A Novel Conformance Testing Technique for Agent Interaction Protocols

Farid Mokhati
Department of Mathematics and Computer Science LAMIS Laboratory, Tebessa University Oum el Bouaghi University- Algeria

Mourad Badri
Department of Mathematics and Computer Science Québec à Trois-Rivières University- Canada

Salim Zerrougui
Department of Mathematics and Computer Science Oum el Bouaghi University- Algeria

***Abstract*—Agent Interaction Protocols (AIP) play a crucial role in multi-agent systems development. AIP allow specifying interactions (sequences of messages) between agents. As agent-oriented development emerges, testing agent-based software is receiving increased research attention. We present, in this paper, a novel conformance testing technique for agent interaction protocols. The approach supports an incremental testing strategy that considers both agent and society testing of Multi-Agent Systems. It uses aspect-oriented technology to support and monitor the testing process. The proposed technique consists in two main phases: (1) Generating adequate test cases from a formal description of Multi-Agent Systems' behavior, and (2) Verifying the execution of test cases. The technique is supported by a visual tool (AIPTE: Agent Interaction Protocols Testing Environment). A case study is presented to illustrate the approach.**

Keywords—Multi-Agent Systems; Agent Interaction Protocol; Conformance Testing; Reduced Testing Sequence; Complete Testing Sequence; JADE; AspectJ

I. INTRODUCTION

Agent-oriented methodologies (e.g., PASSI [3], GAIA [30], SODA [18], INGENIAS [26], DSC-based [7, 8]) address, essentially, Multi-Agent Systems (MAS) analysis, design and implementation. These methodologies are more and more used for developing complex systems that require flexibility and robustness. Although Agent-oriented methodologies provide important solutions in MAS development, they often omit testing activities [20].

Testing activities represent an important task in the quality assurance process of MAS [21]. Many testing techniques have been proposed for traditional and object-oriented software systems. However, despite the rapid evolution of MAS, testing multi-agent systems as autonomous systems is still an open key area [28, 34]. In fact, only few proposals addressing MAS testing are proposed in the literature [22, 27].

Nguyen et al. [22] propose a reference framework that provides a classification of MAS testing levels (such as unit, agent, integration, system, and acceptance testing). Unit testing is concerned with testing all units that make up an agent, while agent testing is about testing the integration of the different modules inside an agent. Integration testing deals with testing the interactions between agents. System testing is about testing MAS as a system running at the target operating environment and, finally, acceptance testing focuses on testing MAS in the customer's execution environment and verifies that it meets stakeholder goals.

Among the few approaches published in the literature during recent years, which are concerned with MAS testing, only the approach proposed by Nguyen et al. [21] deals with all testing levels. Some approaches such as [4, 5, 23, 31-33] deal with agent-oriented unit and/or agent testing as useful techniques for MAS testing. Gomez-Sanz et al [9] address both agent and integration testing of multi-agent systems. Except the approach proposed by Nunez et al. in [23] that deals with conformance testing at the agent level, the other works are only focused on white box testing of MAS.

This paper presents a novel conformance testing technique for agent interaction protocols. The approach is supported by a tool (as a prototype) called AIPTE (Agent Interaction Protocols Testing Environment). Based on a formal specification written in Maude [2, 17], the proposed technique takes into account both agent and society testing (from unit to system testing). In fact, the approach is an extension of our previous work [19] that consists in generating testing sequences from a Maude formal specification of AIP. The testing sequences are defined as lists of: Transitions, Reduced Testing Sequences (RTS) and Complete Testing Sequences (CTS). Each testing sequence is described by: a starting state, an event triggering the sequence, a destination state and eventually a generated event. The different lists of sequences represent the first description used by the approach to begin the conformance testing process.

For testing the conformity between the implementation of an AIP and its Maude specification, we concentrate on input data and execution results. We consider as input data, the starting state and the event triggering the first transition of each sequence. By starting state we mean the agent attributes values before executing the transition. As execution results, we consider the destination state, the generated event and its type (intra, inter of extra-agent). So, we take into account the new agent attributes values which appear in the last transition of the testing sequence. All testing sequences (scenarios) are analyzed in order to extract all possible test cases. The developed tool AIPTE allows executing the implementation of an AIP using these test cases. A testing report is generated for helping users to correct the potential errors.

The contribution of our approach consists basically in using jointly formal methods and aspect technology together to support the conformance testing of AIPs. The technique

considers both micro (agent) and macro (societal) levels of MAS. Formal methods are, in fact, useful in having a precise specification and a healthy strategy for the testing process [14, 15]. Furthermore, the aspect technology is used to support the instrumentation of the code under test in order to monitor the testing process. This procures the advantage that the source code of the program under test remains unchanged. Indeed, traditional instrumentation techniques consist generally in introducing many lines of source code in the program under test. Those fragments of code may introduce involuntary faults [1].

AIP testing is performed using an incremental strategy. We start by transition testing (unit testing), followed by RTS testing (agent testing) and we finish by CTS testing (integration and system testing). The source code implementing a testing sequence is instrumented in order to capture, on the one hand, the execution trace of methods included in the sequence and, on the other hand, potential faults (differences between observed behavior under test and expected behavior). Aspects are used to capture dynamically a trace of the executed methods included in a given sequence.

The remainder of this paper is organized as follows. In Section 2, we give a brief overview of major related work. Section 3 gives a brief introduction to the Maude language. We present briefly, in Section 4, the testing sequences generation process we adopted. Section 5 illustrates the verification process we propose. In Section 6, we present the tool AIPTE using a concrete example. Section 7 gives some conclusions and future work directions.

II. Related Work

In recent years, only few approaches have been proposed in the literature in order to deal with multi-agent systems testing [4, 5, 9, 21, 23, 31-33].

Zhang et al. [31-33] propose a framework for automated unit testing of agent-based systems. The authors extend the framework PDT (Prometheus Design Tool) [25] developed within the Prometheus methodology [24] for incorporating a novel framework for model-based intelligent agent unit testing. The basic units for testing are identified as events, plans and beliefs. Nunez et al. [23] introduce an approach for agent conformance testing applied to autonomous e-commerce agents. The desired behaviors of the agents under test are presented by means of a new formalism, called utility state machine that embodies users' preferences in its states. A utility state machine is like a standard extended finite state machine with the addition of time and a utility function for each state. In essence, the utility function for a state represents what the agent is trying to achieve in this state; when in this state, the agent can exchange resources with other agents if such exchanges increase its utility [11].

The authors propose a strategy for checking whether an implementation of a specified agent behaves as expected. They use, for each agent under test, a special agent that takes the formal specification of the agent to facilitate it to reach a specific state. The operational trace of the agent is then compared to the specification in order to detect faults [22].

