpreter:185) - Reviewing beliefs.
09:40:15,765 DEBUG (BDIAgent$BDIInterpreter:234) - Selected goals to be intentions: 1
09:40:15,766 DEBUG (BDIAgent$BDIInterpreter:251) - BDI-interpreter cycle finished.
Indo para casa de ônibus!!!
09:40:15,766 DEBUG (BDIAgent$BDIInterpreter:183) - Beginning BDI-interpreter cycle.
09:40:15,766 DEBUG (BDIAgent$BDIInterpreter:185) - Reviewing beliefs.
09:40:15,767 DEBUG (Intention:194) - Goal: GoHomeGoal (ACHIEVED) - sma2014.gohome.GoHomeGoal@50d8d322
09:40:15,767 DEBUG (BDIAgent$BDIInterpreter:234) - Selected goals to be intentions: 0
09:40:15,768 DEBUG (BDIAgent$BDIInterpreter:245) - No goals or intentions - blocking cycle.
09:40:15,768 DEBUG (BDIAgent$BDIInterpreter:251) - BDI-interpreter cycle finished.
************************************************************
TESTE CASE FINALIZADO
************************************************************
```

Figura 22 – O log de execução do BDI4JADE para o agente GoHome.

4.1.4. Descrição do cenário de uso – Agente Solicitante

Neste cenário de uso é apresentado um agente que monitora o ambiente e, ao perceber um problema neste, solicita a ajuda de outros agentes. Caso o agente (solicitante) perceba a ocorrência de um roubo, o mesmo deve enviar uma mensagem para um agente policial e aguardar sua resposta. Caso o problema seja um incêndio, a mensagem deve ser enviada para um agente bombeiro e finalmente, se o problema for uma doença um agente médico deve ser chamado conforme representado na Figura 23.

Figura 23 – O agente “solicitante” interage com outros agentes ao identificar a ocorrência de um problema.

Da mesma maneira que no cenário de uso anterior foi utilizada a técnica “error-guessing” para definir o caso de teste a ser automatizado (MEYER, 1997) (BEER e RAMLER, 2008). O caso de teste proposto encontra-se descrito na Tabela 5.

Tabela 5 – Cenário de teste proposto para o agente Solicitante.

Agent Under Test	**Solicitante**
Cenário de Teste	O agente Solicitante percebe a ocorrência de um roubo e procura por um agente policial no serviço de páginas amarelas. Ao encontrar um agente policial, o agente solicitante envia uma mensagem notificando-o do problema. O agente policial envia uma mensagem de confirmação para o agente solicitante informando que vai cuidar do problema.
Resultado Esperado	O agente solicitante recebe uma mensagem do agente policial.

Para apoiar a verificação do comportamento do agente e a identificação de possíveis falhas ocorridas ao longo do desenvolvimento foram propostos os casos de teste descritos na Tabela 6.

Tabela 6 – Casos de testes para o agente Solicitante.

Item a ser testado	**Objetivo do teste**
Crença com o problema	Verifica se o agente possui a crença “solicitacao” para armazenar o problema.
Plano que solicita ajuda	Verificar se o plano que solicita a ajuda de outros agentes foi adicionado à biblioteca de planos do agente.
Plano que solicita ajuda	Verificar se o plano que solicita a ajuda foi executado pelo agente.
Plano que solicita ajuda	Verifica se a mensagem foi enviada corretamente para o agente policial.
Plano que solicita ajuda	Verificar se o agente solicitante recebe a resposta do agente policial.

O próximo passo é codificar os casos de teste na ferramenta e observar os resultados obtidos.

4.1.5. Casos de testes – Agente Solicitante

Nesta seção será demonstrada a construção dos casos de teste definidos de descritos na Seção 4.1.4.

O teste apresentado na Figura 24 verifica se uma determinada crença está presente na base de conhecimento do agente. Através da chamada `startAUT("SolicitanteAgent", new SolicitanteBDIAgent())` o agente "Solicitante" é iniciado (linha 3). O método `waitUntilAUThasFinished(DELAY)` (linha 5) faz com que a thread de execução do teste aguarde até que execução do agente termine. Por último, o método `assertHasBelief(belief)`(linha 9) verifica se existe uma crença com um valor específico na base de crenças do agente, criada na linha 7.

```
1  public void testVerificaExistenciaDaCrencaNaBaseDeConhecimento() {
2
3      startAUT("SolicitanteAgent", new SolicitanteBDIAgent());
4
5      waitUntilAUThasFinished(DELAY);
6
7      Belief<string> belief = new TransientBelief<string>("solicitacao",
8              "ladrao");
9      assertHasBelief(belief);
10
11 }
```

Figura 24 – Verifica a existência da crença na base de conhecimento.

O teste apresentado na Figura 25 verifica se o plano "`SolicitantePlan`" está presente na biblioteca de planos do agente. Neste teste é iniciada a utilização dos agentes mocks que representam os agentes: policial, bombeiro e médico com o qual o agente solicitante precisa se comunicar. As linhas 3-5 criam os agentes mocks no ambiente e registram os mesmos no serviço de páginas amarelas da plataforma. Para o caso de teste que está sendo verificado, o mock do agente policial é criado pelo método `starMocktAUT("MockPoliceMan", new MockPoliceMan())`. O teste da verificação do plano é feito através da chamada `assertHasPlan(plan)`(linha 13) do plano configurado no objeto criando na linha 11.

```
1  public void testVerificaExistenciaDePlanoNaBibliotecaDePlanos() {
2
3      startMockAgent("MockDoctor", new MockDoctor());
4      startMockAgent("MockFireman", new MockFireman());
5      startMockAgent("MockPoliceman", new MockPoliceman());
6
7      startAUT("SolicitanteAgent", new SolicitanteBDIAgent());
8
9      waitUntilAUThasFinished(DELAY);
10
11     Plan plan = new SimplePlan("SolicitantePlan", SolicitantePlan.class);
12
13     assertHasPlan(plan);
14
15 }
```

Figura 25 – Verifica se o plano está na biblioteca de planos do agente.

O teste apresentado na Figura 26 verifica se o plano "`SolicitantePlan`", responsável por localizar o agente desejado no serviço de páginas amarelas da plataforma, foi executado corretamente. Esta verificação é feita pela chamada da linha 13 através do método `assertWasExecutedPlan(plan)`.

