PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 04/06

Testing & Debugging Multi-Agent Systems: A State of the Art Report

Maíra Athanázio de Cerqueira Gatti

Arndt von Staa

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. 04/06 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena February, 2006

Testing & Debugging Multi-Agent Systems: A State of the Art Report *

Maíra Athanázio de Cerqueira Gatti, Arndt von Staa

{mgatti,arndt}@inf.puc-rio.br

Abstract. Current research and development of agent-based systems has focused primarily on architectures, protocols, frameworks, messaging infrastructure and community interactions. As intelligent agent-based systems take over operations in the financial community, transportation, manufacturing, utilities, aerospace, and the military, assurances will need to be given to the owners and operators of these systems assuring that these non-deterministic learning systems operate correctly. In this State of the Art report (SotA), we will give an introduction to work presented in the area of testing and debugging distributed systems composed of complex autonomous entities (agents). We will provide pointers to work by large players in the field. We will also discuss the debugging of multi-agents systems. We will explain why this kind of system must be handled differently than less complex systems.

Keywords: Multi-agent systems, testing, debugging, verification, validation.

Resumo. A pesquisa e o desenvolvimento de sistemas orientados a agentes focam, geralmente, metamodelos, arquiteturas, protocolos, *frameworks*, infra-estrutura de troca de mensagens e organizações de agentes. Considerando que sistemas multiagentes estão presentes no mercado financeiro, na indústria de transporte, utilidades, espaço aéreo e militar, entre outros, faz-se necessário garantir para os seus proprietários e/ou usuários que esses sistemas não-determinísticos operam corretamente. Neste relatório do estado da arte, será apresentada uma introdução dos trabalhos propostos na área de testes e depuração de sistemas multiagentes, que são sistemas complexos, distribuídos, compostos por entidades autônomas (agentes). Será apresentado o trabalho da área de agentes tanto de testes como de depuração, além do porquê esse tipo de sistema precisa ser testado diferentemente dos outros tipos de sistemas.

Palavras-chave: Sistemas multi-agentes, testes, *debug*, verificação, validação.

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

Table of Contents

1 Introduction

A software agent is a computer program that works toward goals in a dynamic environment on behalf of another entity (human or computational), possibly over an extended period of time, without continuous direct supervision or control, and exhibits a significant degree of flexibility and even creativity in how it seeks to transform goals into action tasks.

A multi-agent system (MAS) is a computational environment in which individual software agents interact with each other, in a cooperative or competitive manner, and sometimes autonomously pursuing their individual goals, accessing resources and services of the environment, and occasionally producing results for the entities that initiated those software agents [22]. The agents interact in a concurrent, asynchronous and decentralized manner [6], hence MAS turn out to be complex systems [9]. They are also non-deterministic, since it is not possible to determine a priori all interactions of an agent during its execution. Consequently, they are difficult to debug and test.

Current research and development of agent-based systems has focused primarily on architectures, protocols, frameworks, messaging infrastructure and community interactions. As intelligent agent-based systems take over operations in the financial community, transportation, manufacturing, utilities, aerospace, and the military, assurances will need to be given to the owners and operators of these systems that these non-deterministic, learning systems operate correctly [21]. There are some works which address the problem of building confidence in the owners and users of agent-based systems with particulars techniques which we are going to describe in this work. Some of them are based on testing and monitoring, others are based on debugging, and others on simulation. Moreover these works are still at very early stage. Actually formal methodologies provide validation tests that are applicable in very few and quite irrelevant cases though. The main reason of this lack applicability is that activities, which should assure that the program performs satisfactorily, are very challenging and expensive since it is quite complicated to automate them [2].

Agents are software and developing agents is developing software. And an important phase of developing software is debugging it. It is suggested that debugging and testing may occupy between 25 and 50 percent of the total cost and time of system development with much of this time spent locating the cause of a problem [1]. Unfortunately, debugging multi-agent systems without good debugging tools is highly impractical [16].

The State of the Art Report presented in this work aims to survey techniques and tools for testing and debugging multi-agent systems in the Software Engineering approach. Nevertheless, there are some works that propose strategies and techniques for testing or evaluating multi-agent systems through theorem proving and model checking. However, in section 2 we will see why we are not focusing on those strategies. Thus, this paper surveys all tools found in literature (IEEE Xplore, ACM Digital Library) for the Software Engineering approach.

The paper is organized as follows: section 2 describes the multi-level approach to MAS testing; section 3 describes related works presented in literature which address the techniques for testing and debugging MAS. This section also presents some tools, a comparative table between the approaches and tools found, and also the relative strengths and weakness of each one and the open issues of each one; and, finally, section 5 provides the issues discussed that have arisen during the work on this report, and the essential features of a tool/framework for testing MAS.

2 Software Debugging, Testing and Verification

Testing is an activity in which a system or component is executed under specified conditions, the results are observed or recorded and compared against specifications or expected results, and an evaluation is made of some aspect of the system or component. A test is a set of one or more test cases. The main aim of a test is to find faults.

An error is a mistake made by the developer misunderstanding something. A fault is an error in a program. An error may lead to one or more faults. When a fault is executed an execution error may occur. An execution error, error for short, is any result or behavior that is different from what has been specified or is expected by the user. The observation of an execution error is a failure. Notice that errors may go on unnoticed and hence may play serious havoc with the remaining computation and use of the results of this computation. The longer the period of unobserved operation, the larger is the probability of serious damage due to errors, that is due to unobserved failures.

There are two kinds of tests: static verification and dynamic validation. The former is based on code inspection or "walk through", symbolic execution, and symbolic verification. The later generates test data and execute the program. Figure 1 shows where static verification and dynamic validation tests occur during the software life cycle [8].

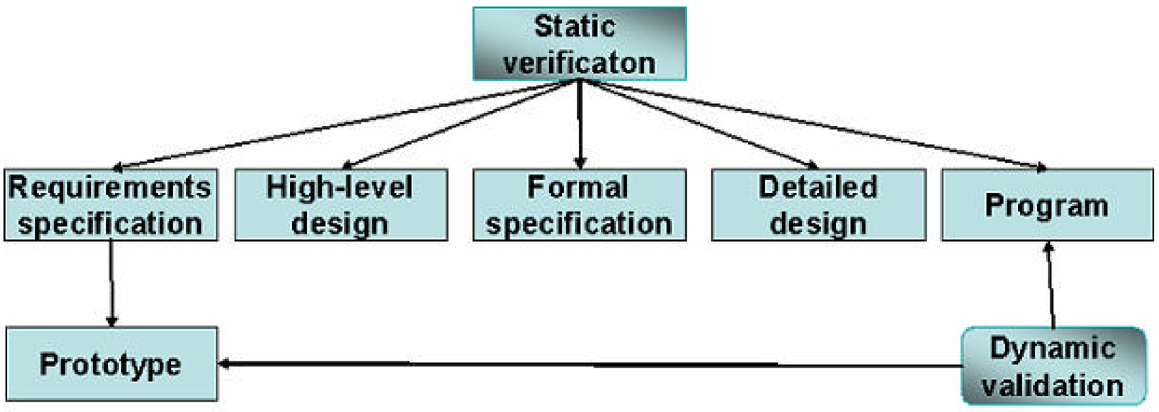


Figure 1 - Kinds of Tests

There are several strategies for testing software and the goal of this survey is not to explain all of them. However, we will describe the main strategies found in literature ([8],[23]) for testing software which are related to some of the works presented in the fourth section. Here they are:

- Black-box testing: also know as functional testing or specification-based testing. Testing without reference to the internal structure of the component or system.
- White-box testing: testing based on an analysis of the internal structure of the component or system. Test cases[1] are derived from the code e.g. testing paths.
- Progressive testing: it is based on testing new code to determine whether it contains faults.
- Regressive testing: process of testing a program to determine whether a change has introduced faults (regressions) in the unchanged code. It is based on re-execution of some/all of the tests developed for a specific testing activity.
- Performance testing: verify that all worst case performance targets have been met, and that any best-case performance targets have been met.