Ekinci et al. [5] propose an approach for goal-based agent unit testing. Each test goal is conceptually decomposed into three sub-goals: setup, goal under test, and assert. The first and last sub-goals prepare pre-conditions and check post-conditions respectively, while testing the goal under test. Coelho et al. [4] present a framework for unit testing of multi-agent systems inspired by JUnit and using Mock Agents. This work focuses on testing roles of agents. A Mock agent is an agent that communicates with just one agent: the Under Test Agent (UTA). Each Mock agent tests a single role of an agent. Gomez-Sang et al. [9] propose an approach for testing and debugging MAS interactions with INGENIAS. INGENIAS [10] is a development kit that provides facilities for MAS testing. This work focuses on agent and interaction testing levels. Testing is based on the assertion of the mental state of individual agents. This assertion covers as well the interaction execution, since its state is recorded into the mental state of its participants.

Nguyen et al. [21] introduce a comprehensive testing methodology called Goal-Oriented Software Testing (GOST) for MAS. The methodology covers all testing levels (from unit to acceptance level) and proposes a testing process that complements goal-oriented analysis and design. Furthermore, GOST provides a systematic way for deriving test cases from goal-oriented specifications and techniques to automate test cases generation and execution.

We present, in this paper, a novel approach for conformance testing of agent interaction protocols. The approach adopts an iterative and incremental strategy for MAS testing and uses aspect-oriented programming techniques for testing agent interaction protocols, which are implemented on JADE platform [6]. It covers several testing levels of MAS (form unit to system testing level). Furthermore, in order to test adequately MAS, it is important to have a precise specification and a healthy strategy for the testing process that takes into consideration various aspects inherent to the system [14, 15]. The proposed testing process is based on a high-level formal description written in Maude allowing facilitating the automation of the testing process. The approach is supported by an aspect-based visual tool called AIPTE.

III. Maude

Maude is a language for specifying and programming systems. It is based on rewriting logic [2, 16, 17]. Maude allows describing easily the intra and inter-object concurrency. Furthermore, Maude has its own model-checker that is used for checking system's properties. In rewriting logic, the logic formulas are called rewriting rules. They have the following forms: R:[t] → [t'] or R:[t] → [t'] if C. Rule R indicates that term t becomes (is transformed into) t'. On its second form, a rule could not be executed except that a certain condition C if verified. Term t represents a partial state of a global state S of the described system. The modification of the global state S of the system to another state S' is realized by the parallel rewriting of one or more terms that express the partial states. The distributed state of a concurrent system is represented as a term whose sub-terms represent the different components of the concurrent state.

IV. An Overview on the Test Sequences Generation Process

A. Basic concepts

We present, in this section, some basic concepts related to the proposed testing approach, namely, Transition, Reduced Testing Sequence (RTS) and Complete Testing Sequence (CTS).

1) Transition

A transition *T = <starting state Si, event Ek, execution of action ACi authorized in state Si, destination state Sj>* is the result of the execution of an action authorized in a given state *Si* of the agent after event *Ek* has appeared, making a change to a destination state *Sj* in the state-chart diagram (Fig. 1).

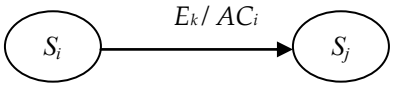

Fig. 1. Description of transition.

Let A be the identifier of an agent. The form of such a transition in Maude can be described by a rewriting rule of the following form:

rl [Action ACi] : Event (A, Ek) <A :Agent| state : Si> => <A :Agent| state : Sj> .

We consider, in the context of our approach, three types of events. An event can be:

- Internal to an agent (Event): meaning the result of a given processing (such as taking a decision).
- Inter-agent synchronization (InternalSynEvent): coming from another agent of the same system.
- External synchronization (ExternalSynEvent): coming from other external objects to the system (such as a Timer).

Let's take as example the following rewriting rule that describes, on the one hand, the reception of the external event *IsInitialized* by the agent *I* that plays the role Initiator in its initial state *StartI* with an empty mailbox and a list of its acquaintances *ACL*. On the other hand, this rule describes the sending of the internal message *ReceiveCFP* to the agent *P* and the agent *I* is becoming in its waiting state.

ExternalSynEvent (I, IsInitialized)

< I : Agent | PalyRole : Initiator, State : StartI, MBox:Empty, AcqList : ACL> => < I : Agent | PalyRole : Initiator, State : WaitI, MBox:Empty , AcqList : ACL> InternalSynEvent (P, ReceiveCFP, false) .

2) Testing sequence

A testing sequence is defined as a series of rewriting rules (transitions) for which starting the first rule allows the execution of the entire sequence. A testing sequence can either be reduced (*RTS*) or complete (*CTS*).

- Reduced Testing Sequence (RTS): An RTS is a sequence composed of one or several rewriting rules that makes an agent changing state from a starting state to a destination state in the state-chart diagram, and transits through a sequence of intermediate states.
- Complete Testing Sequence (CTS) : A CTS is a sequence composed of one or several rewriting rules that makes an agent changing state from an initial state to a final state in the state-chart diagram (if it exists).

Let's take as example the following complete testing sequence CTS4 :

RTSI1, RTSP1.1, RTSI2, RTSP2.1, RTSI2, RTSI5, (RTSP1.4 || RTSP2.5).

By *RTSI* and *RTSP* we mean RTS of Initiator, and RTS of Participant respectively. *RTSP1.1* and *RTSP2.1* describe RTS of the participants *P1* and *P2* respectively. *CTS4* (CTS number 4) expresses one among several feasible scenarios of Contract-Net protocol presented in our previous work [19] in which an initiator (I) and two participants (P1 and P2) are involved in the interaction. In this sequence, the initiator sends a call for proposal to the two participants by executing the reduced testing sequence *RTSI1*. After receiving the call for proposal, the two participants decide to submit proposals to the initiator by executing *RTSP1.1* and *RTSP2.1* respectively. After that, the sequence *RTSI2* is executed for storing the received proposals in the initiator' mail box. Once the deadline has expired, the initiator chooses the best proposal and sends an acceptance to the winner and a reject to the other using the sequence *RTSI5*. As soon as the initiator decision is made, the winner passes to its success state by executing the sequence *RTSP1.4* and the other passes to its failure state by executing the sequence *RTSP2.5*.

B. Test sequences generation process

Fig. 2 shows the process we adopted for the generation of the testing sequences.

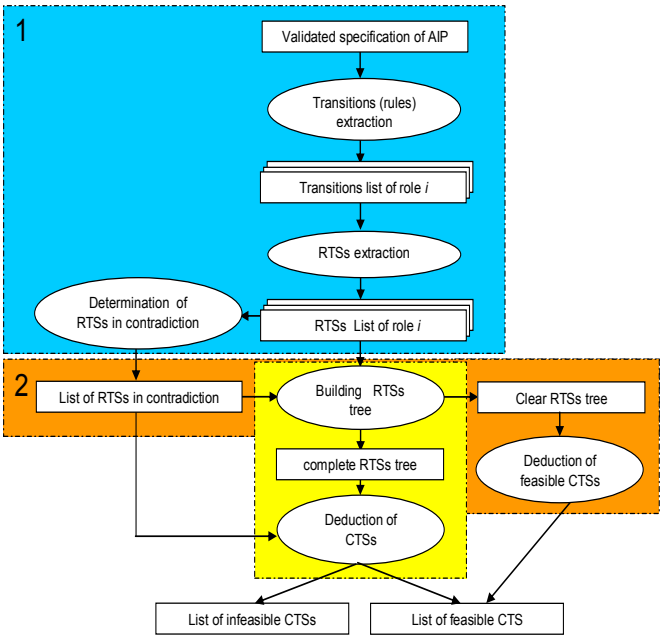

Fig. 2. Test sequences generating process.