```
1  public void testVerificaSePlanoFoiExecutado() {
2
3      startMockAgent("MockDoctor", new MockDoctor());
4      startMockAgent("MockFireman", new MockFireman());
5      startMockAgent("MockPoliceman", new MockPoliceman());
6
7      startAUT("SolicitanteAgent", new SolicitanteBDIAgent());
8
9      waitUntilAUThasFinished(DELAY);
10
11     Plan plan = new SimplePlan("SolicitantePlan", SolicitantePlan.class);
12
13     assertWasExecutedPlan(plan);
14
15 }
```

Figura 26 – Verifica se o plano do agente solicitante foi executado.

O teste apresentado na Figura 27 verifica se o agente "Solicitante" enviou corretamente a mensagem para o agente mock (`MockPoliceMan`). O método `startMockAgent("MockPoliceman", mockAgent)` inicia a execução do agente mock (linhas 3-4). A linha 10 faz com que o teste espere até que a execução do agente mock esteja concluída para continuar a execução do caso de teste (`waitUntilMockHasFinished("MockPoliceman")`). Os métodos das linhas 16 e 17 verificam respectivamente se a mensagem foi enviada para o agente correto (o agente policial, neste caso de teste) e se com o conteúdo correto.

```
1  public void testVerificaMensagemEnviada() {
2
3      Agent mockAgent = new MockPoliceman();
4      startMockAgent("MockPoliceman", mockAgent);
5
6      startAUT("SolicitanteAgent", new SolicitanteBDIAgent());
7
8      waitUntilAUThasFinished(DELAY);
9
10     waitUntilMockHasFinished("MockPoliceman");
11
12     ACLMessage message = new ACLMessage(ACLMessage.INFORM);
13     message.setContent("Existe um roubo em andamento!");
14     message.setSender(mockAgent.getAID());
15
16     assertSentMessageTo(message.getSender());
17     assertContentSentMessageEquals(message);
18
19 }
```

Figura 27 – Verifica se o agente solicitante enviou corretamente a mensagem para o agente policial.

O teste apresentado na Figura 28 verifica se o AUT (Solicitante) recebeu corretamente a mensagem do agente policial. Os métodos das linhas 12 e 13 verificam respectivamente quem enviou a resposta e qual o conteúdo da mensagem.

```
1  public void testVerificaMensagemRecebida() {
2
3      Agent mockAgent = new MockPoliceman();
4      startMockAgent("MockPoliceman", mockAgent);
5
6      startAUT("SolicitanteAgent", new SolicitanteBDIAgent());
7
8      waitUntilAUThasFinished(DELAY);
9
10     waitUntilMockHasFinished("MockPoliceman");
11
12     assertReceivedMessageFrom(mockAgent.getAID());
13     assertContentReceivedMessageEquals("Vou prender o ladrão");
14
15 }
16
```

Figura 28 – Verifica se o agente solicitante recebeu a resposta corretamente.

4.1.6. Execução dos casos de testes – Solicitante

A execução dos casos de testes apresentados anteriormente permite ao desenvolvedor observar o comportamento executado pelo agente “Solicitante” e verificar se o resultado obtido é, de fato, o resultado esperado. A Figura 29 apresenta o resultado da execução dos casos de teste.

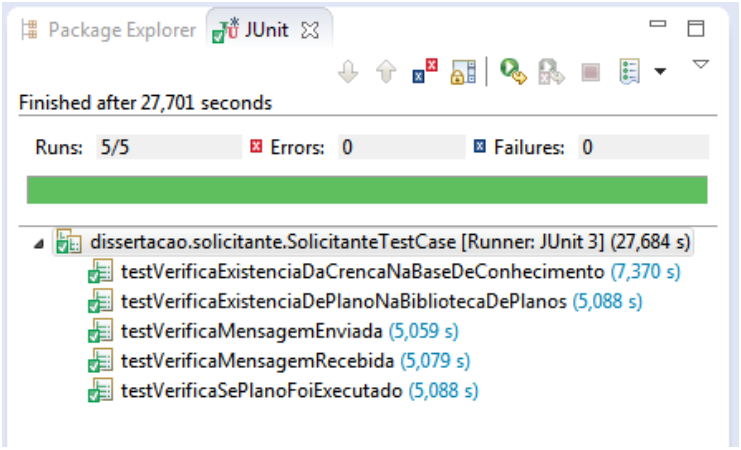

Figura 29 – Resultado da execução dos casos de testes no JAT4BDI.

A Figura 30 apresenta o log de execução do BDI4JADE informando que o agente Solicitante atingiu seu objetivo solicitando a ajuda do agente policial e recebendo a confirmação do mesmo conforme definido no caso de teste do agente.