There are several types of tests. The most frequently performed are the unit test and integration test. A unit test performs the tests required to provide the desired coverage for a given unit, typically a method, function or class. A unit test is white-box testing oriented and may be performed in parallel with regard to other units. An integration test provides testing across units or subsystems. The test cases are used to provide the desired coverage for the system as a whole. It tests subsystem connectivity.

There are several strategies for implementing integration test: (i) bottom-up, which tests each unit and component at lowest level of system hierarchy, then components that call these and so on; (ii) top-down, which tests top component and then all components called by this and so on; (iii) big-bang, which integrates all components together; and (iv) sandwich, which combines bottom-up with top-down approach.

The techniques and strategies presented in this section will appear in the approaches in the following section. The main idea is to relate them with the works presented and classify them according to each strategy or technique.

3 A Multi-level Approach for Testing and Debugging MAS

It is quite hard to verify that agents or multi-agent systems satisfy user requirements, behave correctly and are not malicious. One possible way to help solving this problem is to develop and use design patterns, as well as algorithms and tools for evaluating multi-agent systems [6].

There are several reasons for the increase of the difficulty degree of testing and debugging multi-agent systems: increased complexity, since there are several distributed processes that run autonomously and concurrently; amount of data, since systems can be made up by thousands of agents, each owning its own data; irrepro-

1 A test case is a set of test inputs, execution conditions, and expected results developed for a particular objective.

ducibility effect, which means that it is not ensured that two executions of the systems will lead to the same state, even if the same input is used. As a consequence, looking for a particular error can be difficult if it is not possible to reproduce it each time [6].

There are at least three approaches for evaluating multi-agent systems: (i) theorem proving, which corresponds to checking that a set of formulas satisfies a goal; (ii) model checking, which builds a model of a system and verifies that a temporal logic formula holds for the model; and finally, (iii) testing, which consists in applying test cases to the system and checks if it behaves properly different from model checking which checks if agents are proved correct [6].

The several challenges in developing and debugging multi-agent systems make traditional component testing technologies not applicable [21]. Besides all the challenges due to its autonomy and parallelism, message specifications are often incorrect or incomplete and visualization of the messages is needed. Another challenge is the fact that agents can be programmed to learn, so successive tests with the same test data may give different results [21].

Testing a single agent is different from testing a community of agents. When testing a single agent a developer is more interested in the functionality of the one agent and whether the agent operates for a set of messages, environmental inputs and error conditions. On the other hand, when testing a community of agents, the tester is interested in whether the agents operate together, are coordinated, and if message passing between the agents is correct.

The agent society test is a kind of integration test and the integration strategy depends on the agent system architecture where agents dependencies are usually in terms of communications (but sometimes environment mediated interactions could be present). In the case of a multi-layer (open or closed) architecture, it is possible to adopt common strategies like the top-down or bottom-up composition. With other MAS configurations the integration problem becomes much more difficult to solve because of the proactive nature of agents, in some cases it is not possible to identify the subset of agents that could be tested in a stand-alone basis. Such situations should be faced with a study of the agent responsibilities (in terms of functionalities provided) and ontology (an agent is not expected to interact with something that is out of its knowledge). Society test for mobile agents is also influenced by many configuration management and hosting platform capabilities problems that can affect the agent/system functionality or performance. In the case of functionality, it is not easy for an agent implemented with most diffused platforms to verify before moving to a specific host if all the (not standard) libraries that it needs are available; as regards performance it is easy to imagine the problem that an agent dealing with a great amount of multimedia data could face when moving to a host that cannot provide enough resources (elaboration power, network bandwidth and so on) [2].

3.1 Agent Level Testing

Testing is usually done incrementally during software development. Developers also usually want to incrementally test their agents as they progress, testing functionality as it is added. Much of this testing would require another agent to trigger an event inside the agent to be tested, such as message from another agent, or an event from the environment. Additional testing would also be required to test learning [21]. An important issue related to the bounds of testing MAS is that it is not possible (or very expensive) to predict the agent behavior. Hence, it often is impossible to have an oracle able to give information in order to verify its correctness due to its non determinism.

Therefore, when developing a single agent (for inclusion into a community) developers want to make sure that it responds correctly to given inputs from other agents. Moreover, a sizeable set of agents, none of which have been tested, may be needed to test a single agent.

In [21] some of the errors types are presented, which developers run into while developing an agent, which include:

- Incorrectly addressing a message to another agent;
- Putting an incorrect request in a message so the receiving agent does not recognize the message;
- Incorrectly parsing incoming messages;
- Checking for the wrong performative, which is the type of the communicative act, in an incoming message;
- Not developing code to accept all the messages it is suppose to accept, etc.

An agent can be tested as a black-box which focuses on the behavior of the agents. Testing its behavior corresponds to begin an interaction and evaluating its result. And an agent can be tested as a white-box, which corresponds to test the agent internal behaviors, which is the same of how behaviors are related and their flow of control, where each single behavior is seen as a black-box. The result is therefore a kind of black-box testing of a subsystem (the agent composed of the agent base class and several behavior classes) [2].

3.2 Society Level Testing

Testing communities of agents involves two issues. The first one is how we can ensure that the agents in the community work together as designed previously, which again is related to the oracle problem described before. And second one is how we can ensure that the resultant work is the one expected. During a society test, the validation of the overall results of the different agents it is carried on and the successful integration of the different agents is verified. This involves checking that each agent in the community/society receives the correct messages from the correct agent, provides the correct responses, and interacts with environment correctly as a whole [21], which is related to the first issue. Moreover, it can also involve checking that the goal of this community or organization where the agents are interacting is being achieved, which is related to the second issue.

Some types of errors that can be observed during the developing of a community are miscommunicating the performative (the type of the communicative act), content on agent messages, and designing deadlocks into the messages exchanges. Another issue that has to be considered is the scalability. The larger the agent communities become, the harder it is to test them for proper functionality [21].

Using traditional tools for debugging agent societies is insufficient, that is, it has lack efficiency and it is inadequate, specially because multi-agent systems are distributed systems. Programmers are generally presented with too much inadequately correlated information making it difficult to understand what is really happening in the system. Without a proper procedure for identifying what sorts of information to look for it is unrealistic to know in advance what information will be useful when trying to debug the system. Moreover, most systems have no means of identifying where problems may be occurring, even if the developer notices an error it could take an unnecessarily time to point the location of the error.

Since most of the debugging tools output raw messages, the developer needs to inspect the contents of the messages and the flow of messages and try to determine what is going wrong. With a large numbers of messages between a large numbers of agents this can be extremely difficult [16].

Generally, visualizing overall system behavior in systems with distributed control is a notoriously difficult task. Each agent in the system has only a local view of the organization, and the burden is on the user to integrate into a coherent whole the large amounts of scope-limited information provided by individual agents. Furthermore, because of the complexity of multi-agent interaction and behavior, effective visualization assumes more importance than in single-agent systems. It is necessary in order to reduce the information overload on users, and thus allows then to confirm, understand, control and/or analyze the behavior of the system, and to debug the system [12].

In the next section we describe the current state of the art for debugging and testing techniques and tools. We will also evaluate them considering its approaches, its relative strengths and weakness, and the open issues of each one.

4 A Survey of Testing and Debugging Multi-Agent Systems

In this section we provide an overview of several works that, in one way or another, include some notion of testing and/or debugging multi-agent systems. First, we describe the approaches for testing techniques, which do not address the debugging techniques. Then, we describe the debugging techniques and its tools. It is important to notice that some debugging tools found in literature for multi-agent systems also contain testing techniques (Zeus [12][13], Prometheus [15][16][17]). They are separated in two subsections because the testing techniques may be related only to test cases, and can not be considered a debugging tool. We also evaluate them considering its approaches, its relative strengths and weakness, and the open issues of each one.

4.1 Testing

Passi [2]. The proposal of this work is not to provide an exhaustive testing tool but to propose a new approach based on a simple testing framework which lets developers build a test suite effortlessly in a cheap and incremental way. As a framework, it provides a unifying application model and a partial implementation of it, trying to support the developer in creating and executing tests in a uniform and automatic way. They aim to reduce time and cost when developing MAS, guarantee quality assurance, and provide automatic activities which should assure that the program performs satisfactorily.