Starting from a validated Maude formal description of the AIP (developed in one of our previous work [19]), the proposed method allows, in the first step, analyzing this description and extracting from it all the possible transitions (dashed box 1). Several lists of transitions are extracted. Each list is related to a specific role of an agent. After extracting the different lists of *RTSs*, the user can determine the different *RTSs* that are in contradiction to later eliminate the infeasible *CTSs*. Two *RTSs* are said contradictory if one or both contains at least one rule in contradiction with at least one rule of the other. Once the list of contradictory *RTSs* is determined, the user can generate a *RTSs* tree (while considering the contradictory *RTSs*) (dashed box 2), such that he can generate a complete *RTSs* tree (including the contradictory *RTSs*) (dashed box 3). While choosing the first alternative, the user can generate directly the feasible *CTSs* list. The generated list of *CTSs* while following the second option contains feasible and infeasible *CTSs*. Using the contradictory *RTSs* list, we can separate the list of feasible *RTSs* from those infeasible.

V. VERIFICATION PROCESS

The verification process we propose aims at verifying if the executed sequences are in accordance with the selected ones and if obtained results are conform to the expected ones. That is accomplished in two main phases. The first phase consists in the execution of the program for testing the selected testing sequences. During the second phase, the obtained results are compared to the expected ones in order to detect the potential faults by referring to the AIP's specification.

Fig. 3 illustrates the different steps of the methodology of the proposed approach. It represents a continuation of Fig. 2 for describing the verification process. The proposed approach accepts as input data three lists. A Transitions list, a Reduced Testing Sequences (RTS) list and a Complete Testing Sequences (CTS) list, which are generated from the Maude formal specification describing the AIP to be tested (Fig. 2). These testing sequences are analyzed in order to extract all the possible test cases. We are only considering the following input data: the starting state and the event triggering the first transition of each sequence. As execution results, we focus on the destination state, the generated event and its type (intra, inter of extra-agents). So, we take into account the new agent attributes values which appear in the last transition of the testing sequence.

The testing process of AIP is performed in an incremental way. We start by transition testing, followed by RTS testing and we finish by CTS testing. For that, the user has to select a sequence and implement it. The source code corresponding to the selected sequence must be implemented on the platform JADE and instrumented using AspectJ [13, 29]. The instrumentation process allows, as mentioned previously, capturing dynamically the execution trace of methods included in the sequence and detecting potential faults. Consequently, the instrumented source code contains the original source code written in JAVA and an aspect for capturing the execution trace of methods. This later is used to verify dynamically, if the executed sequence is conform to the selected one (sequence of executed methods, conditions). When a method included in the selected sequence is executed, the aspect captures the execution context of this method which will be used in the rest of the verification process. Once developed, the instrumented code must be executed using the generated test cases. The execution results are compared with the expected ones. Finally, a testing report is generated for helping the user to correct potential faults.

A. The aspect based instrumentation process

As mentioned above, source code implementing a testing sequence is instrumented in order to capture the execution trace of the methods constituting the sequence and potential faults. Note that for optimization reasons (we adopt the same strategy to control the execution of all testing sequences), we used only one aspect for all testing sequences, instead of using an aspect for each test sequence. The testing principle consists in intercepting testing sequences (transition, RTS or CTS) execution by using an aspect which contains:

- A pointcut which defines a set of joinpoints,
- An around advice code which enables checking method preconditions and postconditions before and after a joinpoint respectively.

The pointcut we defined (Fig. 4, line 2) determines all transitions (RI and RP), RTSs and CTSs which will be intercepted. In order to capture the execution of Transitions we used the *call* pointcut. However, we used the *execution* pointcut for RTSs and CTSs to allow keeping track of transitions that are already executed inside RTS or CTS. The around advice code associated to the pointcut trace (Fig. 4, lines [3,..,13]) verifies preconditions and postconditions for each testing sequence.

After defining the sequence to be tested (Fig. 4, line 4), we proceed to loading its initial state and the corresponding expected final state from the test selection tables (Fig. 4, line 5). To execute the intercepted method, its precondition must be true (Fig. 4, line 7). This latter compares the initial state of the implemented testing sequence to the intercepted method arguments. These arguments are returned by *ThisJoinPoint* (Fig.4, line 6) which contains all information (method execution context) about the intercepted method (its name, its class, its super class, its parameters and their types, its return type, etc). After verifying the precondition, the method continues its execution (Fig. 4, line 8). The variable *ret* contains results values (sequence final states) returned by the intercepted method. These results values have are compared to the expected sequence final state in order to verify the post condition of the executed method (Fig. 4, line 10). In line 11, we show and display the results obtained after executing the implemented sequence.

B. Testing criteria

Testing activities often take considerable time. The problem of testing is about the moment in which the testing process has to be stopped. For resolving this problem, it is useful to define adequate testing criteria for deciding whether the program under test has been well tested or not for a specific testing criterion [12]. In the context of our approach, we introduce three testing criteria, namely, Transition coverage criterion, RTS coverage criterion and CTS coverage criterion. The two first criteria are defined to be used during unit and

agent testing (testing of agents' individual behavior). The third criterion is defined in order to test agents' collective behavior (society testing: integration and system).

In all criteria definitions, we use, *TC* as a set of test cases, *TS* as a set of transitions, *RTS-S* a set of reduced testing sequences (*RTSs*), and *CTS-S* as a set of complete testing sequences (*CTSs*).

