A Publish-Subscribe based Architecture for Testing Multiagent Systems

Nathalia Moraes do Nascimento[⋆] Carlos Juliano Moura Viana[†] Arndt von Staa[⋆]
Carlos José Pereira de Lucena[⋆]

[⋆]Software Engineering Lab (LES),
Pontifical Catholic University of Rio de Janeiro,
Rio de Janeiro, Brazil
nnascimento, arndt, lucena@inf.puc-rio.br

[†]Tecgraf Institute,
PUC-Rio,
Rio de Janeiro, Brazil
cviana@tecgraf.puc-rio.br

Abstract - *Multiagent systems (MASs) have been applied to several application domains, such as e-commerce, unmanned vehicles, and many others. In addition, a set of different techniques has been integrated into multiagent applications. However, few of these applications have been commercially deployed and few of these techniques have been fully exploited by industrial applications. One reason is the lack of procedures guaranteeing that multiagent systems would behave as desired. Most of the existing test approaches only test agents as single individuals and do not provide ways of inspecting the behavior of an agent as part of a group, and the behavior of the whole group of agents. Accordingly, we modeled and developed a publish-subscribe-based architecture to facilitate the implementation of systems to test MASs at the agent and group levels. To illustrate and evaluate the use of the proposed architecture, we developed an MAS-based application and performed functional and performance ad-hoc tests.*

Keywords – group test; agent test; multiagent system; test architecture; publish-subscribe; RabbitMQ

1. Introduction

Multiagent systems have been applied to a wide range of application types, including e-commerce, human-computer interfaces, network control, air traffic control and diagnosis [1] [2]. However, few of them have been commercially deployed [2]. According to Pěchouček and Mařík [2], one reason is the lack of procedures guaranteeing that the distributed systems would behave as desired. In addition, agent-based systems involve different characteristics, such as autonomy, asynchronous and social features, which makes these systems more difficult to understand. Thus, more elaborate methods of verification and testing [3] of multiagent operations should be provided [2].

According to Nguyen et al. (2009) [4], a full testing process of a multiagent system consists of five levels: unit, agent, integration (or group), system and acceptance. Agent test tests the capability of a specific agent to fulfill its goal and to sense and affect the environment. Integration test tests the interaction of agents and the interaction of agents with the environment, ensuring that a group of agents and environmental resources work correctly together [4].

As discussed by Serrano et al. (2012) [5], several approaches have been proposed to test multiagent systems at the unit and single agent levels [6] [7] [8] [9] [10], while there are few studies that address the issue of testing a MAS at group level [6] [11] [5] [12]. In addition, to perform group tests, most approaches have focused on capturing and visualizing messages exchanged among agents. They do not provide ways of also tracking the behaviors of two or more agents in the same view and finding a correlation between their behaviors. For example, Serrano et al. (2012) [5], which is one of the most recent papers published about testing MASs at the group level, uses ACLAnalyser [13], a tool for debugging MAS through the analysis of ACL [14] messages. Thus, by using these current test approaches, if an agent exhibits unexpected behavior (failure), a developer has to inspect this failed agent or messages exchanged between agents to find the fault that caused that failure. However, if an agent fails, its failure may be related to a previous and an unexpected behavior of another agent in the environment. It would be a real problem to some MAS-based approaches, such as that one proposed by Malkomes et al. (2017) [15], which promotes the development of cooperative agents without using message communication.

In the general context of distributed systems, Araújo and Staa (2014) [16] also faced the problem of testing a group of asynchronous components. They realized that most approaches to detect error and diagnose a failure in distributed systems rely on distributed log files over various machines, which makes the comprehension of the interaction among the machines more difficult. Thus, Araújo and Staa (2014) [16] proposed the use of a central architecture to receive, store and inspect timestamp-based logs from distributed machines, enabling a developer to further diagnose failures in a single machine and in the whole system during the software development cycle. Further, they presented a diagnosing mechanism based on logs of events annotated with contextual information, allowing a specialized visualization

DOI reference number: 10.18293/SEKE2017-050

tool to filter them according to the maintainer's needs. However, the authors discuss some limitations in their approach, such as the query response time that grows as the database size grows, which impacts the inspection interface usage, and the use of an inefficient solution for creating discarding rules.

