Test Cases Generation for Multi-Agent Systems Using Formal Specification

Yacine Kissoum[1] **and Zaidi Sahnoun**[2]

[1] Département d'Informatique.
Skikda University. Skikda 21000, Algeria.
kissoumyacine@yahoo.fr

[2] Lire Laboratory,
Mentouri University. Constantine 25000, Algeria.
sahnounz@yahoo.fr

Abstract The importance of verifying the accuracy and reliability of software has been universally recognized. Concerning multi-agent systems very few research works have been undertaken in order to provide developers with valuable tools supporting testing activities. Moreover these works are still at very early stage. Actually formal methodologies give validation tests that are however applicable in very few and quite irrelevant cases. The main reason of this lack is that the activities, which should assure that the program performs satisfactorily, are very challenging and expensive since it is quite complicated to automate them. The aim of the paper is to address important issues connected to the transition from the design phase to the implementation phase, with particular attention to testing, formal approaches and diagrammatic notations supporting the development process when this is very close to the actual implementation of MAS.

Résumé L'importance de la vérification de l'exactitude et de la fiabilité d'un logiciel a été universellement reconnue. Concernant les systèmes multi-agents, très peu de travaux de recherches ont été menés dans l'objectif de fournir aux développeurs des outils efficaces supportant les activités de tests. De plus, ces travaux sont toujours à un niveau très modeste. En fait, les méthodes formelles supportent les tests de validation mais ils ne sont applicables qu'à des cas particuliers et généralement hors propos. La principale raison de ce manque de méthodes est due au fait que l'activité de test est une activité difficile, très coûteuse et son processus n'est aussi simple à automatiser. L'objectif de ce papier est de fournir des idées simplifiant la transition de la phase de conception à la phase d'implémentation avec une attention particulière aux méthodes de test, les approches formelles et les diagrammes notationnels supportant le processus de développement lorsqu'il s'agit des systèmes multi-agents.

Keywords Agent, Formal Specification, Maude, Testing.

Mots Clés Agent, Spécification Formelle, Maude, Testing.

1. Introduction

Agent-based systems [6][12] are new paradigm for modeling and building many computer systems ranging from complex distributed systems to intelligent software applications. It proposes new ways for analyzing, designing and implementing such systems based upon the central notions of agents [13], their interactions and the environment which they perceive and in which they act. Although many Multi-Agent Systems (MAS) have been designed [7][16], there is a crucial lack in term of specification, development methodologies and especially, testing phases.

The importance of verifying the accuracy and reliability of software has been universally recognized. Concerning multi-agent systems very few research works have been undertaken in order to provide developers with valuable tools supporting testing activities. Moreover these works are still at very early stage. The main reason of this lack is that the activities, which should assure that the program performs satisfactorily, are very challenging and expensive since it is quite complicated to automate them [4].

This paper proposes automated test case generation for testing multi-agent systems. A system is exhaustively tested if every possible execution of the system is verified. Clearly, this is not possible in any non-trivial system. Not only is there an exponential number of combinations for execution paths, multi-agent systems are usually multithreaded and are non-deterministic. Executing the system several times with the same test case may yield different execution paths and sometimes, different results depending on the execution paths taken. Pragmatically, only a subset of executions can be tested. This subset should then be designed to reveal as many errors as possible. We aim to achieve this by selecting a coverage criterion on which the generated test cases will provide a level of coverage in the system execution.

On the other hand, the process of specification is fundamental to handle the complexity related to build such systems and specifying the desirable behavior of MAS before their implementation phase. A variety of specification formalisms are available in the multi-agent field [1][9]. Such formalisms have focused on the last stages of the software life cycle. This is a serious limitation with regard to their usability, because they do not provide support to manage the coordination constraints from the early stages in the software development process. Indeed, these models provide suitable support to structure the applications giving separate treatment to coordination patterns and functional components at the implementation phase, but they provide little support to deal with adequate separation in the requirements definition or architectural design phases. Thus, the treatment of coordination constraints is relegated to the implementation phase, incrementing the responsibilities of programmers. As a result, they have to accommodate the system specifications to the rules and tools provided by the coordination model, and this accommodation supposes changes that create a mismatch with the system implementation, having critical consequences over the system maintenance and being a possible cause of misinterpretations and incoherencies. To deal with the specification of MAS, we have thus chosen to use Maude formal specification language [5]. We think that the joint use of this formal language and the defined coverage criteria will help testers build a test suite effortlessly in a cheaply way.