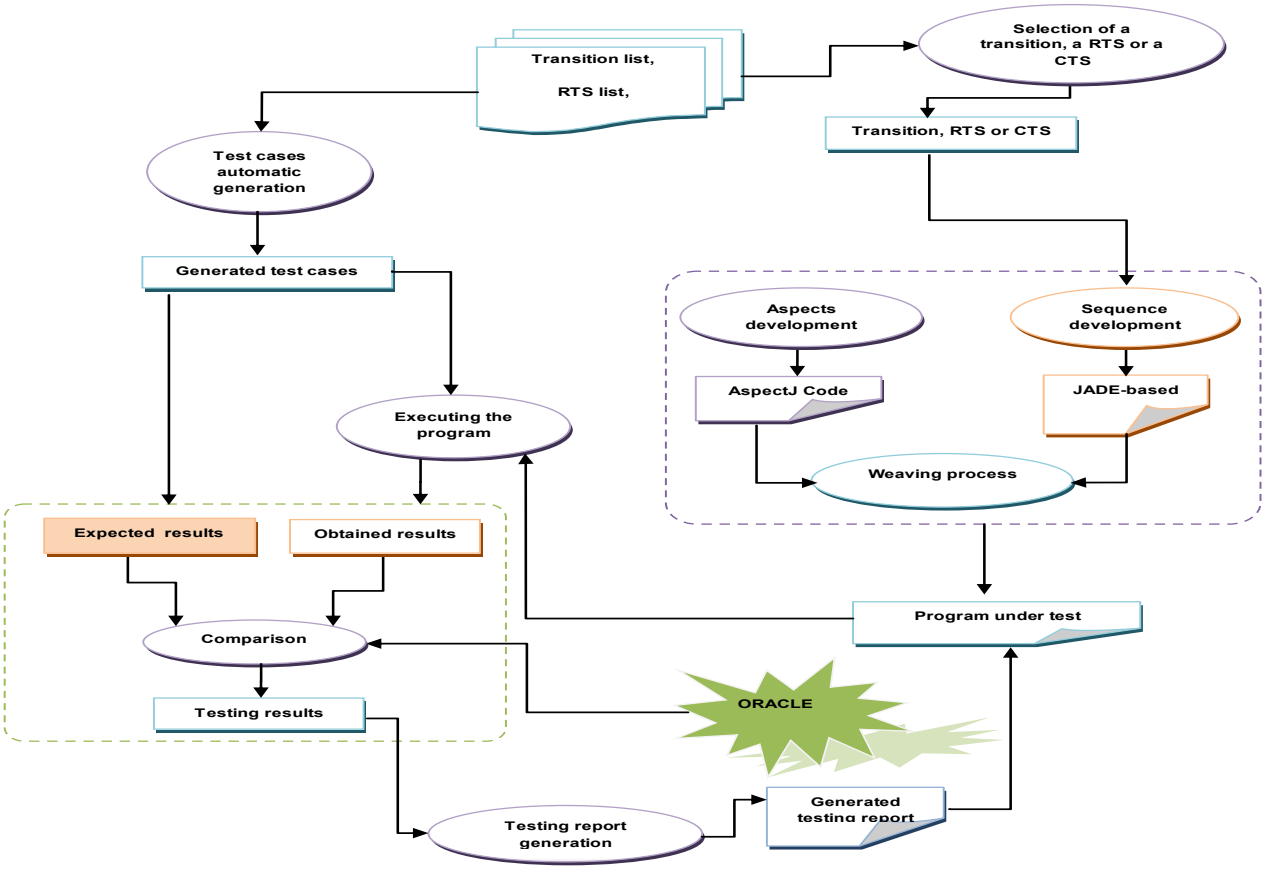

Fig. 3. The methodology adopted by AIPTE.

```
1 privileged aspect TestAspect {
2 pointcut trace(): execution(* *..*.CTS*(..))||execution(*
  *..*.RTS*(..))||call(* *..*.RI*(..))||call(* *..*.RP*(..));
3 Object around(): trace(){
4 /* verify if the sequence is a transition, an
  RTS or a CTS*/
5 /* load the sequence initial states of the
  sequence and the corresponding expected  final
  states from the test selection tables*/
6 Object[] parameters = thisJoinPoint.getArgs(); /*
  intercepting method arguments */
7 /*preconditions (method arguments == sequence
  initial state)*/
8 Object ret = proceed();
9 /* retrieve results from the object ret which
  represent sequence final states*/
10 /* post-conditions (sequence final states ==
  expected sequence final states)*/
11 // display of results
12  return ret;
13        }
```

Fig. 4. Part of the developed aspect.

- *Transition coverage criterion*: This criterion forms the basis of the unit testing of interacting agents. The tester must test at least once each transition (rewriting rule) describing an agent's behavior.

Criterion: The set of test cases TC must contain tests which cause the execution of each transition in TS.

- *RTS coverage criterion*: During its life cycle, an agent may accomplish many consecutive transitions without waiting external events. This transition series constitutes a RTS (Reduced Transition Sequence). This criterion is used for agent testing. After testing all transitions, we pass in a second phase to testing all RTSs, at least once for each one.

Criterion: The set of test cases TC must contain tests which cause the execution of each RTS in RTS-S.

- *CTS coverage criterion*: A CTS (Complete Testing Sequence) represents a series of RTSs relative to the different interacting agents. It allows to testing the agents collective behavior (society testing). We adopt an incremental testing approach. Once the unit and agent testing are accomplished, the tester passes to the society testing using CTS coverage criteria.

Criterion: The set of test cases TC must contain tests which cause the execution of each CTS in CTS-S.

C. Agent testing

This step focuses on unit and agent testing through testing all program fragments relative to the different RTSs corresponding to an agent. Agent testing is performed by testing each RTS independently. Testing an RTS represents the testing of all its transitions in an incremental way (all transitions may be tested separately). So, we test the first transition, followed by the second one until testing the last one. The testing process of an RTS is illustrated by the Fig. 5.

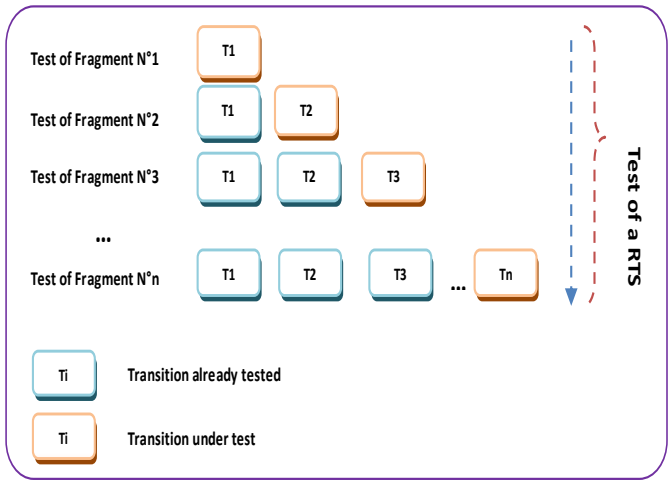

Fig. 5. Testing process of an RTS.

Testing steps of a RTS:

Testing an RTS is accomplished following five steps.

1) Selecting the RTS to be tested by the user.
2) Developing or opening (if it exists) the fragment of code JAVA corresponding to the transition to be tested.
3) Instrumenting this code with Aspect-J by using the proposed aspect.
4) Testing of the instrumented code by using the corresponding test data.
5) Displaying the test report.

The steps 2, 3, 4 and 5 are to be done for all transitions of an RTS. The order of transitions testing must be respected (transition1, transition2,… transition n).

