A Test Driven Development of MAS

Ing. Esp. Juan Pablo Paz Grau[1], Dr. Andres Castillo Sanz[2]

[1]LG CNS Colombia, Bogotá, Colombia
juan.paz@lgcns.com
[2]Departamento de Lenguages, Sistemas Informaticos e Ingenieria de Software, Universidad Pontificia de Salamanca, Campus Madrid, Madrid, España
andres.castillo@upsam.es

Abstract— Testing is a key software lifecycle activity to assure software quality. Although it is of recognized importance, the work on agent testing has been scarce and has been usually tied to a MAS design methodology. This article presents a methodology agnostic testing procedure along with the description of a testing toolkit for MAS, with a practical application to a real, in production MAS. We also present some insights into our practical experience testing MAS.

Keywords: Multi Agent Systems, Agent Oriented Software Engineering, Software testing, Test Driven Development.

1 Introduction

Software testing is the most important tasks in software engineering to guarantee quality during the software development cycle. Having the ability to introduce software testing into the implementation phase of the software lifecycle is a determinant Critical Success Factor for cutting down costs, time and to guarantee quality [1]. The link from requirements analysis to testing is also important, as early tests specifications produce better criterion and better criterion produces better tests. Despite of these facts, MAS methodologies usually address the testing approach only partially [2] [3], paying attention solely to the modeling and implementation phases of the MAS software lifecycle.

Moreover, we have found out that usually, when a MAS methodology introduces a testing framework into its software engineering development cycle, the testing methodology is generally closely linked to the specific MAS design methodology and development paradigm, usually requiring the construction of complex XML documents to configure testing activities [2][4]. This constitutes a big gap to be addressed to make MAS applicable in real world, large scale applications. As there is currently no structured testing process for guiding testing activities for MAS [5], this article intends to bridge this gap, presenting a set of tools and an agnostic MAS testing process, without aiming at a particular MAS methodology.

To illustrate the testing process, the article will start with a dissertation on the proposed test driven development methodology; then, a reference MAS will be

introduced. After that, the design and components of the testing toolkit for MAS that was developed will be shown; next, remarks about the testing procedure that was conducted will be revealed and discussed. Finally, the conclusions will be presented.

2 Test Driven MAS Development

When approaching MAS testing, there are some issues that arise immediately. The first issue is the fact that agents communicate with its surrounding entities via message passing, instead of doing it by method invocation [3][5][6]. This makes Object Oriented testing not directly applicable when tackling the problem of MAS testing. To overcome this limitation, we constraint our agents to be polite, that is, they always respond to messages they receive; this way, we can base our testing tools on analyzing the ACL messages we receive from agents when they are excited by a request. This is not an arbitrary approach; other software entities communicate also by message passing; web services and other artifacts work this way and there are specialized frameworks targeted for these entities extending the xUnit framework.

Another issue is that although agents cooperate with other agents, it is possible for them to run correctly when isolated, but incorrectly when inserted into a community. This means that to correctly test some agents, it is necessary to create a test scenario where a set of mock agents mimic the agent's community, resulting in complex test cases. By using as a development constraint that agents should be polite, we ensure simpler test cases and detect failures more easily.

Finally, some specific kinds of agents pose challenges to the testing activity [5] [6]. Smart agents may learn as they advance during their execution, producing different outputs between executions; on the other hand, scaled agents –agents that work isolated from their environment- provide little or no observable primitives to the outside world which can be analyzed in a test case. These special cases will not be covered in this paper.

2.1 An approach to MAS testing

There are currently two approaches for MAS testing [6]; structured or active (based on test cases) and simulation based or passive (based on running the MAS and analyzing system output). We analyzed both approaches and found no reason for them to be mutually exclusive; moreover, a MAS testing procedure should include both. With active testing, we use test fixtures [7] to set up the agent's environment to an initial state, and then evaluate the agent's response to this state. The set of test cases can be derived from the analysis and design artifacts, allowing the behavior of the MAS to be dynamically evaluated. Concurrently, passive testing is performed to watch agent interactions during test case execution. This implies that a MAS platform should be running while monitoring tools are used to trace the test case execution. This allows the detection of abnormal behaviors or interactions while the test case is running.