```
**********************************************************
TESTE CASE INICIADO -> testVerificaSePlanoFoiExecutado
**********************************************************
Agent container Main-Container@JADE-IMTP://Michele-PC is ready.
------------------------------------------
Iniciando o MockPoliceman
14:40:03,737 DEBUG (BDIAgent$BDIInterpreter:183) - Beginning BDI-interpreter cycle.
14:40:03,737 DEBUG (BDIAgent$BDIInterpreter:185) - Reviewing beliefs.
14:40:03,737 DEBUG (BDIAgent$BDIInterpreter:234) - Selected goals to be intentions: 1
14:40:03,737 DEBUG (BDIAgent$BDIInterpreter:251) - BDI-interpreter cycle finished.
14:40:03,737 DEBUG (BDIAgent$BDIInterpreter:183) - Beginning BDI-interpreter cycle.
14:40:03,737 DEBUG (BDIAgent$BDIInterpreter:185) - Reviewing beliefs.
14:40:03,737 DEBUG (BDIAgent$BDIInterpreter:234) - Selected goals to be intentions: 1
14:40:03,737 DEBUG (BDIAgent$BDIInterpreter:251) - BDI-interpreter cycle finished.
14:40:03,768 DEBUG (BDIAgent$BDIInterpreter:183) - Beginning BDI-interpreter cycle.
14:40:03,768 DEBUG (BDIAgent$BDIInterpreter:185) - Reviewing beliefs.
14:40:03,768 DEBUG (BDIAgent$BDIInterpreter:234) - Selected goals to be intentions: 1
14:40:03,768 DEBUG (BDIAgent$BDIInterpreter:251) - BDI-interpreter cycle finished.
14:40:03,768 DEBUG (BDIAgent$BDIInterpreter:183) - Beginning BDI-interpreter cycle.
14:40:03,768 DEBUG (BDIAgent$BDIInterpreter:185) - Reviewing beliefs.
14:40:03,784 DEBUG (BDIAgent$BDIInterpreter:234) - Selected goals to be intentions: 1
14:40:03,784 DEBUG (BDIAgent$BDIInterpreter:251) - BDI-interpreter cycle finished.
14:40:03,784 DEBUG (BDIAgent$BDIInterpreter:183) - Beginning BDI-interpreter cycle.
14:40:03,784 DEBUG (BDIAgent$BDIInterpreter:185) - Reviewing beliefs.
14:40:03,784 DEBUG (BDIAgent$BDIInterpreter:234) - Selected goals to be intentions: 1
14:40:03,784 DEBUG (BDIAgent$BDIInterpreter:251) - BDI-interpreter cycle finished.
( agent-identifier :name MockPoliceman@Michele-PC:1099/JADE  :addresses (sequence http://Michele-PC:63596/acc )): Vou prender o ladrão
14:40:03,831 DEBUG (BDIAgent$BDIInterpreter:183) - Beginning BDI-interpreter cycle.
14:40:03,831 DEBUG (BDIAgent$BDIInterpreter:185) - Reviewing beliefs.
14:40:03,831 DEBUG (Intention:194) - Goal: SolicitanteGoal (ACHIEVED) - dissertacao.solicitante.SolicitanteGoal@5bede4e1
14:40:03,831 DEBUG (BDIAgent$BDIInterpreter:234) - Selected goals to be intentions: 0
14:40:03,831 DEBUG (BDIAgent$BDIInterpreter:245) - No goals or intentions - blocking cycle.
14:40:03,831 DEBUG (BDIAgent$BDIInterpreter:251) - BDI-interpreter cycle finished.
Classe: JAT4BDITestCase - Método: tearDown()
**********************************************************
TESTE CASE FINALIZADO -> testVerificaSePlanoFoiExecutado
**********************************************************
```

Figura 30 – Log informando que o agente “Solicitante” alcançou seu objetivo.

4.2. Book Trading System

Esta seção apresenta um exemplo bastante conhecido e fornecido em diversas plataformas de sistemas multiagentes: o Book Trading System. Trata-se de uma aplicação de comércio de livros no qual cada agente pode desempenhar o papel de um vendedor, um comprador ou ambos. A Figura 31 (B) detalha o protocolo de interação entre esses papéis. Entre o agente vendedor e o agente

comprador fica estabelecido CONTRACT NET Protocol da FIPA (FIPA, 2000), como ilustrado na Figura 31 (A).

De acordo com a Figura 31 (B), logo que um agente vendedor se junta ao ambiente ele se registra no serviço de páginas amarelas da plataforma como um “book-seller” e fica aguardando por requisições de um comprador. Quando um agente comprador se junta ao ambiente ele procura por agentes “book-seller” registrados nas páginas amarelas e inicia a interação com este.

Quando o agente vendedor recebe uma mensagem do tipo “CPF” de um comprador, ele procura pelo livro requisitado em seu catálogo de livros. Se o livro está disponível o agente vendedor envia uma mensagem “PROPOSE” em resposta à mensagem “CPF”, cujo conteúdo é o preço do livro. Por outro lado, se o agente vendedor não tem o livro em seu catálogo, ele envia uma mensagem “REFUSE” informando ao agente comprador que o livro não está disponível. O agente comprador recebe todas as propostas e rejeições dos agentes vendedores e escolhe aquele com a melhor oferta e, então, envia ao vendedor escolhido uma mensagem “PURCHASE”. Quando o agente vendedor recebe uma mensagem “PURCHASE” ele remove o livro do catálogo e envia uma mensagem “INFORM” para notificar o agente comprador que a venda do livro foi concluída. No entanto, se por alguma razão o livro não está mais disponível no catálogo, o agente vendedor envia uma mensagem “FAILURE” informando ao agente comprador que o livro requisitado não está mais disponível. Se o agente comprador recebe uma mensagem indicando que a compra foi concluída ele pode encerrar. Caso contrário, executará seu plano novamente para tentar comprar o livro de outro agente.

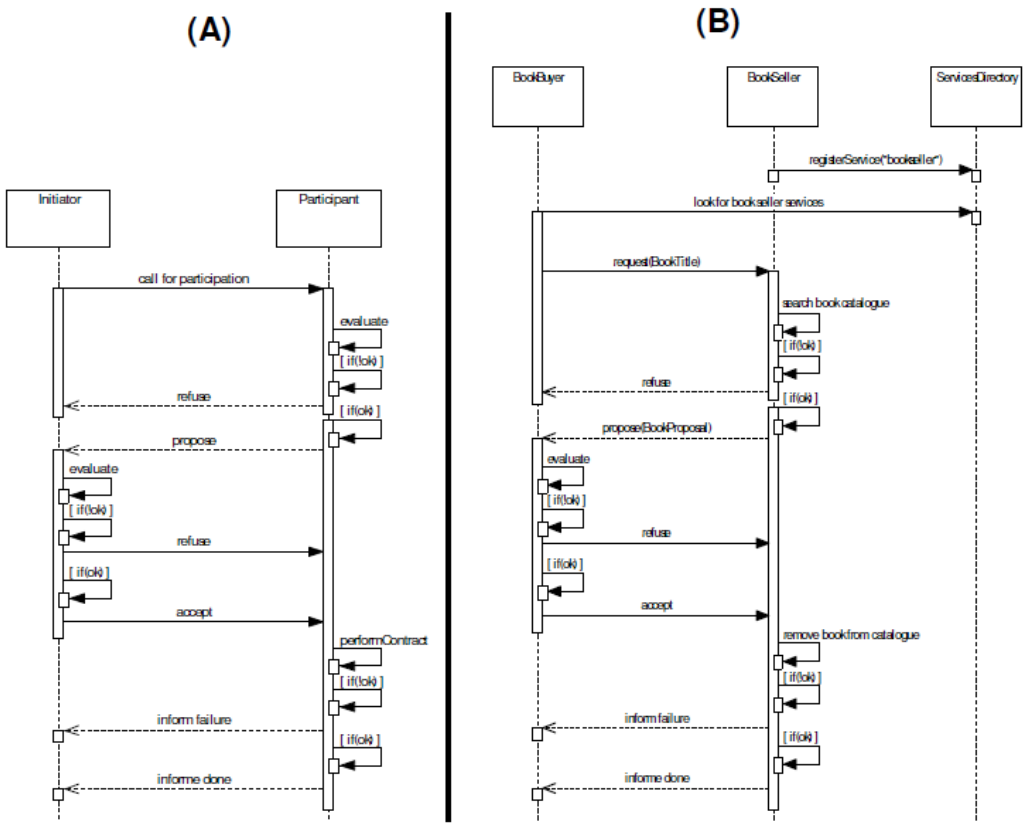

Figura 31 – FIPA CONTRACT-Net Protocol (A) e (B) o Book Trading System.

4.2.1. Descrição do cenário de uso

Recorrendo novamente a técnica “error-guessing”, foi definido o cenário de teste apresentado na Tabela 7. Neste cenário, o agente que deve lidar com a condição excepcional será o AUT do cenário de teste, e todos os outros agentes serão representados por agentes mock.

Tabela 7 – Cenário de teste do exemplo Book Trading System.

Agent Under Test	**Agente Vendedor (BookSeller)**
Cenário de Teste	Dois agentes compradores (BookBuyer) tentam comprar o mesmo livro do agente vendedor (BookSeller) mas ele possui somente um exemplar disponível.
Resultado Esperado	O agente vendedor (BookSeller) deve vender o livro para o primeiro agente que requisitou a compra do livro e rejeitar a solicitação dos

	outros agentes.

A Tabela 8 apresenta os casos de testes que devem ser implementados na ferramenta com o objetivo de apoiar a verificação do comportamento do agente no cenário de teste descrito e a identificação de possíveis falhas.

Tabela 8 – Casos de teste para o agente BookSeller.

Item a ser testado	**Objetivo do teste**
Crença representando o catálogo de livros	Verificar se foi criada uma crença contendo a informação do catálogo dos livros.
Valor da crença	Verificar se o livro está disponível no catálogo.
Plano BookSellerPlan	Verificar se o plano BookSellerPlan está presente na biblioteca de planos do agente.
Plano BookSellerPlan	Verificar se o plano BookSellerPlan foi executado.
Plano BookSellerPlan	Verificar se o plano BookSellerPlan recebeu uma solicitação de compra corretamente.
Plano BookSellerPlan	Verificar se o plano BookSellerPlan informou a disponibilidade do livro corretamente.
Plano BookSellerPlan	Verificar se o plano BookSellerPlan recebeu a proposta de compra do livro.
Plano BookSellerPlan	Verificar se o plano BookSellerPlan informou ao agente comprador que a compra foi finalizada.

4.2.2. Casos de testes – Book Trading System

A verificação do comportamento do agente é iniciada através do teste descrito na Figura 32. Nele é verificada a existência da crença "`catalogue`" na base de conhecimento do agente (linha 7).

