PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 10/2017

Checking the Behavior of BDI4JADE Agents Using an Aspect-Based Approach

Francisco José Plácido da Cunha
Carlos José Pereira de Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 10/2017 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena June, 2017

Checking the Behavior of BDI4JADE Agents Using an Aspect-Based Approach

Francisco José Plácido da Cunha[1] Carlos José Pereira de Lucena[1]

[1]Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
fcunha@inf.puc-rio.br, lucena@inf.puc-rio.br

Abstract. The growth and popularity of the Web has fueled the network-based software development. In this context, multiagent system (MAS) has been considered a promising approach to different areas such as security, or mission critical scenarios, monitoring of environments and people, etc., which means analyzing the choices of this type of software became crucial. However, the methodologies proposed so far by the Agent-Oriented Software Engineering (AOSE) focused their efforts mainly on developing approach to analyze, design and implement a MAS and little attention has been given to how such systems can be tested. Furthermore, with regard to tests involving agents, some issues related to the controllability and observability can difficult the checking of the agents' behavior, for instance: (i) the agent's decision-making process; (ii) the fact of the agent's beliefs and goals are embedded inside the agent, hampering the observation and control of behavior; (iii) how deal with test coverage. This work proposes a new unit test approach for agents written in BDI4JADE, using the JAT Framework ideas and Zhang's fault model.

Keywords: BDI Agents, testing agent, unit test, AspectJ, JAT, BDI4JADE, Mock Agent, JAT4JADE.

Resumo. O crescimento e a popularidade da Web alimentaram o desenvolvimento de uma nova categoria de software baseado em rede. Neste contexto, Sistema multiagente (SMA) foi considerado uma abordagem promissora para diferentes áreas, como segurança, ou cenários de missão crítica, monitoramento de ambientes e pessoas, etc., o que significa que analisar as escolhas desse tipo de software tornou-se crucial. No entanto, as metodologias propostas até agora pela Engenharia de Software Orientada a Agentes (AOSE) concentraram seus esforços principalmente no desenvolvimento de abordagens para analisar, projetar e implementar um SMA, e pouca atenção tem sido dada a como esses tipos de sistemas podem ser testados. Além disso, no que diz respeito aos testes de agentes, algumas questões relacionadas à controlabilidade e observabilidade podem dificultar a verificação do comportamento dos agentes, como por exemplo: (i) o processo de tomada de decisão do agente; (ii) o fato das crenças e objetivos do agente estarem inseridos dentro do agente, dificultando a observação e o controle do comportamento; (iii) como lidar com a cobertura do teste. Este trabalho propõe uma nova abordagem de testes unitários para agentes escritos em BDI4JADE, apoiadas nas ideias do JAT Framework e no modelo de faltas de Zhang.

Palavras-chave: Agentes BDI, teste de agentes, teste unitário, AspectJ, JAT, BDI4JADE, Agentes Mock, JAT4BDI.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Table of Contents

1 Introduction

The growth and popularity of the web have driven the development of network-based software and, according to Zambonelli, the use of agents for these types of systems is considered an appropriate approach (ZAMBONELLI, JENNINGS, *et al.*, 2001) and has been applied in different fields, such as: security, critical business or mission scenarios, advanced monitoring of environments and people, etc., which means that analyzing the choices that this type of software can perform becomes crucial (FISHER, DENNIS and WEBSTER, 2013).

Despite the increasing use of multiagent systems (MAS) in critical scenarios, the methodologies proposed thus far through Agent-Oriented Software Engineering (AOSE) have been concentrated mainly on developing disciplined approaches to analyze, design and code a MAS, with little attention paid as to how such systems could be tested (CAIRE, COSSENTINO, *et al.*, 2004).

Multiagent system are those that decide for themselves what they should do and when they should do it to achieve a goal. Such systems vary in their degree of autonomy employed by the agent and range from systems that are almost completely controlled through human intervention to those that are almost fully automated, that is, with minimal human intervention. Typically, the use of different levels of autonomy is justified for reasons of precision in relation to human capacity and safety regarding the execution of the operation, such as: (i) access to hard-to-reach places; (ii) in hazardous environments; (iii) in long and repetitive activities or; (iv) that require a low response time. Such activities present higher risk when performed by humans due to factors such as fatigue or stress. However, the more autonomy that is employed in an agent, the stricter must be the verification of the performance of this agent (FISHER, DENNIS and WEBSTER, 2013).

In the development of agent-based software, an architecture that is widely known and recommended for the project of agents with high levels of autonomy is BDI (Belief-Desire-Intention). Rao and Georgeff (RAO and GEORGEFF, 1995) proposed this architecture based on Bratman's philosophical model (BRATMAN, 1987). In it, three mental attitudes – belief, desire and intention – make up agents' knowledge base and decision-making mechanism. The agents take decisions from the knowledge they have of themselves and the environment in which they are located in order to achieve their goals.

Contemplating the paradigm of agent-based software development, in this work we focus on the task of testing the software. According to the IEEE 610.12 standard, "software testability is the degree to which a system or component facilitates the establishment of test criteria and performance of tests to determine whether those criteria have been met" (IEEE 610.12, 1990). Also with regard to software testability, two basic characteristics need to be considered: controllability – the ability to control the entry and the internal state of the component being tested, and observability – the ability to observe the result produced by the component being tested. The tests involving software agents bring up some issues related to the controllability and observability of the agents that need to be carefully considered: (i) an agent is an autonomous entity and, therefore, it can be difficult to control its behavior through a testing tool; (ii) the agent's beliefs and goals are embedded in the agent itself, so they cannot be easily observed and controlled by a tool – almost always, a test tool can only observe an agent through its interactions with other agents and with the environment; (iii) without the adoption of

an efficient strategy for coverage of the tests, it certainly becomes unscalable, given the number of possibilities to be tested (BINDER, 1999) and (VOAS and MILLER, 1995).

Winikoff and Cranfield's work presents a quantitative analysis of the coverage effort required and, therefore, to check all the possible decisions that can be taken by the agent. Their conclusion is emphatic, affirming that testing an MAS through the verification of each pathway of the "behavioral space", that is, each pathway of the set of all the pathways that could be possible, is unfeasible (WINIKOFF and CRANFIELD, 2010).

Thus, given the need to check and understand complex behaviors performed by the agent, to evaluate its effectiveness and the challenge and implications regarding the testability of the deliberative mechanism and emergent behavior, this paper focuses on the task of supporting the developer for building test cases for BDI agents. The following questions arise as reasons for research as a result of this work:

RQ1: How can we support the development of multiagent systems by building and maintaining test cases for BDI agents?

RQ2: How can we control and observe the actions performed during the BDI agent reasoning cycle through the use of test cases?

Although it is possible to find works in the literature that discuss and present strategies for testing BDI agents (NGUYEN, PERINI, *et al.*, 2009) (NUNEZ, RODRIGUEZ and RUBIO, 2005) (NGUYEN, PERINI and TONELLA, 2008) (WINIKOFF and CRANEFIELD, 2010) (LOW, CHEN and RONNQUIST, 1999) (ZHANG, THANGARAJAH and PADGHAM, 2007) (ZHANG, THANGARAJAH and PADGHAM, 2009), none of these works is concerned with providing mechanisms to assist in identifying gaps in implementation and observation of the internal state of the agent's elements. Despite the considerable contributions of the aforementioned works, only two (NGUYEN, PERINI, *et al.*, 2009) (LOW, CHEN and RONNQUIST, 1999) deal with approaches that offer tools to support BDI agent testing. Even so, these tools have a greater commitment to coverage strategies and criteria than do control and observation of the agent's internal state.

In view of the existing problem and the limitations of the current approaches to the questions that motivated our research, this work proposes a new approach to BDI agent unit testing based on the combination and adaptation of ideas supported by other research found in the agents' literature.

Support for this paper came from the use of mock agents and multiagent system testing aspects as proposed in the work of Coelho *et al.* (COELHO, KULESZA, *et al.*, 2006), by the ideas of the JAT Framework (COELHO, CIRILO, *et al.*, 2007) and in the failure model proposed by Zhang (ZHANG, 2011) and, thus, assists the BDI4JADE agents developer (NUNES, LUCENA and LUCK, 2011) in the construction and maintenance of test cases and their evaluation. To corroborate the study, a tool was created to serve as proof of concept of the new approach, which we call JAT4BDI. This tool's guideline is to support the development of software agents through the construction of test cases. For this activity, the developer can count on a set of methods that assist in the verification of the decisions taken and the observation of the internal state of the agent during the execution of its reasoning cycle.

2 Theoretical Basis

2.1 Rational and social agents

Wooldridge distinguishes between an agent and a rational or intelligent agent, stating that the latter necessarily needs to be even more reactive, proactive and social (WOOLDRIDGE, 2002).