Our reference model for MAS testing departs from the V-model [8]. It gives a simple and structured view of the testing process, presenting an abstraction of the activities in MAS testing which can be later assembled depending on the specifics of each development effort. Our V-model for MAS testing is depicted in figure 1. We begin our proposed approach with the agent's internal components, specifically, the components that can be tested outside of the targeted agent platform; i.e. agent behaviors can't be tested during this phase, as they have to be triggered according to the agent's plans. At the next test phase, the agent testing takes place. At this stage, we test the agent in isolation from the other agents in the platform under construction; specifically, we test to see if the agent sends the correct responses (i.e. the correct behaviors are activated and the intended protocols are enacted) when receiving a message. At the next stage, sets of agents are tested to ensure they work as expected. Finally, acceptance test is performed to check if the Multi Agent System works at the target environment and the artifacts that interact with the MAS perform their interactions as expected.

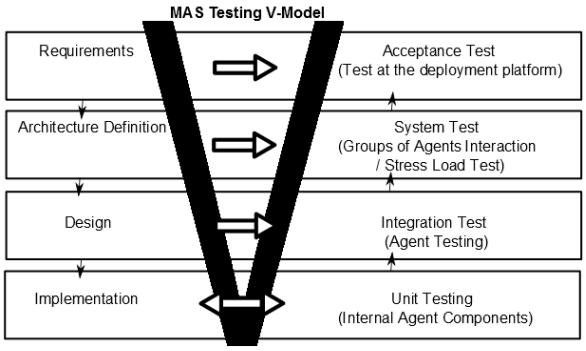

Fig. 1. MAS Testing V-Model

Internal agent components testing.

An agent is a complex set of objects integrated by complex sets of relationships. All these objects must be unit tested before assembly the agent. Testing that the compounding objects meet their contract allows the implementation process to continue into the agent's core functionalities. Internal agent components testing are the finest grain testing and the only one that can be considered white box testing.

Agent testing.

This is the most basic test that needs to be performed in-container on a test agent platform. The agent is the smallest building block of MAS [5], so it should be internally coherent and be tested as a whole before coupling it with the rest of the system. In the same way as an object has a contract to fulfill, an agent also has an ***Agent Contract***: *The agent has to correctly handle all messages passed and return the messages expected*. At this testing stage, we test if the agent is capable of sensing its environment and fulfilling its goals.

To perform agent testing, we rely on stub agents to handle requests and responses to the agent. As the agent is a complex construction, a mock agent usually doesn't suffices to deal with this task; by means of the stub agent we establish a conversation with the Agent under Test (AUT) performing a set of "canned" (previously defined) request/responses and analyzing the responses received from the AUT [3]. If a complex conversation needs to be enacted, the stub agent can create a set of Mock Agents to interact with the AUT. This allows us to test a single agent role per test case. An important objective that arises when performing Agent TDD (Test Driven Development) is the need to question the agent after a complex interaction to get an insight into its current state to determine whether it is correct, as the agent's task executions are black boxes.

System testing.

The aim of system testing is to assemble a set of agents and start testing their interactions [5]. At this stage, the real agents are deployed in the agent platform to guarantee they follow the agreed interactions and semantics and the regulation enforcements of the system. The behaviors of the system are evaluated not from a single agent but from a collective behavior point of view. It is also the moment to test the quality properties and perform stress/load testing on the MAS that will act as interaction points with other components of the enterprise architecture.

This stage also is where agent integration testing takes place. The test cases at this point require much time devoted to passive testing, allowing the agents to interact with each other and monitoring messages exchanges; when a sufficient number of interactions have taken place between agents, then we switch to active testing and ask the agent about its state to demonstrate that the observed behavior matches the internal state of the agent. This allows testing agent dependencies and environmental mediated interactions. Some challenges of this testing stage are the integration of scaled agents, as they rarely or never interact with other agents, mobile agents and testing agents that have not yet been implemented. These yet to be implemented agents can be replaced with mock or stub agents, but the design of such agents may become a very complex task.

Acceptance test.

When the development of a MAS is finished, it should be deployed on a platform as part of an enterprise architecture and interact with other artifacts already deployed there, or with artifacts developed specifically to interact with the MAS. Apart from the non-functional requirements, the functional requirements of the system must be tested. Functional requirements should be tested against the MAS' use cases [9].

