Unit Testing in Multi-agent Systems using Mock Agents and Aspects

Roberta Coelho Uirá Kulesza Arndt von Staa Carlos Lucena

Computer Science Department, Pontifical Catholic University of Rio de Janeiro – PUC-Rio, Brazil

{roberta, uira, arndt, lucena}@inf.puc-rio.br

ABSTRACT

In this paper, we present a unit testing approach for MASs based on the use of Mock Agents. Each Mock Agent is responsible for testing a single role of an agent under successful and exceptional scenarios. Aspect-oriented techniques are used, in our testing approach, to monitor and control the execution of asynchronous test cases. We present an implementation of our approach on top of JADE platform, and show how we extended JUnit test framework in order to execute JADE test cases.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Software Testing and Debugging.

General Terms

Verification

Keywords

Mock Objects, Unit Testing, Dependability, Aspect Oriented Programming.

1. INTRODUCTION

The pervasiveness of World Wide Web and the omnipresence of cellular phones and smart devices are stimulating the creation of a new class of software, composed by applications structured as a collection of distributed components that must deal with inputs from a variety of sources, and often provide real-time responses. Moreover, these applications are intended to address stringent dependability requirements.

Agent technology has emerged as a prominent technique to address the design and implementation of these new and complex distributed systems. While the asynchronous architecture of multi-agent systems (MASs) helps to address current application requirements, it also brings many threats to software dependability. According to [1] there are four complementary means to attain dependability: fault prevention, fault tolerance, fault removal and fault forecasting. In this work, we propose a MASs testing approach which aims at removing faults along the application development.

SELMAS'06, May 22–23, 2006, Shanghai, China.

Although many agent-oriented software engineering methodologies [6] have been recently proposed, only a few define an explicit verification process. On the other hand, recent software engineering methodologies, such as Extreme Programming [2], has emphasized the importance and relevance of unit testing in the development of software systems. Unit testing has been recognized as a useful technique to verify the accuracy and reliability of systems [2, 21]. Regarding multi-agent systems, few research works have been undertaken in order to provide developers with valuable tools to support this level of testing.

This paper presents a unit testing approach for MASs. The main purpose of our approach is to help MASs developers in testing each agent individually. It relies on the use of *Mock Agents* to guide the design and implementation of agent unit test cases. Aspect-oriented techniques are also used in our approach to monitor and control the asynchronous execution of the agents during testing. We present an implementation of our approach on top of JADE platform. In order to allow the execution of our JADE unit test cases we extended JUnit test framework [13].

The remainder of this paper is organized as follows. Section 2 analyses the current status of MASs testing. Section 3 presents an overview of our unit testing approach for MASs. Section 4 shows the implementation of our approach on top of JADE [3] platform. Section 5 describes a worked example, and provides some discussions about this work. Finally, Section 6 presents our conclusions and points to directions for future work.

2. MULTI-AGENT SYSTEMS TESTING

Agent-Oriented Software Engineering (AOSE) methodologies, as they have been proposed so far, mainly proposes disciplined approaches to analyze, design and implement MASs [6]. However, little attention has been paid to how multi-agent systems can be tested [6]. Only a few of these methodologies define an explicit verification process. MaSE [9] and MAS-CommonKADs [15] methodologies propose a verification phase based on model checking to support automatic verification of inter-agent communications. Desire [16] proposes a verification phase based on mathematical proofs - the purpose of this process is to prove that, under a certain set of assumptions, a system adheres to a certain set of properties. Only some iterative methodologies propose incremental testing processes with supporting tools. These include: PASSI/Agile PASSI [5], AGILE [19].

AGILE [19] defines a testing phase based on JUnit test framework [13]. In order to use this tool, designed for OO testing, in MAS testing context, they needed to implement a sequential agent platform, used strictly during tests, which simulates asynchronous message-passing. Having to execute unit tests in an environment

different from the production environment results in a set of tests that does not explore the hidden places for failures caused by the timing conditions inherent in real asynchronous applications. Agile PASSI [5] proposes a framework to support tests of single agents. Despite proposing valuable ideas concerning MAS potential levels of tests, PASSI testing approach is poorly documented and does not offer techniques to help developers in the low level design of unit test cases.

Hence, we can say that few research works have been undertaken in order to provide MASs developers with a detailed testing process and valuable tools to support testing activities.

2.1 Test Levels

Over the last years, the view of testing has evolved, and testing is no longer seen as an activity which starts only after the coding phase is completed. Software testing is now seen as a whole process that permeates the development and maintenance activities. Thus, each development/maintenance activity should have a corresponding test activity. Figure 1 shows a correspondence between development process phases and test levels [22].