Their framework is built on top of JADE and lets developers create tests at different levels (hierarchical approach) simply acting as a support for running tests and visualizing results. The framework is based on a two-level model. At the first level they identify the agent as an atomic entity. In order to check the correctness of the activities carried out by a single agent a number of different cases must be tested. This leads us to the second level where they identify specific agent tasks.

There is an “test-agent” which performs the set of tests related to all the capabilities of a given agent. Tests on specific tasks, on the other hand, will be referred to as “task-test”. In order to reflect the two-level model, the following classes are provided:

- Test class, representing the test of a specific task of an agent.
- TestGroup class, representing the group of all the agent tests. It is basically a collection of Test objects. The list of task-tests to be included in a TestGroup is described in an XML file.

In general, all test methods in a TestGroup share the same fixture, which consists of objects and anything else needed to perform the test. A Test or a TestGroup is executed by a tester agent i.e. an agent that extends the TesterAgent class. Each tester agent has a behavior, an extension of the TestGroupExecutor class, which is in charge of getting the group of tests to be executed and for each test adds the corresponding behavior to the tester agent scheduler. The list of all the agent-tests that can be tested and the list of task-tests to be performed for each of them are described by means of XML files. There is a single main XML file that contains the list of all the agent-tests of the application and one XML file for each agent-test that contains the list of task-tests to be executed. Developing an agent-test means therefore developing a new tester agent in charge of the group of task-tests described in the associated xml file.

Finally the utility class Logger provides methods to create logs. By extending this class it is possible to create sophisticated loggers in order to provide reported information in more suitable formats. To date, reported information can be displayed in a graphical user interface (where very essential information is shown), written to a text file, printed to the standard error or organized into web pages.

In the Figure 2 a class diagram of the framework main classes is represented.

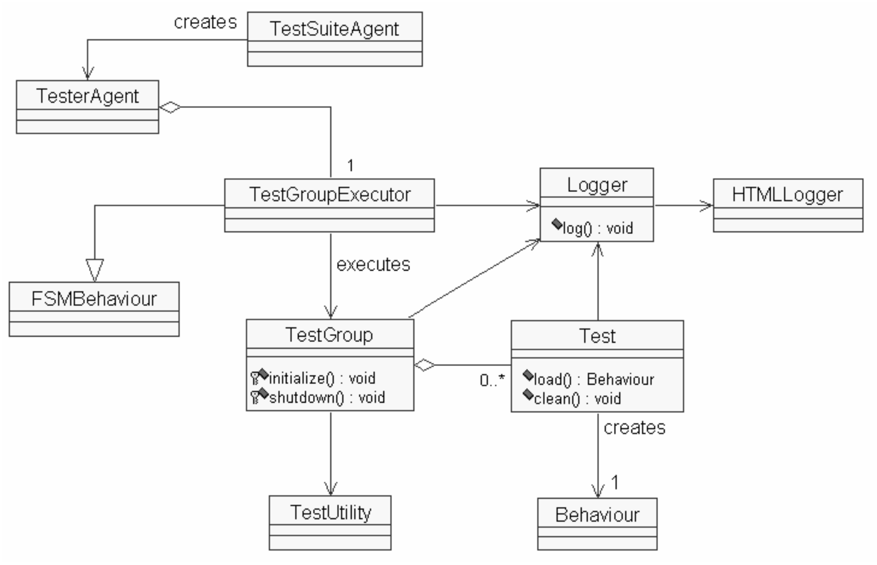
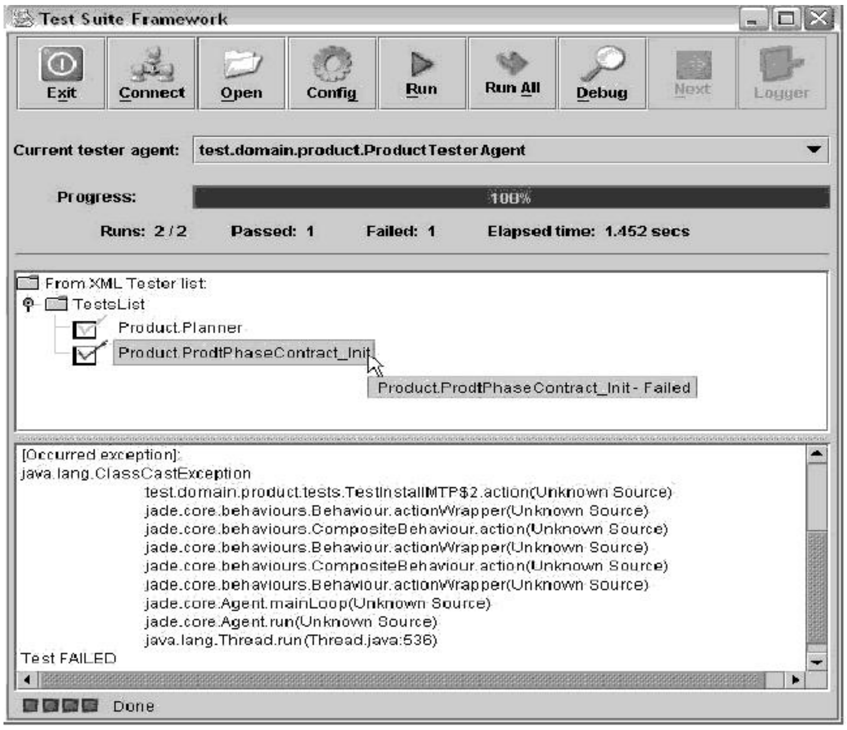


Figure 2 - PASSI - Test Framework main classes

A single test and group of tests can be executed by simply launching the corresponding tester agent. A more convenient way of performing them is by means of the TestSuiteAgent, an agent that provides a valuable graphical interface to run tests. When a test or a group of tests are launched the TestSuiteAgent creates the proper tester agent and delegates to it the execution of the tests. During the testing activity the tester agent will send FIPA ACL messages to the TestSuiteAgent, informing it about the test outcomes and giving eventually detailed information concerning the causes of failure.



Current tester agent: test.domain.product.ProductTesterAgent

Runs: 2 / 2 Passed: 1 Failed: 1 Elapsed time: 1.452 secs

Product.ProdtPhaseContract_Init - Failed

```
[Occurred exception]:
java.lang.ClassCastException
                test.domain.product.tests.TestInstallMTP$2.action(Unknown Source)
                jade.core.behaviours.Behaviour.actionWrapper(Unknown Source)
                jade.core.behaviours.CompositeBehaviour.action(Unknown Source)
                jade.core.behaviours.Behaviour.actionWrapper(Unknown Source)
                jade.core.behaviours.CompositeBehaviour.action(Unknown Source)
                jade.core.behaviours.Behaviour.actionWrapper(Unknown Source)
                jade.core.Agent.mainLoop(Unknown Source)
                jade.core.Agent.run(Unknown Source)
                java.lang.Thread.run(Thread.java:536)
Test FAILED
```

Figure 3 - TestSuiteAgent GUI – The Product agent under test

The TestSuiteAgent, as stated before, provides a graphical interface to run tests, by means of which it is possible to: (i) view information related to the agent-tests and all the task-tests they include; (ii) select and load the tests to be executed; (iii) execute all agent-tests of the list in sequence and produce a final report indicating, for each agent-test, the number of task-tests which have passed and failed and the corresponding causes of failure.

What makes this GUI particularly helpful is the binary visualization of the results, stating whether the test has passed or failed, which cuts down considerably the time needed to analyze tests outcomes. This GUI is similar to the JUnit which is a framework for automated unit tests (for e.g. Figure 4). In fact while the test is running the GUI shows its progress with a progress bar. The bar is initially green but turns into red as soon as there is an unsuccessful test. In this case, the failed tests are marked in the list and detailed information of the causes of failure is reported at the bottom. As a general rule software applications should be tested thoroughly. In particular as changes are made to the system and the added functionality is tested, previously tested functionalities have to be re-tested to assure that the modifications have not corrupted the system. In order to cut down the time of regression testing and to make the task of the developer easier, the button "RunAll" is provided in order to allow the execution of all the tests listed in the main xml file. A snapshot of the TestSuiteAgent GUI is shown in Figure 3. In this figure an example of framework use is reported. The test concerning the ProductProdtPhaseContract_init behavior has failed. The progress bar is red (dark grey in the figure), the unsuccessful test is marked and in the lowest text area information about the causes of failure is reported (the thrown exception, in this specific case).