The input to the MAS acceptance tests are the use cases; the domain model allows the creation of test objects to exercise the use cases and coverage metrics should be taken into account to ensure all use cases flows are exercised and therefore, all test cases

have been identified and exercised to ensure the system meets the functional requirements. Special care should be taken to include some negative, misuse cases [9] to test the system against incorrect usage or attacks. The negative use cases will allow the MAS to demonstrate its resilience and failure control mechanism features, and if it is part of the requirements, the enforcement of the MAS´ policies.

3 Reference Multi Agent System

In the reference application, the MAS is a software component; the design allows many clients to consume data generated and handled by the MAS. The application monitors events and alarms triggered during the daily operation of the automatic fare collection (AFC) system deployed in Bogotá's Public Transportation Integrated System (SITP). The backend of the system is a MAS that queries the system middleware and generates event listings. This structure has two benefits: the event inquiry processing is encapsulated into the MAS, relieving the front application from connecting to the middleware, and on the other hand, the application monitoring panel can be easily exchangeable. The MAS' Use Case is shown in figure 2. The use cases follow:

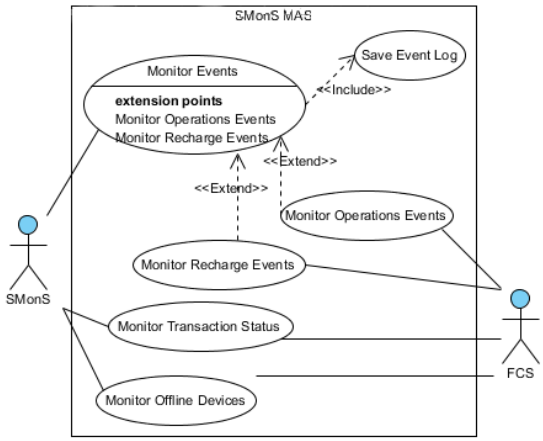

Fig. 2. Monitoring MAS Use Case

3.1 Use Cases

- **Monitor offline devices:** The system periodically requests the middleware to provide a list of devices that are late to communicate with the center on a given time threshold. The list is stored and sent to the monitoring application on demand.
- **Monitor transaction status:** The system periodically asks the middleware to get the transaction load of each transaction type. The last transaction values are stored on a hash map and sent to the monitoring application on demand.
- **Monitor events:** A basic use case that collects events from the system middleware and stores the event details on a list. The list items are sent to the monitoring application on demand, according to the filters specified.

- **Save event log:** It is triggered by the "Monitor Events" use case to store the event list after operations closing. After the event list is stored, it is cleared.
- **Monitor recharge events:** an extension from the Monitor Events use case. It solicits its card recharge and sale devices events from the middleware and delivers them to the "Monitor Events" use case to manage the events.
- **Monitor operations events:** an extension from the Monitor Events use case. It solicits specific operational events (e.g. alarms, failures, etc.) from the middleware and delivers them to the "Monitor Events" use case to manage the events.

This division of monitoring functions has the goal of allowing us to broaden the system further to monitor other events in the future; it also allows us to configure the system to select which type of events we would like to monitor on a given front panel. The agents that compose the Multi Agent System are depicted on figure 3. The Proxy Agent depicted in the diagram is a stub agent that takes the role of the principal actor depicted in figure 2. It is a foreign agent, but as it interacts with the MAS hosted agents, it is included in the diagram.

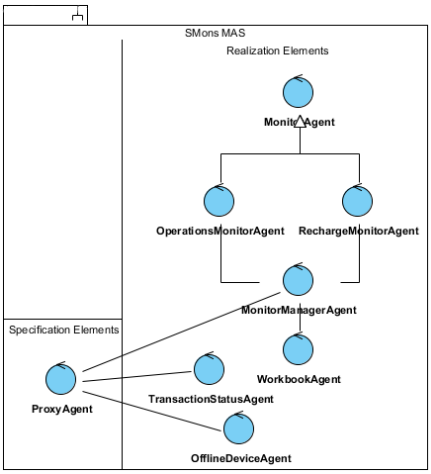

Fig. 3. Monitoring MAS Agent Organization

3.2 Agent Organization / Agent Tasks

- **Monitor Agent:** Is an abstract agent from which monitoring agents are extended.
- **Operations Monitor Agent:** Requests operational events from the middleware. When an event triggers, sends notifications to agents that have subscribed to it.
- **Recharge Monitor Agent:** Requests recharge and sale specific events from the middleware. When an event triggers, sends notifications to agents that have subscribed to it.
- **Monitor Manager Agent:** Subscribes to monitor agents of interest. Organizes a list of events and sends the list to agents that request it. Also, sends requests to the Workbook Agent to save the event list.