The paper is organized as follows: The next section describes the formal specification of agent classes using the Maude algebraic language. Section 3 describes the adopted approach to test the agents and to generate test cases for the specified multi-agent system. Finally, the section 4 concludes with a discussion about issues and future work.

2. The Formal Specification Approach

The implementation phase typically needs a very precisely defined model, which fully refines earlier design models to achieve a specification that is unambiguous enough to be fed to automated tools such as compilers and code generators. Producing such a detailed model generally entails matching the conceptual entities, gathered from analysis and architectural design deliverables, to the current technological artifacts that are going to be used to build the system. Applying the above consideration to the case of MASs development, we have to recognize that with the current state of the technology the final implementation of MAS will most likely use object-oriented technology. In the past years several libraries, frameworks and middleware platforms have been made available, that provide MAS application developers with agent-level abstractions implemented with object-oriented languages and tools [3][15]. When adopting such support systems, developers express interaction, coordination and deliberation concerns through an agent-level API, resorting to plain object orientation when specifying ordinary computation tasks.

However, representing the dynamics of a MAS is quite different from describing the flow of control of an object-oriented system; the first determinant difference is in the greater encapsulation of the agents; in fact despite having a very complex inner structure (they are often composed of several classes), they interact with the remaining part of the system as a whole (the agent). Another important issue is that agents, usually, cannot directly relate to each others but they need a message transport service that in many architectures is provided by the middleware. These interactions are obviously totally different in nature by the direct method invocations that

could take place within the agent among the classes that constitutes it. In short, it seems that agents are an extension of active objects, exhibiting both dynamic autonomy (the ability to initiate action without external invocation) and deterministic autonomy (the ability to refuse or modify an external request). Thus, the basic definition of an agent is "an object that can say 'go' (dynamic autonomy) and 'no' (deterministic autonomy)." For these reasons, and because the Unified Modeling Language (UML) is gaining wide acceptance for the representation of engineering artifacts in object-oriented software, and because agents are viewed as the next step beyond objects, we decided to use extensions to UML and idioms within UML to represent the distinctive requirements of agents. The result is an Agent UML (AUML) [2][8].

Figure 1 depicts the detailed processing, taken from [14] that take place within an Order Processor agent. Here, a sequence diagram indicated that the agent process is triggered by a Place Order communication act and ends with the order completed. The two vertical, or *activation*, bars indicate that the receiving agent is processing the various communication threads concurrently. The internal processing by the Order Processor is expressed as an activity diagram, where the Order Processor accepts, assembles, ships, and closes the order. The dotted operation boxes represent interfaces to processes carried out by external agents—as also illustrated in the sequence diagram. For example, the diagram indicates that when the order has been assembled, both Ship Order and Request Invoice actions are triggered concurrently. Furthermore, only when both the payment has been accepted and the order has been shipped, the Completed Order process can then be invoked.