In this paper, we present an architecture that was implemented[1] to make feasible the implementation of agent and group test activities within the MAS software development process. Our approach is based on the architecture to test distributed systems proposed by Araújo and Staa [16]. Nonetheless, MASs involve some characteristics that are not addressed by non-agent-based systems, such as autonomy and social behaviors. Thus, our goal is to adapt the architecture proposed by Araújo and Staa [16] to create one for testing MASs at different levels. Therefore, in order to represent MAS properties, we changed some tags that are used in their approach to represent logs with meta-information annotations. In addition, to solve the problems of data volume and discarding rules presented in their architecture, we decided to use a publish-subscribe [17] technology, instead of a database one. Through a publish-subscribe based approach, it is possible to develop decoupled and different tests that select logs that are useful for their purposes and ignore the irrelevant ones.

To illustrate and evaluate the use of the proposed architecture for creating systems to test MASs, we used and tested a simple MAS-based application. This experiment is presented in section 2. The remainder of this paper is organized as follows. Section 3 introduces the test architecture. Section 4 evaluates the test architecture, presenting the experimental results and evaluation. The paper ends with conclusive remarks in Section 5.

2. APPLICATION SCENARIO

In order to evaluate our proposed approach to test multiagent systems, we developed a simple multiagent application. This application is based on a scenario commonly used in the MAS literature[14] - a marketplace to buy and sell books on-line. We believe this experiment will assist one to understand our approach and facilitate further comparisons and analysis. We developed this application by using the JAVA Agent Development Framework (JADE) that is a Java software framework implemented to facilitate the development of multiagent systems [14].

2.1. Sellers and Clients

This application implements a simple marketplace where users create autonomous agents to sell and buy books for them, as described in the JADE Guide [14]. Therefore, this scenario contains two kinds of agents: Seller and Client. As part of the JADE platform, there is also a Directory Facilitator Agent (DF) that provides a Yellow Pages service by means of which an agent can find other agents providing the services he requires [14]. This illustrative scenario is depicted in Figure 1.

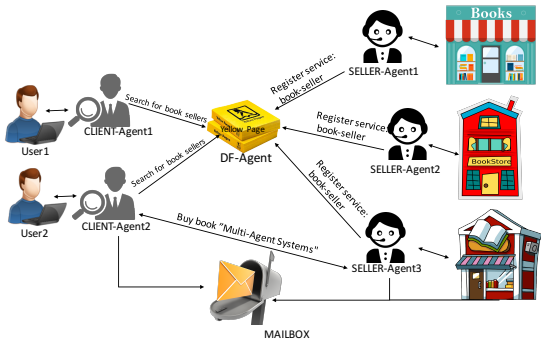

Figure 1: Scenario1: Overview of the general system architecture.

When a user creates a new selling agent, this agent registers itself in the Yellow Page by offering the service of book-seller. A selling agent manages a book catalog for a book store. Each user increments its own catalog at runtime by adding new books for sale. To add a book for sale, the user informs the name of the book and the price that he would like to receive for the book. A client agent is responsible for seeking and buying the book that a buyer user is looking for. Once created, the client agent is released into the marketplace, where it investigates which selling agents have the desired book and it buys the book from the seller that has the best price.

We also added a mailbox to the application. Our goal is to simulate interactions between agents that are different from ACL message communication. In such case, this interaction is performed by sharing a common resource among agents, that is, the mailbox. After selling the book, the seller agent sends a virtual copy of the book to the mailbox, while the client agent verifies if the book has been delivered. If the client agent buys a book and it does not find the book in the mailbox after a time, the client agent will fail.