- **Workbook Agent:** Receives requests and processes them storing a workbook.
- **Offline Device Agent:** Asks the middleware for devices that haven't communicate in a given time threshold. Sends the device list when requested.
- **Transaction Status Agent:** asks the middleware for current transaction load for different transaction types and stores the transaction status. It sends the transaction type status when requested.

3.3 Agent Interactions

The Transaction Status and the Offline Devices Agent enact request / inform interactions; these agents make requests to the middleware at fixed intervals to avoid stressing the middleware, and store a snapshot of the system. When a request arrives, they send the last snapshot stored to the requesting agent. This design allows us to accelerate the MAS in the presence of many Proxy Agents; in contrast, the Workbook Agent performs a simple request/inform interaction.

The Monitor Agents play a subscription protocol interaction along with the Monitor Agents, which act as subscribers. They implement both subscribe and unsubscribe protocols and start sending notifications to the subscribed monitor agent(s) as soon as there are events to report. Finally, the Monitor Manager Agent performs both subscription and request/inform interactions. This agent subscribes to Monitor Agents of interest, and allows Proxy Agents to request subscribe / unsubscribe from Monitor Agents. It sends event listings to Proxy Agents following the request/inform protocol.

3.4 MAS Ontology

Although the MAS ontology contains many ontological classes among concepts, predicates and agent actions, we will concentrate on the set of ontological classes that will be unit tested. We present in figure 4 the basic set of ontological classes used within the Multi Agent System. In particular, we will concentrate on the Event concept, which implements a set of abstract methods that ease the communication with the application front panel and facilitate the extension of the application to manage new events.

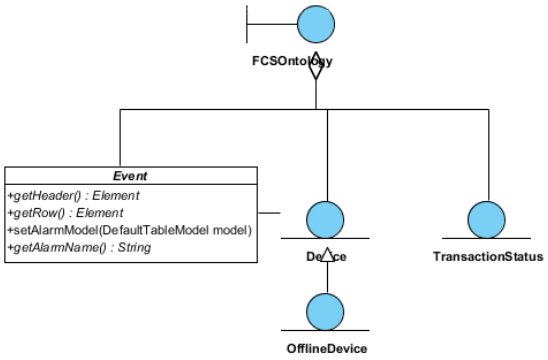

Fig. 4. Basic concepts of the ontology

As each new event has to extend this set of methods we want to guarantee each implementation with unit testing; these tests fall under the category of Internal Agent Components testing. The getAlarmName method returns a printable name for the event. getRow and getHeader methods return JDOM [11] Element objects that represent the event as a row with its row header, respectively. The setAlarmModel method returns a java swing TableModel object for the front panel to display the event's details. The Device ontological class models a device, a concept that is both used at the Event class and the Monitor Offline Devices use case. For this last purpose, we extend the Device concept into the Offline Device concept including its last communication date. The Transaction Status concept models a snapshot of the status of a transaction type.

3.5 Agent Model

As we have stated when we presented our testing approach, an agent can be treated as a complex set of objects and their interactions; we have taken advantage of the component diagram from UML2 to model agents, to depict all the internal artifacts that make up the MAS. In figure 5, we depict three of the agents that make up our system, the Monitor Manager Agent, the Recharge Monitor Agent and the Workbook Agent.