In this work, we are interested in the testing of rational agents in pursuit of their goals. Being rational means that an agent must not do "stupid" things, like simultaneously conducting two conflicting activities such as, for example, planning on spending a lot of money on a vacation and, at the same time, spending this money on buying a car (PADGHAM and WINIKOFF, 2004). A detailed analysis of the meaning of "rational" can be found in the work of Bratman (BRATMAN, 1987), which forms the basis of the agent reasoning model proposed by Rao and Georgeff (RAO and GEORGEFF, 1991).

2.2 Multiagent systems

Generally, a software agent is not found alone in an application or system, but rather in conjunction with other agents, of the same or different types, forming a company or organization (WOOLDRIDGE, 2002). This society is called Multiagent System (MAS). Thus, an MAS consists of a society of agents capable of interacting with each other and that, to interact with success, they need the skills of cooperation, coordination and negotiation among themselves, as occurs in human society (WOOLDRIDGE, 2002).

2.3 Software test

According to Pezzè and Young (PEZZÈ and YOUNG, 2008), engineering disciplines align design and construction activities with activities that check intermediate and final products so that the defects can be identified and removed. The same thing happens with Software Engineering: the construction of high-quality software requires the combination of design and verification activities throughout the development period.

Software is one of the most complex artifacts built on a regular basis. Quality requirements for software used in an environment can be very different and incompatible with another environment or application domain, and as the system grows, its structure evolves and, frequently, deteriorates (PEZZÈ and YOUNG, 2008).

Software checking is an important activity that encompasses the entire development and maintenance process (ADRION, BRANSTAD and CHERNIAVSKY, 1982). The goal is to find defects in the specifications, the design of artifacts and in implementation. On the other hand, another goal is to prevent defects. The test project can discover and eliminate bugs in all stages of the software construction process (SCHACH, 1996).

However, the cost of software checking often represents more than half of the total development and maintenance costs. Advanced development techniques and powerful support tools can reduce the frequency of some error classes (PEZZÈ and YOUNG, 2008).

The purpose of software testing and analysis is to either evaluate the quality of the software or make it possible to improve the software by revealing defects.

2.4 Multiagent systems tests

Agents are distributed and asynchronous: Agents run in parallel and asynchronously. An agent may have to wait for other agents to accomplish their intended goals. Also, it is possible that an agent might work properly when it is alone and incorrectly upon being placed in a community, or vice versa. Testing tools must have a global vision regarding the distribution of agents, in addition to having local and individual knowledge of each of them, in order to decide whether the system is working according to the specifications (CACCIARI and RAFIQ, 1999).

Agents have autonomy: Agents are autonomous entities, which means that the same test entry could result in different behavior under different executions, since agents can update their knowledge base between the two executions, or can learn from previous executions, resulting in different decisions for similar situations.

Agents exchange messages: Agents communicate by exchanging messages. Traditional testing techniques, involving the invocation of methods, cannot be applied directly, because agents may adopt their own strategies (such as simply not responding to a message received) for sending and receiving messages.

Agents suffer the influence of environmental and regulatory factors: The environment and constraints (norms, rules, laws, etc.) are important factors that influence and govern agents' behavior. Different configurations in the environment may affect the test results. Thus, the environment and the restrictive factors are important and must be considered when designing agent tests.

According to Nguyen, Perini, and Tonella, testing in software agents can be classified into different levels: unit test, agent test, integration test, system test and acceptance test (NGUYEN, PERINI, and TONELLA, 2007).

This work is limited to the Unit Test, focusing on verifying the agent elements.

2.5 JAT Framework

The JAT (Jade Agent Testing Framework) (COELHO, CYRILO, *et al.*, 2007) is a framework for multiagent system based on the use of "mock" agents. A mock agent is a "fake" implementation of a real agent, created with the restricted purpose of testing agents (COELHO, KULESZA, *et al.*, 2006).

A mock agent is responsible for sending messages to the agent being tested (AUT), checking its answers and verifying whether the environment was affected as expected. The work of controlling the interaction between the mock agents and AUT is through the Synchronizer element, which is responsible for defining the order in which the mocks interact with the AUT. Another element present in the framework is the Monitor, which is responsible for observing the internal state of the agents and their transitions. Figure 1 depicts all the participants that make up the JAT Framework.

Agent Under Test (AUT): agent whose behavior is checked;

Mock Agent: "fake" implementation of a real agent that interacts with the AUT;

Monitor: responsible for observing the transition of the agents' internal states;

Synchronizer: controls the order in which the mocks interact with the AUT;

Test Scenario: set of conditions to which the AUT will be exposed, to check if it is according to its specification under these conditions;

Test Suite: consists of a set of test scenarios and a set of operations performed to prepare the test environment before starting a test scenario.

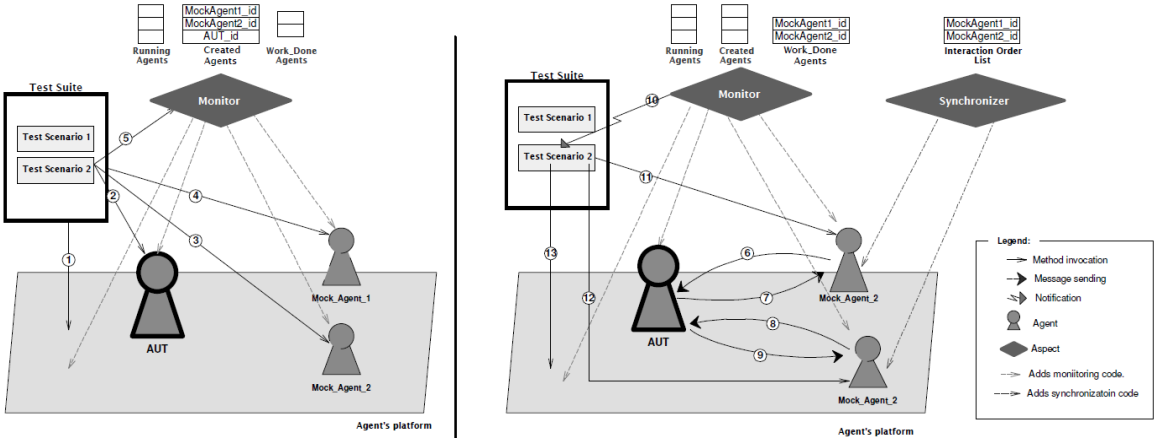

Figure 1 - Flow between the participants of a unit test in the JAT.

Each agent AUT follows the flow shown in Figure 1. In step 1, the test suite creates the platform agents and other elements needed to set up the test environment. Subsequently, the test scenario is initiated. Each test scenario creates one or more mock agents that interact with the AUT (steps 3 and 4) – the number of mock agents varies with the test scenario defined. Next, the AUT is created (step 2). The Monitor agent will be notified when the interaction between the AUT and the mock agents ends (step 5). At this point, the AUT and the mock agents begin their interaction. The mock agents send messages to the AUT, which respond back or vice versa (steps 6 to 9). Steps 6 to 9 are repeated as many times as necessary to run and complete the mock agents' plans. For example, mock agent 1 could answer three messages before finalizing its test activity, and mock Agent 2 could answer only one message from the AUT before its plan ended. During this process, the Monitor observes the interaction of agents and keeps track of changes to the agents' internal states. Three lists as shown in Figure 1 are used for this purpose.

Created Agents List: maintains the identifiers of the agents that were created but that are not yet running;

Running Agents List: maintains the identifiers of the agents that are running;

WorkDone Agents List: maintains the identifiers of the mock agents that have completed their plans;

When a mock agent concludes its plan, the Monitor agent includes the identifier of this agent on the WorkDone list and then notifies the test scenario that the interaction between the mock agent and the AUT was completed (step 10). This notification unblocks the running of the test scenario, which is now capable of: (i) asking the mock agent if the AUT acted or not as expected (steps 11 and 12); and (ii) checking that the environment was affected as expected (step 13). Without such notifications, the test scenario would not be able of knowing when the interactions between the AUT and the mock agents ended, that is, when the test scenario had ended and the verification (final result = expected result) could be made in an intermediate state resulting in a false positive or a false negative in the test. This is the reason why the Monitor is essential in this approach.

The Synchronizer is an optional element. It is only used when the test developer must establish an order in the interaction between the mock agents and the AUT. The Synchronizer maintains a list with the order of interaction that is loaded at the beginning

of the test scenario. This list contains the identifiers of the mock agents that have the right to interact with the AUT in a specific moment in the test scenario. In figure 2, mock agent 1 must send a message to the AUT before mock agent 2. Thus, the test scenario is partially implemented by each mock agent's plan, and the Synchronizer is the element responsible for composing the test scenario. Therefore, the Synchronizer is the element responsible for defining the moment when each mock agent must run in a test scenario.