```
1  public void testVerificaExistenciaDaCrenca() {
2
3      startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
4
5      waitUntilAUThasFinished(DELAY);
6
7      assertHasBelief("catalogue");
8  }
9
```

Figura 32 – Verifica a existência da crença na base de conhecimento.

O próximo passo é a verificação do valor da crença. Dessa forma, é possível verificar se o agente vendedor possui o exemplar que o comprador deseja. A linha 15 apresenta a verificação do valor da crença, conforme Figura 33.

```
1  public void testVerificaSeExisteOLivroNoCatalogo() {
2
3      startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
4
5      waitUntilAUThasFinished(DELAY);
6
7      Book book = new Book("Programação Modular", 100.00, 2);
8
9      List<book> list = new ArrayList<book>();
10     list.add(book);
11
12     Belief<list<book>> belief = new TransientBelief<list<book>>(
13             "catalogue", list);
14
15     assertHasBelief(belief);
16 }
```

Figura 33 – Verifica o valor da crença na base de conhecimento do agente.

O próximo teste verifica se o plano "`BookSellerPlan`" está presente na biblioteca de planos do agente. Este plano é responsável por efetuar a venda do livro para o agente comprador. A linha 9 faz a verificação mencionada conforme Figura 34.

```
1  public void testVerificaSeExistePlanoNaBibliotecaDePlanos() {
2
3      startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
4
5      waitUntilAUThasFinished(DELAY);
6
7      Plan plan = new SimplePlan("BookSellerPlan", BookSellerPlan.class);
8
9      assertHasPlan(plan);
10
11 }
```

Figura 34 – Verifica a existência do plano na biblioteca de planos do agente.

O teste apresentado na Figura 35 verifica se o plano "`BookSellerPlan`" foi executado (linha 14). O teste ainda inicializa os agentes compradores como agentes mock (linhas 3 e 4). Cada agente mock aguarda seu momento de interagir com o AUT (linhas 9 e 10).

```
1  public void testVerificaSeOPlanoFoiExecutado() {
2
3      startMockAgent("MockFirstBuyer", new MockFirstBuyer());
4      startMockAgent("MockSecoundBuyer", new MockSecoundBuyer());
5      startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
6
7      waitUntilAUThasFinished(DELAY);
8
9      waitUntilMockHasFinished("MockFirstBuyer");
10     waitUntilMockHasFinished("MockSecoundBuyer");
11
12     Plan plan = new SimplePlan("BookSellerPlan", BookSellerPlan.class);
13
14     assertWasExecutedPlan(plan);
15
16 }
```

Figura 35 – Verifica se o plano “BookSellerPlan” foi executado.

A Figura 36 apresenta o teste onde o AUT recebe a solicitação de compra do agente mock “`MockFirstBuyer`”. Este caso de teste verifica se o AUT recebeu uma mensagem do tipo “`CPF`” (linha 15) e se o conteúdo da mensagem contendo o livro desejado está correto (linha 16).

```
1  public void testVerificaSolicitacaoDeCompra() {
2
3      startMockAgent("MockFirstBuyer", new MockFirstBuyer());
4      startMockAgent("MockSecoundBuyer", new MockSecoundBuyer());
5      startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
6
7      waitUntilAUThasFinished(DELAY);
8
9      waitUntilMockHasFinished("MockFirstBuyer");
10     waitUntilMockHasFinished("MockSecoundBuyer");
11
12     ACLMessage message = new ACLMessage(ACLMessage.CFP);
13     message.setContent("Programação Modular");
14
15     assertPerformativeReceivedMessageEquals(message.getPerformative());
16     assertContentReceivedMessageEquals("Programação Modular");
17
18 }
```

Figura 36 – Verifica se o tipo e o conteúdo da mensagem estão corretos.

O próximo teste, conforme Figura 37 verifica se a resposta do AUT a solicitação do comprador está correta. Neste caso, o AUT precisa responder enviando uma proposta (linha 15) e o preço do livro requisitado (linha 16).

```
1  public void testVerificaPropostaDeVenda() {
2
3      startMockAgent("MockFirstBuyer", new MockFirstBuyer());
4      startMockAgent("MockSecoundBuyer", new MockSecoundBuyer());
5      startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
6
7      waitUntilAUThasFinished(DELAY);
8
9      waitUntilMockHasFinished("MockFirstBuyer");
10     waitUntilMockHasFinished("MockSecoundBuyer");
11
12     ACLMessage message = new ACLMessage(ACLMessage.PROPOSE);
13     message.setContent("100.00");
14
15     assertPerformativeSentMessageEquals(message.getPerformative());
16     assertContentSentMessageEquals(message);
17
18 }
```

Figura 37 – Verifica se uma proposta foi enviada pelo agente vendedor.

A Figura 38 apresenta o teste que confirma a solicitação de compra. O AUT recebe uma mensagem do agente comprador confirmando a compra do livro (linha 14).