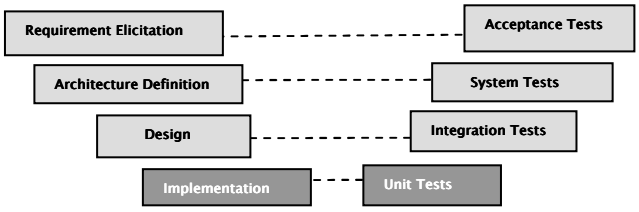

Figure 1: Development and Testing processes correspondence (adapted from [22]).

Each test level, showed in Figure 1, focuses on a particular class of faults [22]: (i) Acceptance Test aims at finding defects in requirements [22]; (ii) System Test aims at finding defects in system specification; (iii) Integration Test intends to find incompatibilities/inconsistencies between elements' interfaces; (iv) and Unit Test verifies whether software units of modularity (e.g. methods, classes, agents) operate as defined in their specification. This principle applies no matter what software life cycle is used [22]. As a consequence, these levels can be used to express a MASs testing process. In this paper we will particularly deal with the unit test level. In the following sections we are going to present an agent unit test approach and an implementation of this approach upon JADE [3] platform.

3. A UNIT TEST APPROACH FOR MULTI-AGENT SYSTEMS

Our testing approach calls attention to the test of the smallest building blocks of the MAS: the agents. Its basic idea is to verify whether each agent in isolation respects its specifications under normal and abnormal conditions. There are different proposals for representing an agent. Our approach is based in the definition detailed in [14] and presented below:

> *An agent is an autonomous, adaptive and interactive element that has a mental state. The mental state of an agent is comprised by: beliefs, goals, plans and actions. Every agent of the MAS plays at least one role in an organization. One of the attributes of a role is a number of protocols, which define the way that it can interact with other roles.*

Agents encapsulate a very complex internal structure (often composed of several classes and/or methods). In order to verify whether these inner components contain faults, we can use traditional unit testing techniques. However, the agent is the unit of modularity of MASs - which is internally coherent and has minimum coupling with the rest of the system – and as such should be tested as a whole.

A running MAS is a web of agents that interact with each other and with an external environment. Since, this kind of interaction differs in nature from the direct method call that takes place within an agent (among the classes that constitutes it) we need to devise specific techniques to test each individual agent. Given that, nearly no agent is an island, almost all agents interact to others, to whom they provide services or on whom they rely for services, one question arises: *How can we define meaningful tests to verify an agent in isolation?*

Figure 2 depicts a test of agent A, which needs a service provided by an agent that plays role B. In this figure, A is the Agent Under Test (AUT). Along this paper, we will use this term to refer to the agent being tested.

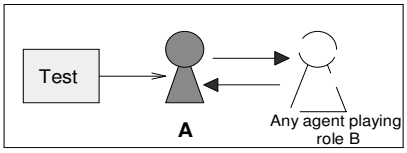

Figure 2: Unit testing Agent A.

In order to test A in isolation, a valuable strategy is to define a "dummy" version of B, usually called stub. Stubs are fake implementations of production code that return canned results. Mackinnon et al [20] proposed the Mock Object test design pattern[1] [4], and since then, Mock Objects have been recognized as a useful approach to the unit test and design of object-oriented software. A *Mock Object* is a regular object that acts as a stub, but also includes assertions to instrument the interactions of the target object with its neighbors. Mock Objects can be used in MAS testing to simulate real environmental resources, under the conditions defined by [20].

In our approach, we adapted Mackinnon et al. idea to MAS testing context and defined the concept of: *Mock Agent*. *A Mock Agent* is a regular agent that communicates with just one agent: the AUT. It has just one plan to test the AUT. The *Mock Agent's* plan is equivalent to a test script, since it defines the messages that should be sent to the AUT and the messages that should be received from it. By testing an agent in isolation using *Mock Agents* the programmer is forced to consider the agent's interactions with its collaborators (or competitors), possibly before those collaborators (or competitors) exist. The next section details our unit test approach, which uses *Mock Agents* to guide the design of each test case.

3.1 Approach's Overview

Figure 2 depicts our agent unit test approach that is composed of five participants:

[1] A Test Design Pattern defines a good design solution for a system intended to test another system.

- *Test Suite*: which consists in a set of *Test Cases* and a set of operations performed to prepare the test environment before a *Test Case* starts.
- *Test Case*: defines a scenario – a set of conditions – to which an *Agent Under Test* is exposed, and verifies whether this agent obeys its specification under such conditions.
- *Agent Under Test* (AUT): is the agent whose behavior is verified by a *Test Case*.
- *Mock Agent:* consists in a fake implementation of a real agent that interacts with the AUT. Its purpose is to simulate a real agent strictly for testing the AUT.
- *Agent Monitor:* is responsible for monitoring agents' life cycle in order to notify the *Test-Case* about agents' states.

