# Understanding Normative BDI Agents Behavior

Francisco Cunha, Marx Viana, Tassio Sirqueira, Marcio Rosemberg and Carlos Lucena
Pontifical Catholic University of Rio de Janeiro (PUC-Rio)
Software Engineering Laboratory (LES)
Rio de Janeiro, Brazil
{fcunha, mleles, tmartins, mrosemberg, lucena} @inf.puc-rio.br

*Abstract*— **Testing the autonomy of, and the interaction between, the agents in Multiagent Systems (MAS) is the frontal challenge of traditional software testing approaches. When we study MAS governed by norms – mechanisms created to restrain the behavior of agents – this challenge increases even further. However, agents are autonomous and it is not guaranteed that they will fulfill all norms. Given the fuzzy notion of "test", especially in the context of MAS, in addition to the difficulties of dealing adequately with normative constraints, the overall understanding of how to handle the creation of tests for normative MAS is still vague. This paper proposes a testing tool to build and run MAS test scenarios and it relies on the use of aspect-oriented techniques to monitor the behavior of autonomous agents. We demonstrated our tool with a simulation of a traffic intersection scenario, based on the Brazilian Transit Code. Our experience shows that the tool can be used to build test scenarios that can achieve high fault detection effectiveness.**

***Keywords – BDI Agent; Autonomous Behavior; Nomative Agents; Testing in Multiagent Systems.***

## I. INTRODUCTION

Multiagent Systems (MAS) are societies in which autonomous, heterogeneous and independently designed entities work toward a common goal [9]. To reach this common goal, it is necessary to deal with the agents' autonomy and establish a strategy that will allow open systems to provide social control mechanisms to ensure the desired order [9]. Agent autonomy is very important in MAS, however, from a testing perspective the characteristics of normative agents add many new challenges to software testability. Traditionally, software behavior can be easily tested and understood when compared to a reference behavior, whereas in multiagent systems, the behavior depends on the interactions with other agents in a dynamic environment.

This means that if on one hand agent technology helps to address application requirements of complex systems, on the other hand, its characteristics, such as the autonomy and the use of norms in the environment, bring obstacles to software testability [2]. According to Voas and Muller [2], testability has two facets: (i) controllability – the ability to control the test input, and (ii) observability – the ability to observe the output of the component under test. Agent autonomy impairs observability since agents may employ some degree of nondeterministic behavior. Consequently, it is hard to define (control) the test input that is not only derived from environment data but also from the messages received from concurrent conversations among agents – this is made worse with the use of norms.

This paper presents a tool named N-JAT4BDI: a JUnit-like testing tool implemented in Java and Aspect-Oriented Programming, which is a technique to improve the modularization of crosscutting concerns. N-JAT4BDI has been developed with the purpose of testing agents built in NBDI4JADE [12], a framework for normative agent-based applications that follows the BDI architecture. We can point out the following contributions: (i) an adaptation of JAT4BDI [13] that adds mechanisms to test the relevant properties of normative BDI agents and their interactions with others; (ii) a tool to support the implementation and automatic execution of test cases; (iii) a real test showcasing a Brazilian traffic scenario and (iv) a quality assessment of this test scenario by using a fault injection technique.

The remained of this paper is organized as follows: Section 2 presents the background and related work. Section 3 presents the testing approach to normative MAS. Section 4 presents design and implementation details of the N-JAT4BDI tool. Section 5 presents the usage scenario. Section 6 presents the evaluation of the results. Finally, Section 7 presents our conclusions and future work.

## II. BACKGROUND

This section summarizes the concepts of norms and their use in Multiagent Systems as well as MAS Testing approaches and their limitations.

### *A. Norms and Normative Multiagent Systems*

In MAS, norms are mechanisms commonly accepted as efficient means of regulating agents behavior and representing the way in which agents understand the responsibilities of other agents [4] [9]. The definition of the norms used in this work is represented by the following properties: Addressee, Condition (for example, Activation, Expiration), Motivation (for example, Rewards, Punishments), Deontic Concept, and State. The description of each property is given as such: (i) *addressee* is used to specify the agents or roles responsible for norm compliance; (ii) *activation* is the condition for the norm to become active; (iii) *expiration* is the validity condition for the norm to become inactive; (iv) *rewards* is used to represent the set of rewards to be given to the agent for norm compliance; (v) *punishments* is the set of punishments to be given to the agent for violating a norm; (vi) *deontic concept* is used to indicate whether the norm establishes an obligation, a permission, or a prohibition, and (vii) *state* is used to describe the set of states or actions that are being regulated [10].