```
1  public void testVerificaConfirmacaoDeCompra() {
2
3      startMockAgent("MockFirstBuyer", new MockFirstBuyer());
4      startMockAgent("MockSecoundBuyer", new MockSecoundBuyer());
5      startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
6
7      waitUntilAUThasFinished(DELAY);
8
9      waitUntilMockHasFinished("MockFirstBuyer");
10     waitUntilMockHasFinished("MockSecoundBuyer");
11
12     ACLMessage message = new ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
13
14     assertPerformativeReceivedMessageEquals(message.getPerformative());
15
16 }
```

Figura 38 – Verifica a confirmação de compra do agente comprador.

Finalmente, cenário de teste se encerra com o caso de teste da Figura 39, que verifica o envio da mensagem do AUT para o agente comprador informando da conclusão da compra (linha 14).

```
1  public void testVerificaConclusaoDaVenda() {
2
3      startMockAgent("MockFirstBuyer", new MockFirstBuyer());
4      startMockAgent("MockSecoundBuyer", new MockSecoundBuyer());
5      startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
6
7      waitUntilAUThasFinished(DELAY);
8
9      waitUntilMockHasFinished("MockFirstBuyer");
10     waitUntilMockHasFinished("MockSecoundBuyer");
11
12     ACLMessage message = new ACLMessage(ACLMessage.INFORM);
13
14     assertPerformativeSentMessageEquals(message.getPerformative());
15
16 }
```

Figura 39 – Verifica o envio da mensagem de conclusão da compra.

Cada etapa da verificação do CONTRACT-NET Protocol foi verificada pelo conjunto de casos de testes expostos acima. A falha em quaisquer das verificações anteriores representa uma falha no protocolo de comunicação proposto para os agentes.

4.2.3. Execução dos casos de testes – Book Trading System

A execução de todos os casos de teste do cenário permite ao desenvolvedor observar o comportamento e a interação entre o agente vendedor (AUT) e o agente comprador. A Figura 40 apresenta o resultado da execução dos testes.

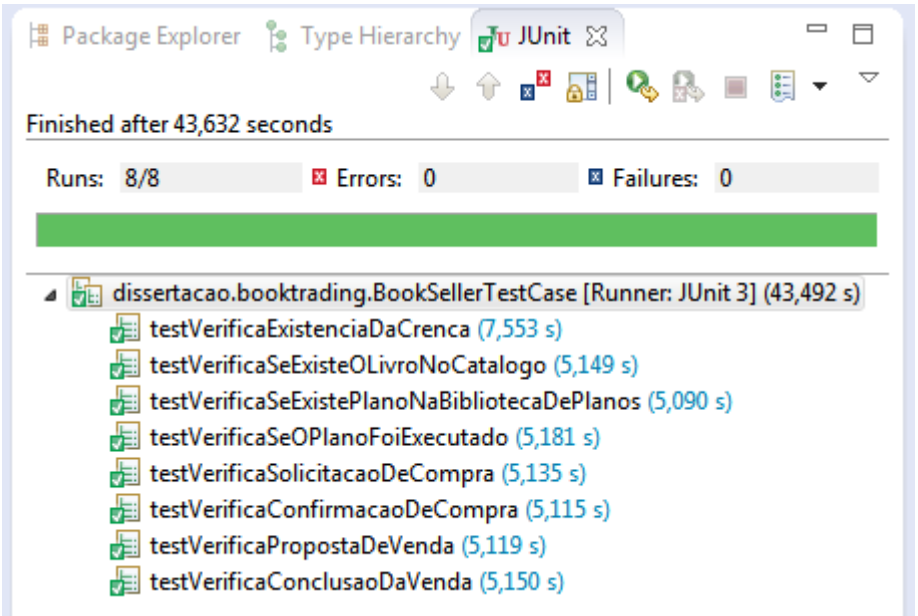

Figura 40 – Resultado da execução dos casos de teste.

A Figura 41 apresenta o log de execução da plataforma BDI4JADE informando que o agente vendedor alcançou seu objetivo.