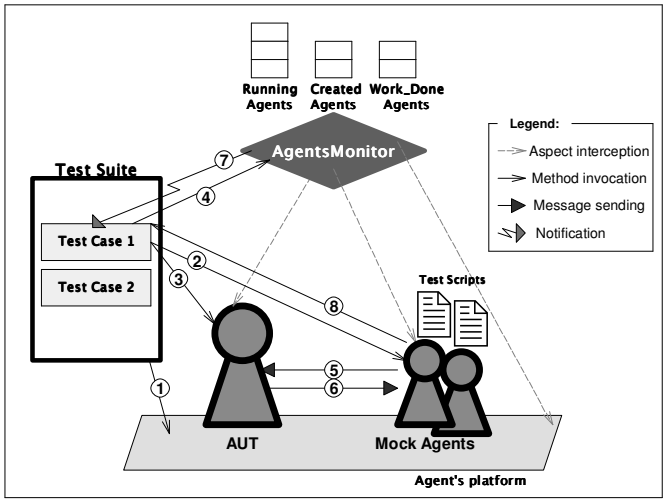

Figure 3: Workflow between the participants of a unit test.

Each agent unit test follows the common structure depicted in Figure 3. In step 1, the *Test Suite* creates the agent's platform and whatever element needed to set up the test environment. After that, a *Test Case* is started. The *Test Case* creates *Mock Agents* to every role that interacts with the Agent Under Test - in the scenario defined by the *Test Case* (step 2). Next, it creates the Agent Under Test (step 3)[2] and asks the *Agent Monitor* to be notified when the interaction between the AUT and the *Mock Agents* finishes (step 4).

At this point, the AUT and the *Mock Agent* start to interact. The *Mock Agent* sends a message to the AUT, and it replies (steps 5 and 6) or vice-versa. They can repeat steps 5 and 6 as many times as necessary to perform the test codified in the *Mock Agent's* plan (for instance, the *Mock Agent* can reply three messages before finalizing its test activity). During all this interaction process, the *Agent Monitor* keeps track of changes in agents' life cycle. In order to do that it uses three lists as illustrated in Figure 3:

(i) **Created Agents List**: maintains IDs of the agents that have been created, but are not running yet – an ID is any information that uniquely identifies an agent.

(ii) **Running Agents List**: maintains IDs of the agents in the running state.

(iii) **WorkDone Agents List**: maintains IDs of the *Mock Agents* that have completed their plan (equivalent to a test script).

When a *Mock Agent* concludes its plan, the *Agent Monitor* includes the *Mock Agent's* ID in the *WorkDone* list, and then notifies the *Test Case* that the interaction between the *Mock Agent* and the AUT have concluded (step 7). Lastly, the *Test Case* asks the *Mock Agent* whether or not AUT acted as expected (step 8).

This agent unit testing approach has two main concerns: (i) the design of a *Test Case* based on the use of *Mock Agents*; (ii) and the *Test Case* execution which relies on the *Agent Monitor* to notify when the test script (codified in *Mock Agent's* plan) was concluded. Next Sections will detail these two concerns.

3.2 Test-Case Design based on Mock Agents

A very important consideration in program testing is the design and creation of effective test cases [22]. Testing, however creative and seemingly complete, cannot guarantee the absence of all errors [22]. Test-case design is so important because complete testing is almost impossible; a test of any non trivial program must be necessarily incomplete. The obvious strategy, then, is to try to make tests as complete as possible. Given constraints on time and cost, the key issue of testing becomes: *What subset of all possible test cases has the highest probability of detecting the most errors?*

The study of test-case design methodologies supplies answers to this question [22]. In general, the least effective methodology of all is to arbitrarily choose a set of test cases. In terms of the likelihood of detecting the most errors, an arbitrarily selected collection of test cases has little chance of being an optimal, or even close to optimal, subset. Unit test approaches for MASs proposed so far does not define a methodology for test-case selection. Below we present an error-guessing [22] test-case-design technique. The basic idea of an error-guessing technique is to enumerate a list of possible error-prone situations and then write test cases based on the list. The process is as follows:

```
1. For each agent to be tested
      1.1. List the set of roles that it plays.
2. For each role played by the AUT
      2.1. List the set of other roles that interacts
           with it.
3. For each interacting role:
      3.1 Implement in a Mock Agent a "plan" that
          codifies a successful scenario.
      3.2. List possible exceptional scenarios that the
          Mock Agent can take part.
      3.3. Implement in the Mock Agent an extra plan
          that codifies each exceptional scenario.
```