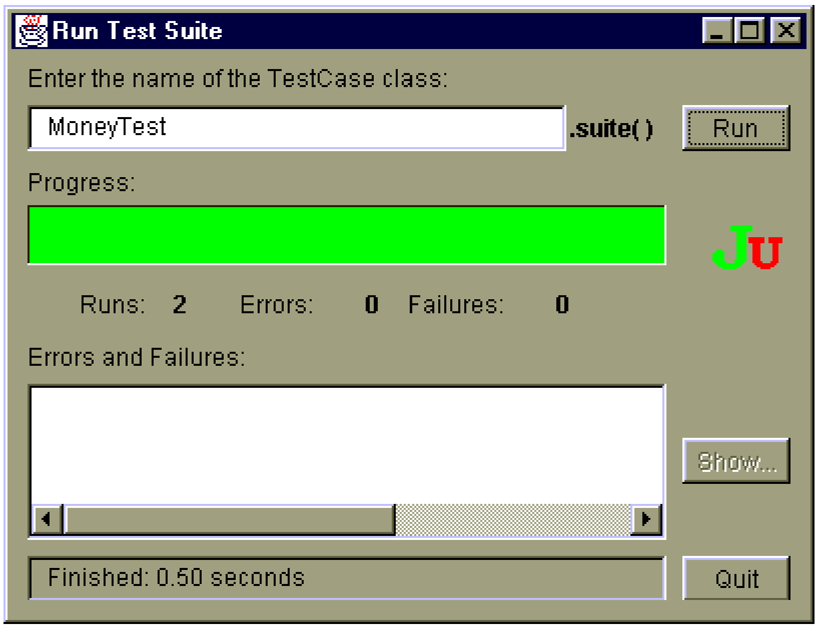


Figure 4 - JUnit: A Successful Run

Test Agent [21]. This work provides a *Test Agent* which is able to be inserted or plugged into a community of agents to exercise each of the agents in the community as well as the community as a whole, and to also monitor the agents after they are deployed to insure proper functionality.

Figure 5 shows the architecture and data flow for the test agent in its completed form. The test agent receives message specification and stores it in the message specification file (potentially as XML, which would allow it to be easily reused). From this information test scripts can be generated to test each of the defined autonomous systems.

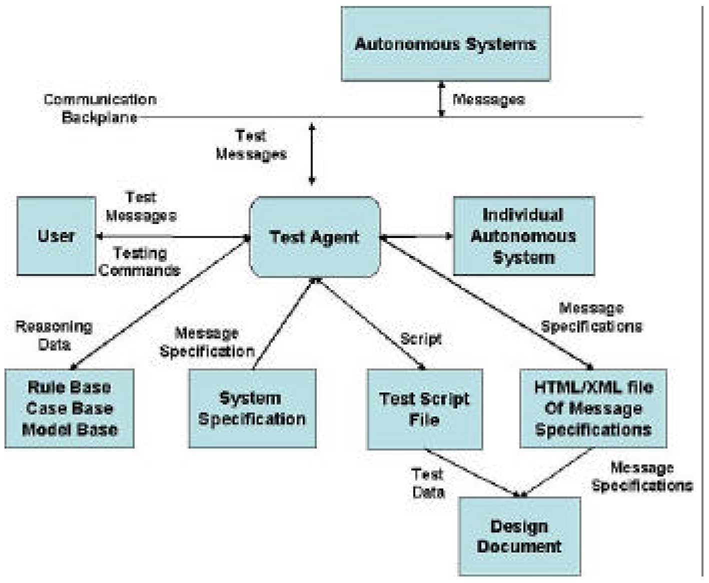


Figure 5 - Architecture of the Test Agent

The test agent tests other agents and communities by:

- Testing a single agent’s ability to send or receive a specific message;
- Testing a single agent’s ability to handle valid as well as invalid messages;
- Testing a community’s ability to handle all defined messages and representative number of invalid messages;
- Maintaining the official message specifications for the agent community;
- Maintaining the message specifications in the design document;
- Collecting metrics on network usage, inter-agent communications and other needed metrics for scalability issues;
- Monitoring an agent system for potential errors and performance problems.

The test agent supports both regression and progression testing of agent message handling capabilities. The regression testing is used to insure that agents perform to their stated specifications and that modifications to agents do not affect existing message handling capabilities. The progression testing supports agent developers as they add new messaging capabilities to their agents.

Since a complete specification of the agent communications is stored in the specification file, it is possible to output the contents of the file into an HTML, XML or other format that could then be used in a design document. The design document is automatically updated when the specification file changes. The messages can be specified in a formal language which allows the message specification file to be generated directly from the content of the formal specification, which in turn will ensure that the content of the message specification file is always correct and reflects the current message specification. In addition, they also ague that deadlock conditions can be prevented.

Testing individual agents is done by sending specific messages from the agent to the target agents and examining the messages returned (if any). The set of valid messages for each agent will be generated from the message specification file. From this definition, the test agent will be able to generate a set of test messages and interpret the responses as valid or invalid. Rules to apply to the test messages will be stored in a rule base and will be generated from user inputs and the agent specification.

Testing a community of agents is also based on the message specification file. From this file the test agent can generate a test script for the entire community of agents. This test script again, can also be run interactively on in a batch mode. The test agent is the successively substituted for each agent in the community (again using automatic testing) with the results being stored in a file or emailed to a developer.

For later tests, developers will also be able to fold progression tests into the regression test script to form a new regression test script. Batch testing will simply execute a predefined set of regression tests and have the results saved to a file.

Madkit [6]. This approach focuses on a specific kind of testing called the Record/Replay mechanism [20] used in regression testing. The Record/Replay mechanism is a test performed during the execution of the system either in simulation or in production. It is realized through system inspection. The record phase records actions in the system (memory, environment, data update, messages, etc.). When an error occurs in the system, designers have a system that they can play and replay until they found the error and fixed it. They also compare model checking and testing, and say that contrarily to the former, testing checks that agents behave properly rather than agents are proved correct. In their approach, the Record/Replay mechanism is coupled to testing via *post-mortem* analysis. It uses the events and data stored during the record phase and checks properties without reexecuting the system.

The information stored depends on what designers want to replay. Actually, there exist two kinds of execution replay: content-based and ordering-based execution replay. The content-based execution replay considers storing all the instructions from systems (instructions from programs as well as memory). The ordering-based execution replay only stores the non-deterministic events in order to make sure they will be replayed in the same order. As stated, the ordering-based execution replay has high level of abstraction. In case of ordering-based execution replay, the record phase uses vector clocks to store the event ordering. Hybrid approaches mixing content-based and ordering-based execution replays offer the better of the two approaches. This is the one chosen for this approach.

There are three monitors agents responsible for the record phase. Two of them were extended from the MadKit platform [3] and the other was created. The agents are:

- The group message tracer agent: is responsible to trace messages sent within the group. Each time this agent receives a message, it records it in a file. During the replay phase, the group message tracer agent reorders the messages based on the trace file.
- The organization tracer agent: is responsible to record group and role actions: when agents enter or leave a group, or when they add or remove a role. It records group and role actions in a trace file. This agent is not used in the replay phase.
- The environment tracer agent: is responsible to record the environment initial configuration and modification. This agent checks the environment at the end of each time interval and records what the modifications (new agent position, resource production or resource consumption) are. The modifications are stored in a trace file.