DOI: 10.18293 / SEKE2018-175

### B. *Multiagent Systems Testing*

According to Nguyen *et al.* [17], the full testing process of a multiagent system consists of the following levels: unit, agent, integration (or group), system and acceptance.

Several approaches have proposed unit testing for individual agents in multiagent systems [6] [5] [7] [13] [16], whereas few studies deal with the issue of testing a MAS at group level [6] [15] [18] [1] [3]. According to Serrano [18], most approaches have focused on capturing and visualizing messages exchanged among agents, and do not provide ways of tracking the correlation among the agents' behavior. It is important to emphasize that none of the papers mentioned above provides an approach to verify the behavior of normative agents. The main focus of this work is to test the capability of an individual agent to fulfill a norm in order to reach its goal.

## III. The Approach Proposed

This section presents a testing tool – the N-JAT4BDI – that proposes test cases to test normative BDI agents encoded in the NBDI4JADE framework [12]. The N-JAT4BDI tool resorted to the main ideas of the JAT4BDI approach [13] and added new features capable of monitoring the behaviors of normative NBDI4JADE agents.

### A. *Overview*

Our tool simulates real agent interactions by using *mock agents* [14]. The *mock agents* interact and exchange messages with the agent under test (AUT) in order to verify the AUT response and to check whether the environment was affected as expected.

Figure 1 depicts all the participants used in our testing approach: (i) *Agent Under Test (AUT)*: agent whose behavior is verified by the unit test execution; (ii) *Mock Agent*: a fake implementation of a real agent that interacts with the AUT; (iii) *Monitor*: responsible for monitoring agents' behaviors (reasoning cycle); (iv) *Synchronizer*: manages the test scenario execution by defining the order in which the mock agents interact with the AUT; (v) *Test Scenario*: defines a set of conditions to which the AUT will be exposed and verifies whether this agent obeys its specification under those conditions. Each scenario encompasses only one AUT and one, or more, *mock agents*, and (iv) *Test Case*: a particular situation that requires verification.

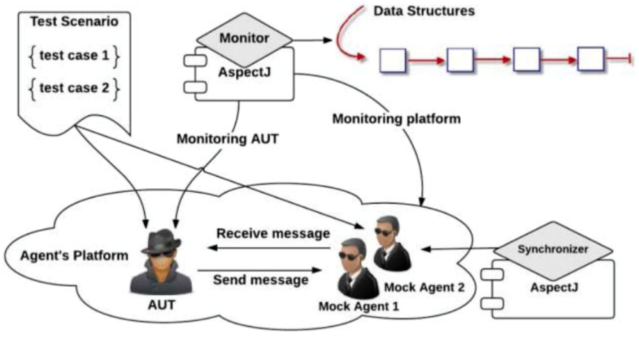


Figure 1. Flow among the participants of the unit test [13]

The workflow steps used by the tool are: (i) to create a *Test Scenario* that will define the test cases involved; (ii) to start the *Test Scenario*; (iii) the test case creates and starts the AUT and *Mock Agents*. The component Monitor starts to observe the AUT's reasoning cycle (its beliefs, executed plans, goals, and norms fulfilled, or refused), and (iv) mock agents exchange messages with the AUT. During this interaction process, the Monitor keeps track of the information gathered during the execution. To do so, it uses a set of data structures to store that information. Both the Monitor and Synchronizer were implemented by aspects [8].

### B. *Normative assertions*

Aiming to support norm fault identification in NBDI4JADE agents, we provide a set of assertive methods that follow the JUnit style and are capable of verifying the agents' decisions. These assertion methods check the information stored by the tool during the agent's execution.

The main assertions are described below: (i) ***assertNormActive****:* It checks whether a norm is active in the environment; (ii) ***assertNormFulfillment***: It checks whether a norm is fulfilled by the agent; (iii) ***assertNormAffectGoal****:* It checks whether a norm affects an agent's goal; (iv) ***assertNormAffectPlan****:* It checks whether a norm affects an agent's plan; (v) ***assertNormAddressee****:* It checks whether a norm is addressed to the agent under test; (vi) ***assertNormExpired****:* It checks whether a norm has expired during the AUT's execution; (vii) ***assertNormReward****:* It checks whether the AUT has received a reward for fulfilling the norm; (viii) ***assertNormPunishment****:* It checks whether the AUT has received a punishment for violating the norm; (ix) ***assertNormDeonticConcept****:* It checks the type of norm constraint (obligation, permission or prohibition) that affects the AUT, and (x) ***assertNormState****:* It checks the internal state of an element that has been regulated.