This technique should be applied for each agent of the MAS, or a subset of the agents responsible for the "core" functionalities of the MAS. Such a choice will be guided by the time and cost constraints previously mentioned. At the end of this process, a *Mock Agent* comprises a set of expected behaviors (under successful and exceptional scenarios) of an agent interacting with AUT. In order to help developers in the definition of *Mock Agents* plans, useful sources of information are sequence diagrams[3] and

[2] In scenarios where the mock agent initiates the interaction process (test drivers) they should be created after AUT.

[3] Many AOSE methodologies propose variations of sequence diagrams in order to represent agent's behavior.

the specification of protocols that regulates the interaction between MAS roles.

As each *Mock Agent* exercises just one role of the AUT, rather than the wide interface that comprises all the features provided it, we call this approach "Role Driven Unit Testing". However, the notion of a role, while supported in many AO methodologies, is not used by some of them. In case the unit test developer is not using a role-based methodology, the process described above should be adapted. Instead of identifying each step according to the agent's role, he/she should define each step according to the agents' plans. Thus, the first two steps would be: (1) For each agent of the MAS, list the set of plans that it performs; (2) For each plan performed by the AUT, list the set of other agents that interacts with it.

Although our approach can help MAS developers in unit test cases construction, it does not intend to be complete. This technique should be combined with other strategies. The reason for such combination it is that: each test-case-design technique contributes a specific set of useful test cases, but none of them by itself contributes a thorough set of test cases [22].

3.3 Test Case Execution

According to our approach, the plan of a *Mock Agent* comprises the logic of the test. Each *Test Case* just starts the AUT and the corresponding *Mock Agent*(s) and waits for a notification from the *Agent Monitor* – informing that the interaction between the agents have finished – in order to ask the Mock Agent(s) whether or not the AUT acted as expected. To keep track of the changes in agents' life cycle, *Agent Monitor* needs to include and remove information from the three lists described previously: *Running Agents*, *Created Agents* and *WorkDone Agents*. In order to access such information the *Agent Monitor* needs to observe specific application events, such as: agent creation, the moment at an agent starts running, agent finalization, and so on.

To prevent *monitoring* concern from becoming scattered across multiple platform modules and tangled with other application concerns, the *Agent Monitor* participant is built upon the facilities of Aspect Oriented Software Development (AOSD) [11, 17]. AOSD has been proposed as a paradigm for improving separation of concerns in software design and implementation. It proposes a new abstraction, called Aspect, with new composition mechanisms which support the modularization of crosscutting concerns. The aspect abstraction aims at encapsulating concerns that crosscut several system modules. Since the *Agent Monitor* deals with the "monitoring" concern, which has a crosscutting nature, it is represented as an Aspect in our approach. Section 4.3 shows the implementation of *Agent Monitor* using AspectJ [18], an aspect-oriented extension to the Java language.

4. APPLYING OUR APPROACH ON TOP OF JADE

In order to validate our testing approach, we have applied it on top of JADE [3]. JADE is an object-oriented framework to develop agent applications in compliance with FIPA specifications for interoperable MASs. Next sections describe step by step how our testing approach was build on top of JADE.

4.1 JADE Mock Agent

An agent in the JADE platform is defined as a Java class that extends the base `Agent` class from JADE framework. Each Agent contains its own thread of execution and defines a set of *behaviors*. A *behavior* is a concept in JADE platform that represents a logical activity unit of a JADE agent [3].

The `JADEMockAgent` class, as any other agent in this platform, extends the `Agent` class, as illustrated in Figure 4. The `JADEMockAgent` plan (equivalent to a JADE Behavior) is analogous to a test script, since it defines the messages that should be sent to AUT and which messages that should be received from it. After executing its plan (equivalent to test script), the `JADEMockAgent` needs to report the test result (success or failure) to the *Test Case*, which in counterpart, will be in charge of examining the test result.

There are many ways of reporting the result of a test. Some of them are: (i) to include the test result in an ACL specific message and send it to another agent that would generate a textual/graphical report; and (ii) to define an interface that contains a set of methods that should be implemented by an agent that wants to report the result of a test script.

In our particular implementation, we have chosen the second alternative. Thus, the `JADEMockAgent` class implements the `TestResultReporter` interface illustrated in Figure 4. `JADEMockAgent` also implements two other methods: `sendMessage()` and `receiveMessage()`.The `receiveMessage()` method performs assertions concerning the received message (e.g whether the message was received within a specific timeout, or if it obeys a pre-defined format). It is implemented following the Template Method design pattern [12] in order to enable the developer to perform additional assertions in the received message.