- **Workbook Agent:** The agent relies on the WorkbookHandler class to parse an XML workbook representation and transform it to an Excel workbook which is stored as a file. The WorkbookHandler class is a candidate for internal agent components testing.
- **Recharge Monitor Agent:** This agent manages its subscribed agent list through the SubscriberManager class. It also relies on the FCSHelper class to communicate with the middleware. Both classes are candidates for intenal agent components testing.
- **Monitor Manager Agent:** This agent manages the list of alarms received from the monitor agents by means of the AlarmManager class. It also manages the list of monitors to whom it is subscribed with the SubscriberManager class. Both classes are candidates for internal agent components testing. The classes that implement the agent's behavior are worth testing at a higher testing stage.

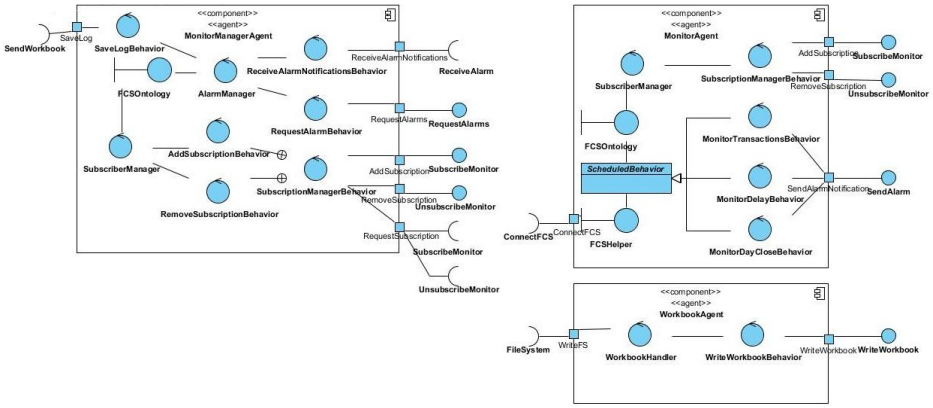

Fig. 5. Artifacts of Recharge Monitor, Monitor Manager and Workbook Agent

4 The MAS Testing Toolkit

To perform the TDD strategy, we developed a toolkit to provide us with a set of tools to ease the execution of agent test cases. These tools extend from the JUnit framework, aiming at facilitating the coding of test cases and to focus on the agent aspects that are under test. This way we can focus on the test case execution flow while coding the tests without worrying about managing platform specific coding features, which of itself would require test cases for each test case.

4.1 ACL Message Matchers

The first set of tools is a collection of matchers that allows the analysis of messages received from the agent under test (AUT) to match against test fixtures previously defined in the test case. The matchers implement a library of common ACL message patterns found in agent communication, and take advantage of the JUnit matchers' property of being able to combine custom matchers with the basic matchers that come with the JUnit framework to structure complex assertions. This behavior let us rely on the default matchers to test the most basic ACL message features (e.g. to assert a given message performative, reply with code, sender, ontology or language name, etc).

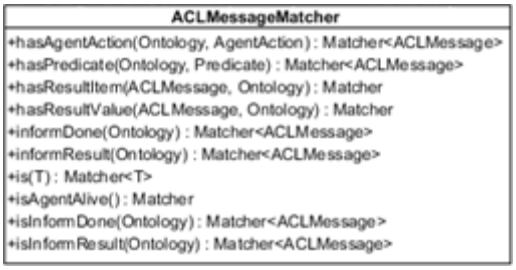

Fig. 6. The ACL matchers set

Figure 6 depicts the matcher library. Among others, we developed matchers for a given agent action or predicate from an ontology, matchers that look for the presence of a value or an item in a set of values from a result ACL message. These matchers need to be aware of the message's ontology in order to decode correctly the message content and perform the assertion. Also, there are matchers to assert the basic ACL Messages INFORM performative with Done predicate, or an ACL message with an INFORM performative and Result predicate. We also developed a matcher that queries the AMS if an agent is present or not in the agent platform.

4.2 The Mock Agent

To implement the mock agent infrastructure, we developed three components: The mock agent itself, a mock ontology and a mock agent gateway to create, destroy and configure mock agents. The mock ontology is depicted in figure 7 [13]. It consists of two agent actions that allow the mock agent gateway to configure the mock agents with their test expectations (i.e. messages that mock agents are going to exchange with the AUT).

