PUC

Monografias em Ciência da Computação

The JAT framework:
A Test Automation framework for Multi-agent Systems

Roberta Coelho

Elder Cirilo

Uirá Kulesza

Arndt von Staa

Awais Rashid

Carlos Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. ___/____ ISSN:_____
Editor: Prof. Carlos José Pereira de Lucena April,2007*(under development)*

The JAT framework: A Test Automation framework for Multi-agent Systems *

Abstract. Testing is a critical activity of software development. Software tests tell developers that the application's pieces are working as designed, and give them confidence that a software maintenance does not break system existing functionalities. Concerning multi agent systems (MASs) very few works have been undertaken in order to provide developers with valuable testing tools. In this work we propose a testing tool which allows the MAS developer to define automatic tests. In this article, 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.

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

---

* This work has been sponsored by the Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

In charge for publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Table of Contents

1 Introduction

Agent technology has emerged as a prominent technique to address the design and implementation of complex distributed systems. It provides a way of solving complex problems through abstractions of a higher level (e.g. agents, roles) than the ones used by the object oriented and procedural paradigms (see Figure 1). A multi agent system (MAS) is a web of agents that interact with each other by sending and receiving messages and with the external environment by observing and affecting it.
A MAS is based on multiple threads of execution and asynchronous exchange of data between the agents. From the testing perspective, these characteristics bring many new challenges to software testability. Traditionally, a software behavior results from a sequential execution of instructions. This kind of behavior can be easily understood and tested, since there is an expected behavior we can reference to. In a MAS, however, the expected behavior is less tangible, since it results from several sequences of instructions (the agents) that operates in parallel and communicates with each other.
According to IEEE std. 610.12, software testability is the degree to which a system or component facilitates the establishment of a test criteria and the performance of tests to determine whether those criteria have been met. The testability has two facets [4, 34]:

- **Controllability**: o test a component we must be able to *control* its input (and internal state).
- **Observability**: o test a component we must be able to *observe* its outputs.

The obstacles to *controllability* and *observability* in a multi-agent system (MAS) result from the following characteristics: (i) an agent is an autonomous entity, and as a consequence, it is hard to be *controlled* by a testing tool; (ii) the agents´ beliefs are embedded in the agents, and thus they cannot be easily *observed* and *controlled* by a testing tool – almost always a testing tool can only *observe* an agent by its interactions with the other agents and with the environment.

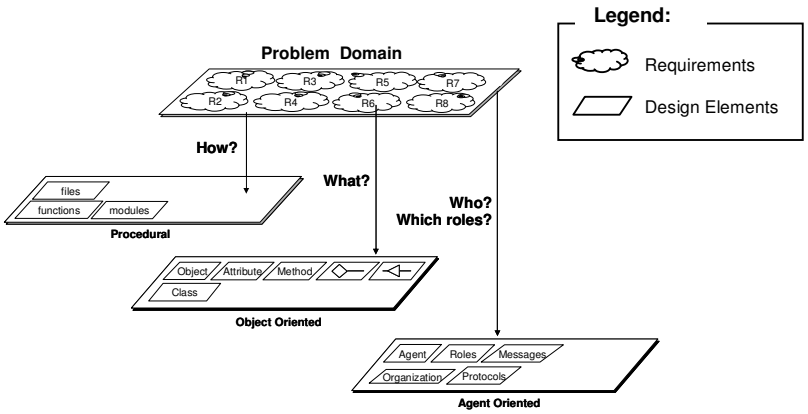

Figure **1.** Agent Oriented Development a new level of abstraction.

Testing is a critical activity of software development. Software tests tell developers that the application's pieces are working as designed, and give them confidence that the system meets its requirements. Testing a whole system manually is an exhaustive and imprecise process. The importance and usefulness of test automation approaches to

assure the software quality as a whole have been advocated by agile development methodologies, such as Extreme Programming. Test automation approaches has been gradually adopted by many software development companies [31, 32]. Experience reports [31, 35, 36, 30] issued by software companies have been shown that automatic tests can significantly improve the system quality as well as the development cost of the system.
A test automation approach is crucial to enable a wide adoption of Agent Technology. Such approach should account for the challenges of MAS testability described previously. In this work, we propose a testing infra-structure, which allows the MAS developer to define automatic tests of different levels (unit, integration and system levels). We also explore the integration of testing activities during the MAS development process, and discuss how a MAS can be gradually tested using the infra-structure proposed here. Although we focus on unit and integration test levels we show how this infrastructure can be easily extended to give a better support to system tests.

2 Background

This section briefly revisits the basic concepts of Aspect-oriented software development (AOSD) and research work on agent-oriented testing approaches.

2.1 Aspect Oriented Software Development

Aspect-oriented software development (AOSD) [19, 13] has emerged as a technique for improving the separation of crosscutting concerns. A concern is part of a problem that we want to treat as a single conceptual unit, and a crosscutting concern is a concern whose implementation is spread over several system modules. AOSD addresses the modularization of these concerns by providing a new abstraction, called aspect, which makes it possible to separate and compose the crosscutting concerns to produce the overall system.
AspectJ [18] is the most used aspect oriented programming language. It includes AOSD concepts in the Java programming language. The main concepts are the following: (i) join points – are well-defined locations within the base code where a concern can crosscut the application. Examples of join points are method calls and method executions; (ii) pointcuts – represent a collection of join points; and (iii) advices – are a special method-like construction of aspects which are used to attach new crosscutting behaviors along the pointcuts.
Over the last years, the software industry has also given more attention to AOSD techniques. Many available software products, such as the JBoss application server and the Spring integration framework, have adopted the use of aspect-oriented techniques to address traditional crosscutting concerns (transaction demarcation, security, monitoring, synchronization, cache, dynamic adaptation) encountered in the development of complex and modern systems. Aspect-oriented techniques have been also used to support testing and static analysis activities [38]. In this work, we use aspect-oriented techniques to support the testing of asynchronous agents.
Researchers on agent-oriented software engineering have also started to explore the benefits of AOSD in the development of MAS, such as: (i) in the modularization of agent mobility [40] and learning [39] properties; and (ii) the incorporation of aspects in agent models [4] and architectures [39].

2.2 Current Approaches for Testing MAS

Agent-Oriented Software Engineering (AOSE) methodologies, as they have been proposed so far, focus mainly on disciplined approaches to analyze, design and imple-

ment MASs [7]. However, little attention has been paid to how multi-agent systems could be tested [7]. How can be seen in table 1 only a few of these methodologies define an explicit verification process. MaSE [12] and MAS-CommonKADs [22] methodologies propose a verification phase based on model checking to support automatic verification of inter-agent communications. Desire [24] 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: AGILE [28] and Agile PASSI [6].
AGILE [28] defines a testing phase based on the JUnit test framework [17]. In order to use this tool, designed for OO testing, in the MAS testing context, it needs to implement a sequential agent platform, used strictly during tests, which simulates asynchronous message-passing. Executing unit tests in an environment different from the production environment results in a set of tests that does not adequately explore the hidden places for failures caused by the timing conditions inherent in real asynchronous applications. Agile PASSI [6] 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.

Table 1. The Testing characteristics of AO methodologies.

Characteristics / AO Methodologies and Process Models	Define an explicit verification phase	Levels of testing defined by the approach	Offers a tool for test automation	Offers a tool for test case generation	Evaluate the effectiveness of the test approach
AGILE [28]	Based on Testing	Unit	Yes	No	No
Agile PASSI [6].	Based on Testing	Unit	Yes	No	No
AOR	No	-	-	-	-
Desire	Based on mathematical proofs	-	-	-	-
Gaia	No	-	-	-	-
Ingenias	No	-	-	-	-
MaSE	Based on model checking	-	-	-	-
MAS-CommonKADs	Based on model checking	-	-	-	-
MASSIVE	No	-	-	-	-
OPM/MAS	No	-	-	-	-
Prometheus	No	-	Offers a debugging tool (*)	-	-
Roadmap	No	-	-	-	-
Tropos	No	-	-	-	-

(*) The debugging tool can be used on test activities to help developers when diagnosing the cause of a fault detected by a test.

A related concept to software testing is software debugging. The first is an activity in which a system or component is executed under specified conditions, and the results are observed and checked against expected results. The last aims at detecting, locating, and correcting faults in a computer program executing in real conditions. Though not being strongly related to the present work some works on MAS debugging are worth

to be mentioned, since they propose interesting ideas that can also be useful when building tests for MAS.
Prometheus methodology [35] propose the use of a central debugging agent, an agent inserted into a standard multi agent system which is responsible for intercepting all messages sent between the agents in the system. The debugging agent uses Petri nets to represent the possible protocols between agents and to detect the possible faults on agent communication. The ACLAnalyser tool [25] uses conventional data mining techniques such as clustering and summarizing to analyze and present results to the developer during a MAS debugging. Both works restate that debugging the communication that occurs between agents can be a quite difficult process.
The difficulty associated to debugging tasks of MASs and to the lack of detailed testing methods for MASs motivated the present work, which proposes a testing approach and a supporting testing infra structure for the design and execution of MAS tests. The testing approach and corresponding infrastructure propose here is not attached to a specific methodology or process model. But it can be easily integrated into any of the AOSE methodology listed above. The testing approach should be integrated in the initial stages of the development approach as stated in section 3.