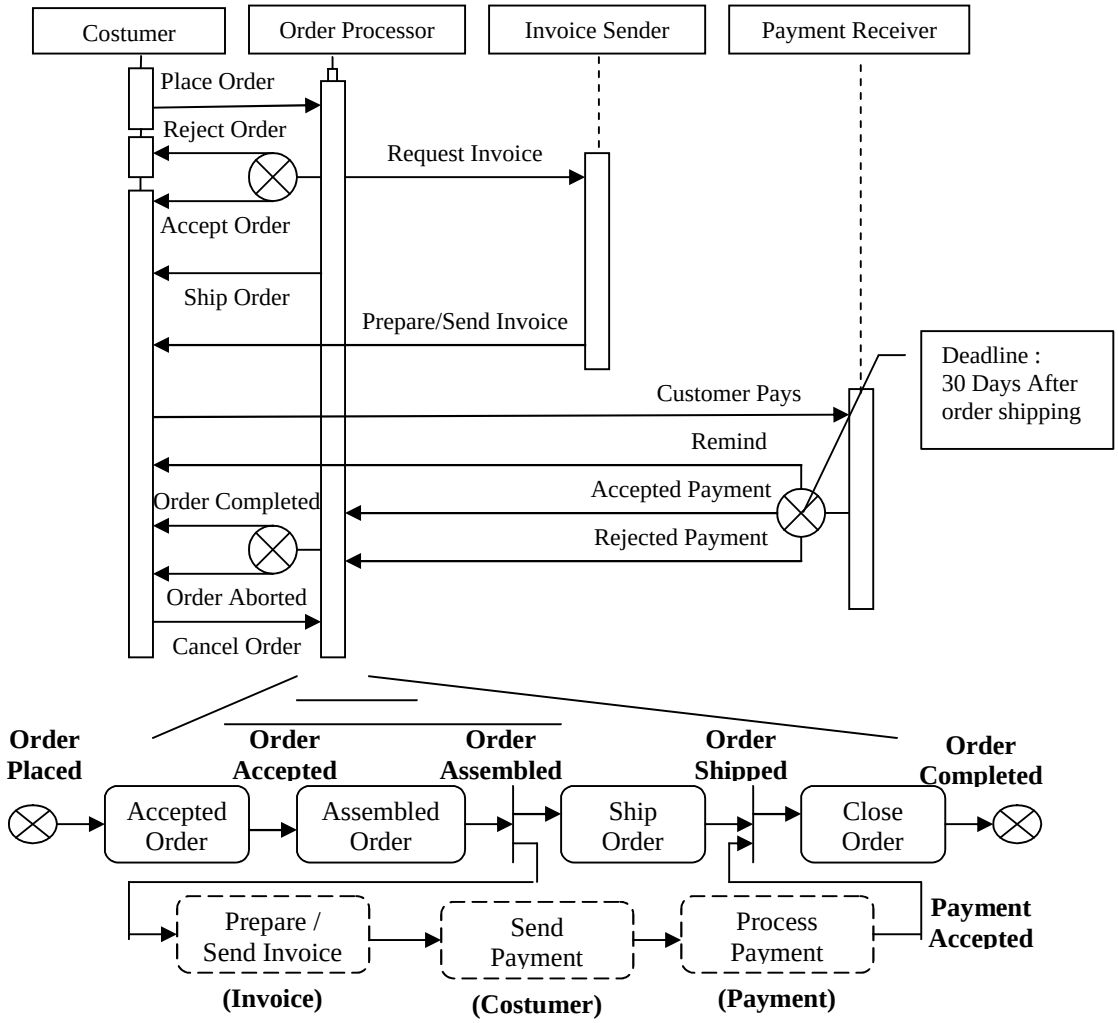

Figure 1: *A sequence and activity diagram that specifies order processing behavior for an Order agent***.**

Where the Order Processor accepts, assembles, ships, and closes the order, all these actions must be coordinated to avoid order may be assembled or shipped before it has been accepted. The receiver can perform the actions *Accept_Order and Reject_Order*. The Assembler will invoke concurrently *Ship_Order* and *Request_Invoice* actions when the order is assembled and so on.

In Maude, the general form required of rewrite rules used to specify the behavior of an object oriented system is as follows:

M1….Mn <O1:C1|attrs1>…<Om:Cm|attrsm>
=> <Oi1:C'i1|attrs'i1>…<Oik:C'ik|attrs'ik>
<Q1:D1|attrs''1>…<Op:Dp|attrs''p>
M'1….M'q
If C

Where the *Ms* are messages expressions, *i1,…,ik* are different numbers among the original 1,…m, and *C* is the rules condition. A rule of this kind expresses a communication event in which n message and m distinct object participate. The outcome of such an event is as follows:

- The messages *M1….Mn* disappear,
- The state and possibly even the class of the object *Oi,…,Ok* may change,
- All other objects *Oj* vanish,
- New objects *Q1,…Qp* are created,
- New messages *M'1,…M'q* are sent.

Object creation is typically initiated by means of a new message of the form *new(C |attrs)* which specifies the new object class and initialization values of its attributes and has the effect of creating a new object with those properties and with a fresh new name. Notice that, since some of attributes of an object as well as the parameters of messages can contain object names, very complex and dynamically changing patterns of communication can be achieved by rules of this kind.