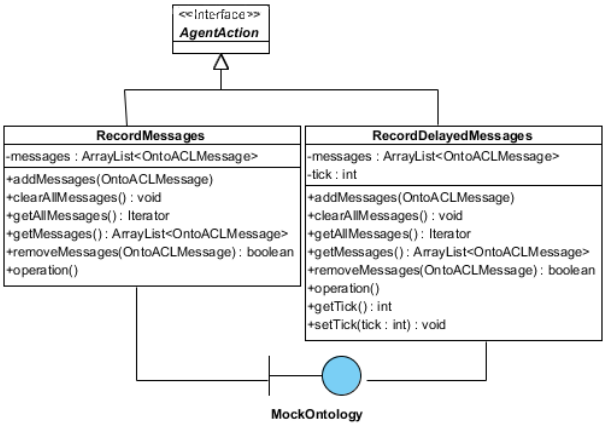

Fig. 7. The Mock Ontology

The "Record Messages" agent action sends the mock agent a set of predefined ACL messages, while the "Record Delayed Messages" agent action adds a tick to set the mock agent´s timer. The mock agent can be configured to operate in two different modes. In the first mode, it listens to ACL Messages and when a message is received, it replies the sender with the next message in its circular message queue. This mode is configured sending the mock agent a "Record Messages" agent action. This allows replaying the mocking actions without the need to reconfigure the Mock Agent.

On the latter mode, it sends the ACL messages on its message queue, on time intervals according to the configuration received. When the last message has been sent, the agent behavior stops. This mode is configured sending the mock agent a "Record Delayed Messages" agent action. The mock agent gateway is depicted in figure 8.

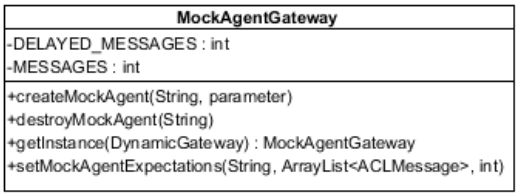

Fig. 8. The Mock Agent Gateway

It is implemented as a singleton, in order to have only one instance for all mock agents in a given test case. The last parameter of the createMockAgent method can be either the MESSAGES constant to configure the agent on the basic mode, or DELAYED_MESSAGES to configure the agent in the ticking message mode.

5 Remarks on TDD of MAS

In the previous sections, we have presented our approach to MAS testing and the tools we developed to follow our testing procedure. We have also summarized as much as possible the MAS developed following this test driven approach. Now in this last section, we will present some practical issues and insights into implementing a test driven development process for MAS.

The first issue is the test case design [3] [6]. As we go up the MAS testing V-model, the test cases become more complex as more actors get involved in each test case. Our starting point was to check all the tasks and roles played by each agent based on the MAS' use cases and interaction model. To perform system testing, it is indispensable to identify all the roles the agent can impersonate in the interaction model. For each agent role, we then identified the agents with which our AUT (Agent under Test) interacted. Depending on the kind of interaction, we decided if those agents were going to be impersonated by mock agents, test stub agent or in the worst case, the real agent itself. Then, we created the positive and negative test cases for the AUT. Negative cases are very important in MAS testing, as we cannot expect to always receive a "positive" answer from the interacting agent. The positive test cases' goal is to discover whether the AUT fulfills its agent contract and conforms to the agent's design [12]. Negative test cases, on the other hand, have two goals; one is to test the agent's error management mechanisms and the other is to reveal faults in the agent's implementation through failures [12].

Some basic negative cases we implemented for each agent were sending ACL messages with FAILURE performative after a request to test the AUT failure handling mechanism, and to send the AUT an ACL message with an unexpected performative or content, in order to assert that the AUT answers the request with a NOT UNDERSTOOD performative. Other interactions needed more complex negative cases. For example, how the AUT handles subscribing / unsubscribing to an already subscribed / unsubscribed agent in a subscription protocol.

The test cases for the MAS agents are shown in table 1. The transaction status and offline devices monitor agents have simple test cases: one positive test case (testRequest) and two negative test cases (testInvalidRequest and testInvalidPerformativeRequest). These agents are tested through the stub agent issuing ACL messages for each of the test cases. These test cases are performed at the agent test level. The workbook agent also has the same positive and negative test cases, but it also includes an internal agent components test level test case to test the workbook handler component. This test case included complex assertions to check that the log file was created, the file name and size, and then during the test case the workbook file was opened to check if it had the correct number of sheets and random cells were looked up in the file to check that they existed.