3. TEST APPROACH: THE SOLUTION ARCHITECTURE

We developed a publish-subscribe based architecture as a foundation for generating different kinds of test applications for MASs. Our goal is to provide mechanisms that capture and process logs generated by agents automatically. As depicted in Figure 2, our architecture consists of three layers: MAS Application (L1), Publish-Subscribe Communication (L2), and Test Applications (L3). The Publish-Subscribe Communication layer uses the RabbitMQ platform [18] for delivering logs from agents (publishers) to be consumed

[1]The source of the test and the MAS application systems are available at http://www.inf.puc-rio.br/ nnascimento/MAS-tests.html

by test applications (subscribers). To understand more about the characteristics of RabbitMQ that we used in our approach, see `https://www.rabbitmq.com/tutorials/tutorial-five-java.html` (Accessed in 03/2017).

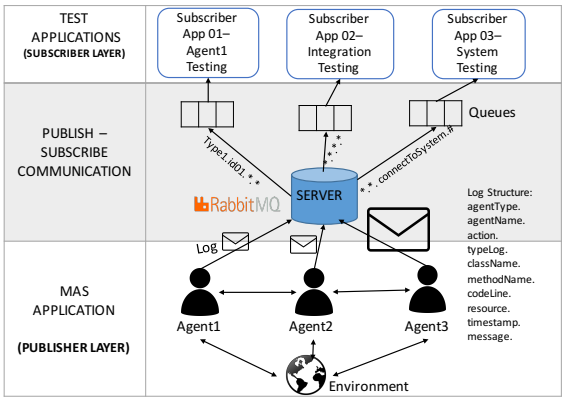

Figure 2: A Publish-Subscribe-based architecture to test MASs.

Each agent publishes logs with annotations that are composed of the following tags:

- *agentType*: the type of the agent (e.g CLIENT, SELLER, VEHICLE). In JADE, it refers to the name of the container where this agent lives;
- *agentName*: the name provided for the agent by the system developer/user (e.g client01, client02, seller01);
- *action*: the event that caused the log generation (e.g connectToSystem, searchBookInCatalogue, beDestroyed);
- *typeLog*: types of logs (e.g error, info, warning);
- *className*, *methodName*, *codeLine*: necessary information to identify the part of the code that generated the event;
- *resource*: the main resource that has been manipulated or requested by an agent during an event execution (e.g book1, book3, memory). It may be used to investigate all events that are related to a specific resource;
- *timestamp*: time that the log was created. Used to sort all events into a single timeline [16];
- *message*: a description of the event.

Thus, a log message must meet the pattern "(agentType).(agentName).(action).(typeLog).(className).(methodName).(codeLine).(resource).(timestamp).(message)." Each agent-based application must specify a set of values that can be used to fill these tag fields.

As depicted in Figure 3, all agents in the MAS application layer are also a TestableAgent type. A Testable agent uses RabbitMQ properties to send logs with annotations as messages. These logs can be published from any part of an agent's code. Some tags fields are automatically filled in by the TestableAgent class and JADE properties, such as agentType, agentName and timestamp.

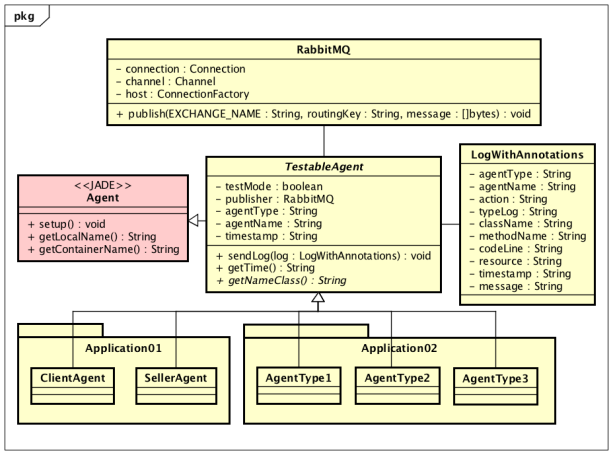

Figure 3: Simplified class diagram for creating testable MASs.