Agents executed on the platform do not need special attention except the use of the *trace* and the *getfromTrace* primitives that are used respectively in the record phase and in the replay phase. First, the initial environment is extracted from the trace of the environment tracer agent. It means that resources and agents are located at the position they were at the beginning of the record phase. Each time, agents need a non-deterministic variable data, they get it from the trace file. For instance, here is an excerpt of the code of an agent:

```
if (MODE_RECORD) { /*record*/

        input = getUserInput();

        trace("input", input);

} else { /*replay*/

        input = getfromTrace("input");

}
```

In the context of ordering-based execution replay, it is necessary that messages arrived in the same order to agents. Agents are constrained in the way they have to send the message to the group message tracer and not to the recipient in order to ensure message ordering. The group message tracer is responsible to reorder messages based on the trace file and deliver them to agents in this order. They also rose the idea of using the environment trace as checkpoints: the user specifies which environment situation to consider as initial situation, then the replay is performed from this environment situation. The agents are necessarily moved to a situation compatible with this environment situation, that is, agents have the state they have just at the moment of this environment situation. Another possibility in the Record/Replay mechanism is to execute a specific process and let it evolving and believing it really interacts with other processes. This is realized thanks to the record phase that can simulate the messages. Agents, in the replay phase, send messages to the group message tracer and not directly to recipients in order to repeat the exact order.

The post-mortem analysis uses the data recorded during the record phase. The data from the trace files are extracted and its properties verified. For instance, if the designer wants to check that after receiving a message, the recipient believes the content of the message, the designer extracts the message from the group message tracer file and checks if there is a belief about the content of the message dated after the message receipt. If yes, the recipient believes the content.

XMLaw [19]. In this work, an approach is proposed for integration tests in open multi-agent systems. This approach supports the creation of test cases based on the information provided by the definition of system rules. They propose to use XMLaw, which is a language for the specification of agents' interactions' regulation in open multi-agent systems.

In open multi-agent systems, agents must obey social conventions in order to maintain predictable integration. Usually, these social conventions are hard coded, leading to unsuitable systems. A solution to hard coded conventions is separate the system's social convention into a separate module insuring agents compliance. This technique is called law enforcement.

A law enforcement approach separates the system's rules from the raw source code, making them explicit so that the development and maintainability may become more efficient. Even though the current approaches deal at some extent with the monitoring and enforcement of such laws, it is not clear how it is possible to perform integration tests over such systems, i.e., how we can write test cases and receive reports about the results of them. Then, they argue that, in a systemic way, integration tests in open multi-agent systems become very important since the behavior of the involved agents may be unpredictable.

XMLaw is a law enforcement language and environment supplying a structural and dynamic model. The former describes how all the law elements are related while the last shows how an event-driven architecture is implemented. XMLaw is based on a mediator agent that has an enforcement mechanism for applying the laws. The system laws are described in an XML file and their hypothesis is to reuse this language to establish system's behavior.

Regarding the source code, some system's assertions can be established, assertions are expressed as observation points. Observation points are an extensible feature that permits us to configure the test infrastructure to specific needs. The first one is the time to live observation point it indicates that a scene can not stay alive for more than a stipulated time, if so, for instance, a treatment is activated someone is notified.

Even with an easier law manipulation and a control over agent's behavior, the law enforcement approach does not solve problems of agent integration in open systems. Leading us to figure out how we should identify failures and errors in open system execution, how to execute tests cases and how to report non-compliant executions. As stated before, the XMLaw environment provides a mediator agent, which enforces all messages exchanged by agents. All agents must use the same mediator, where non-compliant messages are always blocked. Also the environment is built with an event-driven architecture, meaning that messages can fire events and observers can subscribe to them.

Concerning testing and the XMLaw environment, they propose an integration test architecture providing means of executing test cases, block-unblock operation mode, allowing the execution of entire test cases, coverage evaluation of test cases and the generation of testing reports. The system takes advantage of the event-driven model used in XMLaw, all events generated in the system can be observed and be reported to a log database, Figure 6 illustrates the architecture.

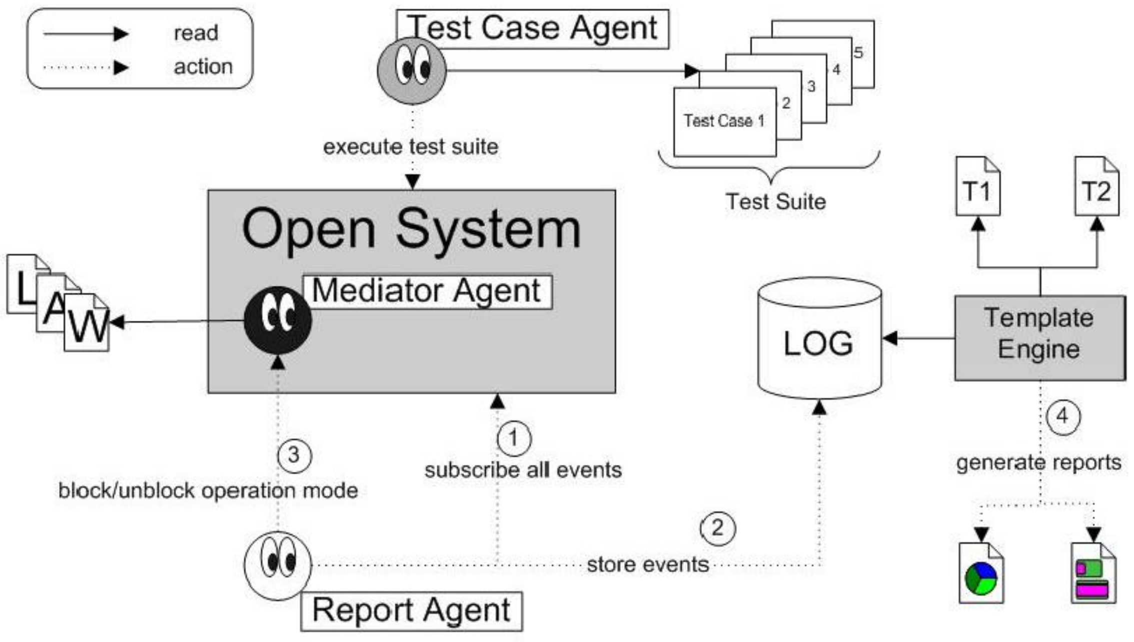


Figure 6 - XMLaw Integration Test Architecture

Since the environment uses a mediator agent which subscribes itself to all events in order to observe all messages, they provided an agent, called Report Agent, which also subscribes itself to all events (1), doing so, all generated events will be stored into a log data base (2). They also proposed to implement a mechanism allowing non-compliant messages cross the system (3), which allows the execution of test suits. However, none of those mechanism have been implemented yet.

At the end of the execution of several test cases, the data-base contains a log of all events fired during the whole execution and also a trace of all messages exchanged. This integration testing approach provides an engine template that, from the log data, generates reports (4) conforming defined in templates (T1 and T2).

4.2 Debugging

Zeus [12], [13]. Zeus is a development toolkit for constructing collaborative agent applications. It provides an integrated environment for the rapid construction of collaborative agent applications. Zeus defines a multi-agent system design methodology, supports the methodology with an environment for capturing user specification of agents, and automatically generates the executable source code of the user-defined agents. It also provides a suite of visualization and debugging tools for the Zeus toolkit.

Zeus debugging tools shift the burden of inference from the user to the visualizer. The visualizer is another agent which just requests local information from agents and tries to build a global view. Some agents may not reply because they are suspended or dead, and in such scenario, the global view presented by the visualizer will be incomplete. The tool-set presented includes:

- A society tool that shows the message interchange between agents in society;
- A report tool that graphs the society-wide decomposition of tasks and the execution states of the various subtasks;
- A micro tool for monitoring the internal states of agents;
- A control tool for remotely modifying the internal states of individual agents. This tools is used for the administrative management of the agents in the toolkit e.g. killing of suspending agents, adding, modifying or deleting goals, resources, tasks, coordination strategies, etc;
- A statistic tool for collating individual agent and society-wide statistics;
- A video tool (within the society, report and statistic tools) which can be used, video-style, to record, replay, rewind, forward, fast-rewind and fast-forward sessions from a database.