D. Society Testing

The aim of society testing is the verification of agents' collective behavior. It represents the second testing phase of our approach. This phase is accomplished by testing each CTS independently. The test of a CTS must be done by testing all its RTSs in an incremental way (integration testing). We begin by testing the first RTS followed by the second one, and so on, until testing the last RTS of the CTS. The testing process of a CTS is illustrated by the Fig. 6.

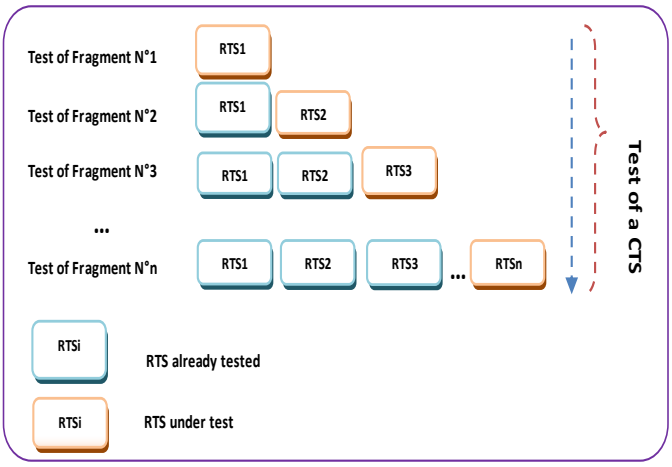

Fig. 6. Testing process of a CTS.

Testing steps of a CTS :

Just as the agent testing process, the test of a CTS is performed following five steps.

1) Selecting among the possible CTSs, the CTS to be tested.
2) Developing or opening (if it exists) the fragment of code JAVA corresponding to the RTS to be tested.
3) Instrumenting this code with Aspect-J by using the proposed aspect.
4) Testing of the instrumented code by using the corresponding test data.
5) Displaying the test report.

The steps 2, 3, 4 and 5 are to be done for all RTSs of a CTS. The order of RTSs testing must be respected (RTS1, RTS2,… RTSn). Once the integration testing is accomplished successfully, the user can test the whole Jade program (system testing) in order to evaluate its compliance with its specified requirements.

VI. Tool Presentation

AIPTE is an environment which automates the proposed verification process (see Fig. 3). As input, the tool takes a list of testing sequences (transitions, RTSs and CTSs). These sequences are analyzed in order to extract test cases.

A. Functioning Principle

As mentioned previously, the developed environment offers for testers several tools allowing testing Jade programs which implement AIPs. For testing a Jade application, tester must perform the following steps:

1) Extraction of the different sequences lists from the basic testing sequences generated by the tool developed in our previous work.

2) Selection of a not tested agent which plays a specific role, if it exists, else go to 6.

3) Generating the possible test cases form the RTS list corresponding to the selected agent.

4) Testing of all RTSs (by testing all transitions which constitute the under test RTS in an incremental way). This activity is accomplished in four steps:

a) Development of the Jade program which implements the RTS to be tested.

b) Instrumentation of the developed code using the proposed aspect.

c) Execution of the instrumented code using test data.

d) If the test report indicates errors, correct the program code and go to 4.3.

5) Go to 2.

6) Selection of a not tested CTS if it existes, else go to 10.

7) Generation of the possible test cases form CTSs list.

8) Testing of all CTSs (by testing all RTSs which constitute the under test CTS in an incremental way). This activity is accomplished in four steps:

a) Development of the Jade program which implements the CTS to be tested.

b) Instrumentation of the developed code using the proposed aspect.

c) Execution of the instrumented code using test data.

d) If the test report indicates errors, correct the program code and go to 8.3.

9) Go to 6.

10) End.

B. Case Study : Contact Net Protcol

This section illustrates the functioning of AIPTE through testing the FIPA contract Net protocol (Fig. 7).

As soon as it is called, the Initiator agent sends a call-for-proposal to a Participant agent. Before a given deadline, the Participant agent can send to the Initiator agent a proposal, refuse to send that proposal (refuse) or indicate that it did not understand properly (not-understood).

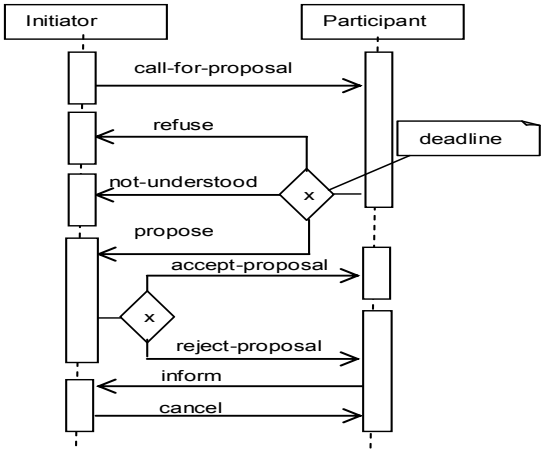

Fig. 7. The FIPA Contact Net protocol.

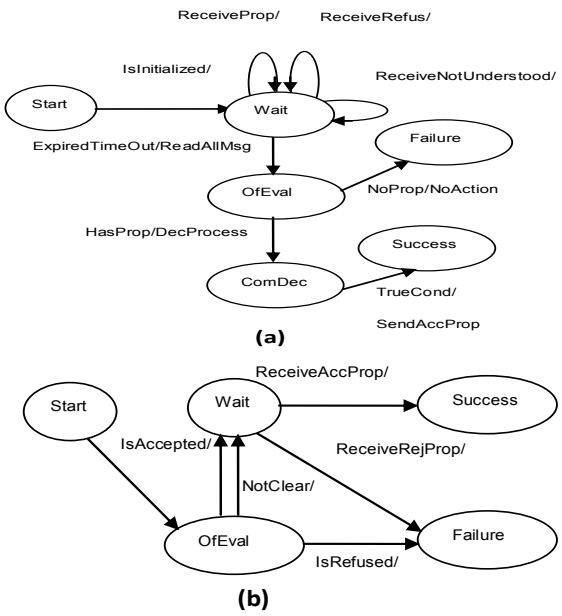

Fig. 8. Internal behaviour of the agents Initiator and Participant.

The proposal formulated by the Participant agent can be accepted or refused by the Initiator agent. When it receives a positive answer to its proposal (accept-proposal), the Participant agent informs the Initiator agent of the execution of its proposition. However, the Initiator agent can cancel the execution of the proposition at any given time. Fig. 8(a) and Fig. 8(b) present respectively the internal behaviour of agents Initiator (I) and Participant (P) of Fig. 7.

C. Implementation

In this section, we illustrate the developed tool using the following complete testing sequence CTS4 quoted above (CTS4 = *RTSI1, RTSP1.1, RTSI2, RTSP2.1, RTSI2, RTSI5, (RTSP1.4 || RTSP2.5))*.