The Monitor and the Synchronizer elements represent two intersecting interests of the approach. Thus, its implementation is intertwined and spread over various parts of the code of the mock agents, the AUT and platform. Therefore, the strategy adopted for implementation was to use an aspect for each one of these elements (COELHO, KULESZA, *et al.*, 2006) (GRISWOLD, SHONLE, *et al.*, 2006) (MEYER, 1997).

2.5.1 The JAT fault model

The agents encapsulate a complex internal structure composed of plans, goals and beliefs. A plan is represented by a sequence of actions, such as sending messages or the running of an internal procedure, which is executed to achieve a specific goal. Goals, like beliefs, can be expressed as agent attributes and be characterized by a type, a name and a default value that can be modified during the agent running (SILVA, CHOREN and LUCENA, 2004). These abstractions associated with the agents are the origins of new classes of errors. In addition to the bugs that can exist in an OO system (arising from implementation in an OO language), there may be errors in: an agent's plan or belief, the interactions between the agents, the emergent behavior of the MAS, or in the restrictions governing a MAS, to name a few. For practical reasons, a test approach should focus on a model of specific errors (BINDER, 1999). The fault model defines a subset of errors considered for the test approach, which includes its ability to detect errors and, therefore, define the type of error it is intended to detect (BINDER, 1999).

The JAT approach defines as an initial candidate for a fault model a set of agent-specific errors that can be expressed as a failure in the running of the plans and that, consequently, undermines the accomplishment of a goal, including these: (i) error in message order; (ii) error in message content; (iii) error to increase the message response time; (iv) error in agents' beliefs – similar to the failures in the OO attributes; (v) error in the agents' internal procedures– similar to the error in the OO methods.

2.6 Fault Model proposed by Zhang

The new abstractions introduced by the software agent paradigm define new error classes. An effective way to reveal the errors in the system or a component under test is to define a fault model for them, which specifies the situations in which, supposedly, this is likely to be found (BINDER, 1999) (MYERS, SANDLER, *et al.*, 2004) (BURNSTEIN, 2002). A testing approach, to be practical, must focus on a model of specific errors, defining a subset of types of failures considered most relevant by the test approach, thus delimiting the types of errors that are intended to be revealed (BINDER, 1999). In this paper, we use an error-driven test approach where the building of test cases is used to reveal and identify failures arising from implementation.

We adopted the fault model proposed by Zhang (ZHANG, 2011) in our approach. In his work, Zhang uses this model to support the development of scenarios and the automated generation of test cases for BDI agents through the Prometheus Design Tool (PDT) (ZHANG, 2011).

Based on Zhang's fault model, definitions are made for testable characteristics, the failure conditions for each of the agent's elements and the errors that are revealed (ZHANG, 2011).

2.6.1 Faults plans

Broadly speaking, a plan consists of a generating fact for the event that triggers the plan, a context condition and the body of the plan. The plan's generating fact indicates the relevance of the plan for the event. The plan's context condition determines the applicability of the plan in relation to the agent's beliefs. The body of the plan has a sequence of actions taken to achieve a specific goal. When considering a plan as a relevant element for the test agent, the intention is to reveal errors such as:

Is a plan, in fact, considered by an event?

A plan should be considered as applicable as soon as it becomes relevant to the event. This requires that the event is of the correct type and also that all the necessary attributes (generating factor and context condition) are present. If this does not occur, an error due to the fact the plan is not considered applicable will be identified.

Is the plan's context condition evaluated correctly in the selection of the plan?

The context condition of a plan indicates in which situation the plan is applicable. The absence of a context condition denotes that the plan is always applicable in any situation. If the developer then specifies a context condition, it is expected that this context condition will be evaluated as true in some situation and false in others.

Does the plan only trigger the events that were specified to be triggered?

There are two possible points of failure: an event that we expected would be triggered by the plan and that does not happen; an event triggered in a test that was not designed to occur.

Does the plan complete its execution?

During the normal running of the program, there may be some reasons that lead to the failure of the plan that is being tested such as, for example, changes in the environment after the plan is selected. However, in a controlled test environment, all the plans that have been selected for execution must be completed. Thus, if the plan being tested does not complete, we should consider there to have been an error in its implementation (ZHANG, 2011).

2.6.2 Faults in cyclic plans

We must consider the hierarchy of the plans when the project is specified. The interaction of a plan with its subplans can form a cycle. A cyclical plan is treated as a special type of agent unit and is tested as if it were a single entity. Some criteria need to be checked in the test of the agent cyclic plans, such as:

Does the cycle occur in run time?

The specification of a cyclical plan in the project implies that a cycle can occur when running. A failure is revealed when an expected cycle is not formed at run time. Another failure possibility occurs in the opposite situation, that is, a cyclic execution is identified even when no cyclical plan was specified in the project.

Is their completion of the cyclic plan execution?

Running a cycle can continue indefinitely and it is up to the agent designer to set the conditions to stop it. However, it is possible that a cycle might run endlessly due to some implementation error. To verify this, an alternative is to introduce a pre-set maximum limit for the number of iterations that occur. If the cyclic execution exceeds this limit, a failure will be identified (ZHANG, 2011).

2.6.3 Faults in events

The characteristics that orient the test of an event are related to the fact of always having a single plan applicable (complete coverage) to respond to the event or more than one plan applicable (overlaid) to respond to this event. It is said, moreover, that the event being tested has incomplete coverage if there is no applicable plan for this in any situation.

The designer may have defined the occurrence of overlapping or incomplete coverage plans for an event and, consequently, the application must permit it. However, such permissions generally are described in natural language and can pass unnoticed in the development of the agent and deserving, in this case, verification of its occurrence, even when specified.

In this way, two characteristics must be considered when testing an event:

Does a plan always exist that applies to the event?

If not, the specification of the project needs to be checked to ensure that incomplete coverage has been foreseen and permitted for the event. Otherwise, an error will be identified.

Does more than one plan applicable for the event exist?

If there is an event that is dealt with by various plans, it is necessary to check if the designer has, in fact, designed the overlay for the event. If not, an error was identified.

Does a plan exist that applies to the event and is never executed?

In this condition, what is most probable is that a coding error has led to a failure situation (ZHANG, 2011).

2.6.4 Faults in beliefs

The test of a belief basically is designed to check two aspects: (i) if the structure for data storage was designed as specified; (ii) if the update of a belief correctly triggers the appropriate events, when specified in this way.

The first check occurs at the application level, because a belief can be structured and implemented in different ways depending on the platform used. However, it must respect the structure foreseen in the agent project. The second is intended for verification of the events triggered by the change of the belief content, its removal from the agent's knowledge base or the inclusion of a new belief.

Was the belief structure implemented pursuant to what was specified?

This error can occur in two situations: the first is when the developer neglects the coding of some element of the belief structure and the second checks if an error occurred in the implementation of a belief, even though it is structurally correct.

Is the appropriate event triggered in the manipulation of the belief?

The test of belief must consider whether, when updated, this action correctly provokes the expected effects. So, if upon the updating of a belief's value an event is triggered, this situation must be verified by the test (ZHANG, 2011).

3 A new approach for testing deliberative agents

The development of an approach for testing deliberative agents, which in this paper is being considered a synonym for BDI agents, is not a trivial task and some restrictions were taken to limit the scope of the work. In this section the restrictions adopted during the development of the proposed approach and the implementation of unit test tool to support the construction of BDI agent test cases are described. The project's first decision was to adopt BDI4JADE as the BDI agent development platform. The reasons for this decision derive from the fact that, in BDI4JADE, the BDI agents are written entirely in the Java language, so it is not necessary to use the platform's own language for the development of the agents or additional configuration files.

We also restricted the scope of the tests to the unit level. Some researchers believe that individually testing each agent comprising an MAS is not a relevant task, because it does not guarantee the operation of the system when these are put together (WINIKOFF and CRANEFIELD, 2010). Our approach is opposite to this way of thinking; rather, like other researchers, we believe that the correct and individual functioning of the agent is indispensable for the functioning of the MAS as a whole (ZHANG, THANGARAJAH and PADGHAM, 2007). The approaches dealing with integration tests and system tests are still necessary and essential to the final quality of the MAS.

We adopted the fault model proposed by Zhang (ZHANG, 2011) for this paper as a guide to the construction of the assertive methods that the tool makes available. This fault model tackles and describes the failure situations of the elements that make up the agent.