## IV. N-JAT4BDI: Design and Implementation Details

This section discusses the main classes of the N-JAT4BDI tool.

### A. *NBDI4JadeMockAgent*

The *NBDI4JadeMockAgent* class implements the mock agent concept in N-JAT4BDI and is an instance of the NBDI4JADE class; it has a simple plan that executes a mock agent's single action. The messages exchanged between the mock agents and the AUT is also stored in the internal data structures and can be accessed by using assertive methods.

### B. *Monitor*

The *Monitor* defines the *pointcut* that will intercept both the normative agents and the NBDI4JADE framework in order to observe the agents' reasoning, their decisions and all the changes that occurred in the environment.

### C. *Synchronizer*

The *Synchronizer* intercepts the code of the *NBDI4JadeMockAgent* class and orchestrates the sequence of interaction between the AUT and mock agents.

### D. *NBDI4JadeTestCase*

This class extends the JUnit framework features to support the NBDI4JADE agent tests. As a result, developers can create agent tests more easily since they will be using their own experience with JUnit tests. This is possible because *NBDI4JadeTestCase* provides a set of Junit-based assertive methods to verify the normative agent's behavior and to manage the execution environment before a test scenario starts.

## V. Usage Scenario

Our case study focuses on the simulation of a traffic scenario in Brazil. This section summarizes our experience with the testing tool and its use in this scenario.

### A. *Motivation*

According to **Article 29** of the Brazilian Transit Code (BTC), the right of way rules for vehicles arriving at an uncontrolled intersection are: (i) ***Norm₁***: vehicles moving on main thoroughfares have the preference; (ii) ***Norm₂***: in the case of a traffic circle, the ones circulating around it have the preference, and (iii) ***Norm₃***: in all other cases, vehicles coming from the right have the preference. In addition, **Article 38**, in its sole paragraph, states that before making a right or left turn, or merging onto traffic, the driver must yield to oncoming pedestrians, cyclists and vehicles, always respecting the norms of preference described in **Article 29**.

### B. *Usage Scenario*

This simulation of Brazilian traffic rules was implemented to briefly demonstrate how N-JAT4BDI can be used to test a normative agent. The complete simulation involves autonomous cars (agents), highways, traffic circles, traffic intersections, and traffic rules. The goal of the autonomous cars is to arrive at their destination without accidents, following local traffic rules.

Figure 2 presents the scenario implemented with the NBDI4JADE framework: three cars arrive at an intersection at the same time. The goals of the autonomous cars are: (i) the pink car wants to proceed on street 1; (ii) the yellow car wants to proceed on street 2 and will have to cross street 1, and (iii) the red car is on street 1 and wants to turn left onto street 2. In this scenario, however, there are no traffic signs and the agents need to make decisions to avoid collision among the cars, taking into account Brazil's traffic rules.

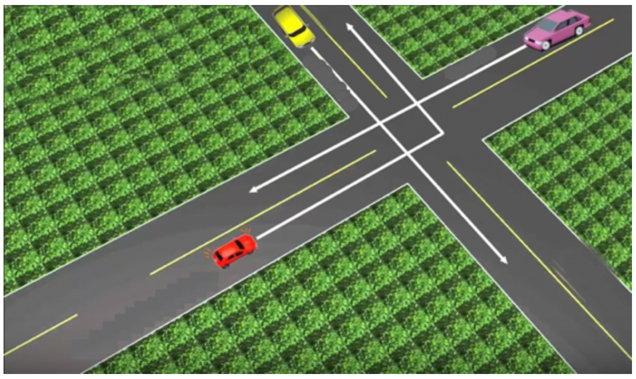


Figure 2. Traffic Intersection rules in Brazil

In our scenario, neither *Norm₁* nor *Norm₂* of **Article 29** of the BTC can be applied. Therefore, the agents need to decide whether they will fulfill, or violate, *Norm₃* of **Article 29**. In our simulation, they all fulfilled *Norm₃*, as follows: (i) The PINK car arrives at the intersection and stops because the YELLOW car is on its right; (ii) The YELLOW car arrives at the intersection and stops because the RED car is on its right; (iii) The RED car arrives at the intersection and there is no car on its right, therefore, the agent's reasoning cannot comply with **Article 29** and must, instead, comply with **Article 38**.