In our case, several object modules have been defined to be included. Some of them are described below. The DEF module defines the *Definition* class, and all classes are instances of this class. It has an attribute *State* which can be null when an element has no state declared. Otherwise it will have a name and a value or list of possible values. Also, it defines a *sort* named *Attrib* which is a generic type that can be use to define any attribute in a specific class if has no a predefined *sort*. It also defines the conditions imposed by the message. That is, how to express the different event notification modes and the priority and/or the messages necessary to perform the actions imposed by the relation. Performing to the Order Processor example, the different modules composing the formal specification of the system are defined. The module Receiver represents the Receiver class. Each class is represented by means of an object module in *Maude*, where a class is defined as a subclass of *Definition* (defined in DEF) with its own attributes and the actions defined as messages. *Recvr* is a *Definition* subclass. The two actions *Accept_Order* and *Reject_Order* are represented as messages.

```
(omod Receiver is
Protecting DEF.
class Recvr | Accepted Opened: Attrib.
subclass Recvr < Definition .
msgs Accept_Order Reject_Order : Oid -> Msg .
var R, C: Oid .
rl[Accept_Order] : Accept_Order(R) < R : Recvr | Opened : false > < R : Recvr |Accepted : false >=>
                   < R : Recvr | Opened : true > < R : Recvr |Accepted : true >
                   New(A : Assembler | Assembled : false)
                   If Event(Eop , Place_Order (C)) . // EoP: End of Process
rl[Reject_Order] : Reject_Order(R) < R : Recvr | Opened : false > < R : Recvr |Accepted : false >=>
                   < R : Recvr | Opened : false > < R : Recvr |Accepted : false >
                   If Event(Eop , Place_Order (C)) .
endom)

(omod Assembler is
Protecting DEF.
class Assbl | Assembled: Attrib.
subclass Recvr < Definition .
msgs Assemble_Order Request_Invoice : Oid -> Msg .
var A, S, I : Oid .
rl[Assemble_Order] : Assemble_Order(A) <A : Assbl | Assembled : false >=> < A : Assbl | Assembled : true>
                   new (S : Shpr | Start_Date : D)
                   Request_Invoice (A) Ship_Order (S).
rl[Request_Invoice] : Request_Invoice(A) < A : Assbl | Assembled :true>=> < A : Assmbl | Assembled : true >
                   < I : Invoicer | Inv_amnt : 0> .
endom)
```

```
(omod Shipper is
Protecting DEF.
class Shpr | Start_Date, Shipped: Attrib.
subclass Recvr < Definition .
msgs Ship_Order  : Oid -> Msg .
var S : Oid .
rl[Ship_Order] : Ship_Order(S) < S : Shipr | Start_Date:D> < S : Shipr | Shipped: false>=>
                    < S : Shipr | Start_Date:D> < S : Shipr | Shipped: true> .
endom)
(omod PayReceiver is
Protecting DEF.
class PRecvr | Completed :Attrib.
subclass Recvr < Definition .
msgs Order_Completed Order_Aborted : Oid -> Msg .
var Pr : Oid .
rl [Order_Completed] : Order_Completed (Pr) < Pr : PRecvr | Completed: False>=>
                                < Pr : PRecvr | Completed: true>.
                                If Event(EoP , Accepted_Payment(P)) .
rl[Order_Aborted] : Order_Aborted(Pr) < Pr : PRecvr | Completed : false>=>
                                < Pr : PRecvr | Completed : false>
                                If Event(EoP , Rejected_Payment(P)) .
endom)
```

3. A Multi-Level Approach to MAS Test Case Generation

Agent-based systems can be considered, from tester's point of view, as a number of different levels of abstraction. These levels will be termed the algorithmic level, class, agent, society, and system levels. They are defined as follow.

- The algorithmic level considers code at method level. It concerns the manipulation made within a routine with respect of some data.
- The class level consists of the interactions of methods and data that are encapsulated within a class.
- The agent level considers the interactions of groups of cooperating classes (the agent composed of the agent base class and several behavior classes).
- The society level consists of the interactions of the overall results of different agents.
- The system level contains all code from all classes and main program necessary to run the entire system.

Of these levels, the first and the last are comparable to normal code and system testing with conventional imperative languages and systems. The second and third levels are perceived years ago by the object oriented community and it has led to the emergence of the XUnit testing frameworks. The founder of this family was JUnit [10], a regression testing framework written by Erich Gamma and Kent Beck. The fourth level is the one that will concern us here, although the society level will be subject of further research. But, just in order to provide a quick idea of the society test we could note that agent society testing is a kind of integration testing 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).

Once code is created, information is extracted from the agent classes by parsing them. This information includes such details as the names of each of member functions within its classes, their parameters, and their types.

First of all the white box testing of the agent structure should be considered (it is known to the test case designer how behaviors are related and their flow of control) where each single behavior is seen as a black box. In traditional programming paradigms, test cases are generated based on a coverage criterion. Coverage criteria such as *statement*, *branch*, *path* and *data flow* [8, 9] have been used to generate test cases. These coverage criteria cannot be used directly in multi-agent systems as the notions of statement, branch, path and data flow are not exactly the same as in the other programming paradigms. Despite the inability to directly use the traditional methods for multi-agent systems, many of these coverage criteria are transferable in principle. For instance, the subsequent coverage criteria are considered *Node coverage* is similar to *statement coverage* [11] in traditional testing. A test case is generated to execute a node. *Arc traversal coverage* is similar to *branch coverage* [11] in traditional testing. *Node path coverage* ensures that every path to every node is tested. Compared to traditional testing, if this criterion is restricted to only the end nodes, then, this criterion is similar to *path coverage* [11].