3.1 Method proposed

As mentioned before, this paper adapted the ideas used by the JAT Framework to enable the BDI agent test written in BDI4JADE. As illustrated in Figure 2, below are the main elements used by the proposed solution:

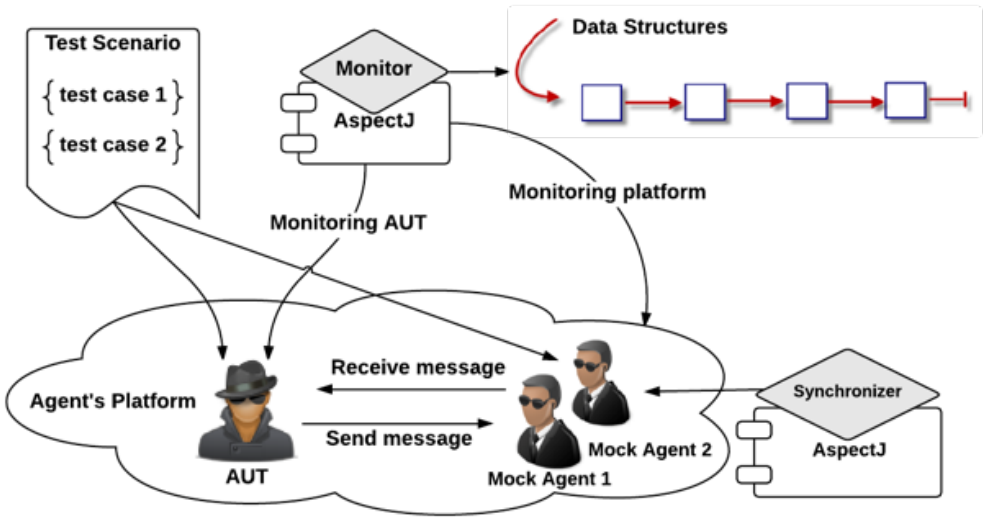

Figure 2 – Participants of the unit test flow.

- *AUT (Agent Under Test)*: represents the agent whose behavior will be checked by running a test scenario;
- *Mock Agent*: element created strictly to interact with the AUT, represents a "fake" implementation of a real agent;
- *Monitor*: element responsible for monitoring the agent's reasoning cycle and filling the data structures with the information extracted during the running of the test case;
- *Synchronizer*: element that organizes the running order of the mock agents in a test scenario and;
- *Test case*: that defines a condition to which the AUT will be exposed, that is, creates the Mock agent(s) that will interact with it and check if the AUT obeys its specification in these conditions;

All the elements presented already were seen in Section 2.5 when the JAT Framework was presented. Some adaptations were made to include in the BDI agents test. They are:

- *AUT*: whereas in the JAT the AUT represents an agent written in JADE, in our approach the AUT represents a BDI agent written in BDI4JADE;
- *Monitor*: in the JAT, it is responsible for monitoring the internal states of the agents while, in our approach, the monitoring is carried out in the agent reasoning cycle;
- *Test cases*: in our approach, the developer uses assertive methods to test and verify agent behavior;

3.2 The data structured involved

As has been said, one of an agent's main characteristics is autonomy. Each agent has its own execution "thread" and its behavior is determined by a set of inferences made from its beliefs, goals and plans (GARCIA, LUCENA and COWAN, 2004). With the objective of effectively testing the agent – taking into account that the tests are based on some form of comparison of the expected result with the result produced – it is necessary to know how each of the agent's components behaved in each step of the reasoning cycle during execution. Therefore, the information on the state of the beliefs, plans and events is useful for checking if the agent's behavior was as expected.

The idea of creating and maintaining structures containing the transitions of an agent's states adopted by the JAT Framework led to the building of a new set of data structures capable of storing the information about the internal components of the agents, seeking to identify the occurrence of possible errors according to the fault model introduced in Section 2.6. To obtain this result, we must add code (in the agents and platforms involved, which in our case is the BDI4JADE and JADE itself). These codes should be added at all points where changes occur in the state of a component and where important decisions are taken in the deliberative mechanism. Doing so, however, the added code would be scattered in many places and in many platform modules. We realized then that, as occurs with the JAT, monitoring the agent's reasoning cycle is, of course, a transversal interest; in these cases, one solution, widely adopted with regard to the monitoring of transversal interests, is to define an aspect to point directly to the agents' execution locations, that is, in their reasoning cycle, and in the platform code representing the component state transitions (BRIAND, LABICHE and LEDUC, 2005) (COELHO, *et al.*, 2006). We use the strategy to monitor the agent and platform reasoning

cycle and platform through an aspect, implemented in the ASPECTJ language. To store the internal state of the components of the AUT during the running of the test case, the following data structures were created, as seen in Table 1.

Table 1. Populated data structures in the running of the AUT

Data structures with the information about the AUT	
Set of Plans	Storage of agent's plans.
Set of Capacities	Storage of agent's capacities.
Set of Events	Storage of triggered events.
Set of Messages	Storage of messages received and sent by the agent.
Set of Goals	Storage of agent's goals.

The structures presented previously are used to classify the internal elements of the agent as follows:

Beliefs inserted: are the beliefs inserted in the agent's beliefs base during running;

Beliefs removed: are the beliefs removed from the agent's beliefs base during running;

Beliefs updated: are the beliefs whose contents have been changed;

Plans executed: are the plans that the agent executed;

Plans not executed: are the plans from the agent's plans library that were not executed;

Goals achieved: are the goals that the agent managed to achieve;

Unreached goals: are the goals the agent failed to achieve during its execution;

Intentions: are the goals that, at some point in the moment the agent was running, were pursued by the agent.

Messages sent: messages sent by the agent (with all message information)

Messages received: messages received by the agent (with all message information)

Events triggered: are the events that occur during agent running

The data structures presented above were used by the assertive methods that, upon consulting their contents, are able to check if when the agent was in execution an internal component of the agent was able or not to reach an expected state.

3.3 Synchronizing the parts involved

To synchronize the parts presented in Sections 3.1 and 3.2, a test scenario is chosen and partially implemented by the plan executed by each mock agent that participates in this scenario. Sometimes, depending on the test scenario, it is necessary to define in which order the mocks will interact with the AUT. This synchronization is, traditionally, handled through construction of synchronized code snippets and data structures, in the places that need to be synchronized. Similar to what occurs with the data structures, the synchronization code can be spread by the code of the mock agents and by the main functionalities, preventing the reuse of the mock agents in different test scenarios. Thus, in a manner analogous to the monitoring of the agent reasoning cycle, synchronization is also a transversal interest and can effectively be implemented as an aspect.

In our approach, like with the JAT, an aspect is defined to implement this interest. The *Synchronizer Aspect* is the element responsible for the composition of the test scenario that is partially implemented in each mock agent's plans. This element is responsible for defining the time each mock agent must take action in a test scenario. However, the Synchronizer defines only the order in which each mock agent will interact with an AUT. It does not define the actions executed by the mock agents, which must be implemented in the behavior of each mock agent involved in the AUT scenario.

Representing the synchronization of the mock agents as an aspect allows the reuse of this mock agent in different test scenarios without the need to change its code.

3.4 The JAT4BDI tool: design and implementation

This section presents, in detail, the JAT4BDI tool, developed to serve as proof of concept for the proposed testing approach. The components, structures and the use of the tool for its purpose of supporting the development of test cases for agents written in BDI4JADE will be presented. Figure 3 presents the approaches and technologies used in the JAT4BDI solution of its interdependencies.

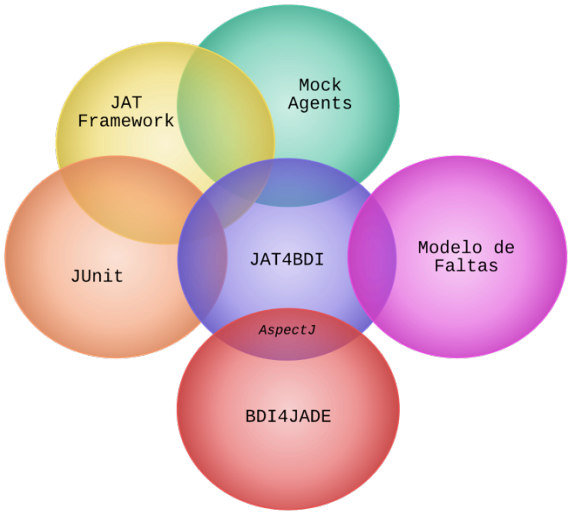

Figure 3 – Dependencies between the JAT4BDI and the approaches used

3.4.1 JAT4BDI details

The tool is organized internally in four packages: (i) the core package; (ii) the annotations package; (iii) the aspects package and; (iv) the faultinjection package, as shown in Figure 4. The *core package* contains the classes responsible for supporting the creation and execution of test cases and the checking of the information collected from the reasoning cycle and the agents' internal states during the running of an agent in the test (AUT). The *annotations package* contains the classes (JAVA annotations) associated with each failure of the fault model and used to annotate parts of the code where it is desired to inject a certain falt for verification. The *aspects package* also contains the classes (aspects developed in the AspectJ language) responsible for the monitoring of the agents' reasoning cycle and for the synchronization of the execution of the test cases between the AUT and the mock agents. Finally, the *faultinjection package* contains a class (this is also an aspect implemented in the AspectJ language) responsible for injecting errors in the behavior of the AUT.