3 Levels of Testing

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 2 shows a correspondence between development process phases and test levels [27].

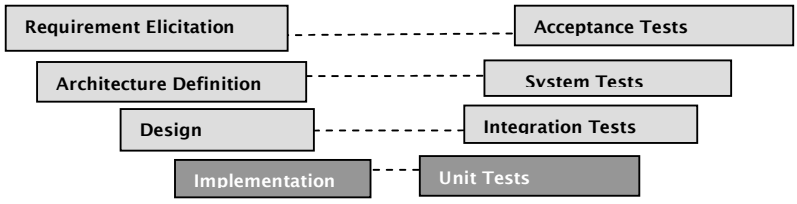

Figure 2. Development and Testing processes correspondence.

Each test level, showed in Figure 2, focuses on a particular class of faults [27]: (i) Acceptance Test aims at finding defects in requirements [27]; (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 operate as defined in their specification. This principle applies no matter what software life cycle is used [27]. These same levels can be used to express a MASs testing process. In this paper we particularly deal with the unit and integration test levels in MASs.

3.1 Object Oriented Testing x Agent Oriented Testing

There are different proposals for representing an agent. Our approach is based in the definition detailed in [29] 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 interact with other agents by sending and receiving messages, and observe/affect the environment. This kind of interaction differs in nature from the direct method call that takes place in OO systems. However, we can make a parallel between the way agents and classes can be unit tested. Figure 3 illustrates white box tests and black box tests of agents and objects.

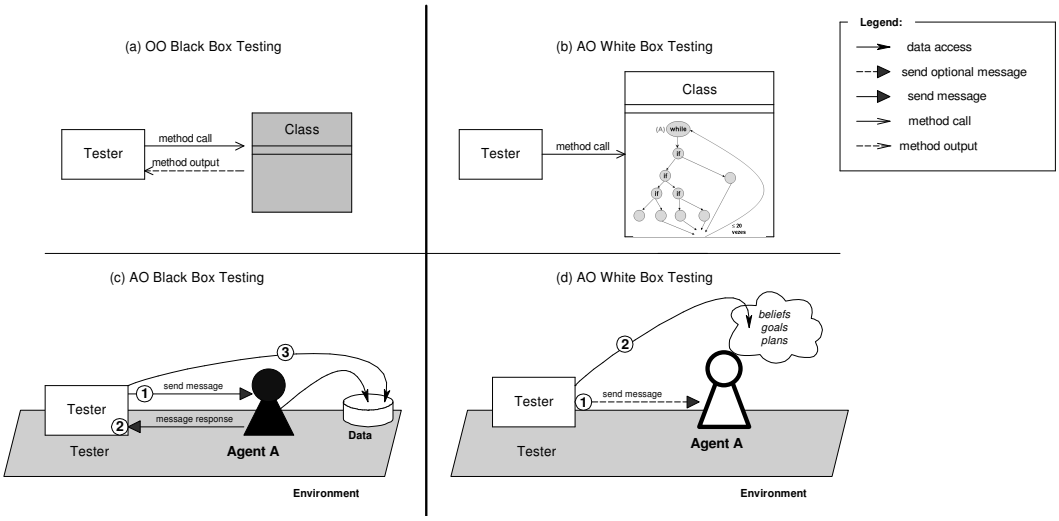

Figure 3. Black Box Unit Testing x Unit White Unit Box Testing

In the black box approach, in order to test a class the developer calls each class method, passing valid and invalid arguments, and checks whether the method output corresponds to the expected output (see Figure 3(a)) [27]. Similarly, a developer can test an agent (see Figure 3(c)) by sending a message to it (step 1), checking whether the message response corresponds to what was expected1 (step 2) and checking whether environment was affected as expected (step 3).

The white box approach needs the tester to look into the code and find out which parts of the code is malfunctioning, even if the method output is correct there can be a bug inside it. The class white box testing analyses the method execution flow to detect bugs. Figure 3(d) depicts a white box testing approach in which an agent receives an input which can be a message or an event from the environment event (clock event) and the testing tool verifies whether the agent performs as expected by accessing the agents beliefs, goals and plans. The white box testing is strongly related with the agent implementation – the way the beliefs, goals and plans are implemented. On the other hand, the black box testing approach is based on the agent interface with the external environment. As a consequence, modifications on the agent implementation that does

1 The message response check comprises a set of assertions: if the agent does not send a response message with a specific content, of a specific type or within a pre-defined amount of time.

not affect its "interface" will not break the tests implemented in the black box approach but will probably break the tests developed according the white box one.
The testing infrastructure described in this paper support a black box testing approach for agents. We have been performing some experiments using a white box testing approach which accesses the agents beliefs using computational reflection and aspect-oriented programming but it is out of the scope of this work.
Knowing that the single agents work well in isolation is not enough. We need to test the interactions between them: we need to progressively integrate agents and select tests to detect possible problems in their communications – integration test level. Different approaches can be used to progressively integrate the agents. To define these integration approaches some concepts can be borrowed from Object Oriented software engineering: (i) top-down; (ii) bottom-up; (iii)Big-Bang. Such integration strategies are based on the dependencies between system elements. In OO context, such dependencies are represented by: which module calls which other. In AO context such dependencies are represented by: to whom an agent provides services or on whom it relies for services.
After incrementally testing subsets of agents, the whole society should be tested. The system test is the final validation step, and is performed once the whole system is developed. System tests can check whether the MAS functional and performance requirements were addressed. It can check for instance whether a specific task that needs agents' cooperation can be solved.

4 Overview of the Approach

In our approach we defined a "mock" version of a system agent to unit test agents. It is called Mock Agent. A Mock Agent interacts with an agent under test. It is responsible for sending messages to it, checking its response and checking whether the environment was affected as expected.
As we mentioned previously, in order to test a component we should be able to control its inputs and observe its outputs. Thus, to test an agent we need: (i) to control the environment and the interaction between the Mock Agents and the agent under test (the test inputs); and (ii) to observe the behavior of the agent under test (test output).
The job of controlling the interaction between the Mock Agents and the agent under test in our approach uses a Synchronizer element. This element is responsible for defining the interaction order between the Mock Agents and the agent under test. A Monitor element is adopted by our approach to do the job of observing the agents' behaviors. Figure 4 depicts all the participants that compose our agent unit test approach:

- **Agent Under Test (AUT):** is the agent whose behavior is to be verified by the execution of unit testing;
- **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;
- **Monitor:** is responsible for monitoring agents' states;
- **Synchronizer:** orchestrates the test scenario. It defines the order in which the Mock Agents are going to send messages to the agent under test;
- **Test Scenario:** define a scenario – a set of conditions – to which the AUT will be exposed, and verifies whether this agent obeys its specification under these conditions. Every scenario comprises only one AUT and one or more Mock Agents;
- **Test Suite:** consists of a set of Test Scenarios and a set of operations performed to prepare the test environment before a Test Scenario starts.

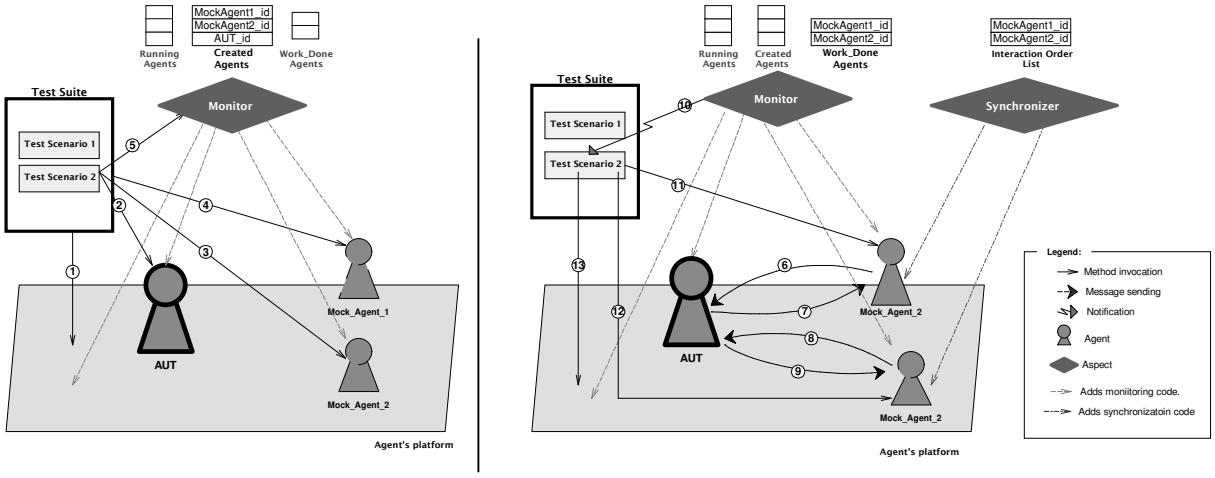

Figure 4. Workflow between the participants of a unit test.