Applying those coverage criteria on the order processor agent, we shall, first, describe the internal structure of the agent as shown in figure 2. After that and based on the selected coverage criterion, test case designer can launch black box testing of the agent behavior. Indeed, in most common cases, an agent interacts

with the remaining part of the system using communications. Testing its behavior corresponds in generating of a series of test sequences that are, in reality, a combination, in an arbitrary order, of the agent communications acts (a series of messages). The test process becomes a searching problem, that is, a search for the order of messages with various parameters that yields errors. In this way, only test cases that invoke the first message in the sequence are generated. All the others messages in the sequence will be invoked automatically, because the outputs of one message are the inputs of subsequent one in the sequence. This lets testers build a test suite effortlessly in a cheaply way. At any time tester can stop the execution and evaluating its result. The specification of the result to be expected from message can be derived from the study of the specification (expected results, constraints violation, etc), while the specification could not be sufficient when agents interact with the environment (using sensors or actuators). In this case, more information should be extracted from other parts of the design. We usually perform this kind of testing by using a driver/stub agent when the agent under test is supposed to be a participant/initiator in the communication. The construction of these driver/stub agents is easily performed with the Agent Factory tool [9] that allows the composition of an agent starting from a repository of patterns.

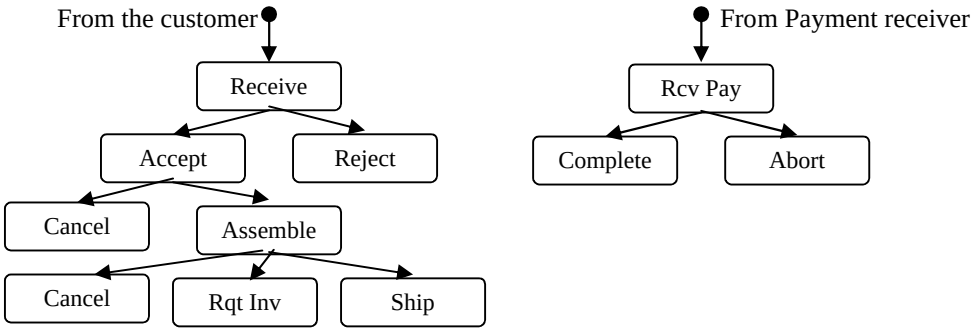

Figure 2: The Order Processor Plans Body.

Our framework is comprised of the following components (figure 3):

- The agent under test file which is created from the system by mean of instrumentation (we assume that the type of instrumentation will not affect the performance of the application),
- The agent information file, which store details of information extracted from the agent classes under test as well as test performed and results given,
- The test driver generator,
- The test driver which guides the testing process,
- The test sequences generator which uses the specification in terms of Maude Modules.

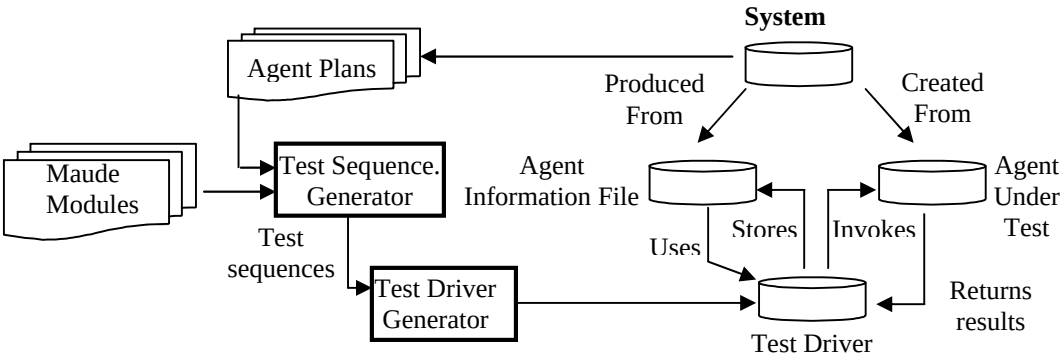

Figure 3: *Framework Structure.*

Now suppose that we have simple classes, written in some programming language, which provides the facilities of the order processor example. The parsing of this file yields information on the constructor and destructor of each class together with the member routines of each agent. This information is placed in the agent information file. The test sequences generator generates sequences of tuples representing combinations of messages specified in Maude object modules. The following is the algorithm for the test suite generation.