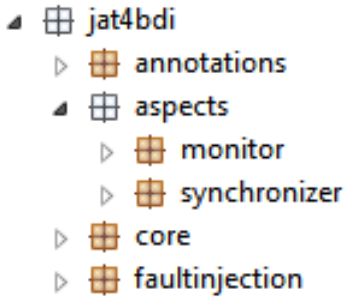

Figure 4 – Packages structure used by the tool

The classes contained in the core package provide the mechanisms to support the construction and maintenance of the test cases. The main class of the package is *JAT4BDITestCase*. This class extends the TestCase class of JUnit, which provides the entire infrastructure necessary for running automated unit tests. The JAT4BDITestCase class also provides methods to run mock agents and a set of assertive methods for checking the agent's state, as shown in Figure 5.

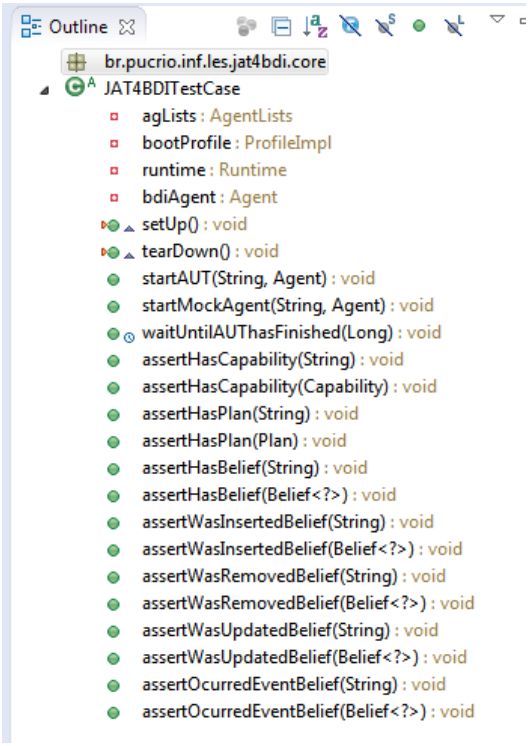

Figure 5 – Attributes and operations supplied by the JAT4BDITestCase

Another important core package class is *JAT4BDIMockAgent*, which implements the concept of mock agents in the JAT4BDI tool. Unlike the agents being tested (AUT), the mocks are simple JADE agents and extend the Agent class. This decision is based on the simplicity of behavior that the mock agents should represent. Thus, the plan of a mock agent is represented by a JADE behavior. Another important point is that the mock agents must report the result of the AUT interaction. For this, an interface was defined that contains a set of methods that must be implemented by the agent that wants to report the result of the interaction with the AUT. Therefore, to report the results, the JAT4BDIMockAgent class implements the *JAT4BDITestReporter* interface. Another way to check the result of the interaction between the mock agent and the AUT is to use the assertive methods to verify the information exchanged (content, performative, etc.) with the mock agents.

The aspects package contains the classes responsible for monitoring agents and the storage of information for analysis and verification. The main classes contained in this package are: *AgentLists*, *ReasoningCycle* and *Synchronizer*. The AgentLists class maintains the data structures set capable of storing the reasoning cycle information and the internal state of the agents. These structures are filled during the running of the test agent through the ReasoningCycle aspect, responsible for agent monitoring.

The ReasoningCycle class defines the aspect responsible for monitoring and collecting information about the reasoning cycle and the internal state of the AUT and the filling out of the previously submitted data structures. For this, the platform or agent points are intercepted to collect information on its operation, as shown in figure 6 where a code snippet responsible for this monitoring is presented.

```
1  public aspect ReasoningCycle {
2
3      private AgentLists agLists = AgentLists.getInstance();
4
5      pointcut addbeliefMap(Belief belief) : execution(void BeliefBase.addBelief(..)) && args(belief);
6
7      after(Belief belief) : addbeliefMap(belief) {
8          agLists.addMapOfBelief(belief);
9          agLists.addInsertedBeliefSet(belief);
10     }
11
12     pointcut removedBeliefSet() : execution(* BeliefBase.removeBelief(..));
13
14     after() returning (Belief belief) : removedBeliefSet() {
15         agLists.addRemovedBeliefSet(belief);
16     }
17
18     pointcut updatedBeliefSet(String name, Object value) : execution(* BeliefBase.updateBelief(..)) && args(name, value);
19
20     after(String name, Object value) returning (boolean updated) : updatedBeliefSet(name, value) {
21         if (updated) {
22             Belief<string> belief = new TransientBelief<string>(name, value.toString());
23             agLists.addUpdatedBeliefSet(belief);
24         }
25     }
26 }
```

Figure 6 – The ReasoningCycle aspect fills out the data structures.

Another aspects package class is the Synchronizer. This is an aspect responsible for "orchestrating" the execution of mock agents in the test cases. In its operation, the aspect adds a snippet of code before the messaging code (lines 6-15), causing the mocks to check whether it is their turn to send the message to the AUT. The OrderList class contains the identifiers of the mock agents that must send a message to the AUT in a specific test scenario, ordered by the interaction priority (line 7). So, if the return from the orderList.checkTurn() method is true, the agent can send the message to the AUT; otherwise, the agent waits a few seconds and checks again if it is its turn to send the message pursuant to the code presented in the lines (lines 10-12) of Figure 7.

```
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 a sua vez, senão ele dorme
9          while(!orderList.checkTurn(agent.getAID())){
10             ...
11             Thread.sleep(500);
12             ...
13         }
14     }
15 }
16
```

Figure 7 – Partial Synchronizer aspect Code.

There is also the annotations package containing the JAVA annotations. These notes are responsible for configuring the faults that will be injected into the agents.

Finally, the faultinjection package contains the class responsible for injecting faults in the AUT verifying the effectiveness of the assertions provided. This is an aspect that

during the running of the test case injects a specific fault into the AUT that the developer wants to check.

Figure 8 shows the class diagram with the main tool classes. Thus, it is possible to have a view of the main classes and their dependencies.

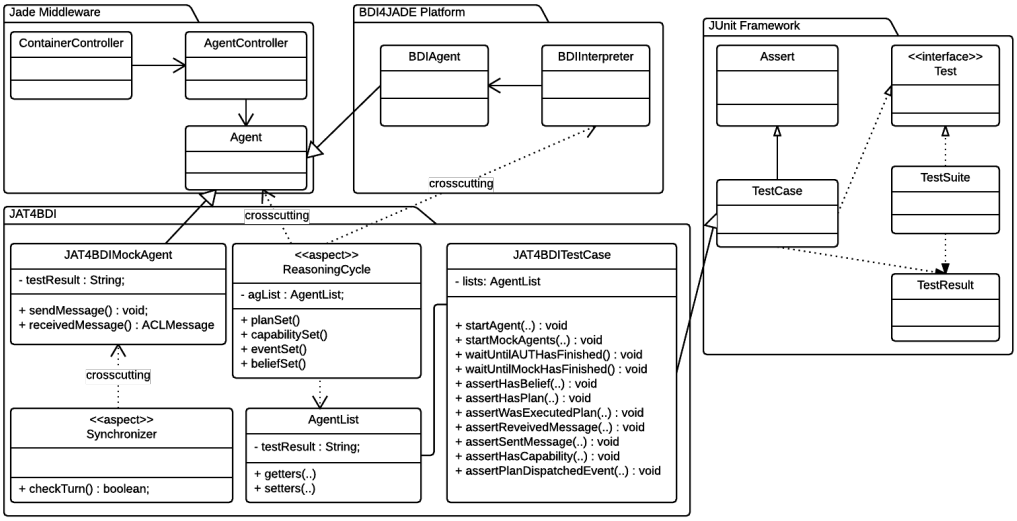

Figure 8 – Class diagram with the main tool classes

3.4.2 Assertions

To assist the development of test cases as well as the verification and analysis of the behavior of the internal state and the test agent reasoning cycle, the tool provides a set of assertive methods in JUnit style in order to identify possible errors in the running of an agent considering, for this, the error conditions presented in the fault model used. These are not JAVA assertions that are inserted in the code of the agents and guarantee a certain behavior or state of AUT, but rather methods offered by the tool itself to support the identification of the errors or unwanted behavior that occurred.

Below is the list of methods provided by JAT4BDI to support the development of test cases:

Methods for configuration of the environment:

- *setUp*: method run before the test case, used for settings;
- *tearDown*: run at the end of the execution of the test case;
- *startAUT*: start AUT execution;
- *startMockAgent*: start a mock agent execution on the platform;
- *waitUntilAUThasFinished*: waiting for the end of the AUT execution;
- *waitUntilMockHasFinished*: waiting for the end of the mock agent execution;

Tables 2-6 above lists methods that support the identification of faults associated with beliefs, plans, capabilities, messages and events, respectively.