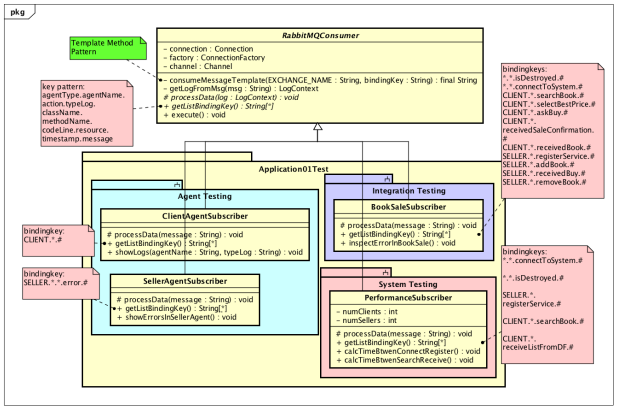

Figure 4: Simplified class diagram for creating applications for testing a MAS application at agent, integration and system levels.

The RabbitMQ autonomously delivers log messages to queues according to their tags' values. As shown in Figure 4, each test application defines a binding key in order to subscribe itself to consume messages from a specific queue. For example, a test application that monitors only error logs from SELLER agents must have the binding key "SELLER.*.*.error.#." Therefore, this application will

consume any log with the tuples (agentType,SELLER) and (typeLog,error). It is also possible to create applications that use multiple bindings. For example, if a performance test application needs to calculate the number of SELLER and CLIENT agents that are connected to the system, this application will have to consume logs with different action values. Thus, it needs to consume logs with the tuples (action,connectToSystem) and (action,beDestroyed).

Test applications do not interfere on the execution of each other. Each test class extends the class RabbitMQConsumer that starts an independent process to consume messages from a specific queue. We used the Template Method Pattern [19] to model the consumeMessage method. Thus, to consume and process particular log messages, a test class must overwrite and customize the methods getListBindingKey() and processData().

4. TESTS AND RESULTS

By using our proposed architecture, we created some test applications to execute functional tests at agent and group levels. Thus, this section presents part of the test plan that we created and performed for testing the application presented in the section 2. We provide the complete list of the functional and performance tests that we executed in [20].

4.1. Agent and Group Tests

We executed various test cases, taking eight parameters into account: (i) level (i.e. agent or group); (ii) function (e.g it is composed of a set of actions, for example, the function buy book may be composed of askBuy and soldBook actions); (iii) procedure (e.g a general description of the test); (iv) input (i.e a resource, a component); (v) expected value (e.g the result that will be produced when executing the test if the program satisfies its intended behavior); and (vi) validation method (e.g strategies that a tester performs to evaluate the system, comparing the program execution against expected results). Each test case execution produced several logs with meta-information annotations, which were consumed by test applications. Then, we used these logs as a validation method, as shown in table 1.

To validate a test case, the test application must verify if logs are appearing in the order described in the Validation Method column. Therefore, after the developer informs the logs from the validation column, the test application will automatically create a state machine, where each state represents an action. For example, Figure 5 illustrates the state machine that was created to validate the execution of the "buy Book" function. As shown, the verification program defines the transition between states as a log. A transition will only occur when the expected log appears. Each state has a maximum wait time for the expected log. Thus, if the maximum wait time exceeds, an error linked to the current state will be generated. This situation indicates that an agent performed an unexpected behavior and the action was not successful executed.

Table 1: Functional tests at agent and integration (group) levels (Simplified Table).