Each agent unit test follows the common structure depicted in Figure 4. In step 1, the Test Suite creates the agent's platform and whatever elements needed to set up the test environment. After that, a Test Scenario is started. Each Test Scenario creates one or more Mock Agents that interact with the AUT (steps 2 and 3) – the number of Mock Agents varies according to the Test Scenario being defined. Next, it creates the AUT (step 4) and asks the Agent Monitor to be notified when the interaction between the AUT and the Mock Agents finishes (step 5).

At this point, the AUT and the Mock Agents start to interact with the AUT. The Mock Agents send messages to the AUT, which then replies or vice-versa (steps 6 to 9). They can repeat steps 6-9 as many times as necessary to perform the test codified in the Mock Agents' plans. For instance, the Mock Agent 1 may reply three messages before finalizing its test activity, and the Mock Agent 2 may reply only one message from AUT before its plan is done. During this interaction process, the Monitor observes the agents interaction and keeps track of changes in their life cycle. In order to do that it uses three lists as illustrated in Figure 4:

- **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;.
- **Running Agents List:** maintains IDs of the agents in the running state;.
- **WorkDone Agents List:** maintains IDs of the Mock Agents that have completed their plan - which is 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 Scenario* that the interaction between the *Mock Agent* and the AUT have concluded (step 10). This notification unblocks the *Test* Scenario which is now able: (i) to ask the *Mock Agents* whether or not the AUT acted as expected (steps 11-12); and (ii) to check whether the environment was affected as expected (step 13). If there was no such notification, the *Test Scenario* would not be able to know when the interactions between the AUT and the Mock Agents had finished – which means the end of the test scenario. As consequence, the *Test Scenario* could make the test final check (final result = expected result?) in an intermediary state, which could result in a false positive or a false negative test status. This is the reason why the Monitor is essential to our approach.

The Synchronizer is an optional element in our approach. It is only used when the test developer needs to establish an order of interaction between the Mock Agents and the AUT. The Synchronizer keeps an *interaction order list* which is loaded at the beginning

of the test scenario. This list contains the Mock Agent id that has the right to interact with the AUT at a specific moment in a test scenario.
In the example illustrated in Figure 4, the Mock Agent 1 must send a message to AUT before the Mock Agent 2. Thus, we can realize that the test scenario is partially implemented by the plan of each mock, and the Synchronizer is the element responsible for composing the testing scenario. We can make a parallel between the Synchronizer element and an Opera Conductor. The same way an opera conductor defines the order in which each instrument takes place on a Symphony. The Synchronizer is responsible for defining the moment at which each Mock Agent should take place in a test scenario.
The Monitor and the Synchronizer elements represent two crosscutting concerns from our approach. Their implementation is necessarily tangled and spread over the code of mock agents and the AUT. In order to prevent the code of these concerns from being tangled with the code of the agents (been monitored and synchronized, respectively), we have implemented these elements in our approach as aspects [14, 26].

4.1 Overcoming Obstacles to Testability

As mentioned in Section 1, to test an agent we should be able to *control* its inputs (i.e messages received from other agents and data read form the environment) and *observe* its outputs (i.e messages sent by the agent and how it affects the environment).
In order to *control* the agents inputs the approach uses: (i) the *Mock Agent* which is responsible for sending messages to the agent under test; and the Synchronizer which defines the order in which the Mock Agents should send messages and the agent under test; and (iii) the test scenario which configurates the environment before the test execution.
In order to *observe* the agents outputs the approach uses: (i) the Monitor element that observes and store information about the agents´ states; and the (ii) Mock Agent which besides sending messages to the agent under test, is also responsible for checking its response and checking whether the environment was affected as expected.
The Monitor and the Synchronizer elements represent two crosscutting concerns from our approach. Their implementation is necessarily tangled and spread over the code of mock agents and the AUT. In order to prevent the code of these concerns from being tangled with the code of the agents (been monitored and synchronized, respectively), we have implemented these elements in our approach as aspects.

4.2 Test Scenarios Design

A very important consideration in program testing is the design of effective test scenarios [27]. Testing, however creative and seemingly complete, cannot guarantee the absence of faults [27]. Test scenario design is important because complete testing is almost impossible; a test of any non trivial program is usually 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 scenarios has the chance of detecting the largest number of faults?
In general, the least effective method of all is to arbitrarily choose a set of test scenarios. In terms of the likelihood of detecting most of the faults, an arbitrarily selected collection of test scenarios has little chance of being an optimal or even close to optimal subset. Test approaches for MASs proposed so far do not define a method for test-scenario selection. This approach adopts an error-guessing test case design technique [27]. The basic idea of an error-guessing technique is to enumerate a list of scenarios in which exceptional conditions may occur and then write tests based on the list.

Thus, for each agent of a MAS[2] we should define a set of exceptional scenarios in which the agent is responsible for performing a specific task. This agent will be the AUT of the test scenario and all the other agents that interact with it in this scenario will be represented as Mock Agents. We can also have exceptional scenarios in which two or more real agents (AUTs) interacts with a set of mock agents. These scenarios represent an integration test between two or more agents. Table 1 contains a template for a test scenario description in this testing approach. Next section, gives an example of how this template is used to define test scenarios for the Expert Committee system (Section 3.1).

Table 2. Template for a test scenario.

Agent		*<The Agent Under Test>*
Scenario 1	**Input**	*<Describes the test scenario input which comprises on the messages that should be sent to AUT in this scenario and the state of data repositories or other environmental resource in the test scenario>*
	Output	*<Describes the expected behaviors for the AUT – whether it should send an specific message or should affect an specific resource from the environment>*

5 JAT Design and Implementation

Figure 5 depicts the main elements of JAT (implemented on top of JADE as mentioned earlier). We discuss these elements next.

JADEMockAgent

The JADEMockAgent implements the Mock Agent concept in JAT. The JADE-MockAgent, as any other agent in this platform, extends the Agent class (see Figure 5). The JADEMockAgent plan (represented by a JADE Behavior) is analogous to a test script, since it defines the messages that should be JADEMockAgent needs to report the test result (success or failure) to the Test Scenario, that is in charge of examining the test result (see JADETestCase description below). There are many ways of reporting the result of an agent interaction test. For example, one may include the test result in an ACL (Agent Communication Language) message and send it to another agent that would generate a test report. Another alternative is to define an interface that contains a set of methods that should be implemented by an agent that wants to report the result of the interaction with the AUT. In our particular implementation, we have chosen the second option. Thus, the JADEMockAgent implements the TestReporter interface illustrated in Figure 5.

2 If there are strict costs or time constraints, this technique should be applied only to the subset of MAS agents responsible for the "core" functionalities.

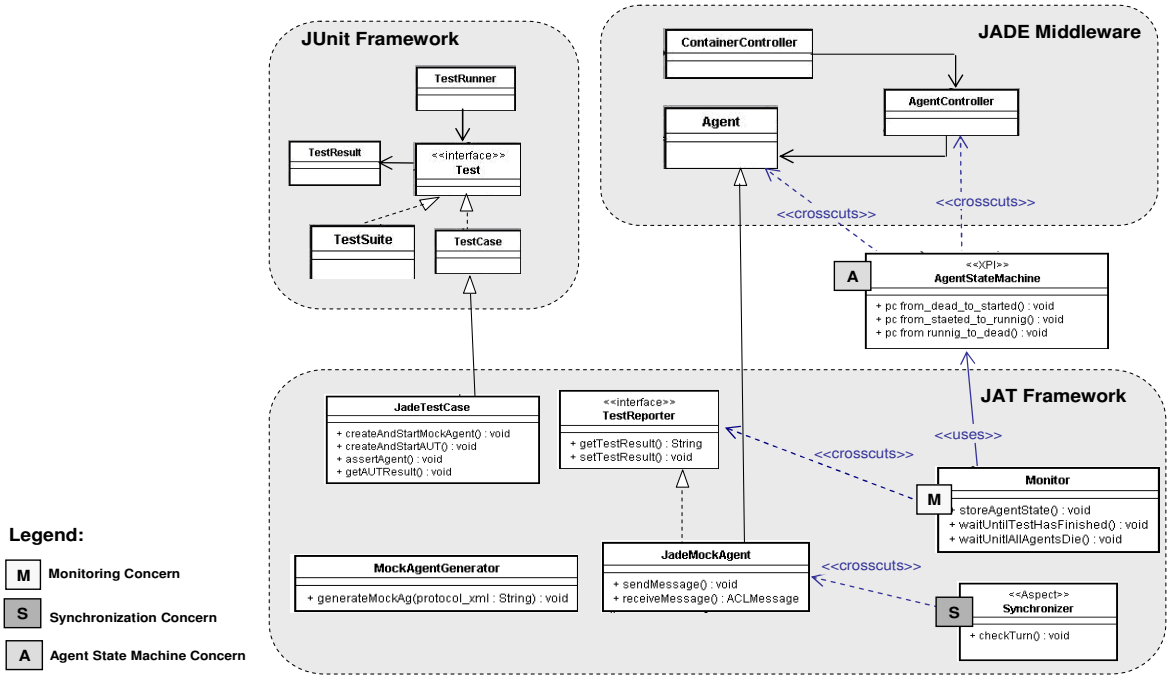

Figure 5. The JAT framework Design.