Table 2. Methods that support error identification associated with the beliefs

Verification Method	**Goal Description**

assertHasBelief	checks if the agent has a belief in its belief base
assertWasInsertedBelief	checks whether a belief was inserted into the belief base during agent execution
assertWasRemovedBelief	checks whether a belief has been removed from the belief base during agent execution
assertWasUpdatedBelief	checks whether a belief's value was changed during agent execution

Table 3. Methods that support the identification of errors associated with the plans

Verification Method	Goal Description
assertHasPlan	checks if a plan is part of the agent's plan library
assertWasExecutedPlan	checks whether a plan was executed
assertHasAssociatedGoal	checks if a plan is associated with a goal (agent objective)
assertHasCyclePlan	checks if there was a cycle during agent execution
assertPlanDispatchedEvent	checks if the plan triggered an event

Table 4. Methods that support error identification associated with capabilities

Verification Method	Goal Description
assertHasCapability	checks whether an agent has a capacity
assertHasInCapability	checks whether a component has a capacity which can be: a belief or a plan

Table 5. Methods supporting the identification errors associated with message exchanges

Verification Method	Goal Description
assertReceivedMessageFrom	checks if the agent received a message from another agent
assertSentMessageTo	checks the contents of the message received by the AUT from another agent
assertContentReceivedMessageEquals	checks the contents of the message received by the AUT from another agent
assertContentSentMessageEquals	checks the message content sent by the AUT to another agent
assertPerformativeReceivedMessageEquals	checks the performance of the message received by the agent
assertPerformativeSentMessageEquals	checks the performance of the message sent by the AUT

Table 6. Method which supports the identification of errors associated with events

Verification Method	Goal Description
assertDispatchedEvent	checks if an event was triggered by a plan or a belief

3.4.3 Steps for execution

The *JAT4BDITestCase* class performs the construction of test cases. The developer must build test cases in the JUnit style in order to include the entire specification of the specified test scenario. For this, the verification methods presented in the previous section are used. The execution of a test scenario follows the flow shown in Figure 9.

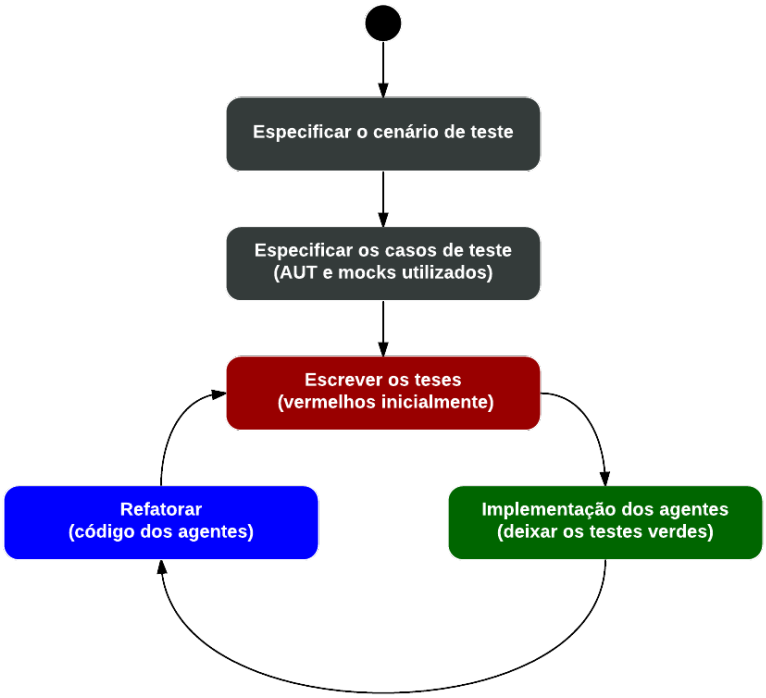

Figure 9 – Workflow to run unit tests in JAT4BDI.

The use of the tool starts with the specification of the test scenario in which we want to check the behavior. Once the test scenario has been specified, the next step is to specify test cases that cover the specified scenario. Table 7 presents a test scenario model.

Table 7. Model for describing a test scenario

Agent	**Agent Under Test (AUT)**
Test scenario (input)	Describes the entry of the test scenario that includes the entries for the AUT in this scenario and the initial state of the agent and other variables in the environment
Expected Outcome	Describes the expected behavior for AUT - if it should send a message or change the environment

The focus of the proposed work is to build test cases and, thus, the test case developer must create a class extending the JAT4BDITestCase class. After creating the class that represents the test case, the developer can implement the tests using the methods offered by the tool to create the mock agents, the AUT and verify the result after running the agent. Figure 10 shows a simple example for the verification of a belief through the *testVerifyingBelief* method that checks if the agent has the "message" belief in its belief base.

```
1  public class HelloWorldTestCase extends JAT4BDITestCase {
2
3      static final long DELAY = 50001;
4
5      public void testVerifyingBelief() {
6
7          startAUT("HelloWorld", new BDIAgentHelloWorld());
8
9          waitUntilAUThasFinished(DELAY);
10
11         assertHasBelief("message");
12
13     }
14 }
```

Figure 10 – An example of execution of a test case in JAT4BDI.

The result of the test execution above is represented by figure 11.

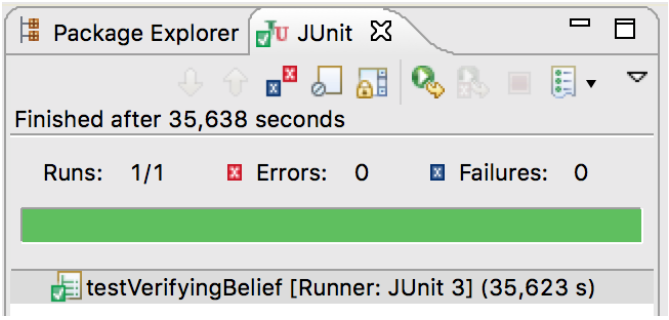

Figure 11 – Results of the execution of a test of the test case.

4 Use Scenarios

This section demonstrates the applicability of the approach proposed by the JAT4BDI tool.

4.1 Book Trading System

This section presents an example that is well known and provided on various multiagent platform system: the Book Trading System. This is a book commerce application in which each agent can play the role of a seller, a buyer, or both. Figure 12 (B) details the interaction protocol between these roles. The FIPA CONTRACT NET Protocol (FIPA, 2000) between the seller agent and the buyer agent is established, as illustrated in Figure 12 (A).

According to Figure 12 (B) as soon as a seller agent joins the room it registers on the platform's yellow pages service as a "book-seller" and awaits orders from a buyer. When a buyer agent joins the environment it looks for "book-seller" agents registered in the yellow pages and start interacting with them.

When the seller agent receives a "CPF" type of message from a buyer, it looks for the book requested in its catalogue of books. If the book is available, the seller agent sends a "PROPOSE" message in response to the "CPF", whose content is the book' price. On the other hand, if the book is not in the seller agent's catalogue, it sends a "REFUSE" message to the buyer agent that the book is not available. The buyer agent receives all proposals and rejections of selling agents and chooses the one with the best offer and, then, sends the chosen seller a "PURCHASE" message. When the seller agent receives a

"PURCHASE" message it removes the book from the catalogue and sends an "INFORM" message to notify the buyer's agent that book's sale was completed. However, if for some reason the book is no longer available in the catalogue, the seller agent sends a "FAILURE" message informing the buyer agent that the requested book is no longer available. If the buyer agent receives a message indicating that the purchase was completed it can shut down. Otherwise, it will again execute its plan to try to buy the book from another agent.