Fig. 9 and Fig.11 illustrate the RTSs lists relative to Initiator and Participant, show the Jade-based Java code implementing RTSI1 and RTSP1and the used aspect code. These figures also illustrate selection tables that contain testing data (input data, and the expected results). Testing reports relative to the execution of the two sequences using testing data which are extracted from their corresponding testing cases are illustrated by Fig. 10 and Fig. 12. These reports show that the obtained results are in accordance with the expected ones and the executed sequences are also conform to the expected ones.

RTS1 (because we are testing the Initiator, we wrote RTS1 instead of RTSI1) is composed of only one transition (RI1) which represents the rewriting rule describing the call for proposal. Fig. 10 shows that the executed RTS1 has as input data: StartI (the initial state of the Initiator), Empty (the initiator mailbox is empty), IsInitialized and ExternalSynEvent (*IsInitialized* is an external synchronization event), and as excepted results: WaitI (the Initiator passes to its waiting state), Empty (the initiator mailbox is empty), ReceiveCFP and IntrenalSynEvent (*ReceiveCFP* is an inter-agents synchronization event). The results obtained after executing

this sequence are illustrated by Fig. 10. They are in accordance with the expected ones. Fig. 10 also illustrates that the executed sequence (RTS1) is in accordance with the expected one.

In the same way we tested the first RTS for the Participant (Fig. 11), RTS1_4 which denotes that it is composed of the two rewriting rules RP1 and RP4. This sequence describes that the participant decides to submit a proposal. Fig. 12 shows that the obtained results after executing RTS1_4 are in accordance with the expected ones. This sequence has as input: StartP (initial state of the Participant), ReceiveCFP and InternalSynEvent (*ReceiveCFP* is an inter-agents synchronization event). These input data are extracted from the left part of the first rewriting rule (RP1) of the RTS1_4. The expected results of this sequence are extracted from the right part of the last rewriting rule (RP4) of RTS1_4. For this sequence the expected results are: WaitP (the Participant passes to its waiting state), ReceivePropose and InternalSynEvent; *ReceivePropose* is an inter-agents synchronization event which expresses that the Participant has sent a proposal to the Initiator. Fig. 12 also illustrates that the executed sequence (RTS1_4) is in accordance with the expected one.

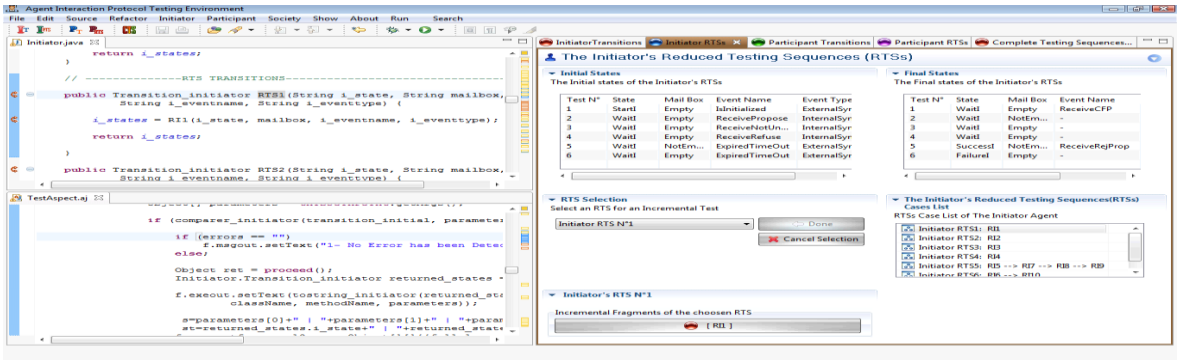

Fig. 9. Testing of the agent *Initiator*

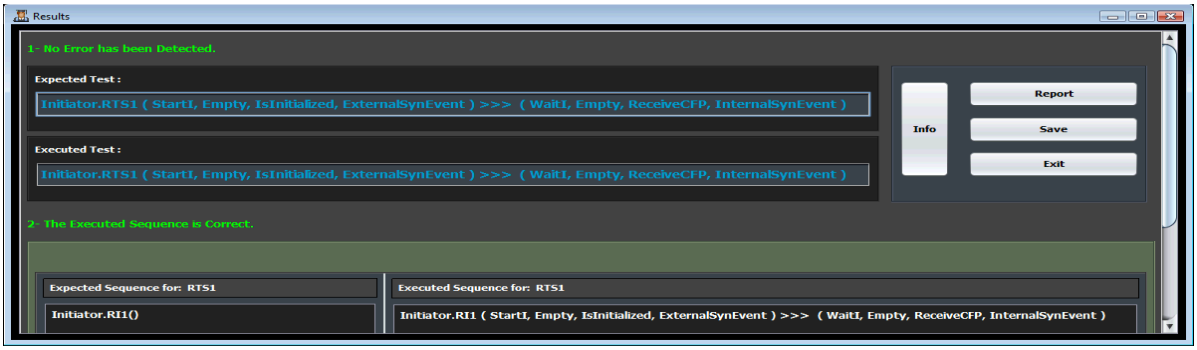

Fig. 10. Execution results of the RTSI1.

After testing all RTSs related to the agents Initiator and Participants, the tester may pass to society testing (test of collective behavior). Fig. 13 visualizes the execution of the scenario CTS4 quoted above relative to the society testing. This figure shows the list of different complete testing sequences and mentions that CTS4 is the one subtest (part 3). Parts 1 and 2 show a part of the Jade-based Java code which implements the sequence CTS4 and the AspectJ code corresponding to CTSs.

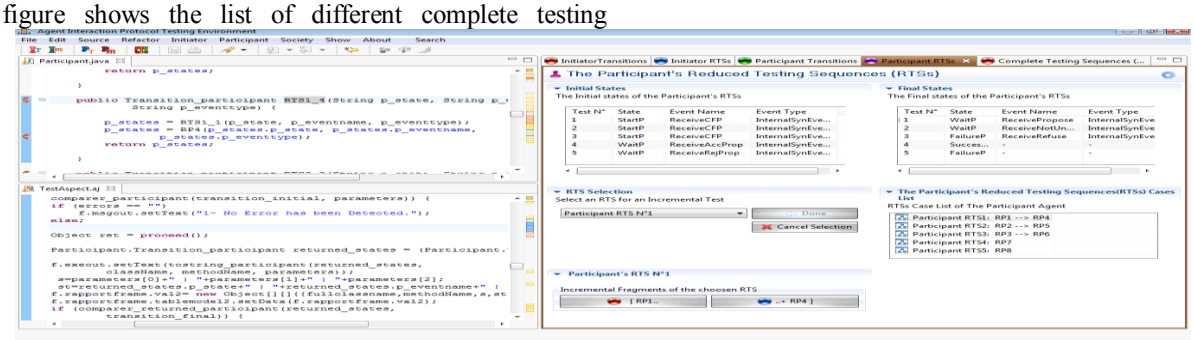

Fig. 11. Testing of the agent Participant.