AgentStateMachine

To keep track of an agent's state transitions we could define an aspect to directly advise the points to be tracked (monitored) [6, 9]. However, whenever the JADE platform evolves our framework would have to be evolved too, since such a monitoring aspect would be strongly coupled to the JADE platform implementation.

Recently, Sullivan et al [31, 16] have proposed a new approach to aspect-oriented design and implementation based on the definition of crosscutting interfaces (XPI) between aspects and classes. An XPI represents a crosscutting abstraction that is not well represented in OO design. Such abstraction is useful for improving the reasoning about how aspects affect the code, and can be perceived as a kind of an *aspectual hot spot* when applied to frameworks [11,21].

The agent state machines themselves, though a valuable concept, are not exposed as explicit design-level abstractions in the JADE platform's OO design. Nor is the case for a design that includes a monitoring aspect that tracks the agents' state transitions. The agent state machine is a crosscutting concept and as such can be represented by an XPI, and then, exposed as an explicit design-level abstraction, in the form of AgentStateMachine XPI, depicted in Figure 5.

As a result of this design decision, the Monitor aspect (detailed next) is not strongly coupled to the JADE platform's implementation details. It accesses the AgentStateMachine XPI, which acts as intermediary between the Monitor aspect and the platform classes. When a new release of the JADE platform is deployed only the AgentStateMachine XPI will need to be updated, making the JAT framework resilient to changes in the underlying platform. The AgentStateMachine XPI can be represented by an aspect [16] in AspectJ[3] language which contains a set of pointcuts, as illustrated in the code snippet below.

3 Please refer to [6] for AspectJ syntax details.

```
public aspect AgentStateMachine {

  pointcut created(): call(public
    ContainerController.createNewAgent(..));

  pointcut from_created_to_running(Agent o):
     execution(protected void Agent.setup()) &&
     target(o);

  pointcut from_running_to_dead (Agent o):
     execution(public void Agent.doDelete()) &&
     target(o);

   pointcut from_runnig_to_mockagworkdone(Agent o):
       ( execution( public void
         org.jadeunit.TestReporter.setTestResult(..))
         && target(o);
 }
```

Listing 1. AgentStateMachine XPI.

Monitor

The Monitor aspect reuses the pointcuts defined on the AgentStateMachine XPI in order to monitor agents' state transactions. It keeps track of the agents' current states in a set of internal data structures, and makes this information available to other framework components, such as the JADEMockAgents and the JADETestScenario, during test execution. The agents' state information is available through a set of accessor methods and some blocking methods such as waitUntilTestFinishes() that are detailed in Section 6.

```
privileged public aspect Monitor {

  private StateLists stLists = StateLists.getInstance();
  ...
  before(Agent o):
  AgentStateMachine.from_created_to_running(o){
        String nome = o.getLocalName();
        stLists.includeInRunningAgent(nome);
  }
  ...
  public void waitUntilTestHasFinished (String mockAgentID){
     ...
     while (!stList.isWorkDone(mockAgentID) &&
           (MAXTIMEOUT > timeSpent )){
        wait(20000);
        ...
    }
  }
}
```

Listing 2. AgentMonitor partial code.

Synchronizer

The Synchronizer aspect intercepts the code of the JADEMockAgent class responsible for sending messages and appends additional behavior before the send message code, which makes the mock agent check whether it is its turn to send a message to AUT. The Synchronizer element was implemented using AspectJ language [27]. The partial code of the Synchronizer is illustrated below.

```
1. public aspect Synchronizer {
2.
3.    pointcut MockSendMessage(...):
4.      call(... org.pucjadeunit.JADEMockAgent.send(..))...;
5.
6.    before(...) : MockSendMessage(agent,message) {
7.      OrderList orderList = OrderList.getInstance();
8.      //The agent only sends the message if it is
9.      //his turn otherwise he sleeps.
10.     while ( !orderList.checkTurn(agent.getAID()) {
11.               ...
12.               Thread.sleep(500);
13.          ...
14.     }
15.  }
16.}
```

Listing 3. Synchornizer partial code.

The `Synchronizer` aspect intercepts the code of the `JADEMockAgent` responsible for sending messages (lines 3-4). It adds an extra piece of code before the send message code (lines 6-15), which makes the mock agent checks whether it is its turn to send a message to AUT. The `OrderList` contains the ids of the mock agents that should send a message to AUT in a specific test scenario, ordered by the interaction priority (line 7). Thus, if the `orderList.checkTurn()` returns true it means that the agent can send a message to AUT, otherwise the mock agent is going to sleep and a few seconds latter it wakes up to check whether its turn had arrived (lines 10-12).

JADETestCase

As depicted in Figure 5, instead of creating a testing tool from scratch to build and execute the JADE agent test scenarios, we extend the JUnit framework [14] to support JADE agent tests. This should lower the developers' learning curve if they are already familiar with JUnit. Hence, JADETestCase consists of a set of Test Scenarios (JUnit test methods) and a set of operations to prepare the test environment before a Test Scenario starts.

5.1 Mock Agents Library

Since the implementation of mock objects during class tests can be a costly task, libraries of mock objects for commonly used components [20] have been developed to support OO testing (e.g. mocks for Web Servers and Java.io classes). In our approach we have been using interaction protocols specifications as a source of information to enable the implementation of a library of Mock Agents.

Often the conversation between agents follows typical patterns, in which a certain sequence of messages is expected, at each point in the conversation. These patterns of message exchange are called *interaction protocols*. The Foundation for Intelligent Physi-

cal Agents (FIPA) has catalogued a set of Interaction Protocols (IPs)[4] that are used in various scenarios of MASs. Examples of the IPs available in FIPA library are: the contract-net protocol – which are often followed between a buyer agent and a seller agent of a product; the request interaction protocol; and the query interaction protocol. Each protocol illustrates the roles that take part on a conversation and the sequence of messages exchanged between these roles. We have implemented mock objects for the roles that takes part on some of the FIPA compliant interaction protocols.

5.2 Generating the Mock Agents Code

A mock agent is a simple implementation of a real agent, which follows a deterministic behavior and expects a set of messages to be received from the AUT. Each mock agent obeys a similar structure: (i) it is a JADE agent that contains one single behavior; and (ii) inside this behavior the mock agent receives and send messages according to a specific protocol (application specific or FIPA compliant protocol).

To reduce the cost of generating a set of mock agents per test scenario, we defined a generation template-based approach for mock agents which generates the code of a mock from a partial[5] protocol specification defined in an XML file. The `MockAgentGenerator` class illustrated in Figure 4 is responsible for reading this XML file and generating a `JADEMockAgent`. Next section illustrates one example of a mock agent generation.

5.3 JAT-Based Testing Approach Workflow

The workflow illustrated in Figure 6, shows each activity performed in the agent testing approach proposed here. During the test case design the developer refers to the system documentation, the interaction protocols specification and the test design technique defined in Section 3.2, in order to specify a set of test cases. During the JADE test case implementation, the developer implements the JADE TestCases, and TestSuites and the Mock Agents specified in the test case design tables – some Mock Agents can also be reused from the library. After implementing these testing components the test of JADE agents are executed in JAT framework. After the test execution, the framework can generate any one of the Test Result reports available in JUnit framework.

4 FIPA compliant interaction protocols are defined in the context of a conversation. By their nature, agents can engage in multiple dialogues, simultaneously. The term *conversation* is used to denote any particular instance of such a dialogue. Thus, the agent may be concurrently engaged in multiple conversations, with different agents, within different IPs.

5 The protocol specification is partial because it illustrates the messages that should be sent by only one participant of a protocol.

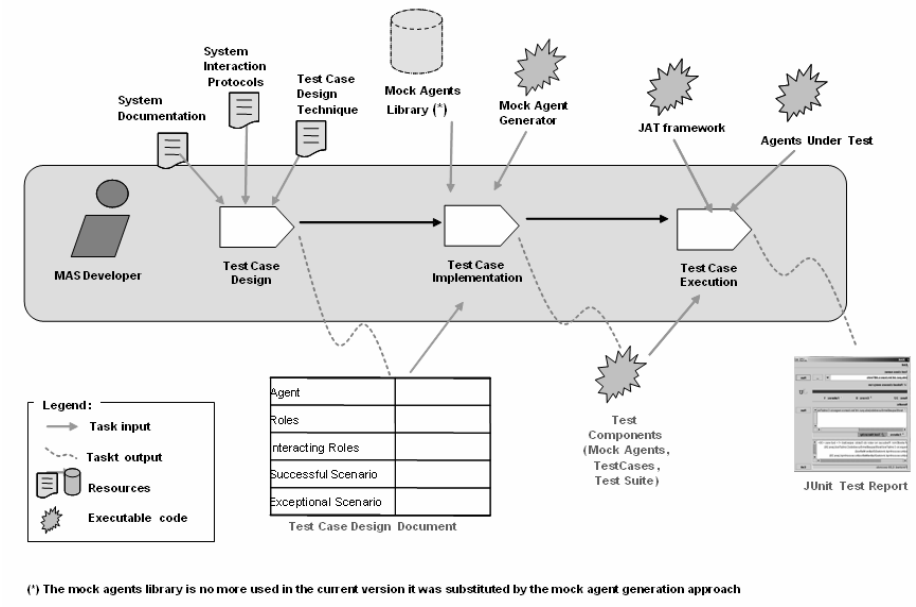

Figure 6. JAT Workflow.