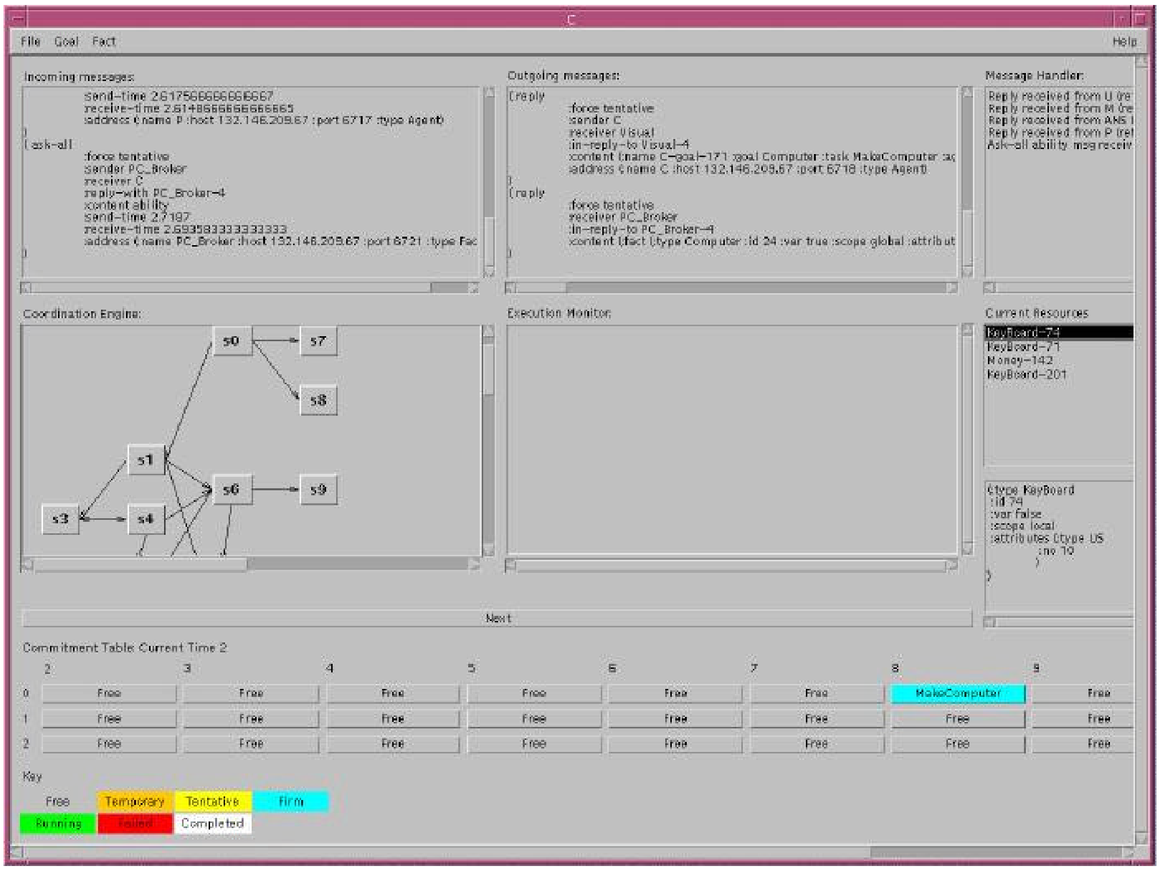
The report tool allows a user to select a set of agents and request that they report to it the status of all their tasks/goals. Then this tool collates the local views to provide a more complete picture.

Although these tools were developed in order to visualize agent applications built using our ZEUS toolkit, it was designed to be relatively independent. Hence with some modifications to the message content language it could be used to visualize non-ZEUS agent systems. The overall architecture of the visualizer comprises a central hub made up of a mailbox, a message handler, a message context database, and a tool launcher.

The mailbox provides facilities for messaging between the visualizer and other agents. Currently, communication is via KQML messages. The tool launcher launches (on demand by the user) instances of the different tools in the tool set, namely, instances of the Society, Statistics, Micro, Report, and Control tools. This is the natural architecture that emerges for providing different viewpoints on the visualization/debugging process.

The message handler processes incoming messages received by the mailbox and delivers them to tool instances that have registered an interest in messages of that type. Tool instances register an interest in receiving messages of a particular type by using the message context database that is consulted by the message handler during its processing. Thus, each tool instance interested in receiving report messages of a particular type from a set of agents will (i) use the mailbox to send requests to those agents that they report to the visualizer whenever events of that type occur, and (ii) register in the message context database an interest in receiving all incoming report messages of the desired type. They utilized the KQML reply-with (outgoing message) and in-reply-to (incoming message) fields to associate an identifier to each message, which is unique to tool instances and event-types. This way, the message handler does not need to scan the contents of a message to determine which tool instance requires it. This arrangement allows users of the visualizer to decide at runtime the set of events they are interested in monitoring, and also to change this set at any time.

As mentioned earlier, the society tool also provides facilities for the storage in a database for later playback of the messages being sent and received by the agents. Again, message filters can be used during storage and/or replay to select particular messages of interest. Replay facilities support the video metaphor, allowing a user to review the messages in forward or reverse mode, in single step fashion or animated. Further, users can make normal database type queries on the stored messages, e.g. "how many messages did agent C send out regarding task 141". The video replay facilities with message filters, and database query support significantly enables users to dynamically change perspectives whilst trying to make sense of the interactions between the agents.



```
            :send-time 2.617566666666667
            :receive-time 2.614866666666665
            :address (:name P :host 132.146.209.67 :port 6717 :type Agent)
)
(ask-all
            :force tentative
            :sender PC_Broker
            :receiver C
            :reply-with PC_Broker-4
            :content ability
            :send-time 2.7197
            :receive-time 2.693583333333333
            :address (:name PC_Broker :host 132.146.209.67 :port 6721 :type Fac
)
```

```
(reply
            :force tentative
            :sender C
            :receiver Visual
            :in-reply-to Visual-4
            :content (:name C-goal-171 :goal Computer :task MakeComputer :a
            :address (:name C :host 132.146.209.67 :port 6718 :type Agent)
)
(reply
            :force tentative
            :receiver PC_Broker
            :in-reply-to PC_Broker-4
            :content (:fact (:type Computer :id 24 :var true :scope global :attribut
)
```

```
Reply received from U (re
Reply received from M (re
Reply received from ANS (
Reply received from P (re
Ask-all ability msg receiv
```

```
KeyBoard-74
KeyBoard-71
Money-142
KeyBoard-201
```

```
(:type KeyBoard
 :id 74
 :var false
 :scope local
 :attributes (:type US
          :no 19
          )
)
```

Commitment Table: Current Time 2

	2	3	4	5	6	7	8	9
0	Free	Free	Free	Free	Free	Free	MakeComputer	Free
1	Free	Free	Free	Free	Free	Free	Free	Free
2	Free	Free	Free	Free	Free	Free	Free	Free

Figure 7: A micro tool view of an agent (C). From left to right, we have views of the incoming and outgoing mail boxes, the message handler summary, the coordination graph, the execution monitor summary and a list of the items in the agent's resource database. ("Adapted from [13]")

The micro tool (Figure 7) provides users with a finer look at the internal processing of an agent. It allows them to select an agent and request to see one or more of:

- (i) the messages being received by the agent;
- (ii) the messages being sent out by the agent;
- (iii) a summary of the actions taken in response to incoming messages, for example, to what module of the agent the message was dispatched to for detailed processing;
- (iv) a graphical depiction of the co-ordination process of different goals by the agent. Each node of the co-ordination graph indicates a particular state of the process, and the trace of a goal on the graph summarizes how it is being coordinated and indicates the current state of the co-ordination process;
- (v) a diary detailing the tasks the agent has committed itself to performing, and the current status of those tasks (i.e. waiting, running, completed or failed);
- (vi) a list of the resources available to the agent including those allocated to the different tasks it has committed itself to;
- (vii) a summary of the results of monitoring executing tasks or tasks scheduled to start execution; e.g. this might indicate a task which failed to start because of inadequate resources or it might flag an executing task which has been stopped because it over-ran its scheduled allocation of processor time.