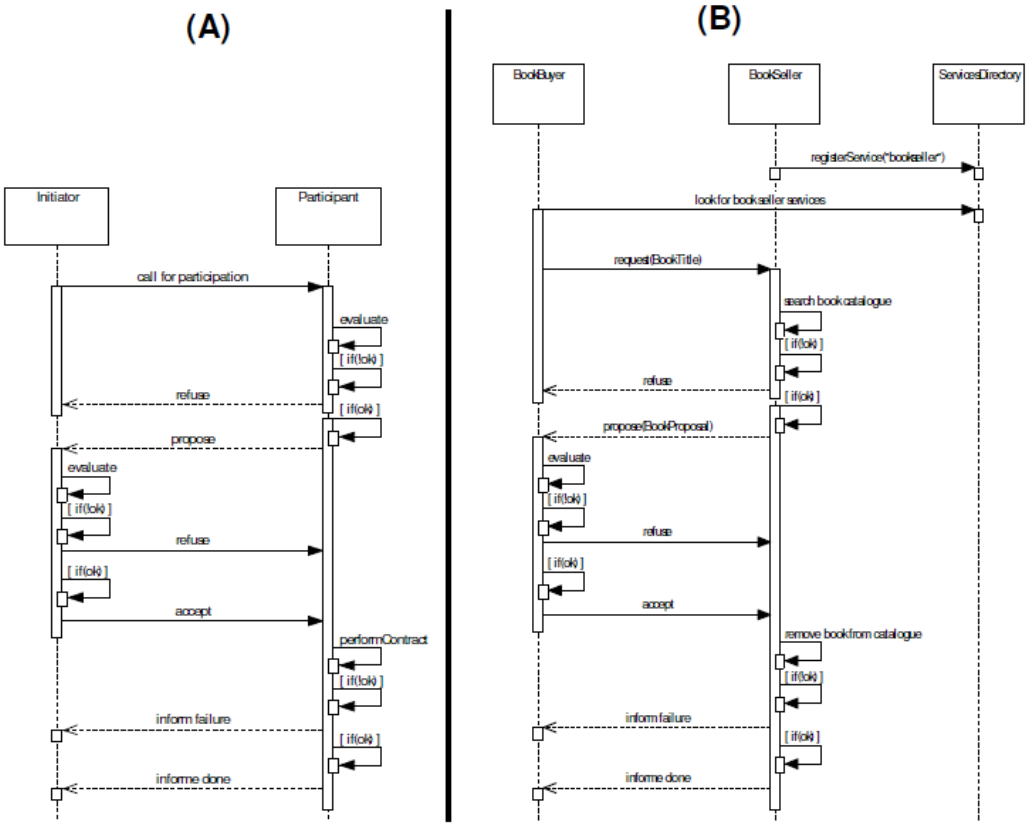

Figure 12 – FIPA CONTRACT-Net Protocol (A) and (B) the Book Trading System.

4.1.1 Description of the usage scenario

Using the *error-guessing technique*, the test scenario presented in Table 8 was defined. In this scenario, the agent that must deal with the exceptional condition will be the test scenario AUT, and mock agents will represent all other agents.

Table 8. Book Trading System test scenario example

Agent Under Test	**Seller Agent (BookSeller)**
Test Scenario	Two buyer agents (BookBuyer) try to buy the same book from the seller agent (BookSeller) but it has only one copy available
Expected Outcome	The seller agent (BookSeller) should sell the book to the first agent who requested its purchase and reject the request of other agents

Table 9 presents test cases that must be implemented in the tool in order to support verification of the agent behavior described in the test scenario and identification of possible errors.

Table 9. Bookseller agent test cases

Item to be tested	Test Objective
Belief representing the book catalog	Check if a belief containing the books catalog information was created
belief Value	Check if the book is available in the catalogue
plan BookSellerPlan	Check if the BookSellerPlan plan is present in the agent's plan library
plan BookSellerPlan	Check if the plan was BookSellerPlan
plan BookSellerPlan	Check if the BookSellerPlan plan received a purchase request properly
plan BookSellerPlan	Check if the BookSellerPlan plan announced the availability of the book correctly
plan BookSellerPlan	Check if the BookSellerPlan plan was the purchase of book proposal
plan BookSellerPlan	Check if the plan BookSellerPlan informed the buyer agent that the purchase was finalized

4.1.2 Test cases – Book Trading System

The verification of the agent behavior is initiated by the test described in Figure 13. It checks the existence of the "catalogue" belief in the agent's knowledge base (line 11).

```
1  public class BookSellerTestCase extends JAT4BDITestCase {
2
3      static final long DELAY = 50001;
4
5      public void testVerifyingBelief() {
6
7          startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
8
9          waitUntilAUThasFinished(DELAY);
10
11         assertHasBelief("catalogue");
12     }
13 }
```

Figure 13 – Checks the existence of the belief in the knowledge base.

The next step is to check the value of the belief. Thus, it is possible to verify that the seller agent has the copy of the book the buyer wishes. Line 18 shows the verification of the value of the belief, as shown in Figure 14.

```
1  public class BookSellerTestCase extends JAT4BDITestCase {
2
3      static final long DELAY = 5000l;
4
5      public void testVerifyingBeliefInCatalogue() {
6
7          startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
8
9          waitUntilAUThasFinished(DELAY);
10
11         Book book = new Book("Programação Modular", 100.00, 2);
12
13         List list = new ArrayList();
14         list.add(book);
15
16         Belief belief = new TransientBelief("catalogue", list);
17
18         assertHasBelief(belief);
19     }
20 }
```

Figure 14 – Checks the value of the belief in the agent's knowledge base.

The next test checks whether the *BookSellerPlan* plan is present in the agent's plan library. This plan is responsible for selling the book to the buyer agent. Line 13 is the check mentioned pursuant to Figure 15.

```
1  public class BookSellerTestCase extends JAT4BDITestCase {
2
3      static final long DELAY = 5000l;
4
5      public void testVerifyingPlan() {
6
7          startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
8
9          waitUntilAUThasFinished(DELAY);
10
11         Plan plan = new SimplePlan("BookSellerPlan", BookSellerPlan.class);
12
13         assertHasPlan(plan);
14     }
15 }
```

Figure 15 – Checks the existence of the plan in the agent's plans library.

The test shown in Figure 16 checks whether the plan *BookSellerPlan* was executed (line 17). The test also initializes the buyer agents as mock agents (lines 7 and 8). Each mock agent awaits its moment for interacting with the AUT (lines 12 and 13).

```
1  public class BookSellerTestCase extends JAT4BDITestCase {
2
3      static final long DELAY = 5000l;
4
5      public void testVerifyingExecutedPlan() {
6
7          startMockAgent("MockFirstBuyer", new MockFirstBuyer());
8          startMockAgent("MockSecoundBuyer", new MockSecoundBuyer());
9          startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
10
11         waitUntilAUThasFinished(DELAY);
12         waitUntilMockHasFinished("MockFirstBuyer");
13         waitUntilMockHasFinished("MockSecoundBuyer");
14
15         Plan plan = new SimplePlan("BookSellerPlan", BookSellerPlan.class);
16
17         assertWasExecutedPlan(plan);
18     }
19 }
```

Figure 16 – Checks if the *BookSellerPlan* plan was executed.

Figure 17 shows the test where the AUT receives the purchase request from the mock agent *MockFirstBuyer*. This test case verifies that the AUT has received a *CPF* type message (line 18).

```
1  public class BookSellerTestCase extends JAT4BDITestCase {
2
3      static final long DELAY = 5000l;
4
5      public void testVerifyingRequest() {
6
7          startMockAgent("MockFirstBuyer", new MockFirstBuyer());
8          startMockAgent("MockSecoundBuyer", new MockSecoundBuyer());
9          startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
10
11         waitUntilAUThasFinished(DELAY);
12         waitUntilMockHasFinished("MockFirstBuyer");
13         waitUntilMockHasFinished("MockSecoundBuyer");
14
15         ACLMessage message = new ACLMessage(ACLMessage.CFP);
16         message.setContent("Programação Modular");
17
18         assertPerformativeReceivedMessageEquals(message.getPerformative());
19     }
20 }
```

Figure 17 – Checks whether the type and message contents are correct.

The next test, as per Figure 18, checks if the response of the AUT to the buyer's request is correct. In this case, the AUT must respond by sending a proposal (line 18) and the price of the requested book (line 19).

```
1  public class BookSellerTestCase extends JAT4BDITestCase {
2
3      static final long DELAY = 5000l;
4
5      public void testVerifyingSellerProposal() {
6
7          startMockAgent("MockFirstBuyer", new MockFirstBuyer());
8          startMockAgent("MockSecoundBuyer", new MockSecoundBuyer());
9          startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
10
11         waitUntilAUThasFinished(DELAY);
12         waitUntilMockHasFinished("MockFirstBuyer");
13         waitUntilMockHasFinished("MockSecoundBuyer");
14
15         ACLMessage message = new ACLMessage(ACLMessage.PROPOSE);
16         message.setContent("100.00");
17
18         assertPerformativeSentMessageEquals(message.getPerformative());
19         assertContentSentMessageEquals(message);
20     }
21 }
```

Figure 18 – Checks that a proposal was sent by the seller agent.

Figure 19 shows the test that confirms the purchase request. The AUT receives a message from the buyer agent confirming the purchase of the book (line 17).

```
1  public class BookSellerTestCase extends JAT4BDITestCase {
2
3      static final long DELAY = 50001;
4
5      public void testVerifyingBuyerConfirmation() {
6
7          startMockAgent("MockFirstBuyer", new MockFirstBuyer());
8          startMockAgent("MockSecoundBuyer", new MockSecoundBuyer());
9          startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
10
11         waitUntilAUThasFinished(DELAY);
12         waitUntilMockHasFinished("MockFirstBuyer");
13         waitUntilMockHasFinished("MockSecoundBuyer");
14
15         ACLMessage message = new ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
16
17         assertPerformativeReceivedMessageEquals(message.getPerformative());
18     }
19 }
```

Figure 19 – Checks the buyer agent's purchase confirmation.