4.2 JADE Test Suite and Test Cases

Instead of creating a unit testing tool from scratch to exercise the *Test Cases* defined in our approach, we decided to extend JUnit framework [13] to support JADE agents' tests. The reason for that is to lower the developers' learning curve providing a simple, and widely used testing framework architecture. JUnit was already ported to many other languages in different paradigms (e.g. CppUnit, NUnit, PyUnit, XMLUnit). All these frameworks, known as xUnit family of tools, have the same basic architecture. Figure 4 illustrates the main modules of JUnit Framework.

Before diving into JADE Agents testing, we need to relate our testing concepts, described in Section 3.1, with the concepts used in JUnit framework: our concept of *Test Case* is represented in JUnit Framework by what they call a test method, and JUnit's concept of *Test Case* is equivalent to our concept of *Test Suite*.

The `JADETestCase` class (which implements our *Test Suite* concept) extends the `TestCase` class from JUnit [13], as shown in Figure 4. This class defines a set of concrete and abstract methods which support the implementation of the test methods (equivalent to our *Test Case* concept). The `createEnvironment()` method is called inside the `JADETestCase` constructor. It is responsible for creating the JADE environment that will be active during the execution of all test methods. Each test method will be able to include agents in such environment by calling `createAgent()`. The `tearDown()` method removes all agents from the environment after the execution of each test method.

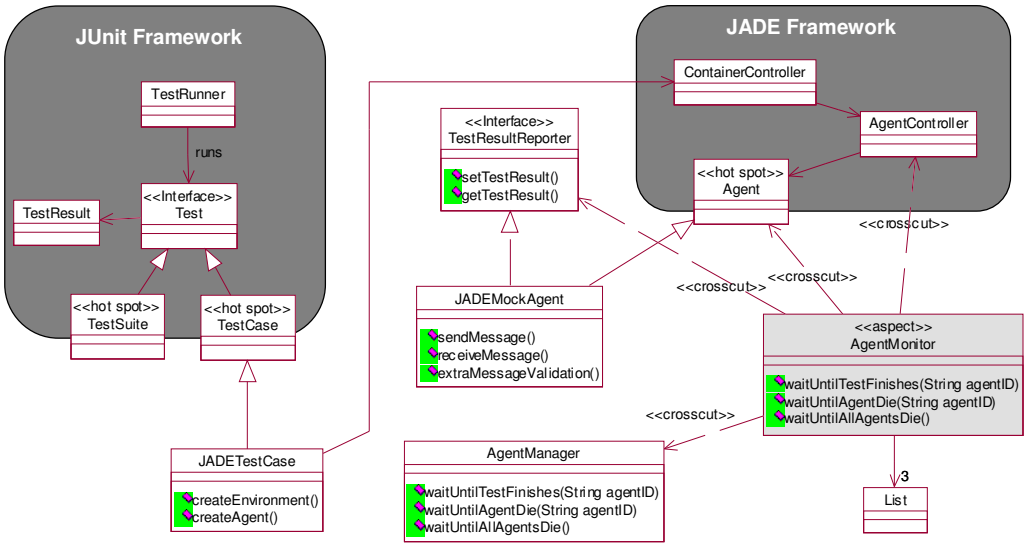

Figure 4: Implementing our testing approach upon JADE Platform - a low level design view.

4.3 Creating Agent Monitor with AspectJ

The *Agent Monitor* of our approach was implemented using AspectJ [18], which is an aspect-oriented extension to the Java programming language. In AspectJ, an aspect comprises one or more advices (code snippets - like methods) and a list of pointcuts which specify points in the execution of classes (e.g. method execution, method call, constructor execution) where the advice code should be included. Aspect weaving is the mechanism responsible for composing the Java classes and the AspectJ aspects. For more detail about AspectJ refer to [18].

Figure 5 shows the partial code of `AgentMonitor` using AspectJ. `AgentMonitor` is represented by an aspect which encapsulates the three agent lists described previously. In order to access the information about agents life cycle – and include such information in the agents lists – it crosscuts the components responsible for: agent creation (a specific class inside JADE-Platform), agent execution (`Agent` class), and test execution (`TestResultReporter` interface – described in Section 4.1). Figure 4 illustrates such crosscutting relations. Similar design solutions were adopted by [10][8] to instrument OO systems.