Level	Func.	Procedure	Input	Expected Value	Validation Method (Logs sorted into a timeline)
Agent	create Selling Agent	User creates a new agent	1.Type: SELLER 2. Name: seller1	seller1 agent is registered as book-seller	1)SELLER.seller1. connectToSystem. info.# 2)SELLER.seller1. registerService.info.#
	add Book	User adds a new book to seller1	1.Book's name: book1 2.Book's price:10	book1 is in seller1's catalogue	1)SELLER.seller1. addBook.info.*.*. .*.*.'name:book1 and price:10'
Group	buy Book	client1 asks seller1 to buy book1	1.client1 2.seller1 3.book1	client1 receives book1 after two seconds	1)CLIENT.client1. askBuy.info.# 2)SELLER.seller1. receivedBuy.# 3)SELLER.seller1. removedBook.# 4)SELLER.seller1. soldBook.# 5)CLIENT.client1. receivedSaleConf.# 6)CLIENT.client1. receivedBook.#

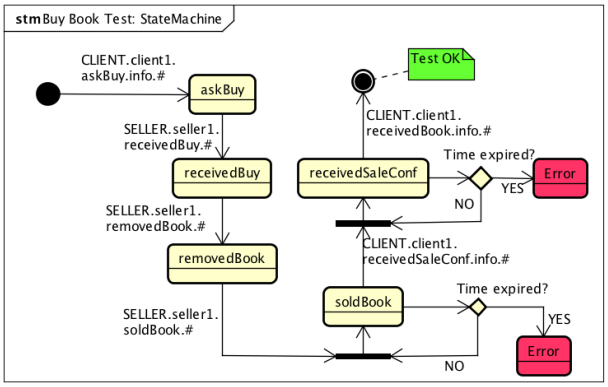

Figure 5: Simplified state machine for verifying test cases generated for the function buy book.

In order to force test failure and verify if these test applications were able to identify faults, we forced certain classes to act incorrectly during the execution of the program over some integration tests. For example, to test the function "buy Book", we inserted a defect that makes a Seller agent to die after accepting a book sale. Therefore, a client agent that bought a book from this seller, did not find this book in the mailbox and failed. As the test application did not receive the log "CLIENT.client1.receivedBook.info.#", its state machine indicated a failure in the state "receivedSaleConf." Figures 6 and 7 depict the logs that were generated by agents while this situation was executing.

```
[x] Sent 'SELLER.seller1.create.INFO.SellerAgent.initAgent.40.agent.2017.3.5.22.30.18.41.
[x] Sent 'SELLER.seller1.connectToSystem.INFO.SellerAgent.setup.56.agent.2017.3.5.22.30.18.47.
[x] Sent 'SELLER.seller1.createCatalogue.INFO.SellerAgent.setup.55.catalogue.2017.3.5.22.30.18.299.
[x] Sent 'SELLER.seller1.registerService.INFO.SellerAgent.registerService.71.yellowpage.2017.3.5.22.30.18.
[x] Sent 'SELLER.seller1.addBook.INFO.SellerAgent.addBookToCatalogue.109.book.2017.3.5.22.30.18.541.book1:
[x] Sent 'SELLER.seller1.receivedBudget.INFO.SellerAgent.receiveBudgetRequest.133.book.2017.3.5.22.31.58.7
[x] Sent 'SELLER.seller1.receivedBuy.INFO.SellerAgent.receiveMsgToBuy.171.book.2017.3.5.22.31.58.777.clien
[x] Sent 'SELLER.seller1.soldBook.INFO.SellerAgent.receiveMsgToBuy.171.book.2017.3.5.22.31.58.788.client:
[x] Sent 'SELLER.seller1.isDestroyed.WARNING.SellerAgent.takeDown.98.agent.2017.3.5.22.31.59.570.
[x] Sent 'SELLER.seller1.isDestroyed.ERROR.SellerAgent.takeDown.98.agent.2017.3.5.22.31.59.653.
```