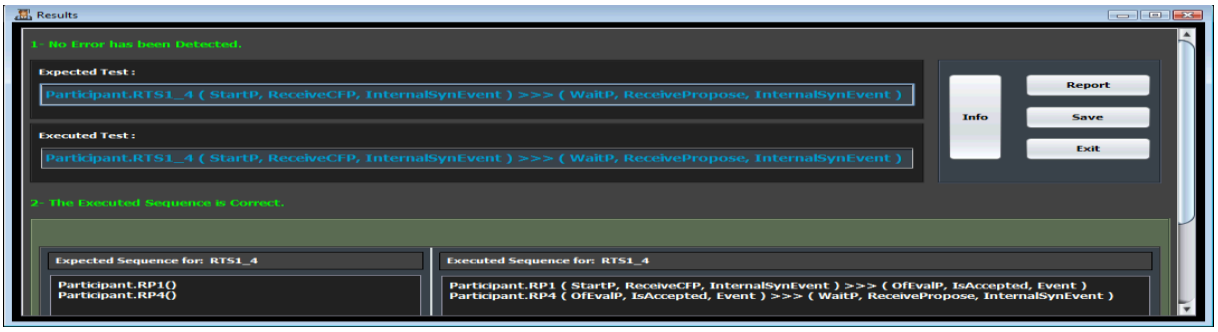

Fig. 12. Execution results of the RTSP1.

Part 4 illustrates the console that displays the result of launching JADE and the different messages exchanged between agents running. This window shows the two proposals submitted by participants P1 and P2 with the cost of 8 and 0 respectively. The generated testing report (Fig. 14) indicates that the obtained results are in accordance with the expected ones. Furthermore, it illustrates the execution trace (execution sequence) of the different methods which implement the different transitions. The executed sequence is conforming to the expected one.

The input data for CTS4 are extracted from the left part of the first rewriting rule (RI1) of its first RTS (RTSI1). The expected results of this sequence are extracted from the right part of the two last rewriting rules (RP7 and RP8) of its two last RTSs (*RTSP1.4* and *RTSP2.5* respectively) which are executed in parallel. Fig. 14 shows that CTS4 has as input data: StartI (the initial state of the Initiator), Empty (the initiator mailbox is empty), IsInitialized and ExternalSynEvent (*IsInitialized* is an external synchronization event) and, as expected results: ((SuccessP1, -,-) (FailureP2, -,-) or (FailureP1, -,-) (SuccessP2, -,-)). This expresses that the results we have to obtain are either the Participant P1 in its success state and the Participant P2 in its Failure state, or vice versa. The obtained results show that P1 is the winner (because it proposed the cost 8 and it is in its Success state) and P2 is the loser (because it proposed the cost 0 and it is in its Failure state). The symbol (-) expresses that no event has been sent form the agent state.

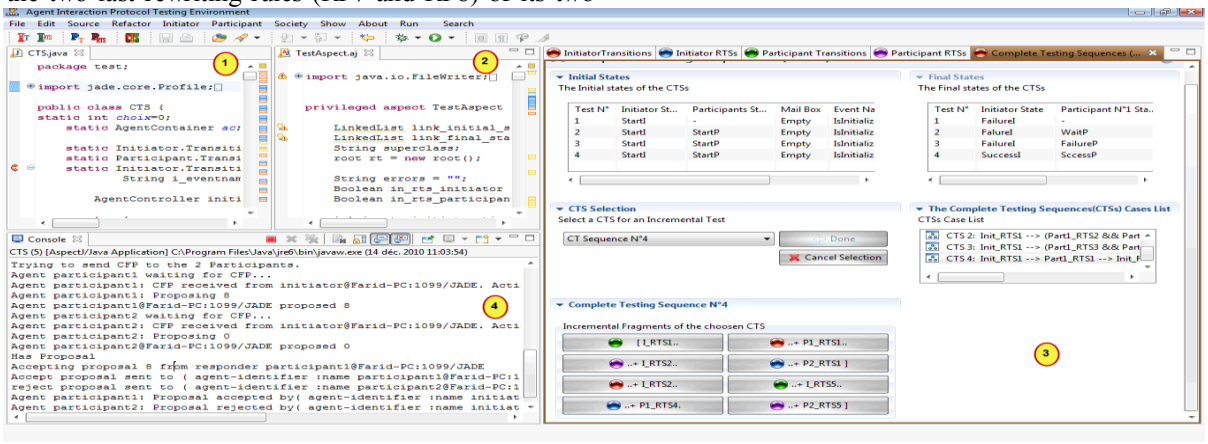

Fig. 13. Society Testing.

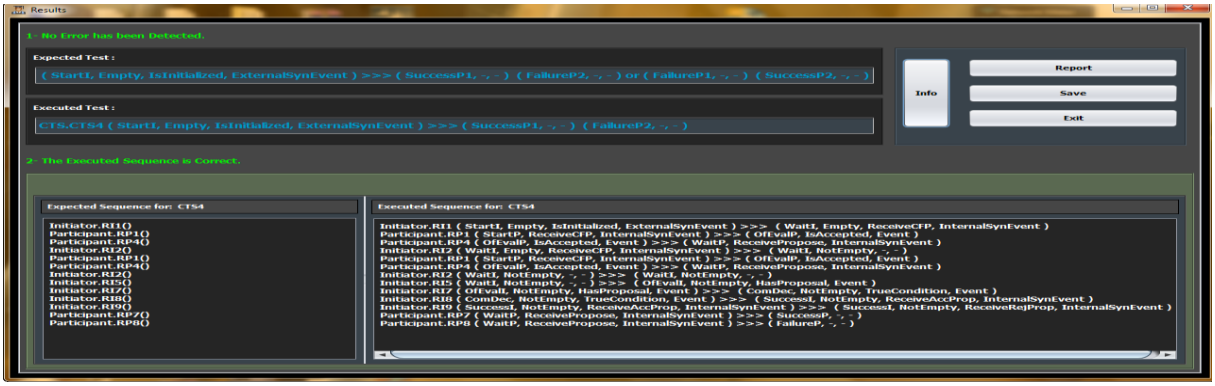

Fig. 14. Execution Results of CTS4.

VII. CONCLUSION

Conformance testing is a well-known approach for the validation of software systems. It consists of checking the consistency between two representations of a system. Testing complex systems such as multi-agent systems is a difficult, but necessary task. Nowadays, few studies have addressed this issue by proposing approaches dedicated to testing different agents models.

In this context, we have proposed in this paper a novel conformance testing approach of MAS and specifically for testing agent interaction protocols implemented on JADE. To monitor and control the execution of the agents under test while capturing potential errors and execution trace methods involved in a testing sequence, the source code that implements the testing sequence is instrumented using the aspect paradigm.

We developed a visual tool called AIPTE that assists the tester during an application testing by providing an interactive interface. AIPTE adopts an incremental testing strategy. It takes into account both agent and society testing (form unit to system testing level). Agent testing is performed by testing all Transitions and all Reduced Testing Sequences (RTS) relatives to the agent under test, while society testing requires testing all Complete Testing Sequences (CTS).