```
21:28:23,989 DEBUG (BDIAgent$BDIInterpreter:183) - Beginning BDI-interpreter cycle.
21:28:23,989 DEBUG (BDIAgent$BDIInterpreter:185) - Reviewing beliefs.
21:28:23,989 DEBUG (BDIAgent$BDIInterpreter:234) - Selected goals to be intentions: 1
21:28:23,989 DEBUG (BDIAgent$BDIInterpreter:251) - BDI-interpreter cycle finished.
21:28:23,989 DEBUG (BDIAgent$BDIInterpreter:183) - Beginning BDI-interpreter cycle.
21:28:24,005 DEBUG (BDIAgent$BDIInterpreter:185) - Reviewing beliefs.
21:28:24,005 DEBUG (BDIAgent$BDIInterpreter:234) - Selected goals to be intentions: 1
21:28:24,005 DEBUG (BDIAgent$BDIInterpreter:251) - BDI-interpreter cycle finished.
21:28:24,005 DEBUG (BDIAgent$BDIInterpreter:183) - Beginning BDI-interpreter cycle.
21:28:24,005 DEBUG (BDIAgent$BDIInterpreter:185) - Reviewing beliefs.
21:28:24,005 DEBUG (Intention:194) - Goal: BookSellerGoal (ACHIEVED) - dissertacao.booktrading.BookSellerGoal@3b3166fc
21:28:24,005 DEBUG (BDIAgent$BDIInterpreter:234) - Selected goals to be intentions: 0
21:28:24,005 DEBUG (BDIAgent$BDIInterpreter:245) - No goals or intentions - blocking cycle.
21:28:24,005 DEBUG (BDIAgent$BDIInterpreter:251) - BDI-interpreter cycle finished.
```

Figura 41 – Log informando que o agente vendedor alcançou seu objetivo.

4.3. Resultados Observados

Foram realizados testes incrementais utilizando o JAT4BDI para os exemplos descritos previamente. Primeiramente foram definidos os casos de testes onde foi utilizada a técnica “error-guessing” (MEYER, 1997) (BEER e RAMLER, 2008). Dado o cenário de teste, para cada caso de teste definido havia somente um AUT e os demais agentes que interagiam com o AUT foram representados por agentes mock.

A ferramenta JAT4BDI foi utilizada no apoio ao desenvolvimento de todos os exemplos descritos anteriormente e auxiliou por diversas vezes na identificação de falhas tais como: (i) falha na configuração do valor da crença do agente; (ii) falha na implementação de um plano, isto é, por vezes o plano não era executado por não estar associado ao objetivo do agente; (iii) falha na troca de mensagens entre os agentes devido a erro no conteúdo da mensagem ou na performativa utilizada.

4.4. Sumário

Este capítulo apresentou alguns cenários de uso cujo objetivo foi de exercitar a abordagem proposta através de uma prova de conceito da mesma. Para isso, utilizamos a ferramenta JAT4BDI que implementa a abordagem proposta para testes de agentes deliberativos escritos em BDI4JADE. Foram apresentados dois exemplos simples cujo objetivo foi explorar a ferramenta, seus recursos e utilização. Em seguida, foi apresentado um exemplo para o sistema

Book Trading System. Escolhemos este exemplo por ser bastante conhecido e estar presente em diversas plataformas de desenvolvimento de agentes.

5
Trabalhos relacionados

Neste capítulo, são descritos alguns trabalhos relacionados ao trabalho proposto os quais nortearam as decisões sobre o escopo desta pesquisa.

5.1.
On the testability of BDI agents

O trabalho de Winikoff e Cranefield (WINIKOFF e CRANEFIELD, 2010) apresenta uma análise da flexibilidade e das características adaptativas dos agentes BDI observando o espaço comportamental do agente, ou seja, o número de caminhos possíveis para alcançar um objetivo. O trabalho buscou ainda entender quais são os fatores que influenciam no tamanho desse espaço comportamental e a viabilidade de assegurar a eficácia dos sistemas multiagentes através dos testes.

Assim, Winikoff e Cranefield relacionaram a viabilidade do teste de um sistema multiagente à proporção de caminhos percorridos do espaço comportamental, considerando ainda que, executar um teste consiste em observar um caminho de execução e determinar se este está correto ou não.

Para melhor compreensão segue uma síntese da análise feita por Winikoff e Cranefield.

Primeiramente, os objetivos (*goal*) e os planos (*plan*) dos agentes, assim como a relação entre eles foram representados como uma árvore *goal-plan,* como ilustrado na figura 42.

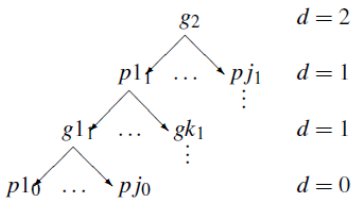

Figura 42 – Árvore goal-plan do agente.

Assumiu-se que a árvore *goal-plan* é uniforme e que para um *goal* de profundidade *d*, todos os filhos são de profundidade *d* - 1 (e semelhantemente

para os planos). Assumiu-se também que todos os planos ou possuem *k* ou *0* *sub-goals* e ainda que cada *goal* possui *j* instâncias de planos aplicáveis ($p_1...p_j$) e que cada instância de plano não folha possui *k* *sub-goals* $g_1...g_k$. Foi usada a notação $n^{\checkmark}(x_d)$ para representar o número de caminhos de execução bem sucedidos de uma árvore de profundidade *d* e com raiz x (onde x ou é um *goal* ou é um *plan*).

Para um *goal,* o número de caminhos que podem ser alcançados é o somatório do número de caminhos no qual cada filho pode ser alcançado. Para um plano, o número de caminhos que podem ser alcançados é o produto do número de caminhos no qual seus filhos podem ser alcançados.

$$\begin{aligned} n^{\checkmark}(g_d) &= j n^{\checkmark}(p_{d-1}) \\ n^{\checkmark}(p_0) &= 1 \\ n^{\checkmark}(p_d) &= n^{\checkmark}({g_d}^k) = n^{\checkmark}(g_d)^k \end{aligned}$$

Figura 3 – Representação das definições da árvore Goal-Plan

$$\begin{aligned} n^{\checkmark}(g_1) &= j n^{\checkmark}(p_0) = j\,1 = j \\ n^{\checkmark}(g_2) &= j n^{\checkmark}(p_1) = j\,(n^{\checkmark}(g_1)^k) = j\,(j^k) = j^{k+1} \\ n^{\checkmark}(g_3) &= j n^{\checkmark}(p_2) = j\,(j^{k+1})^k = j^{k^2+k+1} \\ n^{\checkmark}(g_4) &= j n^{\checkmark}(p_3) = j\,(j^{k^2+k+1})^k = j^{k^3+k^2+k+1} \end{aligned}$$

Figura 3 – Expandindo a definição anterior

$$n^{\checkmark}(g_d) = j^{\sum_{i=0}^{d-1} k^i}$$

Figura 3 – Generalização da definição

Em última análise, os autores concluem que testar o espaço comportamental de um sistema multiagente como um todo é inviável dado o seu comportamento assintótico.

A preocupação do trabalho de Winikoff e Cranefield não está em verificar se um determinado caminho está correto, mas em determinar se é possível garantir a eficácia do sistema por meio de testes. As conclusões de Winikoff e Cranefield sobre a viabilidade do teste de sistemas multiagentes corroboraram para a decisão de delimitar o escopo da abordagem proposta ao teste unitário de agentes e não nos testes de sistemas ou integração.

5.2. JAT: A Test Automation Framework for Multi-Agent Systems

Em seu trabalho, Coelho *et al*. apresentam um framework para testes de sistemas multiagentes (COELHO, CIRILO, et al., 2007) baseado na utilização de "agentes mock", ou seja, na implementação "falsa" de um agente real com o propósito exclusivo de testar a comunicação entre os agentes (COELHO, KULESZA, *et al.*, 2006). Através do monitoramento da transição do estado interno dos agentes, o JAT controla e observa a interação entre os agentes mock e o AUT. Este monitoramento é feito através de aspectos escritos na linguagem ASPECTJ.