6 Case Studies

Our case studies include: a Book Trading System, which is available with the JADE Platform distribution [1]; the Expert Committee System [32] a MAS system for paper submission, distribution and notification which is a benchmark for agents technologies; and the Auction System which was developed by one of the authors in a Test Driven approach that used JAT as a design and test supporting tool [10].

6.1 Book trading System

To illustrate our agent 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 7 (B) details the interaction protocol between these roles. We can realize that the BookBuyer agent and the BookSeller agent follow the contract-net FIPA compliant protocol illustrated in Figure 7 (A).

According Figure 7 (B), 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-execute its plan and try to buy the book again from some other agent.

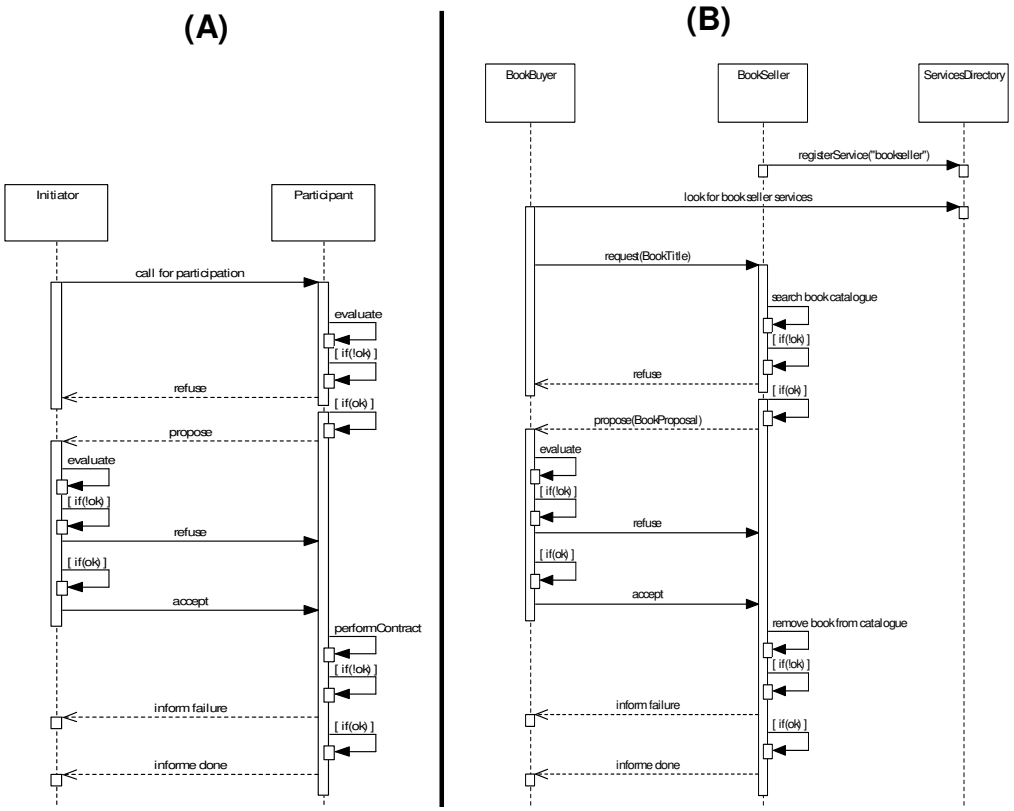

Figure 7. Sequence diagrams of (A) FIPA contract-net interaction protocol and (B) the Book Trading system.

The first step is to specify the test scenario to be automated. At this step, we adopt an error-guessing test case design technique [26], how was described in section 4.2, to support the definition of test scenariosWe pick one of the test scenarios (see Table 3). The agent(s) that should deal with the exceptional condition will be the AUT(s) of the test scenario, and all the other agents that interact with it/them in this scenario (causing the exceptional condition) will be represented as Mock Agents. Table 3 contains a description of an exceptional scenario to be automatically tested using JAT.

Table 3. Example of a Test Scenario

Agent Under Test	BookSeller agent
Test Scenario	Two BookBuyer agents try to buy the same book from the BookSeller, but it has only one copy available.
Expected Result	The BookBuyer should sell the book to the first agent that requests the book and reject the buying request from the other.

To implement a Mock Agent for the BookBuyer agent - that will be responsible for testing the BookSeller agent in this scenario - the developer must define the communication protocol in an XML file, and the MockAgentGenerator generates the code of the Mock Agent as illustrated in Listing 3. In this example the shaded code was included by the developer after the code generation.

```
public class BookBuyerMockAgent extends JADEMockAgent{
  ...
  public class MockAgentBehaviour extends OneShotBehaviour{
    public void action() {

      try {
        sendMessage(ACLMessage.CFP,seller,targetBookTitle);

        ACLMessage msg = blockReceiveMessage(30000,ACLMessage.PROPOSE);

        checkFloatValue(msg.getContent);

        sendMessage(ACLMessage.ACCEPT_PROPOSAL,seller,targetBookTitle);

        msg = blockReceiveMessage(30000,ACLMessage.INFORM,ACLMessage.FAILURE);

        if ( order == FIRST &&
          msg.getPerformative == ACLMessage.INFORM){
          setTestResult(“OK”);
        else if ( order == SECOND &&
          msg.getPerformative == ACLMessage.FAILURE) {
          setTestResult(”OK”);
        } else {
          setTestResult(“ERR:Protocol Broken.”);
        } catch (Exception e) {
          setTestResult("ERR: "+ e);
        }
      }
  }
```

Listing 4. BookBuyer Mock Agent partial code.

The next step is to compose the test scenario in which two instances of the BookBuyerMockAgent depicted above are going to interact with the BookSeller agent (AUT). The code in Listing 5 illustrates the JADETestCase that implements this test scenario.

```
1. public class BookTradingTesteCase extends JADETestCase {
2.
3.
4.   public void testBookTradingScena1(){
5.
6.    //Load Data Repositories
7.          …
8.    startAUT("seller","BookSeller");
9.    startMockAgent("buyer1","BookBuyerMockAgent",FIRST);
10.   startMockAgent("buyer2","BookBuyerMockAgent",SECOND);
11.
12.   //The Test Scenario block until the mock
13.   //agents finishes their interaction with the AUT.
14.   Monitor.aspectOf().waitUntilTestHasFinished("buyer1");
15.   Monitor.aspectOf().waitUntilTestHasFinished("buyer2");
16.
17.   //1. Checking Expected Result:
18.   assertInteractionOK("buyer1");
19.   assertInteractionOK("buyer2");
20.
21.   //2.Check Agent’s Beliefs
22.   Object belief = getBelief(“seller”, “catalogue”);
      ...
23.   assertEquals(expectedBelief,belief);
23.  }
25.  ...
26. }
```

Listing 5. A JADETestCase partial code.

According to this partial code, firstly, the test harness is prepared (line 6), which comprises the load of the test data necessary for the test scenario. Then, the BookSeller agent as well as the Mock Agents are created through methods available in the JAT framework (lines 8-10). The method waitUntilTestHasFinished (lines 14-15) blocks the test scenario execution until the Monitor detects that the plan of each mock agent has finished. When the test scenario is unblocked, it is time to check the test scenario results against the expected ones (lines 24-30). The getBelief() method (line 22) gives a white-box flavor to the mock-agent-based black-box tests developed in JAT. This method access through reflection information embedded in the agent under test, which can be useful to the test scenario. The returned belief is then compared to its expected value (line 23).

Since the order in which the BookBuyer mock agents interact with the AUT is relevant to build this scenario, the test developer needs to build a synchronization file. This file is loaded by the Synchronizer aspect mentioned before the execution the test scenario. The Synchronizer also appends additional code before the send message code of each mock agent on this scenario, which makes the mock agents check whether it is turn to send a message to AUT. Since the synchronization code is included on an aspect it is not necessary to add extra code inside the mock agents in order to synchronize them. As a consequence, the same mock agent can be used in a non-synchronized and on a synchronized testing scenario without the necessity of changing the code of the mock agent. The code bellow illustrates how should be an order list to synchronize the interaction between the mock reviewers and the chair.

```
<?xml version="1.0" encoding="UTF-8"?>
<JAT>
  <interactioOrder>
    <agentturn>buyer2</agentturn>
    <agentturn>buyer1</agentturn>
    <agentturn>buyer1</agentturn>
    <agentturn>buyer2</agentturn>
  </interactioOrder>
</JAT>
```

Listing 6: A synchronization file.