The monitor manager agent and the monitor agent were tested at the system testing level, as they needed the existence of a set of agents to operate. In each case, the other participant in the conversation was impersonated by a mock agent. The first test cases tested the subscribe / unsubscribe protocol. Tests included adding subscriptions and then adding again the same subscription as a negative case. On the first test case, it was affirmed that the subscriber/participant be included on the subscriber/participant list of the AUT; in the latter test case it was affirmed that the correct messages were exchanged between the agents and the AUT managed the error condition. The latter test cases are composite as the subscription protocol should be enacted first in order for the AUT to send/receive events notifications.

Table 1. Test cases

Agent	Test Stage	Test Name
Transaction Status Monitor Agent	Agent Testing	testRequest testInvalidRequest testInvalidPerformativeRequest
Offline Device Monitor Agent	Agent Testing	testRequest testInvalidRequest testInvalidPerformativeRequest
Workbook Agent	Internal Agent Components Agent Testing	testWorkbookHandler testRequest testInvalidRequest testInvalidPerformativeRequest
Monitor Manager Agent / Monitor Agent	System testing	testAddSubscription testAddInvalidSubscription testRemoveSubscription testRemoveInvalidSubscription testExportAlarms testReceiveAlarm

An interesting fact of this testing level was that it was conducted concurrently with passive and active testing mechanisms. This is, while the test cases were running, we were monitoring the message exchange trace between the agents and analyzed both sets of results when the test case concluded. A transcription of the messages exchanged during the subscription protocol testing (from the monitor manager agent point of view) and captured by a sniffer agent is shown on figure 9.

In the figure, the ACL messages traces correspond to the following operations: (1-4) depict the creation of two mock agents that will assist in testing the subscription protocol; (5-6) are the messages with the Mock ontology setting the mock agents expectations. These operations set the test fixture needed for the test. Messages (7-10) depict a subscription request to the first mock agent while messages (11-14) depict a subscription request to the second mock agent. The test case finalizes asking the Monitor Manager Agent for the list of subscribed agents (15-16) to proceed to perform the test assertions. Messages (17-24) are a replay of the test fixture (negative test case). Messages (25-26) again request the list of subscribed agents from the Monitor Manager Agent to proceed to perform the test assertions. At this point, we check also that the Monitor Manager Agent has enacted the protocol for failure, has reacted to the error condition and hasn't died. Finally, messages (27-30) destroy the mock agents after concluding the test cases.

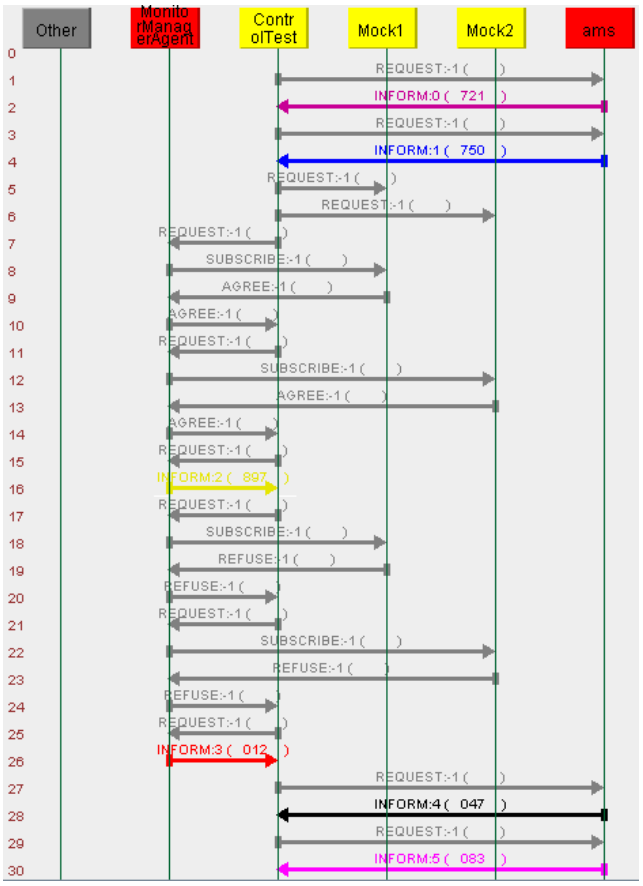

Fig. 9. ACL messages exchanged during the subscription protocol test cases (Monitor Manager Agent).