Apesar da boa contribuição no que se refere ao teste de agentes de software, o trabalho de Coelho *et al.* se limitou ao teste de agentes de comportamentos reativos. O trabalho limitou-se, ainda, a utilização de um modelo de faltas que se concentra basicamente na identificação de falhas no protocolo de comunicação entre os agentes.

O trabalho de Coelho *et al*. contribuiu de maneira relevante com a abordagem proposta nessa dissertação inspirando a utilização do uso de mocks para testar a interação entre os agentes e o modelo de execução do framework para criação da ferramenta JAT4BDI. Adotamos ainda a ideia de utilizar aspectos para monitorar o ciclo de raciocínio dos agentes BDI4JADE e armazenar seus estados em estruturas de dados internas para observação posterior.

5.3. Model based testing for agent systems

O trabalho de Zhang *et al.* (ZHANG, THANGARAJAH e PADGHAM, 2009) apresenta um framework para geração automática de casos de testes para sistemas multiagente. Este trabalho considera a construção de sistemas multiagentes baseados em modelos (APFELBAUM e DOYLE, 1997) (EL-FAR e WHITTAKER, 2001), neste caso, o Prometheus (PADGHAM e WINIKOFF, 2004), desenvolvidos durante o projeto do SMA. Nesta abordagem, o próprio modelo projetado para o SMA fornece os insumos de entrada para o framework daquilo que deve ser testado. Neste trabalho é apresentado também um modelo para identificação de possíveis falhas e das condições em que as mesmas podem ocorrer. O trabalho concentra-se no teste unitário dos componentes

internos do agente, realizando um teste dirigido a falhas, onde o objetivo é revelar possíveis falhas na implementação (BINDER, 1999).

Apesar de apresentar uma abordagem interessante e relevante no que ser refere ao teste de agentes, o foco principal do trabalho de Zhang *et al.* é a geração automatizada de casos de teste. Em relação ao modelo de faltas, apesar de descrever precisamente as situações e condições de erros, o mesmo é utilizado para apoiar as decisões da geração dos casos de teste não fornecendo nenhum mecanismo para observação do estado interno dos componentes do agente quando uma falha é identificada.

A contribuição do trabalho de Zhang *et al.* para a abordagem proposta nessa dissertação foi bastante relevante, fornecendo a indicação daquilo que deveria ser testado nos agentes e indicando precisamente a situação e condição de possíveis falhas na tentativa do agente em alcançar seus objetivos.

6
Conclusão

O uso da tecnologia de agentes para o desenvolvimento de softwares distribuídos tem se mostrado promissora para esse tipo de sistema. Sua utilização em diversos domínios de negócios, principalmente em cenários críticos para a atuação humana, é uma estratégia que vem sendo cada vez mais adotada. Para esses cenários críticos, a análise e verificação do comportamento do software tornam-se crucial. Contudo, as metodologias propostas até o momento pela Engenharia de Software Orientada a Agentes (AOSE) concentraram seus esforços principalmente em abordagens disciplinadas para analisar, projetar e codificar um SMA e pouca atenção tem sido dada a forma como tais sistemas poderiam ser testados.

Neste contexto, este trabalho propôs uma abordagem para apoiar o desenvolvimento de agentes de software através da construção e manutenção de casos de testes para agentes deliberativos (BDI) escritos em BDI4JADE. Tal abordagem apoiou-se nas ideias suportadas pelo JAT Framework (o uso de agentes mock para simular a interação entre o agente em testes e um agente real e o monitoramento do comportamento dos agentes através de aspectos) e no modelo de faltas proposto por Zhang (ZHANG, 2011), descrevendo as possibilidades de faltas e quais elementos dos agentes precisam ser observados.

Neste trabalho foi proposta ainda, uma ferramenta para apoiar a construção e execução de casos de testes automatizados, o JAT4BDI. Através de exemplos simples, e de caráter exploratório, foram apresentados alguns cenários de uso da ferramenta e seus recursos. Através de métodos de verificação, semelhantes aos existentes no framework de testes JUnit, o desenvolvedor dos testes tem acesso as informações do agente ocorridas durante o seu ciclo de raciocínio auxiliando na identificação de possíveis falhas.

6.1. Contribuições

A seguir são apresentadas as contribuições diretas resultantes do desenvolvimento deste trabalho:

1. **Uma abordagem para testes de agentes deliberativos:** foi proposta uma nova abordagem para apoiar o desenvolvimento de agentes escritos em BDI4JADE baseada na combinação de ideias suportadas por trabalhos relacionados e existentes na Engenharia de Software Orientada a Agentes.

2. **Implementação de uma ferramenta para construção de casos de testes para agentes BDI4JADE:** construção de uma ferramenta para apoiar o desenvolvimento de agentes através da construção de casos de testes.

6.2. Trabalhos futuros

As contribuições apresentadas anteriormente são um esforço para prover aos desenvolvedores de agentes BDI4JADE maior aparato ferramental no que se refere à verificação do comportamento dos agentes. Diversos pontos podem ser melhorados e a seguir são apresentados os trabalhos futuros que podem ser realizados como desdobramentos do trabalho proposto nesta dissertação:

1. **Criação de um framework**: evoluir a ferramenta atual para que a mesma se torne um framework. Possíveis pontos de extensão para o framework seriam: o modelo de faltas utilizado, o mecanismo de monitoramento do ciclo de raciocínio dos agentes permitindo que agentes escritos em outras plataformas pudessem ser testados e não apenas agentes BDI4JADE, um mecanismo para criação de métodos assertivos personalizados pelos desenvolvedores dos casos de teste.
2. **Realização de um estudo experimental**: foram utilizados cenários de uso com a finalidade de apresentar a abordagem e a ferramenta. Contudo, propomos a realização de um estudo para testar sua eficácia e eficiência.

3. **Verificação de comportamentos normativos**: estender a ferramenta para permitir o teste unitário de agentes normativos, isto é, agentes cujo comportamento é regulado por alguma norma externa.

7
Referências bibliográficas

ADRION, W.; BRANSTAD, M.; CHERNIAVSKY, J. **Validation, verication, and testing of computer software**. ACM Computing Surveys. v.14, p.159-192, 1982.

APFELBAUM, L.; DOYLE, J. **Model Based Testing.** International Software Quality Week Conference. CA – USA, 1997.

BEER, A.; RAMLER, R. **The Role of Experience in Software Testing Practice**. Euromicro Conference Software Engineering and Advanced Applications, 2008.

BELLIFEMINE, F.; CAIRE, G.; GREENWOOD, D. **Developing Multi-Agent Systems with JADE**. Wiley Series in Agent Technology, 2007.