The entire visualizer is implemented in the Java programming language and runs on Solaris UNIX and Windows NT. To establish that the visualizer meets its required non-functional requirements of being domain-independent, customizable and scaleable, it has been evaluated on (i.e. used in visualizing and debugging) four distributed MAS applications from diverse domains:

- A travel management demonstration application involving 17 agents, in which a travel manager generates an itinerary for a transatlantic trip for a user.
- A telecommunications network management demonstration application with 27 agents. For this application the report tool was customized to support a different display format for task decomposition graphs.
- An electronic commerce demonstration application where agents buy and sell goods and services using different negotiation strategies.
- A business process management demonstrator in which agents collaborate in the provision of complex end-to-end business processes within a call centre setting.

Prometheus [15][16][17]. Prometheus is a methodology which has been developed to support the building of intelligent agent systems. This methodology focuses on covering all phases of development, which includes testing and debugging that are linked to analysis and design.

Their central thesis is that the design documents and systems models developed when following an agent-based software engineering methodology can be incorporated in an agent and used at run-time to provide for run-time error detection and debugging [16].

The idea is that the protocols during design specify allowable message exchanges between agents. Therefore, it is possible to check that a given execution of a system does not violate existing protocols and a debugging tool which monitors execution can hence detect violations of the protocols as specified and can notify the developer.

Monitoring can be done via eavesdropping on the communication medium (e.g. [4]), or more directly by requiring that a carbon copies of any messages sent also be sent to the debugging agent as done in the Zeus toolkit. The approach developed in relation to Prometheus involves automatically adding code that sends copies of messages to a monitoring agent. For the debugger to reason about the correctness of a message within a particular conversation it needs to compare it against the protocol specification.

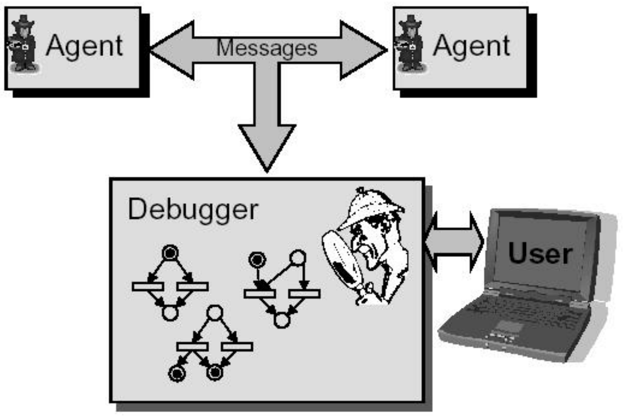


Figure 8 - Debugging System Design

The debugger has a library of the specified interaction protocols that it uses to detect errors. The debugger keeps a list of active conversations so that it can monitor multiple interleaved conversations simultaneously. When a message is received from an agent it is added to the appropriate conversation (or a new conversation is instantiated), and is then processed by firing any enabled transitions (Figure 8).

For any given conversation the debugger does not know what protocol the agents are following. Therefore the debugger keeps a list of possible protocols, instantiated from the interaction protocol library, which currently match the sequence of messages within the conversation. As the conversation progresses the possible protocols list is reduced whenever a message is received that causes an error in the individual protocol. The conversation is still considered valid as long as there is at least one entry in the possible protocols list.

Each time the debugger receives a message for a specific conversation an analysis is done on each protocol within the possible protocols list of that conversation to identify error situations. As long as there is more than one protocol in the possible protocols list, errors simply lead to the conclusion that this protocol was not in fact the one that was being followed, and it is discarded from the list. However if an error is detected in the only remaining protocol, then this is reported.

This technique of monitoring conversations between agents is capable of automatically identifying incorrect interaction patterns. An incorrect execution pattern is typically the result of a lower level coding error in one of the agents. The debugger provides information about the protocol that the agents were engaging in, the agent types and instances that were executing, and the point at which a conversation diverged from the allowed behavior. This provides valuable information to the developer to assist in locating the exact cause of the error.

The debugging mechanisms proposed by this work described have been implemented and also been tested on a substantial implementation project of a meeting scheduler. They implemented a debugging agent using the algorithms for converting and have tested in some examples. They also converted and tested several FIPA interaction protocols, including the call for proposal, request interaction and query interaction protocols. They stated that their debugging agent was able to identify incorrect message exchange whenever it occurred. They also performed testing to ensure that correct message exchanges are not erroneously flagged by the debugging agent.

4.3 Analysis

This section is concerned with the results of the evaluation approach. First we are going to present the analysis for each framework or tool investigated on this survey. Then we are going to present a table that categorized them by strategy (testing/debugging) and level (agent/society) with the values of evaluation of each one.

PASSI. The testing framework proposed in the PASSI methodology lets developers build a test suite effortlessly in a cheaply and incrementally way. The uniformity of the approach and the binary representation of the results helps the developers to create test cases and interpret the information. The framework allows the developer to run regression tests which is good, since it allows the developer do run the tests while develops it. In particular as changes are made to the system and the added functionality is tested, previously tested functionalities have to be re-tested to assure that the modifications have not corrupted the system. However the framework is bound to the agent level testing. No society level approach has been provided, hence it is still difficult to find a behavior organization deviation easily. The visualization tool provided also needs some enhancements to show the messages exchanged by the current agent being tested.

Test Agent. The test agent proposed in this work can be used to test a single agent or can be inserted into a community of agents to test the community as a whole. In addition, a subset of the test agent can be deployed with an agent community that will provide data on how well the community is performing and to give feedback on their correct operation. The test agent supports both regression and progression testing of agent message handling capabilities. However, despite all the efforts on testing, there is no debugging tool for helping developers to visualize the execution of the agent or the system. Moreover, the framework doesn't provide any monitoring approach. Monitoring agents can handle behaviors errors not related to message exchanging between agents. If the agent crash, for example, it is not possible to know if the agent isn't answer messages because it is in a deadlock or if it is crashed.

Madkit. This work proposed the Record/Replay mechanism used in parallel or distributed systems for testing multi-agent systems in the Madkit platform. Its main advantage is to deal with scaling and combinatorial explosion since only non-deterministic events (and optionally data) are recorded. Besides the Record/Replay mechanism, a post-mortem analysis is added, which allows some verification of properties. It is then possible to record data and verifies the relation between them. This could help for instance, when verifying that agents are compliant to the ACL semantics or when verifying that an agent respects some norms if it belongs to a specific group. However, the supplementary code necessary for the record mechanism can generate the probe effect. Moreover, it would be better if there was a tool that inserts those primitives were required. This approach also doesn't have any tool for visualization. Then it is too hard to verify the data evolution into the trace files. Another issue is the overhead of the record phase. Depending on the number of properties to be traced, it may affect the system performance.

XMLaw. This approach is very useful for open systems that have to be regulated by a law enforcement approach. As XMLaw allows designers specifying the interaction between agents, the enforcement by itself allows the verification of the interaction protocols. The test cases are proposed as observations points in the law specification which can allow an integration test. However, the approach proposed does not cover all points needed for a complete integration test, since it depends of report interpretation and analysis. Moreover, the points addressed take advantages of already know system information, which were wasted before. Applying these information to testing aims to improve the system in the way that you do not have to redefine what your system must do in law clauses, and what it expect to do in test cases, since the information is already declared. Of course there will always be the test case definition, but in this case the test will be concerned exactly with the agent behavior, i.e., it will test how compliant the agent's messages will be with the system's law without worrying with unacceptable cases. The reporting analysis is also a good tool since it concerns several views of the system's execution. Sometimes, reading log files is an unwise and wasteful task. The idea of focusing on the data to be observed avoids the tedious tasks as the analysis of log and trace files. Also, the template definition can be used in future analysis, so the user does not have to redefine a new report view when re-inspecting a log data base.

ZEUS. Despite the fact that this approach is addressed to multi-agent systems based on collaboration, this one was the best approach found in literature for testing and debugging multi-agent systems. All the debugging verification and validation were integrated to the software life cycle and were related to the previous phase of analysis and design. There are a lot of views which allows a better comprehension of how the agents' behavior goes on during its execution, and a specific tool which exposes different views of an agent execution. It is scaleable, customizable and generic. Even with all those tools, there is a problem that Zeus doesn't solve: as the collected information are done through messages request, it is not possible to predict if an agent is behaving as expected since it may not respond with information of its state. Hence, the society report cannot report informations about that specific agent into the organization. Another issue related to that problem is the fact that in open systems, the agents may be malicious and send wrong messages about their states. A solution for this problem is the implementation of a monitoring agent which checks the agents' state and generates reports for the tool.