6.2 The Expert Committee System

Consider a MAS that supports the management of paper reviewing processes for a conference, from herein referred to as Expert Committee [18, 20]. The Expert Committee system encompasses user agents that are software assistants to represent system users in submission, reviewing and chairing activities. The basic functionality of the user agents is to infer and keep information about the system users as well as collaborate with other user agents in order to perform the main functionalities of the Expert Committee system. Each user agent can play one of the following roles:

- **Author**: it is responsible for receiving a paper submission request from an author user and submitting the paper to a specific event. Its beliefs include information about the author (e.g. author's institution);
- **Chair**: its responsibilities include distributing papers among reviewers, contacting the Coordinator asking for new reviewers when the number of reviewers for a specific paper is not enough, and notifying the authors about the paper acceptance and rejection. The chair beliefs comprise information about the conference, including deadlines, the list of reviewers, and the list of submitted papers;
- **Reviewe**r: it judges the chair proposals. A reviewer can accept or reject a review proposal and once it accepts to review a paper it should send a paper evaluation to the chair before an evaluation deadline. Its beliefs include information about the reviewer (e.g. reviewer's institution and research interests);

• **Coordinator**: it is in charge of inviting additional reviewers to help in the revision of specific papers, in case the chair notifies the Coordinator that there are not enough reviewers to a specific paper.

Figure 8 illustrates the main interactions between the Expert Committee agents' roles as well as the information repositories that can be accessed by each role.

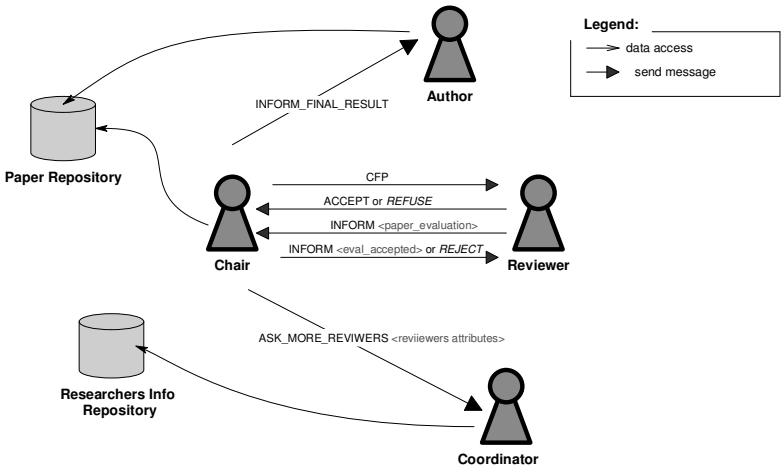

Figure 8. The Expert Committee System.

The first step is to define a set of scenarios related to each agent role of this system. We are going to define these scenarios to the chair role, which is the one responsible for performing most of the tasks related to the Expert Committee system. Table 4 illustrates a set of scenarios related to chairs responsibilities. The words written on italic represent the agents roles that interact with the chair in the scenario.

Table 4. Test scenarios related to Chair's responsibilities.

Agent		Chair
Scenario 1	**Input**	The *reviewer* does not accept to review the paper, but after that it sends the paper evaluation to the chair.
	Output	The char should reject the paper evaluation
Scenario 2	**Input**	All the *reviewers* for an specific paper are of the same institution of the author.
	Output	The chair should ask the coordinator for more reviewers
Scenario 3	**Input**	There is nor *reviewer* to review the paper.
	Output	The same as Scenario 2.
Scenario 4	**Input**	The only available *reviewer* for a specific paper is its author.
	Output	The same as Scenario 2.
Scenario 5	**Input**	Only two *reviewers* evaluate the article before the revision deadline.
	Output	The same as Scenario 2.

After defining test scenarios related to some of the chair's responsibilities. The next step according this approach is to define automatic tests scenarios. We are going to define a test scenario to automatically check the Scenario 1 - the other scenarios follows the same structure.

In this scenario a reviewer should: (i) reject the paper review proposal, and then (ii) send a message to the chair agent with a paper evaluation. In a correct implementation

of the Expert Committee system, the chair agent should refuse the paper evaluation sent by this reviewer. To implement the reviewer mock agent we define a XML file which contains the messages the reviewer mock agent should send. This XML file is illustrated bellow.

```
<?xml version="1.0" encoding="UTF-8"?>
 <MockAgent package="test" name="ReviewerMockAgent">
  <protocol>
     <receive>
            <performative>ACLMessage.CFP</performative>
            <time>30000</time>
     </receive>
     <send>
            <performative>ACLMessage.REJECT_PROPOSAL</performative>
            <to>chair</to>
            <content>I reject</content>
     </send>
     <send>
            <performative>ACLMessage.INFORM</performative>
            <to>chair</to>
            <content></content>
     </send>
     <receive>
            <performative>ACLMessage.REFUSE</performative>
            <time>30000</time>
     </receive>
  </protocol>
</MockAgent>
```

Listing 7. A JADETestCase partial code.

Based on this XML file, the `MockAgentGenerator` element generates the code for the mock agent which can be modified and extended by the test developer. In this example the shadowed code was included by the developer after the generation.

```
public class ReviewerMockAgent extends JADEMockAgent{
  …
 public class MockAgentBehaviour extends OneShotBehaviour
 {
    public void action() {
       try {
            CLMessage msg = receiveMessage(30000,ACLMessage.CFP);
            //Rejecting the paper review
            sendMessage(ACLMessage.REJECT_PROPOSAL,msg.getSender(),"I reject");

            //Sending the paper evaluation
            sendMessage(ACLMessage.INFORM,chair,paperRevision);

            //The chair expected behavior
            msg = receiveMessage(30000,ACLMessage.REFUSE);

        } catch (ReplyReceptionFailed e) {
             setTestResult("ERROR: "+ e);
        }
     }
}
```

Listing 8. A JADETestCase partial code.

After implementing the Mock Agent(s), the test developer needs to compose the test scenario in which the one or more Mock Agent reviewers are going to interact with the chair (AUT). The code bellow illustrates the `JADETestCase` that implements this test scenario.

```
1. public class ExpertCommitteeTestCase
2.                     extends JADETestCase {
3.
4. //Test method for Scenario1
5.  public void testRejectReviewAndPaperAcceptd() {
6.
7.     //Load Data Repositories
8.          …
9.      startMockAgent("reviewer1","test.MockAgentReviewer");
10.
11.     startMockAgent("reviewer2","test.MockAgentReviewer");
12.
13.     startAUT("chair","agent.Chair", conference);
14.
15.
16.
17.     //These methods make the Test Scenario block until
18.     //the mock agents finishes its interaction with
19.     //the AUT.
20.     Monitor.aspectOf().waitUntilTestHasFinished("reviewer1");
21.     Monitor.aspectOf().waitUntilTestHasFinished("reviewer2");
22.     //Checking Expected Result:
23.     //1. Mock Agents Final Interaction Result
24.     assertInteractionOK("reviewer1");
25.     assertInteractionOK("reviewer2");
27.
28.     //2. Status of Data Repositories
29.     Paper paper = PapersRepository.get(paperTitle);
30.     assertFalse(paper.getStatus(), Paper.ACCEPTED);
31.  }
32.   …
33.}
```

Listing 9. A JADETestCase partial code.

According to this partial code, firstly the test harness is prepared (on line 8) - which comprises the load of the test data necessary to the test scenario. Then, the Mock Agents as well as the chair agent are created through methods available JAT framework (lines 9-15). Since the order in which the reviewers interact with the AUT is not relevant to build the test scenario, the test developer does not need to build a synchronization file to this test scenario.

The method `waitUntilTestHasFinished()` (lines 20-21) blocks the test scenario execution until the `Monitor` detects that the plan of each mock agent has finished[6]. When the test scenario is unblocked, it is time to check the test scenario results against the expected ones (lines 24-30).

Since JAT framework is an extension of JUnit framework, we use the same graphical user interface (GUI) of JUnit framework to execute the tests and show the test report (see Figure 9).

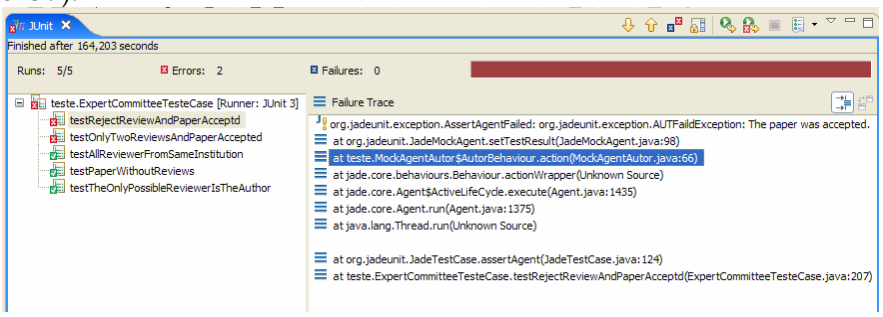

Figure 9. Test Report generated after the execution of agent tests.

6 We used the `aspectOf()` static method (lines 20 and 21) available in all AspectJ aspects. The `aspectOf()` method returns the singleton instance of an aspect, which can be used by any class to call the public methods from an aspect.

We have defined automatic tests to many scenarios existing in the Expert Committee system, including those ones described in each of the five scenarios described in Table 4. During the execution of the Expert Committee tests, we found many errors, such as: (i) the chair agent accepted the evaluation sent by a reviewer that had rejected to evaluate the article and also accepted the paper evaluated by them; and (ii) the chair finalized the reviewing process of a paper even without receiving the minimum number of paper evaluations for a paper. The right side of JUnit GUI shows the exact place (the line of code) in which the errors were detected.