Figure 6: Logs generated by Seller1.

```
[x] Sent 'CLIENT.client1.connectToSystem.INFO.ClientAgent.setup.41.agent.2017.3.5.22.30.58.358.
[x] Sent 'CLIENT.client1.create.INFO.ClientAgent.initAgent.54.agent.2017.3.5.22.30.58.358.
[x] Sent 'CLIENT.client1.searchBook.INFO.ClientAgent.setup.53.book.2017.3.5.22.31.58.661.book:
[x] Sent 'CLIENT.client1.receiveListFromDF.INFO.ClientAgent.setup.73.yellowpage.2017.3.5.22.31.
[x] Sent 'CLIENT.client1.askPrice.INFO.ClientAgent.buyBook.145.book.2017.3.5.22.31.58.703.ask b
[x] Sent 'CLIENT.client1.receivedBudget.INFO.ClientAgent.buyBook.163.agent.2017.3.5.22.31.58.75
[x] Sent 'CLIENT.client1.selectBestPrice.INFO.ClientAgent.buyBook.182.agent.2017.3.5.22.31.58.7
[x] Sent 'CLIENT.client1.askBuy.INFO.ClientAgent.buyBook.145.agent.2017.3.5.22.31.58.776.seller
[x] Sent 'CLIENT.client1.receivedSaleConfirmation.INFO.ClientAgent.buyBook.180.book.2017.3.5.22
[x] Sent 'CLIENT.client1.receivedBook.ERROR.ClientAgent.getBookInPostOffice.229.book.2017.3.5.2
```

Figure 7: Logs generated by Client1.

As each agent is running separately, to identify and understand what caused this failure, the tester would have to inspect the log file from each one of the agents. This work could be so difficult if the number of agents or the number of log messages was higher. Thus, by using our proposed solution, a test application can automatically select those logs from different agents that are probably to be essential for a specific test case and show them sorted in a single timeline.

In order to specify which characteristics need to be monitored from logs during the execution of a test cases set, the developer must establish a list of binding keys and override the method getListBindingKey() from the RabbitMQConsumer class (Figure 4). The list of binding keys will determine which test cases can be covered by the test application. For example, to cover the test cases that are listed in Table 1, it is necessary to establish binding keys that will make the test application able to receive all logs that are described in their "Validation Method" columns.

The code below shows the method getListBindingKey() that was used by a test application to cover these three test cases. If the developer wants this test application covering more test cases, he needs to add more binding keys to allow the test application to consume different logs.

```
@Override
public String [] getListBindingKey () {
BindingKey bd = new BindingKey();
String [] listKey = new String [11];
listKey [1] = bd.createBindingKey(LogValue.Action.connectToSystem);
listKey [2] = bd.createBindingKey(LogValue.AgentType.CLIENT, LogValue.Action.askBuy);
listKey [3] = bd.createBindingKey(LogValue.AgentType.CLIENT, LogValue.Action.receivedSaleConfirmation);
listKey [4] = bd.createBindingKey(LogValue.AgentType.CLIENT, LogValue.Action.receivedBook);
listKey [5] = bd.createBindingKey(LogValue.AgentType.SELLER, LogValue.Action.registerService);
listKey [6] = bd.createBindingKey(LogValue.AgentType.SELLER, LogValue.Action.addBook);
listKey [7] = bd.createBindingKey(LogValue.AgentType.SELLER, LogValue.Action.receivedBuy);
listKey [8] = bd.createBindingKey(LogValue.AgentType.SELLER, LogValue.Action.removedBook);
return listKey ;
}
```

As a result, the interface depicted in Figure 8 shows all logs that were consumed by a test application according to this binding key list. In addition, all logs are organized in a single timeline. As shown, not all logs depicted in Figures 6 and 7 were presented in this interface, but only the relevant logs to inspect the execution of these test cases. Thus, we were able to verify these logs in order to find the fault that generated the failure indicated by the state machine.