Prometheus. This approach proposes a debug agent which monitors the messages' exchanges between agents and checks them against interaction protocols. Violations of the interaction protocols such as a failure to receive an expected message or receiving an unexpected message can then be automatically detected and precisely explained. Their debug agent was able to identify incorrect message exchange whenever it occurred in a protocol execution. They also performed testing to ensure that correct message exchanges are not erroneously flagged by the debugging agent. However, there are some issues they didn't address in this approach. One of them is that fact that they assume that the system isn't timing dependent. It would be necessary to extend their approach for constructing time-dependent systems. Another issue is that the debugging agent cannot translate preconditions, i.e., it cannot know when an arbitrary condition becomes true since it doesn't have access to the internal beliefs of other agents. Finally, the last issue rose was about the debug of the society level considering the environments. They didn't propose anything for the integration of multiple agents in a number of multi-environments.

To finish the evaluation, considering all the strengths and weakness of each approach, as stated before, we associated three values to each approach according to its strategies, where: **0** means that the approach doesn't apply at all the strategy, **1** means that the approach apply weakly the strategy, and **2** means that the approach apply totally the strategy.

For the debugging approaches, a criterion for given the values was based on the fact that the tool had or not a visualization tool for that strategy.

Table 1 - APPROACH COMPARISON

	TESTING				**DEBUGGING**	
	Agent Level		**Society Level**		**Agent Level**	**Society Level**
	Black-box	White-box	Black-box	Integration		
PASSI	2	2	0	0	1	0
Test Agent	2	2	2	2	0	0
Madkit	2	2	2	2	0	0
XMLaw	0	0	0	1	0	1
ZEUS	2	2	2	2	2	2
Prometheus	0	0	0	1	0	1

5 Conclusions and Future Work

The State of the Art presented in this work survey the recent efforts on debugging and testing multi-agent systems. There are a few work on this research and some of the work found in the literature is specific to a methodology or a subsystem which does not address all the characteristics presented in any multi-agent system.

First, we have explained the main strategies and techniques for testing systems in general. Then we showed why those strategies and techniques could not be applied to multi-agent systems and how other techniques (for instance, the ones used in parallel systems) can improve those traditional strategies for the context of agent based systems.

After that we started the survey itself presenting four strategies for testing multi-agent systems ([2],[6],[19],[21]) and two for debugging it ([12], [15]). When studying those strategies we realized that a deep evaluation would be more appropriate to verify with they are so efficient as their authors say they are. But there was no time for developing an example that would experiment each one of them and check all the statements done by the authors. Then, we evaluated them considering the strategies used. Actually, it was not an evaluation itself but a categorization of each work found.

We let as a future work the experiments that should be done in order to verify which strategy should be improved or not. We didn't check the conformity, for instance, of the strategies that assure that they check the analysis and design phase ([2], [12], [15]) with the implemented phase.

However, considering the evaluation done in section four, we realized that the most suitable and complete strategy would be the integration of the one done implemented in MadKit ([6]) which uses the record/replay mechanism, with the one implemented in Zeus ([12]) which have all views necessary for the debugging phase, and with the one implemented in Prometheus ([15]), which has a thoroughly verification between the design phase with the multi-agent execution.

6 Bibliography

[1] Boehm, B.W., Software engineering economics. *Prentice hall, Englewood Clifs*, NJ, 1981.

[2] Caire, G., Cossentino, M., Negri, A., Poggi, A., and Turci, P., Multi-agent systems implementation and testing. In Proc. of From Agent Theory to Agent Implementation - Fourth International Symposium (AT2AI-4), Vienna, Austria, April 2004.

[3] Gutknecht, O., and Ferber, J., The MADKIT agent platform architecture. In Agents Workshop on Infrastructure for Multi-Agent Systems, pages 48–55, 2000.

[4] Heselius, J., 2002. Debugging parallel systems: A state of the art Report. MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-63/2002-1-SE, Mlardalen Real-Time Research Centre, Mlardalen University.

[5] Hewitt, C., Jong, P., Open systems on conceptual modelling. Technical report, Massachusetts Institute of Technology, 1982.

[6] Huget, M.-P.; Demazeau, Y.; Evaluating multiagent systems: a record/replay approach. Intelligent Agent Technology, 2004. (IAT 2004). Proceedings. IEEE/WIC/ACM International Conference on 2004 Page(s):536 - 539 Digital Object Identifier 10.1109/IAT.2004.1343013.

[7] Huget, M.-P., Odell, J., 2004. Representing agent interaction protocols with agent uml. Proceedings of the AAMAS04 Agent-oriented Software Engineering (AOSE) Workshop.

[8] Ian Sommerville, Software Engineering (Sixth edition), Addison Wesley, 2000.

[9] Jennings, Nicholas R., An Agent-Based Approach for Building Complex Software Systems, Communications of the ACM, 44(4), 35-41, April 2001.

[10] Jürgen Lind. MASSIVE: Software Engineering for Multiagent Systems. PhD thesis, University of the Saarland, 2000.

[11] Jürgen Lind. *Iterative Software Engineering for Multiagent Systems - The MASSIVE Method.* Volume 1994 of Lecture Notes in Computer Science, Springer Verlag, Heidelberg, May, 2001

[12] Ndumu, D. T., Nwana, H. S., Lee, L. C., and Collis, J. C. 1999. Visualising and debugging distributed multi-agent systems. In *Proceedings of the Third Annual Conference on Autonomous Agents* (Seattle, Washington, United States). O. Etzioni, J. P. Müller, and J. M. Bradshaw, Eds. AGENTS '99. ACM Press, New York, NY, 326-333. DOI= http://doi.acm.org/10.1145/301136.301220

[13] Nwana H, Ndumu D, Lee L and Collis J, ZEUS: an Advanced Tool-kit for Engineering Distributed Multi-Agent Systems, In Proceedings of PAAM98, London, 1998.

[14] Padgham, L., and Winikoff, M., Developing Intelligent Agent Systems A practical guide. Wiley Series in Agent Technology, Wiley, New York (2004).

[15] Poutakidis, D., Padgham, L., Winikoff, M., An exploration of bugs and debugging in multi-agent systems to appear in the proceedings of the 14th International Symposium on Methodologies for Intelligent Systems (ISMIS 2003)

[16] Poutakidis, D., Padgham, L., Winikoff, M., Debugging Multi-Agent Systems using Design Artifacts: The case of Interaction Protocols, in the proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002). July 15-19, 2002, Bologna, Italy

[17] Padgham, L., Winikoff, M., Poutakidis, D., Adding Debugging Support to the Prometheus Methodology. Engineering Applications of Artificial Intelligence, special issue on Agent-oriented Software Development, Volume 18, Issue 2, March 2005, Pages 173-190, doi:10.1016/j.engappai.2004.11.018.

[18] Reisig, W., Petri Nets An Introduction. EATCS Monographs on Theoretical Computer Science, Springer, Berlin (1985).

[19] Rodrigues, L.F; Carvalho G. R.; Paes, R. B.; Lucena, C.J.P.; Towards an Integration Test Architecture for Open MAS (SEAS), SBES, 2005.

[20] Ronsse, M., Bosschere, K. D., Christiaens, M., Kergommeaux, J. C., and Kranzlmüller, D., Record/replay for nondeterministic program executions. Communications of the ACM, 46(9), September 2003.

[21] Rouff, C.; A Test Agent for Testing Agents and Their Communities. Aerospace Conference Proceedings, 2002. IEEE Volume 5, 2002 Page(s):5 - 2638 vol.5 Digital Object Identifier 10.1109/AERO.2002.1035446

[22] http://agtivity.com/agdef.htm, accessed in Oct/2005.

[23] Pressman, R. S., Engenharia de Software, 5ªed., Rio de Janeiro, McGraw-Hill, 2002.