6.3 Results

We progressively tested the three systems described previously using JAT. Firstly, we defined some unit tests to each agent; based on the error-guessing test case design technique [26], we defined scenarios in which there were one AUT and a set of mock agents. Then, integration tests were implemented comprising more then one AUT and the mock agents that interact with them – here most mock agents were reused from the unit test scenarios. Finally, we implemented some system tests, in which we did not check the interaction between the agents – which were already checked in the previous tests – but we checked whether real agents collaborations addressed the system's requirements (i.e., the test scenario verified whether the environment was effected as expected)[7]. During the execution of these tests we found 21 faults summarized on Table 5.

Table 5, Faults detected when testing Book Trading (BT), Expert Committee (EC), and Auction (AC) systems.

Fault Type	Detected Faults			
	BT	EC	AC	All
On MAS Constraints	**-**	**2**	**-**	**2**
Unimplemented MAS constraint	**-**	**2**	**-**	**2**
On Interaction Protocol	**3**	**6**	**4**	**13**
Mismatching Protocols	1	-	-	1
Incomplete Main Flow	-	2	1	3
Missing Alternative Flows	2	4	3	9
On Internal Procedures	**1**	**1**	**1**	**3**
Boundary values	-	-	1	1
Logic Mistakes	1	1	-	2
TOTAL	**4**	**9**	**5**	**19**

According to Table 5, some faults were related to unimplemented MAS constrains (e.g., there should be at lest 3 reviewers per article), and internal procedures which that did not properly dealt with boundary values or had logic mistakes. However, most of the faults (13 faults) were found on agents' interaction protocol implementations. Some agents did not agree on the protocol to be followed (1 fault) or did not implement the complete main flow of the interaction protocol (3 faults), as a consequence some agents waited forever for a message that would never be sent. Another interesting finding during tests was that most of the agents were not implemented to deal with exceptional conditions - they did not implement exceptional flows. A reason may reside on the fact that the interaction models of multi-agent systems almost always define only the main execution flow and neglect "what should the agent do if an unexpected message is received?". Thus it is up to the developer to decide what should be done, and as a consequence, most often than not, this flow is not adequately imple-

7 A detailed description of the test process adopted the tasks and tasks inputs can be found in [10].

mented. The error-prone test selection technique in combination with the mock agents was a valuable tool to examine how the agents behaved under these exceptional scenarios (implemented in alternative flows), and was responsible for uncovering most of the faults.

7 Evaluation

Fault injection is considered a very useful technique to evaluate the effectiveness of testing approaches. The key idea of the fault injection technique is to produce faults during system execution, and verify whether the testing approach precisely detects the injected fault. Which faults should be injected depends on the fault model associated with a testing approach.
We have implemented a fault injection [33] tool to try to estimate the effectiveness of the test cases developed using the mock-agent-based test approach in JAT. The tool, implemented using Java Annotations and AspectJ, intercepts JADE Agents and introduces the faults specified in our fault model (see Section 3.4).

7.1 The Fault Injector

The fault injector was implemented as an aspect in ApectJ language. It intercepts the JADE agents code and adds the faulty code on it. The aspects intercepts the agent's send() method and adds the fault specified on an annotation, that specifies the specific fault to be injected. We have define one annotation to each fault described in our fault model described before. For instance:

- @changePerformative: changes the performative of the message to be sent by an agent. One attribute of this annotation is the new message performative. This faults simulates to change the order in which the message was sent, in systems on which the order of the message is specified by its performative. @changeContent: replaces the message content by na unexpected one. The attribute of this annotation is a string that comprises the new message content;
- @sleep: blocks the message to be sent during an specified time. The time is specified in milliseconds, is the annotation attribute.

The Listing 10, illustrates the implementation of each one of these annotations.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface ChangeContent {
       String content();

}

@Retention(RetentionPolicy.RUNTIME)
public @interface ChangePerformative {
       int performative() default 0;

}

@Retention(RetentionPolicy.RUNTIME)
public @interface Sleep {
       int time();

}
```

Listing 10. Definição das anotações criadas.

In order to inject one of the faults of the fault model on agents code, the developer only needs to include annotations (and specify values to its attributes) on its meth-

ods. Listing 11 illustrates the partial code of three methods from three different agents in which faults were included using the annotations specified above.

```
@ChangePerformative(performative = ACLMessage.AGREE)
public void action() {

      ACLMessage msg = myAgent.receive(mt);
      if (msg != null) {
          String title = msg.getContent();
          ACLMessage reply = msg.createReply();
      ...
      send(msg);
}


@Sleep(time = 3000)
public void action() {

      ACLMessage msg = myAgent.receive(mt);
      if (msg != null) {
          String title = msg.getContent();
          ACLMessage reply = msg.createReply();
      ...
      send(msg);
}

@ChangeContent(content = "JAT")
public void action() {

      ACLMessage msg = myAgent.receive(mt);
      if (msg != null) {
          String title = msg.getContent();
          ACLMessage reply = msg.createReply();
      ...
      send(msg);
}
```

Listing 11. A JADETestCase partial code.

Listing 12 illustrates the partial code of the Fault Injector Aspect which uses the information of Java annotations, detailed above, in order to inject faults on agents methods.