According to the unit testing approach detailed in Section 3, the main purpose of managing the information inside these three lists is to notify *Test Case* when the interaction between AUT and the Mock Agent finishes. To fulfill this requirement `AgentMonitor` aspect implements the following methods: `waitUntilAllAgentsDie()`, `waitUntilTestFinishes()`, `waitUntilAgentDies()`. Each method periodically analyzes the agents lists and make the *Test Case* (JUnit test method), that called them, wait until the condition specified in methods signature is reached.

Since we do not want the developer of a JADE *Test Case* to learn about aspect-oriented programming (AOP) in order to call such auxiliary wait* methods, we created an auxiliary class called `AgentManager`. This class contains empty-body implementations of all wait* methods defined in `AgentMonitor` aspect. The `AgentMonitor` aspect intercepts all methods of the `AgentManager` class, and replaces each empty body method by the implementation provided by the `AgentMonitor` (using `around` advices). Figures 4 and 5 illustrate this crosscutting relation.

5. WORKED EXAMPLE

To illustrate our agent unit test approach, we will work through an example. Consider an application of book-trading, in which, each agent can play a BookSeller role, a BookBuyer role or both.

Figure 5 details the interaction protocol between these roles. According to it, as soon as a BookSeller agent joins the environment, it registers itself in a Service Directory as a "book-seller" and starts to wait for book-buying requests. When a BookBuyer agent joins the environment, it initially looks for the agents already registered in the Service Directory as "book-sellers". After that, it sends a "cfp" message to all the agents registered as "book-sellers". When the BookSeller agent receives a "cfp" message from a BookBuyer, it searches in its catalogue for the requested book. If it is available, the BookSeller agent sends a "propose" message in reply to "cfp" message, whose content is the book price. If on the other hand, the BookSeller agent does not have the book on catalogue it will send a "refuse" message informing the BookBuyer Agent that the book is not available. The BookBuyer agent receives all proposals/refusals from seller agents and chooses the agent with the best offer. Then, it sends the chosen seller a "purchase" message. When the BookSeller agent receives a "purchase" message it removes the book from the catalogue and sends an "inform" message to notify to the BookBuyer agent that the book sale was complete. However, if for any reason the book is no more available in the catalogue the BookSeller agent sends a "failure" message informing BookBuyer agent that the requested book is no more available. If the BookBuyer agent receives a message indicating that the sale was complete, the agent can terminate. Otherwise, it will re-executes its plan and try to buy the book again from some other agent.

In order to implement this simple MAS in the JADE platform two subclasses of `jade.Agent` class were created: the `BookSellerAgent` and the `BookBuyerAgent`. These classes obey the interaction protocol detailed above. Here, we will illustrate the `BookSellerAgent` unit test. The first step is to follow the procedure described in section 3.2. Table 1, summarizes the information collected while following these steps.

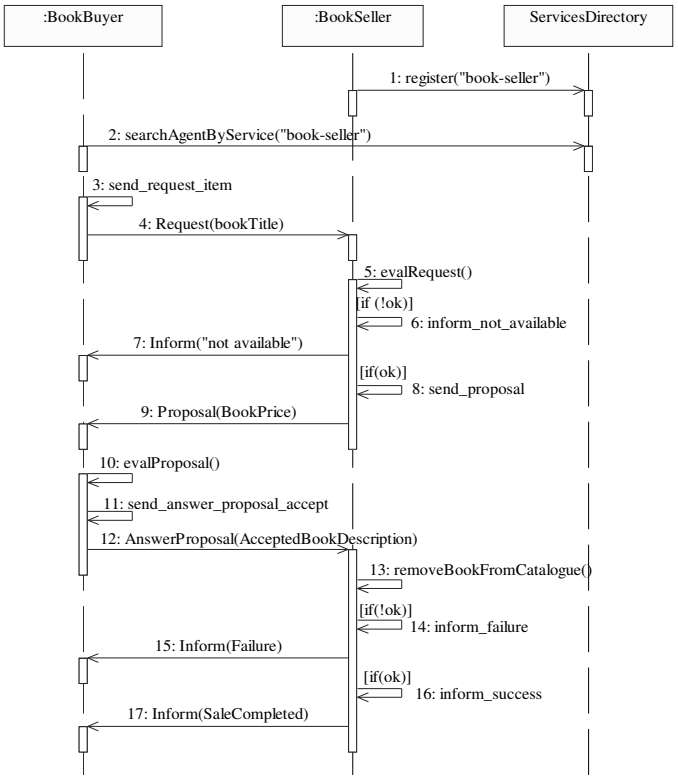

Figure 6: Book-trading workflow

Our first test-case should be the simple success case briefly described in Table 1. To implement this test-case all we need is to write a *Mock Agent* that simulates this scenario. In order to implement a second test case which verifies an exceptional scenario we just need to implement an extra plan (JADE Behavior) in our *Mock Agent* class.