Finally, test scenario ends with the test case shown in Figure 20, which checks the sending of the AUT's message to the buyer agent informing the completion of purchase (line 17).

```
1  public class BookSellerTestCase extends JAT4BDITestCase {
2
3      static final long DELAY = 50001;
4
5      public void testVerifyingSellConclusion() {
6
7          startMockAgent("MockFirstBuyer", new MockFirstBuyer());
8          startMockAgent("MockSecoundBuyer", new MockSecoundBuyer());
9          startAUT("BookSellerBDIAgent", new BookSellerBDIAgent());
10
11         waitUntilAUThasFinished(DELAY);
12         waitUntilMockHasFinished("MockFirstBuyer");
13         waitUntilMockHasFinished("MockSecoundBuyer");
14
15         ACLMessage message = new ACLMessage(ACLMessage.INFORM);
16
17         assertPerformativeSentMessageEquals(message.getPerformative());
18     }
19 }
```

Figure 20 – Checks the sending of the purchase completion message.

The group of test cases shown above verified each stage of the checking of the CONTRACT-NET protocol. An error in any of the above checks represents a failure in the communication protocol proposed for the agents.

4.1.3 Running of the test cases – Book Trading System

The execution of all scenario test cases allows the developer to observe the behavior and interaction between the seller agent (AUT) and the buyer agent. Figure 21 shows the test results.

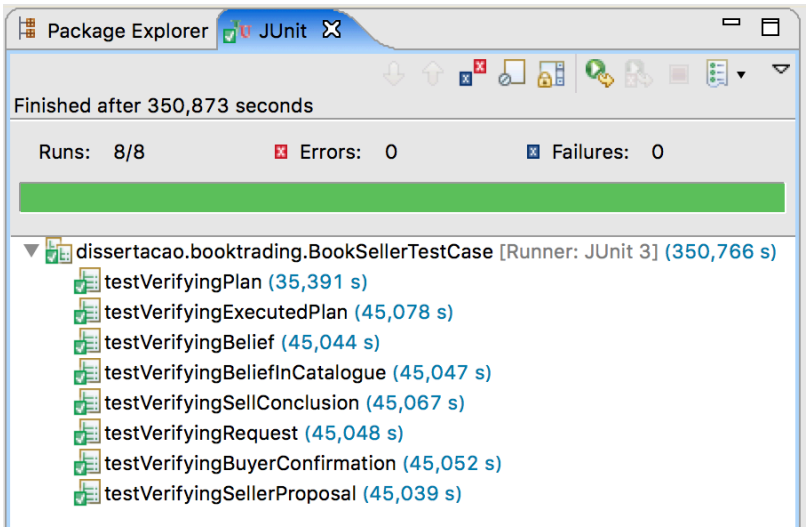

Figure 21 – Results of the running of the test cases.

Figure 22 shows the BDI4JADE platform execution log reporting that the seller agent achieved its goal.

```
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.
```

Figure 22 – Log informing that the seller agent achieved its goal.

4.2 Results observed

Incremental tests were performed using JAT4BDI for the previously described example. First, test cases were defined where the error-guessing technique (Meyer, 1997) (BEER and RAMLER, 2008) was used. Given the test scenario for each defined test case, there was only one AUT and mock agents represented the other agents that interact with the AUT. The JAT4BDI tool was used to support the development of all the examples described above and helped several times with regard to error identification, such as: (i) error in the agent belief value configuration; (ii) failure to implement a plan, that is, sometimes the plan was not executed for not being associated with the agent's goal; (iii) failure to exchange messages between agents because of errors in the message content or the performative used.

5 Limitation of the related works

This section describes some of the works related to the proposed work that guided decisions about the scope of this research.

The work of Winikoff and Cranefield (WINIKOFF and CRANEFIELD, 2010) presents an analysis of the flexibility and the adaptive characteristics of the BDI agents observing the behavioral space of the agent, that is, the number of possible ways of achieving a

goal. Their work also has sought to understand what are the factors that influence the size of the behavioral space and viability to ensure the effectiveness of multiagent system through tests. Thus, Winikoff and Cranefield related the feasibility of the test of a multiagent system to the proportion of paths taken in the behavioral space, and moreover, considering that running a test consists of observing a way of execution and determining whether this is correct or not. The concern of Winikoff and Cranefield's work is not to verify whether a given path is correct, but rather whether it is possible to ensure the effectiveness of the system through testing. The conclusions of Winikoff and Cranefield on the feasibility of multiagent system test corroborated the decision to delimit the scope of the proposed approach to the unit testing of agents and not for system or integration testing.

In their work, Coelho *et al*. present a framework for testing multiagent system (COELHO, CIRILO, *et al.*, 2007) based on the use of "mock agents," that is through a "fake" implementation of a real agent for the sole purpose of testing the communication between agents (COELHO, KULESZA, *et al.*, 2006). By monitoring the transition of the internal state of the agents, the JAT controls and observes the interaction between the mock agents and the AUT. This monitoring is done through aspects written in the ASPECTJ language. Despite the good contribution in relation to the software agents test, the work of Coelho is limited to the testing of agents with reactive behavior. The work was further limited, by using an fault model that focuses basically on finding errors in the communication protocol between agents. The work of Coelho contributed in a significant way to our approach, inspiring the use of mocks to test the interaction between the agents and the framework execution model for the creation of the JAT4BDI tool. We also adopted the idea of using aspects to monitor the reasoning cycle of the BDI4JADE agents and store their states in internal data structures for subsequent observation.

The Zhang's work (ZHANG, THANGARAJAH and PADGHAM, 2009) presents a framework for automatic generation of test cases for multiagent system. This work considers the construction of model-based multiagent systems (APFELBAUM and DOYLE, 1997) (EL-FAR and WHITTAKER, 2001), in this case the Prometheus (PADGHAM and WINIKOFF, 2004), developed during the MAS project. In this approach, the model designed for MAS provides inputs to the framework of what should be tested. This work also presents a model to identify possible errors and the conditions under which they may occur. The work focuses on the unit test of the agent's internal components, conducting a directed error test, where the goal is to reveal possible implementation errors (BINDER, 1999). Despite presenting an interesting and relevant approach regarding the agent test, the main focus of the work of Zhang is automated generation of test cases. In relation to the fault model, despite precisely describing the situations and error conditions, it is used to support the decisions of the generation of test cases, not providing any mechanism for observation of the internal state of the agent's components when an error is identified. The contribution of the work of Zhang to the approach proposed was quite important, providing an indication of what should be tested on agents and precisely indicating the situation and condition of possible errors in the agent's attempt to achieve its goals.

6 Conclusion and future work

The use of agent technology for developing distributed software has shown promise for this type of system. Its use in various business domains, especially in critical scenarios for human activity, is a strategy that increasingly is being adopted. For these critical

scenarios, analysis and software behavior verification become crucial. However, the methods proposed so far by Agent-Oriented Software.

Engineering Agents (AOSE) have focused their efforts primarily on disciplined approaches to analyze, design and code an MAS, with little attention given to how such systems could be tested. In this context, this work proposed an approach to support the development of software agents through the construction and maintenance of test cases for deliberative agents (BDI) written in BDI4JADE. This approach was based on the ideas supported by the JAT Framework (the use of mock agents to simulate the interaction between the agent in tests and a real agent and the monitoring of agent behavior through aspects) and in the fault model proposed by Zhang (ZHANG, 2011), describing the possibilities of errors and which agent elements must be observed.

Also in this paper, a tool was proposed to support the construction and execution of automated test cases, the JAT4BDI. Through an exploratory example about how to use the tool and its resources were presented. Through verification methods, similar to those existing in the JUnit Test Framework, the test developer has access to the information from the agent that occurred during its reasoning cycle, helping identify possible errors.

A number of points could be improved and the following is the future work that could be conducted as an outcome of the work proposed: (i) creation of a framework: evolve the current tool so that it becomes a framework. Possible extension points for the framework would be: the fault model used, the monitoring mechanism of the agent reasoning cycle making it possible that agents written on other platforms could be tested and not only BDI4JADE agents, the mechanism for the creation of customized assertive methods by the test case developers; (ii) conducting an experimental study: usage scenarios designed to present the approach and the tool were used. However, we propose carrying out a study to test its effectiveness and efficiency; (iii) checking normative behaviors: extending the tool to allow the unit testing of regulatory agents, that is, agents whose behavior is governed by some external rule.

References

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.

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.

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, p. 84-93, September 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.

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

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 Multi-agent 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 International Conference on Multi-agent 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.

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.

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. edition. Wiley, 2002.

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 Multi-agent Systems, 2007.

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