Algorithm: Generate Test sequence.
Input: The specification in term of Maude modules and agent plans derived from the implementation.
Output: A list of sequences and constraints on the test data for each test sequence.
Generate sequence
FOR each plan body
WHILE untested coverage criterion exists
- Select one of the defined coverage criterions
- Generate a list of all transitions that satisfies the selected coverage criterion and mark each transition as not covered.
- Generate a set of test data that invoke the first message in the sequence and verify that constraints are not violated (test data are generated from Maude modules specification).

ENDWHILE
ENDFOR
Return the test suite consisting of the test sequences together with the constraints on the test data.
END

Next, the test driver is generated with the necessary links into the agent under test. Upon the invocation of the framework, the test driver attempts to execute the list of generated test sequences on the instrumented version. Results comprising the tuple tested, the returned values of any message, and errors generated are output to a file.

4. Conclusion

In this paper, we have described a formal approach for specifying, developing and testing multi-agent systems. This approach based on the use Maude algebraic language which facilitates the descriptions and representations of relations between elements in a transparent manner with regards to the internal behavior of each agent class. We also used a set of coverage criterion that can guide the test sequences generation and compute the efficiency of the testing approach.

The aim of this paper is to provide tools for:

- Specifying agent interaction as a whole
- Expressing the interaction pattern among agent classes
- Representing the internal behavior of an agent

An issue, that needs to be addressed, deals with testing the MAS not at the single agent level but at the society level where a composition of different agents interact and their social behavior should be evaluated and compared with the expected results. Our framework can be a valid support for these kinds of test but improvements are desirable in order to give a more appropriate infrastructure.

5. References

[1] Arbad F. What do you mean coordination? *Bulletin of the Dutch Association for Theoretical Computer Science* (NVTI). March 1998.

[2] Bauer B., J. P. Müller, and J. Odell. Agent UML: A Formalism for Specifying Multiagent Interaction. *Agent-Oriented Software Engineering, Paolo Ciancarini and Michael Wooldridge eds*., Springer, Berlin, pp. 91-103, 2001.

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

[4] Blackburn M., R. Busser, A. Nauman. Why Model-Based Test Automation is Different and What You Should Know to Get Started. *Software Productivity Consortium, NFP*. 2004

[5]. Clavel M. and al. Maude: Specification and programming in rewriting logic. *Computer Science Laboratory. SRI International*. March. 1999.

[6] Cossentino M., L. Sabatucci, A. Chella - A Possible Approach to the Development of Robotic Multi-Agent Systems. *IEEE/WIC Conf. on Intelligent Agent Technology (IAT'03).* October, 13-17, Halifax Canada. 2003.

[7] Ferber J. and Gutknecht O.. Aalaadin: a meta-model for the analysis and design of organizations in multi-agent systems. *In ICMAS'98*, july 1998.

[8] FIPA Modeling Technical Committee – Home Page – Available at http://www.fipa.org/activities/modeling.html

[9] Halpern J. and Y. Moses, A guide to completeness and complexity for modal logics of knowledge and belief. *Artificial Intelligence*, 54:319–379. 1992

[10] JUnit – JUnit Home Page. Available at http://www.junit.org

[11] Myers G., The Art of Software Testing, Wisley: New York, 1979.

[12] Nwana H. S., “Software agents: An overview,” *The Knowledge Engineering Review* vol. 11(3), pp.205–245, 1996.
[13] Oates T., Nagendra Prasad M. V., and V. R. Lesser, “Cooperative information-gathering: A distributed problem-solving approach,” *IEE Proceedings—Software Engineering* vol. 144(1), pp. 72–88, 1997.
[14] Odell, J. Parunak H. Bauer B. Extending UML for Agents. In proceedings of the agent-Oriented Information Systems Workshop. 17th National Conference on Artificial Intelligence , 2000.
[15] Poslad S., P. Buckle, R. Hadingham. The FIPA-OS Agent Platform. Open Source for Open Standards. *Proc. of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*. Manchester, UK, April 2000, 355-368
[16] Rao A. S. and Georgeff M. P., ‘BDI agents: From theory to practice,” in: *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS-95)*, 1995, pp. 312–319.