For the sake of simplicity, Figure 7 illustrates the partial code of the *Mock Agent* which contains only the behavior that verifies the exceptional scenario described in Table 1.

Table 1: Unit Test Case template.

Agent	BookSellerAgent
Roles	BookSeller
Interacting Roles	BookBuyer
Successful Scenario	BookSellerAgent sells a book to an agent playing BookBuyer role.
Exceptional Scenario	A BookBuyer agent can send a "cfp" message requesting a specific book, and afterwards send a purchase message trying to buy a different book.

As we can see the *Mock Agent* has just one plan (represented by a JADE Behavior called `TestScenario`) to test BookSellerAgent. `TestScenario` class codifies the "logic" of the test-cased. This Behavior allows the agent to send, receive and check the content of the messages received from BookSellerAgent (the AUT in this example). The methods detached in gray are methods from `JADEMockAgent` class - described in Section 4.1– that eases the implementation of the *Mock Agent*. The `sellerID` variable (used in line 12) contains the identification (agent's local name) of the `BookSellerAgent` instance under test.

```
1. public class BookSellerTestCase extends JADETestCase {
2.   ...
3.   public void testBookSelling_Success(){
4.        …
5.     createAgent("seller","BookSellerAgent",argS);
6.     createAgent("buyer","MockBookBuyerAgent",argB)
7.     AgentsManager.waitUntilTestFinishes("buyer");
8.     mockAg=environment.getLocalAgent("buyer");
9.     res=((TestReporter) mockAg).getTestResult();
10.    if(!res.equals("OK"))){
11.         fail(res);
12.     }
13.   }
14. }
```

Figure 8: Partial code of a `JADETestCase`.

To execute this Test Case all we need is to create a subclass of `JADETestCase` and to implement a test method that creates an instance of AUT (`BookSellerAgent`), and an instance of the *Mock Agent* (`MockBookBuyerAgent`). After that, this test method waits until their interaction finishes and asks the Mock Agent about the test result. Figure 8 presents a partial code of this Test Case (we have left out local variables definitions for brevity).

```
1. public class MockBookBuyerAgent extends JADEMockAgent {
2.      ...
3.      protected void setup() {
4.           ...
5.           addBehaviour(new TestScenario());
6.       }
7.}
8. private class TestScenario extends OneShotBehaviour {
9.   public void action(){
10.     try {
          ...
11.       sendMessage(msgType.CFP,sellerID, bookTitle);
12.       reply = receiveReply(6000, msgType.PROPOSE);
13.       sendMessage(msgType.Accept,sellerID,otherTitle);
14.       reply2 = receiveReply(6000, msgType.FAIL);
15.    } catch (ReplyReceptionFailed e) {
16.       setTestResult( prepareMessageResult(e));
17.    }
18.    setTestResult("OK");
19. }
20.}
```

Figure 7: Partial code of a Mock Agent

Working through this example has shown how programmers could test agent roles gradually. Writing tests provides a framework to think about MAS functionality, Mock Agent provide a framework for making assertions about relationships between agents. All this encourages programmers to think about "failure scenarios" during agent development, and verify the way an agent responds to these scenarios. *Mock Agents* also allow programmers to make their tests as precise as they need to be. In scenarios, where an AUT needs to interact with more than one *Mock Agent*, we can have simple *Mock Agents* that only send canned messages and more sophisticated *Mock Agents* that validate the order the messages are exchanged and its content (as the one we used in this example).

The overhead incurred by our approach are as follows:

1. *Development time overhead*: The tester's time spent to write the Mock Agents. However, such cost is mitigated by the re-execution of tests along development and maintenance phases.

2. *Compilation overhead*: The application code needs to be re-compiled. Currently, our approach uses a weaving process [18] that works at the source code level, thus recompiling it is necessary. This overhead might be eliminated by using an aspect-oriented language that supports runtime weaving.

3. *Run-time overhead*: The Agents Monitor aspect inserts additional code in agent's platform, in order to monitor specific events. This extra code will cause an overhead in runtime. However, this overhead is small since the extra code was limited to atomic operations in the lists of agent IDs previously described.

6. CONCLUSIONS & FUTURE WORK

In this paper, we presented a unit testing approach for MASs. Our approach aims at helping MASs developers in testing each agent individually. It relies on the use of *Mock Agents* to guide the design and implementation of agent unit test cases. Each *Mock Agent* performs a test script, in which it sends and receives messages from the agent being tested. Each *Mock Agent* is responsible for testing a single role of an agent (AUT), under successful and exceptional scenarios.