As future work, we plan to extend the proposed approach to support testing argument-based negotiation protocols.

REFERENCES

[1] B. Beizer, Software Testing Techniques, International Thomson Comuter Press, 1990.

[2] M. Clavel and al. “Maude : Specification and Programming in Rewriting Logic”. Internal report, SRI International, 1999.

[3] M. Cossentino, “From Requirements to Code with the PASSI Methodology,” In Agent-Oriented Methodologies, Eds. B. Henderson-Sellers and P. Giorgini, Idea Group Inc., Hershey, PA, USA, 2005, pp. 79–106.

[4] R.S. Coelho et al, “Unit Testing in Multi-agent Systems using Mock Agents and Aspects”. Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems. Shanghai, China 2006. pp. 83-90.

[5] Ekinci, E.E., Tiryaki, A.M., Cetin, O., Dikenelli, O.: Goal-Oriented Agent Testing Revisited. In: Proc. of the 9th Int. Workshop on Agent-Oriented Software Engineering, pp. 85–96 (2008).

[6] Fabio L.B et al, ‘’Developing Multi-Agent Systems with JADE’’. Wiley Edition April 2007.

[7] G. Fortino, W. Russo, and E. Zimeo, “A Statecharts-based Software Development Process for Mobile Agents”, In Information and Software Technology, 46(13), pp.907-921, Elsevier, Amsterdam, The Netherland, 2004.

[8] G. Fortino, A. Garro, and W. Russo, “An Integrated Approach for the Development and Validation of Multi Agent Systems”, In Computer Systems Science & Engineering, 20(4), pp. 94-107, CRL Publishing Ltd., Leicester (UK), Jul. 2005.

[9] Gomez-Sanz, J.J., Botıa, J., Serrano, E., Pavon, J.: Testing and debugging of MAS interactions with INGENIAS. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 199–212. Springer, Heidelberg (2009).

[10] Gómez-Sanz, J. & Pavón. INGENIAS Development Kit (IDK) Manual, vers. 2.5.2. Universidad Complutense Madrid (2005).

[11] R.M. Hierons: Editorial: Validating our findings. Softw. Test., Verif. Reliab. 15(4): 209-210 (2005)

[12] A. Jefferson Offutt, Shaoying Liu, Aynur Abdurazik, Paul Ammann: “Generating test data from state-based specifications”. Softw. Test., Verif. Reliab. 13(1): 25-53 (2003).

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In Proc. Europ. Conf. Object-Oriented Programming. Springer, 2001.

[14] S-K. Kim and al. ‘’A UML Approach to the Generation of Test Sequences for Java- based Concurrent Systems’’. In Proceedings of the 2005 Australian Software Engineering Conference (ASWEC’05).

[15] B. MERMET. “Formal model of a Multiagents System’’ , TRAPPL R., Ed., Cybernetics and Systems, Austrian Society for Cybernetics Studies, 2002, p. 653-658.

[16] J. Meseguer, “A Logical Theory of Concurrent Objects and its Realization in the Maude Language”. In G. Agha, P. Wegner, and A. Yonezawa, Editors, Research Directions in Object-Based Concurrency. MIT Press, 1992.

[17] J. Meseguer, “Software Specification and Verification in Rewriting Logic”. In M. Broy and M. Pizka, editors, Models, algebras and logic of engineering software, pages 133-193. IOS Press, 2003. ISBN 1-58603-342-5.

[18] A. Molesini, A. Omicini, E. Denti, and A. Ricci, “SODA: A Roadmap to Artefacts,” 6th International Workshop on Engineering Societies in the Agents World VI, (ESAW 2005), Kusadasi, Aydin, Turkey, October 2005. LNAI 3963, Springer, 2006.

[19] F. Mokhati, M. Badri, L. Badri, F. Hamidane, S. Bouazdia: Automated testing sequences generation from AUML diagrams: a formal verification of agents' interaction protocols. IJAOSE 2(4): 422-448 (2008)

[20] M. Moreno, J. Pavón, A. Rosete: Testing in Agent Oriented Methodologies. IWANN (2) 2009: 138-145.

[21] Nguyen, C.D., Perini, A., Tonella, P.: Goal-oriented testing for MASs. Int. J. Agent-Oriented Software Engineering 4(1), 79–109 (2010).

[22] Nguyen, C.D., Anna Perini, Carole Bernon, Juan Pavón, John Thangarajah “Testing in multi-agent systems”. In Proceeding AOSE'10 Proceedings of the 10th international conference on Agent-oriented software engineering Pages 180-190, Springer-Verlag Berlin, Heidelberg ©2011.

[23] Nunez, M., Rodrıguez, I., Rubio, F.: Specification and testing of autonomous agents in e-commerce systems. Software Testing, Verification and Reliability 15(4), 211–233 (2005).

[24] L. Padgham and M. Winikoff. Prometheus: A pragmatic methodology for engineering intelligent agents. In Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies, pages 97.108, Seattle, Nov. 2002.

[25] Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. John Wiley and Sons, Chichester (2004)

[26] [26] J. Pavón, J. Gómez-Sanz, and R. Fuentes, “The INGENIAS Methodology and Tools,” In Agent-Oriented Methodologies, Eds. B. Henderson-Sellers and P. Giorgini, Idea Group Publishing, 2005, pp.236-276.

[27] Tomas Salamon. A Three-Layer Approach to Testing of Multi-agent Systems. G.A. Papadopoulos et al. (eds.), Information Systems Development, DOI 10.1007/b137171_41, Springer Science + Business Media, LLC 2009.

[28] Weiss, L.G.: Autonomous Robots in the Fog of War. IEEE Spectrum 48(8), 30–57 (2011), doi:10.1109/MSPEC.2011.5960163.

[29] Tao Xie, Jianjun Zhao, Darko Marinov, David Notkin: Detecting Redundant Unit Tests for AspectJ Programs. ISSRE 2006: 179-190.

[30] F. Zambonelli, N. Jennings, and M. Wooldridge, "Developing multiagent systems: The Gaia methodology," ACM Trans. Software Eng. Meth., vol. 12, no. 3, pp.417-470, 2003.

[31] Zhang, Z., Thangarajah, J., Padgham, L.: Automated unit testing for agent systems. In: 2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2007), Barcelona, Spain, pp. 10–18(2007)

[32] Zhang, Z., Thangarajah, J., Padgham, L.: Automated unit testing

intelligent agents in pdt. In: AAMAS (Demos), pp. 1673–1674 (2008)

[33] Zhang, Z., Thangarajah, J., Padgham, L.: Model based testing for agent systems. In: AAMAS, vol. (2), pp. 1333–1334 (2009)

[34] Zoltán Micskei, Zoltán Szatmári, János Oláh, István Majzik: A Concept for Testing Robustness and Safety of the Context-Aware Behaviour of Autonomous Systems. KES-AMSTA 2012: 504-513.