INTEGRATION LOGS

Agent Type	Agent Name	Class	Log Type	Action	Resource	Timestamp(ms)	Message
SELLER	seller0	SellerAgent	INFO	connectToSystem	agent	3766039.0	
SELLER	seller1	SellerAgent	INFO	connectToSystem	agent	3766256.0	
CLIENT	client1	ClientAgent	INFO	connectToSystem	agent	3766471.0	
SELLER	seller0	SellerAgent	INFO	registerService	yellowpage	3766605.0	
SELLER	seller1	SellerAgent	INFO	registerService	yellowpage	3766668.0	
SELLER	seller0	SellerAgent	INFO	addBook	book	3766669.0	book1: 200
SELLER	seller1	SellerAgent	INFO	addBook	book	3766708.0	book1: 100
CLIENT	client1	ClientAgent	INFO	searchBook	book	3826505.0	book: book1
CLIENT	client1	ClientAgent	INFO	selectBestPrice	agent	3826600.0	seller: seller1 price: 100
SELLER	seller1	SellerAgent	INFO	receivedBuy	book	3826617.0	client: client1 Title: book1 Price: 100
CLIENT	client1	ClientAgent	INFO	askBuy	agent	3826617.0	seller: seller1 price: 100
CLIENT	client1	ClientAgent	INFO	receivedSaleConfirmation	book	3826670.0	seller: seller1 price: 100
SELLER	seller1	SellerAgent	WARNING	isDestroyed	agent	3828085.0	
CLIENT	client1	ClientAgent	ERROR	receivedBook	book	3830698.0	book: book1

Figure 8: Application View - Agent and Integration tests. Fault detection in test case execution.

Test Results

As shown in Table 1, we executed some functional tests at agent and group levels. By using state machines, the test applications were able to validate these test cases by comparing the logs consumed from the MAS publisher against the logs listed in the "Validation Method" column. In addition, we also conducted some tests by inserting software failures and verifying if our test software could be useful for detecting faults. As a result, after the state machine had indicated a failure, the developer could use the interface to identify the fault and reduce the time of diagnosis.

5. Conclusions and Open Challenges

We believe these results are promising. We presented a decoupled architecture that allows a developer to execute tests simultaneously and independently while running a MAS. In addition, we provided evidence of the usability of our proposal, using it to test a known MAS application. We showed that it is possible to develop different tests for a multiagent system at different levels by using logs containing meta-information annotations and a publish-subscribe technology.

5.1. Testing Non-Deterministic Applications

However, MASs usually are much more complex than the experiment that was used in this work. Current approaches modeled by using a MAS may involve non-deterministic characteristics that were not addressed by this paper, such as *learning* [21], *self-adaptation* and *self-organization* (SASO) [22]. In fact, there is a gap in the literature regarding the test of systems with these features. There are few approaches to inspect the emergence process in a self-organizing MAS system [23] [24], and all of them do MAS design based only on simulation techniques. One reason is the difficulty of specifying expected results for non-deterministic applications, especially in actual environments. Nonetheless, we believe our approach opens the way for more experiments in testing Multiagent Systems, since it provides ways for testing a MAS at different levels. For example, as a self-organizing MAS system enables the emergence of social features based on the behavior of individual

agents, to test this kind of system it is necessary to perform tests at single and group levels.

5.2. Deploying Agents in a Distributed Environment

For instance, all agents are running on a single machine and the proposed approach assumes that there is a central clock so that the timestamp in each message can be used to sort events and measure throughput. If the tester needs to evaluate the MAS application in a distributed environment, he can use RabbitMQ to create a common publisher that publishes logs from agents localized on different machines. However, new challenges will arise. In the general case of distributed agents, possibly running on machines in different continents, a synchronization mechanism is required.