A test case in our approach consists of one or more *Mock Agents* interacting with an AUT - each one has its own thread of execution. In order to monitor and control the execution of such asynchronous tests cases we used the facilities provided by AOP. We believe that our aspect oriented solution, embodied by *Agent Monitor* concept, can be extended in order to introduce other classes of instrumentation in MAS.

Our work represents an initial step in the definition of a complete MAS testing process which will provide strategies to the integration and system testing levels. We also intend to address, in future works, the integration of this testing process with existing development methodologies. We are currently investigating, how our *Mock Agents* can be used to progressively test, agents' integration scenarios.

Finally, we are also investigating the complete specification of a generative approach [7] which can integrate and generate the source code of *Mock Agents*, *Test Suites* and *Test Cases*. Agent interaction diagrams and specification of agent communication protocols, for example, can work as a source of information to generate the *Mock Agents* source code. The definition of this generative approach can improve the productivity of our agent unit testing proposal and motivates even more developers to adopt it in the development of large scale multi-agent systems.

Acknowledgements. This work has been partially supported by CNPq under grant 150678/2004-7 for Roberta Coelho and by FAPERJ under grant No. E-26/151.493/2005. The authors are also supported by the ESSMA Project under grant 552068/02-0.

7. REFERENCES

[1] Avizienis, A; Laprie, J-C; Randell, B; Landwehr, C. Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1(1), pp. 11-33, 2004.

[2] Beck, K. Extreme Programming Explained. Reading, MA: Addison-Wesley, 2000

[3] Bellifemine, F., Poggi, A., Rimassa, G. JADE - A FIPA2000 Compliant Agent Development Environment. In Proc. Agents Fifth International Conference on Autonomous Agents, pp. 216-217, 2001.

[4] Binder, R. Testing object-oriented systems: models, patterns, and tools.Addison-Wesley Longman Publishing Co., Inc., 1999

[5] Caire, G., Cossentino, M., Negri, A., Poggi, A., Turci, P., Multi-agent systems implementation and testing. In Proc. Of 4th International Symposium - From Agent Theory to Agent Implementation (AT2AI-4), 2004.

[6] Cernuzzi, L., Cossentino, M., Zambonelli, F. Process Models for Agent-based Development, Journal of Engineering Applications of Artificial Intelligence, 18(2), 2005

[7] Czarnecki, K. and Eisenecker, U. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000

[8] Dantas, A., Cirne, W., Saikoski, K., Using AOP to Bring a Project Back in Shape: The OurGrid Case, Journal of the Brazilian Computer Society, 2005.

[9] DeLoach, S., Wood, M. and Sparkman, C. Multiagent Systems Engineering. International Journal of Software Engineering and Knowledge Engineering, vol. 11, No. 3, pp. 231-258, 2001.

[10] Deters, M.; Cytron, R. K. Introduction of Program Instrumentation using Aspects. Workshop on Advanced Separation of Concerns in OO Systems, 2001.

[11] Filman, R., Elrad, T., Clarke, S., Aksit, M. Aspect-Oriented Software Development. Addison-Wesley, 2005.

[12] Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[13] Gamma, E. and Beck, K. JUnit: A regression testing framework. http://www.junit.org, 2000. ,

[14] Garcia, A., Lucena, C., Cowan D. Agents in Object-Oriented Software Engineering. Software Practice & Experience, Elsevier, 34 (5), pp. 489-521, 2004.

[15] Iglesias, C., Garijo, M., Gonzalez, J.C., Velasco, J.R. Analysis and Design of Multiagent Systems using MAS-CommonKADS. Springer, LNCS 1365, pp. 312-328, 1997.

[16] Jonker, C.M., and Treur, J. Compositional Verification of Multi-Agent Systems: a Formal Analysis of Pro-activeness and Reactiveness. Proc. of COMPOS'97, Springer, LNCS 1536, 1998.

[17] Kiczales, G.,Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J. Aspect-Oriented Programming. European Conference on Object-Oriented Programming (ECOOP), Springer, LNCS (1241), 1997.

[18] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G.Getting Started with AspectJ. Communication of the ACM, 44(10), pp. 59-65, 2001.

[19] Knublauch, H. Extreme programming of multi-agent systems. In Proc. 1st International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 704 – 711, 2002.

[20] Mackinnon, T., Freeman, S., and Craig, P. EndoTesting: Unit Testing with Mock Objects. Proc. of XP2000, 2000.

[21] McConnell, Code Complete, 2nd Ed., Microsoft Press, 2004.

[22] Myers, G. J. The Art of Software Testing. Wiley, 2nd Ed. 2004.