```
1. privileged aspect FaultInjection {
2.
3.      pointcut sendMessage(Object b) : call(* Agent.send(..)) && args(b);
4.      pointcut annotedWithChangePerformative(ChangePerformative cp) :
5.                                          @withincode(cp);
6.
7.      pointcut annotedWithSleep(Sleep s) : @withincode(s);
8.
9.      pointcut annotedWithChangContent(ChangeContent con) : @withincode(con);
10.
11.     ...
12.
13.     void around(Object b,ChangeContent con) :
14.                           sendMessage(b) && annotedWithChangContent(con) {
15.
16.         String content = con.content();
17.
18.         ACLMessage message = (ACLMessage)b;
19.         message.setContent(content);
20.
21.         proceed(b,con);
22.     }
23. }
```

Listing 12. Fault Injector Aspect.

This aspect defines the pointcut sendMessage() which intercepts every message sent by a JADE agent. This pointcut is used in combination with the other pointcuts which

details the fault to be injected (lines 4-9). Such pointcuts intercept the execution point in which annotations were defined (e.g., @ChangePerformative, @Sleep and @ChangeContent). This pointcuts are used in combination with sendMessage() pointcut in each advice of the Fault Injector aspect (line 13). This advice intercepts all execution points in which a message is sent - `sendMessage()-` and is annotated with change conted annotation - `annotedWithChangeContent()`. It then replace the content of the message to be sent by the agent (lines 18-19) by a new content specified as the annotation attribute (line 16) and then send the corrupted message (line 21).

7.2 Results

We have injected 83 faults inside all the three systems to check whether the test scenarios were able to diagnose the injected faults. Table 6 summarizes the results of our fault injection experiment.

Table 6: Faults injected and detected by the test scenarios.

Fault Type	Faults Injected/Detected			
	BT	EC	AC	All
On Interaction Protocol	**16/16**	**23/23**	**21/21**	**60/60**
Message Ordering	6/6	10/10	8/8	24/24
Message Timing	6/6	10/10	7/7	23/23
Message Content	4/4	3/3	6/6	13/13
On Beliefs	**4/4**	**1/1**	**2/2**	**7/7**
Corrupt Belief	4/4	1/1	2/2	7/7
On Internal Procedures	**5/5**	**7/2**	**4/4**	**15/10**
Corrupt Return and Parameter values	**2/2**	6/2	**2/2**	8/4
Logic Mistakes	3/3	1/0	3/3	7/6
TOTAL	**24/25**	**31/26**	**27/27**	**83/78**

Every interaction protocol faults were precisely detected by one of the developed test scenarios. This percentage assures the quality of the test scenarios implemented. The reason for such results can be twofold. Firstly, because the test scenarios have been developed since Jan. 2006, and since then have been continuously improved and maintained after each development iteration. Secondly, because the developers have been acquiring expertise in developing agent tests (also since Jan. 2006) and, as a consequence, the new test scenarios have become more effective. Some internal procedure faults (5 faults) were not detected by the test cases; some of them (2 faults) because they were not exercised by the interaction between agents, they were only used by the other system's elements - such as graphical interface components; others (2 faults) because there was no test scenario that exercised the interaction in which the operation was exercised; and one because the injected fault generated an equivalent mutant. Concerning the agent's beliefs, every time a belief was corrupted it was detected by the test scenario. This can be explained due to the possibility of using of the `getBelief()` construction – based on Java Reflection - which improves the beliefs' visibility during tests.

8 Lessons Learned

This section provides further discussion of issues and lessons we have learned while using JAT to test the previously described systems, and injecting agent-specific faults to asses the test scenarios developed in these systems.

Mock Agents Library. The implementation of mock agents can be a costly task. Our first approach relied on the creation of a library of mock agents – similar to the libraries of mock objects which have been developed to support OO testing (e.g., mocks for Web

Servers and Java.io classes) [23]. But, unfortunately, our implementation of a mock library did not succeed as, mostly, the Mock Agents were too specific to test scenarios and could hardly be reused. After this frustrating initiative we came up with the idea of generating mocks from a protocol specification which turned out to be a valuable solution to reduce the cost of mock agent creation, reducing drastically the time spent to create a test scenario.

The Synchronization crosscutting concern. The design decision of representing the synchronization concern as an aspect enabled the same mock agent to be used in a non-synchronized and in a synchronized test scenario without the necessity of changing the mock agent's code.

Framework Maintainability. The design decision of representing the Agent State Machine crosscutting abstraction as an XPI improved JAT's maintainability. When the JADE platform version evolves we do not need to update all the frameworks aspects that intercepted it, but only the XPI which acts as a "glue" between the JADE framework and the JAT framework.

MAS Tester Expertise. The effectiveness of the test activity depends upon the test developers' ability to define error prone scenarios. Testing, like design, is a creative task which depends on developers' experience and expertise. Such expertise is particularly necessary, when testing MAS, where we have different kinds of agents and several possibilities of message-based interaction scenarios.

Complementing JAT Test Scenarios. Our preliminary results on fault injection justifies the need of unit testing the agents' internal procedures that are not exercised during agents' interaction and that are only exercised by other system elements (e.g., the system graphical interface components). Such procedures can be tested using conventional OO unit testing using JUnit for example. Since JADETestCase class is an extension of JUnit TestCase class, it recognizes and executes conventional test methods. As a consequence the mock-agent based test scenarios and the conventional OO tests can run jointly and continuously.

Test Case Generation. The current approach for test generation could be improved, through the leverage of an existing test tool such as Parasoft's JTest [28] – one of the most used test input generation tools. JTest generates test inputs in the form of JUnit tests. It has been extended to generate test inputs for different development paradigms, such as the aspect-oriented paradigm [37]. We could also leverage this tool to generate JADETestCases besides JUnit test cases.

Improving Test Coverage. Sometimes the agents' behaviors may not be sufficiently exercised by the test scenarios initially developed. Works such as [37] have used coverage results not only to give feedback to developers on what parts of code are to be exercised but also to give further guidance to developers on how to improve the coverage. We are currently investigating the use of coverage information as a feedback to improve effectiveness of the test scenarios defined using this framework.

9 Conclusions and Future Work

We have presented JAT, a framework for building and running tests for Multi Agent Systems implemented in JADE platform. The test scenarios in JAT are based on the use of Mock Agents and aspect-oriented techniques to monitor and control the execution of the asynchronous agents during tests.

JAT has been used to test three multi-agent systems from different sources. In these experiments, JAT was used as a generic testing framework for JADE developers, same as the JUnit framework is for Java developers. We have used JAT to build and execute

unit tests, integrations tests and even system tests. We could benefit from a set of automatic tests that could run constantly, improving the confidence in the system and supporting the maintainability tasks. Although JAT automatic tests could not assure the absence of faults in the tested systems, they could guarantee that whenever scenarios were executed during MAS tests, they had executed as expected. Moreover, we also used a fault injection approach to assess the quality of the test scenarios developed to each one of theses systems. Our preliminary results have shown that JAT can effectively uncover bugs related to MASs fault model.

Although JAT framework implementation is based on Java and AspectJ[18], its underlying concepts (Section 3) can be applied to test any MAS system, given that the language the system is implemented in can be extended to support aspect-oriented concepts.

In future work, we plan to improve the generative approach: generating Mock Agents of a higher complexity, and also the Test Scenarios; as well as leveraging existing test generation tools such as Parasoft's JTest – which generate test inputs in the form of a set of JUnit tests – to generate set of test inputs for JADE systems.

10 References

[1]Bellifemine, F., Poggi, A., Rimassa, G., "JADE - A FIPA2000 Compliant Agent Development Environment", Proc. of AAMAS, 2001, pp. 216-217.

[2]Bellifimine, F., Caire, G., Poggi, A., and Rimassa, G., "JADE– A White Paper", EXP in Search of Innovation, 3(3), 2003, pp. 6–19.

[3]Bernon, C., Massimo, C., Pavón, J., "An Overview of Current Trends in European AOSE Research", Informatica (Slovenia) 29(4): 379-390 (2005).

[4]Binder, R., Testing Object-Oriented Systems. Models, Patterns, and Tools, Addison-Wesley, 1999.

[5]Botıa, J., Hernansaez, J., and Gomez-Skarmeta A., "Towards an approach for debugging MAS through the analysis of ACL messages", Computer Systems Science and Engineering, 20, 2005.

[6]Briand, L., Labiche, Y., Leduc, J., "Tracing Distributed Systems Executions Using AspectJ". Proc.of ICSM 2005.

[7]Caire, G., Cossentino, M., Negri, A., Poggi, A., Turci, P., "Multi-agent systems implementation and testing", Proc. of 4th International Symposium - From Agent Theory to Agent Implementation, 2004.

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

[9]Coelho, R, Dantas, A., Kulesza, U.; Staa, A.v.; Cirne, W.; Lucena, C., "The Application Monitor Aspect Pattern", Proc. of PLoP 2006, 2006.

[10]Coelho, R. et al, "The JAT Testing Framework", Technical Report, PUC-Rio, Brazil, 2007 (available at: www.puc-rio.br/~roberta/reports.html).

[11]Coelho, R., Alves, V., Kulesza, U., Neto, A., Garcia, A. A. v. Staa, C. Lucena, P. Borba, "On Testing Crosscutting Features using Extension Join Points". 3rd Workshop on Product Line Testing (SPLiT'2006), SPLC'2006, 2006.

[12]Coelho, R., Kulesza, U., Staa, A.V., Lucena, C., "Unit Testing in Multi-agent Systems using Mock Agents and Aspects". Proc. of Workshop of Software Engineering for Large-Scale Multi-Agent Systems,ICSE 2006, 2006,p.83-90.

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

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

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

[16]Griswold, W.G. Shonle, M. Sullivan, K. Song, Y. Tewari, N. Cai, Y. Rajan, H., "Modular Software Design with Crosscutting Interfaces", IEEE Software, Special Issue on Aspect-Oriented Programming, 2006.

[17]Jennings, N. R. and Wooldridge, M., Agent Technology: Foundations, Applications, and Markets, Springer, 1998.

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

[19]Kiczales, G.,Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J., "Aspect-Oriented Programming", Proc. of ECOOP, 1997.

[20]Knublauch, H., "Extreme programming of multi-agent systems". Proc. of AAMAS, 2002, pp. 704 – 711.

[21]Kulesza, U., Coelho, R., Alves, V., Garcia, A., Staa, A., Lucena, C., Borba, P., "Implementing Framework Crosscutting Extensions with EJPs and AspectJ", Proc. of ACM SIGSoft XX Brazilian Symposium on Software Engineering, 2006.

[22]Kung, D., Bhambhani H., Nwokoro S., Okasha W., Kambalakatta R., Sankuratri P., "Lessons Learned from Software Engineering Multi-Agent Systems", Proc. of COMPSAC 2003.

[23]Mackinnon, T.,Freeman, S.,and Craig, P., "EndoTesting Unit Testing with Mock Objects", Proc. of XP2000, 2000. [24]Maximilien, E., Williams. L., "Assessing Test-Driven Development at IBM", Proc. of ICSE'2003, pp. 564-569.

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

[26]Meyer, B., Object-oriented software construction, Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1997.[2]

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

[28]Parasoft: http://www.parasoft.com [29]

[29]Poutakidis, D., Padgham, L., Michael, W., "Debugging multi-agent systems using design artifacts the case of interaction protocols", Proc. of AAMAS 2002, pp. 960-967.

[30]Silva, V.; Choren, R.; Lucena, C., "A UML based approach for modeling and implementing multi-agent systems". In AAMAS 2004, pp.914-921.[35]

[31]Sullivan, K.,Griswold, W., Song, Y., Cai, Y.,Shonle, M., Tewari, N., Rajan, H., "Information hiding interfaces for aspect-oriented design",ESEC/SIGSOFT FSE,2005, pp.166-175.

[32]The Expert Committee System: http://www.tec comm.les.inf.pucrio.br/SoCAgents/CI/expertcommittee.htm

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

[34]Voas, J., and Miller, K.. Software Testability: The New Verication. IEEE Software, 12 3:1728, 1995.

[35]Wooldridge, M., and N. Jennings, "Pitfalls of agentoriented development". Proc. of the 2nd AAMAS, 1998, pp. 385–391.

[36]Xie, T., Marinov, D. and Notkin D., "Rostra: A framework for detecting redundant object-oriented unit tests", In Proc. 19th ASE, 2004, p.196–205, 2004.

[37]Xie, T., Zhao, J., "A framework and tool supports for generating test inputs of AspectJ programs", Proc. of AOSD 2006, pp. 190-201.

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

[39] Garcia, A., Lucena, C., Taming Heterogeneous Agent Architectures with Aspects. Communications of the ACM, 2007. (to appear)

[40] Lobato, C., Garcia, A., Lucena, C., Romanovsky, A. A Modular Implementation Framework for Code Mobility. Proceedings of the 3rd IEE Mobility Conference 2006, October 2006, Bangkok, Thailand.

[41]Pace, A., Trilnik, F., Campo. M., Assisting the Development of Aspect-based MAS using the SmartWeaver Approach. Software Engineering for Large-Scale Multi-Agent Systems. LNCS 2603, Springer-Verlag, April 2003.