Because the BTC does not deal with similar situations at uncontrolled intersections, it creates an impasse, and requires that the agent's reasoning process be improved.

Due to space limitation, we describe only one simple test scenario and its implementation, as demonstrated in Table 1.

TABLE I. Test Scenario Example

| Agent Under Test | Test Scenario Description |
|---|---|
| The RED autonomous car (Red car) | The RED car arrives at the intersection and there is no car on its right, therefore, the agent's reasoning cannot comply with article 29. |

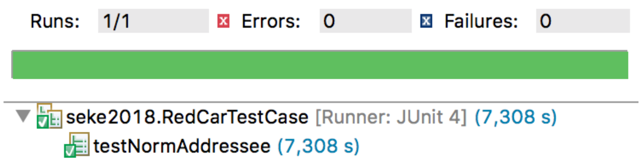
### C. *Encoding test scenario*

Figure 3 illustrates the implementation of the test scenario. Line 13 of the test case starts the agent under test (Red car) and line 15 configures the concurrence from the test environment execution. Line 17 creates a local norm that emulates a real norm in the environment. Line 18 checks whether the norm is *active* in the environment and line 19 checks whether the norm is *addressed* to the agent under test.

```
 7 public class RedCarTestCase extends NJAT4BDITestCase {
 8
 9     static final long DELAY = 5000l;
10
11     public void testNormAddressee() {
12
13         startAUT("RedAutonomousCar", new RedAutonomousCarAgent());
14
15         waitUntilAUThasFinished(DELAY);
16
17         Norm norm = new Norm("article29");
18         assertNormActive(norm);
19         assertNormAddressee(norm);
20     }
21 }
```

Figure 3. Checks if the norm is active and was addressed to the Red agent

Figure 4 depicts the result of the test case execution visualized in a JUnit style.


Runs: 1/1 Errors: 0 Failures: 0

seke2018.RedCarTestCase [Runner: JUnit 4] (7,308 s)
testNormAddressee (7,308 s)

Figure 4. The result of test casse execution

## VI. Evaluation

Fault injection is considered a very useful technique to evaluate the effectiveness of testing approaches. The key idea is to introduce faults during system execution and verify whether the testing approach precisely detects the injected fault [11], which depends on the fault model associated with a testing approach.

In order to estimate the effectiveness of the test cases development, we implemented a module in the tool that uses Java Annotations and aspect-oriented programming that intercepts the execution of the N-BDI4JADE agents and introduces faults in our normative agent.

### *A. The Fault Injector*

The fault injector component adds specific faults defined by the annotation. Each annotation describes one type of fault and aims to check how agents react to the fault. For instance: (i) *@ActivateNorm*: forces the activation of a norm in the environment. The attribute of this annotation is the norm that will be activated; (ii) *@DeactivateNorm*: forces the deactivation of a norm in the environment. The attribute of this annotation is the norm that will be deactivated; (iii) *@IncreaseReward*: forces the increase in the agent's reward score even when a norm is not fulfilled; (iv) *@IncreasePunishment*: forces the increase in the agent's punishment score even when a norm is not fulfilled; (v) *@ChangeAddressee*: forces a change in the addressee of a norm, and (vi) *@ChangeFulfillment*: forces the agent to fulfill, or not, a norm.

### *B. Results*

We have injected 22 faults inside our simulation to check whether the test scenarios were able to diagnose the injected faults. According to the results, the N-JAT4BDI tool helps the developer in the identification of these types of faults. We attribute these results to the use of the testing driven development technique during the development of our testing tool. Thus, the test cases became consistent and accurate whereas the injection of faults that involved reward and punishment, failed completely. Table 2 summarizes the results.

TABLE II. FAULTS INJECTED AND DETECTED BY THE TEST SCENARIOS

| **Fault Type** | **Faults Injected / Detected** |
|---|---|
| Activate Norms | 9 / 9 |
| Deactivate Norms | 1 / 1 |
| Reward Norms | 3 / 0 |
| Punishment Norms | 3 / 0 |
| Addressee Norms | 3 / 3 |
| Fulfillment Norms | 3 / 3 |

## VII. CONCLUSION AND FUTURE WORK

This work presented N-JAT4BDI, a testing tool for building and running automated test cases for normative agents with N-JAT4BDI to verify a Brazilian traffic simulation involving traffic intersections. To evaluate our approach, we used a fault injection technique to assess the quality of the test scenarios developed for this simulation. The results have shown that N-JAT4BDI can effectively uncover bugs in normative agents. As future work, we plan to improve the normative fault model and add features when testing other normative properties such as reward and punishment.