5.3. Predictive Analysis

For each application, there is a set of predetermined values that can be used in tags. Thus, we can codify and normalize these values to use them as inputs of a temporal neural network, which is a known structure of predictive analysis. By consuming temporal logs, a test application may use a temporal neural network to process log information in order to predict errors.

Acknowledgements This work has been supported by the Laboratory of Software Engineering (LES) at PUC-Rio. Our thanks to CNPq, CAPES, FAPERJ and PUC-Rio for their support through scholarships and fellowships.

References

[1] C. Lucena, *Software engineering for multi-agent systems II: research issues and practical applications*. Springer Science & Business Media, 2004, vol. 2.

[2] M. Pěchouček and V. Mařík, "Industrial deployment of multi-agent technologies: review and selected case studies," *Autonomous Agents and Multi-Agent Systems*, vol. 17, no. 3, pp. 397–431, 2008.

[3] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[4] 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.

[5] 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.

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

[8] 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 2015 International Conference on Autonomous Agents and Multiagent Systems*, 2015, pp. 1071–1079.

[9] F. Cunha, A. D. da Costa, M. Viana, and C. J. P. de Lucena, "Jat4bdi: An aspect-based approach for testing bdi agents," in *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2015 IEEE/WIC/ACM International Conference on*, vol. 2. IEEE, 2015, pp. 186–189.

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

[11] 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.

[12] A. Ferrando, D. Ancona, and V. Mascardi, "Decentralizing mas monitoring with decamon," *Autonomous Agents and Multi-Agent Systems*, vol. 16, 2017.

[13] J. Botía, A. Lopez-Acosta, and A. Skarmeta, "Aclanalyser: A tool for debugging multi-agent systems," 2004.

[14] F. Bellifemine, G. Caire, T. Trucco, G. Rimassa, and R. Mungenast, "Jade administrator's guide," *TILab (February 2006)*, 2003.

[15] G. Malkomes, K. Lu, B. Hoffman, R. Garnett, B. Moseley, and R. Mann, "Cooperative set function optimization without communication or coordination," in *Proceedings of the 16th International Conference on Autonomous Agents and Multiagent Systems*, 2017.

[16] T. P. de Araújo and A. von Staa, "Supporting failure diagnosis with logs containing meta-information annotations," *Technical Reports in Computer Science. PUC-Rio. ISSN 0103-9741*, vol. 14, p. 21, 2014.

[17] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi, "Meghdoot: content-based publish/subscribe over p2p networks," in *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Springer-Verlag New York, Inc., 2004, pp. 254–273.

[18] RabbitMQ, "Rabbitmq," Available in https://www.rabbitmq.com/, 10 2016.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *European Conference on Object-Oriented Programming*. Springer, 1993, pp. 406–431.

[20] N. M. do Nascimento, C. J. M. Viana, A. v. Staa, and C. J. P. de Lucena, "A publish-subscribe based approach for testing multi-agent systems," *Technical Reports in Computer Science. PUC-Rio. ISSN 0103-9741*, vol. 2016, p. 21, 2016.

[21] J.-P. Briot, N. M. de Nascimento, and C. J. P. de Lucena, "A multi-agent architecture for quantified fruits: Design and experience," in *28th International Conference on Software Engineering & Knowledge Engineering (SEKE'2016)*. SEKE/Knowledge Systems Institute, PA, USA, 2016.

[22] N. M. do Nascimento and C. J. P. de Lucena, "Fiot: An agent-based framework for self-adaptive and self-organizing applications based on the internet of things," *Information Sciences*, vol. 378, pp. 161–176, 2017.

[23] L. Gardelli, M. Viroli, and A. Omicini, "On the role of simulation in the engineering of self-organising systems: Detecting abnormal behaviour in mas," 2005.

[24] C. Bernon, M.-P. Gleizes, and G. Picard, "Enhancing self-organising emergent systems design with simulation," in *International Workshop on Engineering Societies in the Agents World*. Springer, 2006, pp. 284–299.