BINDER, R. **Testing Object-Oriented Systems: Models, Patterns, and Tools**. Addison-Wesley, 1999.

BRATMAN, M. **Intentions, Plans, and Practical Reason**. Harvard University Press, Cambridge – MA, 1987.

BRIAND, L.; LABICHE, Y.; LEDUC, J. **Tracing Distributed Systems Executions Using AspectJ**. Proceedomgs of ICSM, 2005.

BURNSTEIN, I. **Practical Software Testing**. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

CACCIARI, L.; RAFIQ, O. **Controllability and observability in distributed testing**. Information and Software Technology. V.41, p.767-780, 1999.

CAIRE, G.; COSSENTINO, M.; NEGRI, A.; POGGI, A.; TURCI, P. **Multi-agent systems implementation and testing**, Proceedings of 4th International Symposium - From Agent Theory to Agent Implementation, 2004.

CARROLL, J. **Scenario-based design: envisioning work and technology in system development**. Book: Scenario-based design: envisioning work and technology in system development. John Wiley & Sons, Inc. New York, 1995. ISBN: 0-471-07659-7.

CHOREN, R.; LUCENA, C. **Modeling Multi-Agent Systems with a Note**. Softwarte and Systems Modeling. v. 4, n. 2, p. 199-208, Maio 2005.

COELHO, R.; DANTAS, A.; KULESZA, U.; STAA, A.; CIRNE, W.; LUCENA, C. **The Application Monitor Aspect Pattern**. PLoP'06, 2006.

COELHO, R; CIRILO, E.; KULESZA, U.; STAA, A.; RASHID A.; LUCENA, C. **JAT: A Test Automation Framework for MultiAgent Systems.** International Conference on Software Maintenance. ICSM, 2007.

COELHO, R; CIRILO, E.; KULESZA, U.; STAA, A.; RASHID A.; LUCENA, C. **The JAT Testing Framework - Technical Report**. PUC-Rio, Brazil, 2007.

COELHO, R.; KULESZA, U.; STAA, A.; LUCENA, C. **Unit Testing in Multi-agent Systems using Mock Agents and Aspects**. International Workshop on Software Engineering for Large-Scale Multi-Agent Systems. ICSE, 2006.

EL-FAR, I.; WHITTAKER, J. **Model-Based Software Testing.** Encyclopedia of Software Engineering, pages 825-837. Wiley, Chichester, 2001.

FIPA Contract Net Interaction Protocol Specification, 2000. Disponível em: http://www.fipa.org/specs/fipa00029/.

FISHER, M.; DENNIS, L.; WEBSTER, M. **Verifying Autonomous Systems**. Communications of the ACM, Vol. 56 No. 9, Pages 84-93, Setembro de 2013.

GARCIA, A.; LUCENA, C.; COWAN, D. **Agents in Object-Oriented Software Engineering**. Software Practice & Experience. Elsevier, 34(5), pages 489-521, 2004.

GRISWOLD, W.; SHONLE, M.; SULLIVAN, K.; SONG, Y.; TEWARI, N.; CAI, Y.; RAJAN, H. **Modular Software Design with Crosscutting Interfaces**. IEEE Software, Special Issue on Aspect-Oriented Programming, 2006.

IEEE 610.12 - **IEEE Standard Glossary of Software Engineering Terminology**, 1990 - DOI: 10.1109/IEEESTD.1990.101064.

JENNINGS, N.; WOOLDRIDGE, M. **Software Agents**. IEEE Review. v. 42, n.1, p. 17-20, Janeiro 1996.

LÓPEZ, F. L. **Social Power and Norms**, 2003.

LOW, K; CHEN, T.; RONNQUIST, R. **Automated Test Case Generation for BDI Agents**. Autonomous Agents and Multi-Agent Systems. v.2, No. 4. Pages 311-332, 1999.

LUCENA, C. **Inteligência Artificial e Engenharia de Software**. Zahar. Rio de Janeiro – RJ, 1987.

MEYER, B. **Object-oriented software construction.** Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1997.

MYERS, G.; SANDLER, C; BADGETT, T.; THOMAS, T. **The Art of Software Testing**. Wiley, Second Edition, June de 2004.

NGUYEN, C. D.; PERINI, A.; TONELLA, P. **Automated Continuous Testing of Multi-Agent Systems**. European Workshop on Multi-Agent Systems (EUMAS), 2007.

NGUYEN, C.; PERINI, A.; TONELLA, P.; MILES, S.; HARMAN, M.; LUCK, M. **Evolutionary Testing of Autonomous Software Agents**. International Conference on Autonomous Agents and Multiagent Systems (AAMAS), 2009.

NGUYEN, D.; PERINI, A.; TONELLA, P. **A Goal-Oriented Software Testing Methodology**. Springer - Berlin, April 29, 2008.

NUNES, I.; LUCENA, C.; LUCK, M. **BDI4JADE: a BDI layer on top of JADE**. International Workshop on Programming Multi-Agent Systems - ProMAS, 2011.

NUNEZ, M.; RODRIGUEZ, I.; RUBIO, F. **Specification and testing of autonomous agents in e-commerce systems**. Journal of Software: Testing, Verification and Reliability. v.15, issue 4, p. 211-233, 2005.

PADGHAM, L.; WINIKOFF, M. **Developing Intelligent Agent Systems: A practical guide**. Wiley Series in Agent Technology. RMIT University, Melbourne, Australia, 2004.

PEZZÈ, M.; YOUNG, M. **Teste e Análise de Software: processos, princípios e técnicas**. 1ª. ed. [S.l.]: Bookman, 2008.

RAO, A.; GEORGEFF, M. **BDI-agents: from theory to practice**. Proceedings of the First Intl. Conference on Multiagent Systems, 1995.

RAO, A.; GEORGEFF, M. **Modeling rational agents within a BDI-Architecture**. In: J. Allen, R. Fikes, E. Sandewall (eds.) Principles of Knowledge Representation and Reasoning, Proceedings of the Second International Conference, p. 473–484, Morgan Kaufmann, 1991.

ROUFF, C. **A test agent for testing agents and their communities**. Aerospace Conference Proceedings. v. 5, 2002.

SCHACH, S. **Testing: Principles and practice**, Journal ACM Computing Surveys. v. 28, n. 1, 1996, p. 277-279, Março 1996.

SILVA, V.; CHOREN, R.; LUCENA, C. **A UML based approach for modeling and implementing multiagent systems**. Pages.914-92, AAMAS 2004.

Standard Glossary of Terms used in Software Testing – Documento de referência do International Software Testing Qualification Board (ISTQB). Disponível em: http://www.istqb.org/downloads/glossary.html (acessado em Novembro de 2014).

VOAS, J.; MCGRAW, G. **Software Fault Injection: Inoculating Programs Against Errors**. Wiley, 1998.

VOAS, J.; MILLER, K. **Software Testability: The New Verication**. IEEE Software, 1995.

WEGENER, J. **Stochastic Algorithms: Foundations and Applications.** Springer Berlin / Heidelberg, chapter Evolutionary Testing Techniques, p. 82-94, 2005.

WINIKOFF, M.; CRANEFIELD, S. **On the testability of BDI agents**. European Workshop on Multi-Agent Systems, 2010.

WOOLDRIDGE, M. **An Introcuction to MultiAgent Systems**. 2ª. ed. [S.l.]: Hoboken, NJ: Wiley, 2002.

WOOLDRIDGE, M.; JENNINGS, N. **Intelligent Agents: Theory and Practice**. The Knowledge Engineering Review. v. 10, n. 2, p. 115-152, 1995.

ZAMBONELLI, F.; JENNINGS, N.; OMICINI, A.; WOOLDRIDGE, M. **Agent-oriented software engineering for internet applications**. Coordination of Internet Agents, p. 326–346. Springer Verlag, 2001.

ZHANG, Z. **Automated Unit Testing of Agent Systems**. Tese de Doutorado - RMIT University, Outubro de 2011.

ZHANG, Z.; THANGARAJAH, J.; PADGHAM, L. **Automated Unit Testing For Agent Systems**, International Conference on Autonomous Agents and Multiagent Systems, 2007.

ZHANG, Z.; THANGARAJAH, J.; PADGHAM, L. **Model based testing for agent systems**, International Conference on Autonomous Agents and Multiagent Systems, 2009.