## REFERENCES

[1] A. Ferrando, D. Ancona and V. Mascardi, "Decentralizing mas monitoring with decamon", Proceedings of the Conference on Autonomous Agents and MultiAgent Systems, pp. 239–248, 2017.

[2] J. M. Voas and K. W. Miller, "Software testability: The new verification.", IEEE software, v.12, n.3, pp. 17–28, 1995.

[3] N. M. do Nascimento, C. J. M. Viana, A. von Staa and C. J. P. de Lucena, "A Publish-Subscribe based Architecture for Testing Multiagent Systems", 2017.

[4] M. Alberti, A. Gomes, R. Gonçalves, J. Leite, and M. Slota, "Normative systems represented as hybrid knowledge bases," Computational Logic in Multi-Agent Systems, pp. 330–346, 2011.

[5] R. Coelho, E. Cirilo, U. Kulesza, A. von Staa, A. Rashid and C. J. P. de Lucena, "Jat: A test automation framework for multi-agent systems", in IEEE International Conference on Software Maintenance, pp. 425–434, 2007.

[6] D. T. Ndumu, H. S. Nwana, L. C. Lee and J. C. Collis, "Visualising and debugging distributed multi-agent systems", in Proceedings of the third annual conference on Autonomous Agents. ACM, 1999, pp. 326–333.

[7] Y. Abushark, J. Thangarajah, T. Miller, J. Harland and M. Winikoff, "Early detection of design faults relative to requirement specifications in agent-based models", in Proceedings of the International Conference on Autonomous Agents and Multiagent Systems, pp. 1071–1079, 2015.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier and J. Irwin, (1997, June). "Aspect-oriented programming", In European conference on object-oriented programming (pp. 220-242). Springer, Berlin, Heidelberg.

[9] F. L. y López, "Social power and norms: Impact on agent behavior", Ph.D. dissertation, University of Southampton, June, 2003.

[10] V. T. da Silva, "From the specification to the implementation of norms: an automatic approach to generate rules from norms to govern the behavior of agents," Autonomous Agents and Multi-Agent Systems, vol. 17, no. 1, pp. 113–155, 2008.

[11] J. Voas and G. McGraw, "Software Fault Injection: Inoculating Programs Against Errors", Wiley, 1998.

[12] F. J. P. da Cunha, T. F. M Siqueira, M. L. Viana and C. J. P. de Lucena, "Extending BDI Multiagent Systems with Agent Norms", International Conference on Intelligent Agent Technology, 2018 – In Press.

[13] F. J. P. da Cunha, A. D. da Costa, M. L. Viana and C. J. P. de Lucena, "JAT4BDI: An Aspect-based Approach for Testing BDI Agents", Web Intelligence and Intelligent Agent Technology (WI-IAT), 2015 IEEE/WIC/ACM International Conference on. Vol. 2. IEEE, 2015.

[14] R. Coelho, U. Kulesza, A. von Staa and C. J. P. de Lucena, "Unit testing in multi-agent systems using mock agents and aspects", In Proceedings of the international workshop on Software engineering for large-scale multi-agent systems, 2006, pp. 83–90, ACM.

[15] J. J. Gomez-Sanz, J. Botía, E. Serrano, and J. Pavón, "Testing and debugging of mas interactions with ingenias," in International Workshop on Agent-Oriented Software Engineering. Springer, 2008, pp. 199–212.

[16] V. J. Koeman, K. V. Hindriks and C. M. Jonker, "Automating failure detection in cognitive agent programs", Proceedings of the International Conference on Autonomous Agents & Multiagent Systems, 2016, pp. 1237–1246.

[17] C. D. Nguyen, A. Perini, C. Bernon, J. Pavón and J. Thangarajah, "Testing in multi-agent systems," in International Workshop on Agent-Oriented Software Engineering. Springer, 2009, pp. 180–190.

[18] E. Serrano, A. Muñoz and J. Botia, "An approach to debug interactions in multi-agent system software tests", Information Sciences, vol. 205, pp. 38–57, 2012.