From this exercise, it is clear how the monitoring tools from the agent platform can be effectively used during the test cases execution to follow the test case execution path and obtain an insight to what is happening inside the agent platform during the test. They also give traces of faulty message exchanges or misprocessing that otherwise would be very hard to detect during the forensic review of the ACL messages traced. This demonstrates how active and passive MAS testing approachs can be combined to holistically perform the agent testing activity

Another great lesson from this exercise is how the test driven development helps improve the system while designing the test cases. This is evident as the Query Agent Subscriptions agent action was not initially considered as an action for the Monitor Manager Agent, and it had to be included in order to check the subscription list of the agent and perform the assertions needed to validate the test case.

6 Conclusions

In this article we present a testing approach for Multi Agent Systems. We have carefully avoided making references to any development paradigm or agent platform in order to keep the approach as methodology agnostic as possible; our goal was to present a testing method that can be applied to any type of MAS system without regard for the underlying design methodology or platform. Based on the V-Model, we have presented a set of testing stages to follow while developing a Multi Agent System. The blueprints of the testing toolkit developed for our test driven development activities were also presented, in an attempt to share their functionality and enable other researchers to implement and extend their functionality [12].

We also presented some insights into the test plan development and showed how running the test cases can be performed with active and passive testing approaches concurrently, in order to have a more holistic view of the system's behavior while performing the test cases.

7 REFERENCES

1. G. Caire, M. Cossentino, et. al., "Multi-Agent Systems Implementation and Testing". Proc. of the 4th "From Agent Theory to Agent Implementation" Symposium, (2004)
2. D. Nguyen, A. Perini, P. Tonella, "A Goal-Oriented Software Testing Methodology". Springer Berlin / Heidelberg, Lecture Notes in Computer Science Vol. 4951 pp. 58-72, ISSN 978-3-540-79487-5 (2008)
3. R. Coelho, U. Kulesza, A. von Staa, C. Lucena, "Unit Testing in Multi-agent Systems using Mock Agents and Aspects". Proc. Of the SELMAS'06, (2006)
4. E. Cortese, G. Caire, R. Bochicchio, "JADE Test Suite User Guide". TILab, (2005)
5. Z. Houhamdi, "Multi-Agent systems Testing: A Survey". International Journal of Advanced Computer Science and Applications Vol. 2, No. 6, ISSN 2156-5570 (2011)
6. C. D. Nguyen, A. Perini, C. Bernon, et. al., "Testing in Multi Agent Systems". Proceedings of AOSE'10 International Conference, ISBN: 978-3-642-19207-4 (2011)

7. P. Tahchiev, F. Leme, V. Massol et. al., "Junit in Action 2nd Edition". Manning Publications, ISBN: 978-1-935-18202-3 (2011)
8. Haskins, C., Forsberg, K.: Systems engineering handbook: A guide for system life cycle
9. processes and activities; incose-tp-2003-002-03.2. 1 (2011)
10. S. Kariyuki, H. Washizaki, Y. Fukazawa, et. al., "Acceptance Testing based on Relationships among Use Cases". 5th World Congress for Software Quality (2011)
11. JDOM Project, JDOM, http://www.jdom.org
12. [Z. Zhang, J. Thangarajah, L. Padgham, "Automated Unit Testing for Agent Systems". 2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE-07) (2007)
13. [G. Isaza, A. Castillo, M. López, L. Castillo, "Towards Ontology-Based Intelligent Model for Intrusion Detection and Prevention", Computational Intelligence in Security for Information Systems Advances in Intelligent and Soft Computing Volume 63, Springer, 2009, pp 109-116 (Book Chapter).
14. G. A. Isaza, A. G. Castillo, M. López, L. F. Castillo, M. López, "Intrusion Correlation Using Ontologies and Multi-agent Systems", In proceeding of: Information Security and Assurance - 4th International Conference, ISA 2010, Miyazaki, Japan, June 23-25, 2010. Proceedings 01/2010; pp.51-63.