Multi-Agent System Testing: A Survey

Zina Houhamdi
Software Engineering Department, Faculty of Science and IT
Al-Zaytoonah University
Amman, Jordan

***Abstract*—In recent years, agent-based systems have received considerable attention in both academics and industry. The agent-oriented paradigm can be considered a natural extension to the object-oriented (OO) paradigm. Agents differ from objects in many issues which require special modeling elements but have some similarities. Although there is a well-defined OO testing technique, agent-oriented development has neither a standard development process nor a standard testing technique. In this paper, we will give an introduction to most recent works presented in the area of testing distributed systems composed of complex autonomous entities (agents). We will provide pointers to work by large players in the field. We will explain why this kind of system must be handled differently than less complex systems.**

Keywords-Software agent; Software testing; Multi-agent system testing.

I. INTRODUCTION

As the technology evolving, the more we are driven towards abstraction and generalization. The increasing use of Internet as the spine for all interconnected services and devices makes software systems highly complex and in practice open in scale. These systems nowadays need to be adaptive, autonomous and dynamic to serve different user's community and heterogeneous platforms. These systems are developed very fast in past few decades. They are changed continuously to satisfy the business and technology modifications.

Software agents are key technologies to meet modern business needs. They offer also an efficient conceptual methodology to design such complex systems. In practice, research on software agents' development and Multi-Agent System (MAS) has become too large and used in different active area focusing mainly on architectures, protocols, frameworks, messaging infrastructure and community interactions. Thus, these systems receive more industrial attention as well.

Since these systems are increasingly taking over operations and controls in organization management, automated vehicles, and financing systems, assurances that these complex systems operate properly need to be given to their owners and their users. This calls for an investigation of appropriate software engineering frameworks, including requirements engineering, architecture, and testing techniques, to provide adequate software development processes and supporting tools.

Software agents and MAS testing is a challenging task because these systems are distributed, autonomous, and deliberative. They operate in an open world, which requires context awareness. In particular, the very particular character of software agents makes it difficult to apply existing software testing techniques to them. There are issues concerning communication and semantic interoperability, as well as coordination with peers. All these features are known to be hard not only to design and to program [3], but also to test.

There are several reasons for the increase of the difficulty degree of testing MAS:

- 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;
- Irreproducibility effect, since we can't ensure 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 impossible to reproduce it each time [22].
- They are also non-deterministic, since it is not possible to determine a priori all interactions of an agent during its execution.
- Agents communicate primarily through message passing instead of method invocation, so existing object-oriented testing approaches are not directly applicable.
- Agents are autonomous and cooperate with other agents, so they may run correctly by themselves but incorrectly in a community or vice versa.

As a result, testing software agents and MAS asks for new testing methods dealing with their specific nature. The methods need to be effective and adequate to evaluate agent's autonomous behaviors and build confidence in them. From another perspective, while this research field is becoming more advanced, there is an emerging need for detailed guidelines during the testing process. This is considered a crucial step towards the adoption of Agent-Oriented Software Engineering (AOSE) methodology by industry.

Several AOSE methodologies have been proposed [17, 34]. While some work considered specification-based formal verification [11, 14], others borrow object-oriented testing techniques, taking advantage of a projection of agent-oriented abstractions into object-oriented constructs, UML for instance

[9, 33]. However, to the best of our knowledge, none of existing work provides a complete and *structured testing process* for guiding the testing activities. This is a big gap that we need to bridge in order for AOSE to be widely applicable.

II. SOFTWARE TESTING

Software testing is a software development phase, aimed at evaluating product quality and enhancing it by detecting errors and problems. Software 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 intended results, and an estimation is made of some aspect of the system or component. A test is a set of one or more test cases.

The principal test goal is to find faults different from errors. 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 then an execution error may occur. An execution error 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 be unobservable and as a consequence may play severe disrupt with the left over computation and use of the results of this computation. The greater the period of unobserved operation, the larger is the probability of serious damage due to errors that is caused by unnoticed failures.

As showed in Figure 1, software testing consists of the dynamic verification of the program behavior on a set of suitably selected test cases. Different from static verification activities, like formal proofing or model checking, testing implies running the system under test using specified test cases [22].

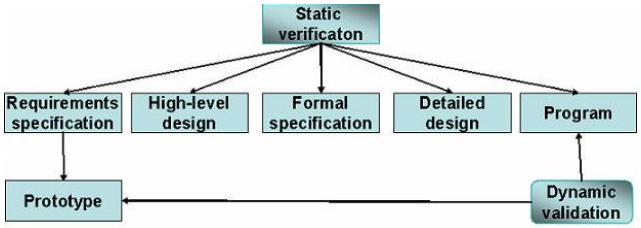

Figure 1. Kinds of Tests

There are several strategies for testing software and the goal of this survey is not to explain all of them. Nevertheless, we will describe the main strategies found in literature [22, 35]. 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 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: it is the process of testing a program to determine whether a change has introduced faults (regressions) in the unchanged code. It is based on reexecution of some/all of the tests developed for a specific testing activity.
- Performance testing: verify that all worst case performance and 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 needed system, as a whole, coverage. It tests subsystem connectivity. There are several strategies for implementing integration test:

- Bottom-up, which tests each unit and component at lowest level of system hierarchy, then components that call these and so on;
- Top-down, which tests top component and then all components called by this and so on;
- Big-bang, which integrates all components together;
- Sandwich, which combines bottom-up with top-down approach.

On the other hand, the goal of software testing is also to prevent defects, as it is clearly much better to prevent faults than to detect and correct them because if the bugs are prevented, there is no code to correct. This approach is used in cleanroom software development [22]. Designing tests is known as one of the best bug prevention activities. Tests design can discover and eliminate bugs at every stage in the software construction process [2]. Therefore, the idea of "test first, then code" or test-driven is quite widely discussed today. To date, several techniques have been defined and used by software developers [1].

Recently, a new testing technique called Evolutionary testing (ET) [27, 41] has been presented. The technique is inspired by the evolution theory in biology that emphasizes natural selection, inheritance, and variability. Fitter individuals have a higher chance to survive and to reproduce offspring; and special characteristics of individuals are inherited. In ET, we usually encode each test case as an individual; and in order to guide the evolution towards better test suites, a fitness measure is a heuristic approximation of the distance from achieving the testing goal (e.g., covering all statements or all branches in the program). Test cases having better fitness values have a higher chance to be selected in generating new test cases. Moreover, mutation is applied during reproduction in order to generate more different test set. The key step in ET is the transformation from testing objective to search problem, specifically fitness measure. Different testing objective gives rise to different fitness definitions. Once a fitness measure has been defined, different optimization search techniques, such as local search, genetic algorithm, particle swarm [27] can be used to generate test cases towards optimizing fitness measure (or testing objective, i.e. finding faults).

III. SOFTWARE AGENTS AND MAS TESTING

A software agent is a computer program that works toward goals in a dynamic context on behalf of another entity (human or computational), perhaps for a long period of time, with discontinuous direct supervision or control, and exhibits a significant flexibility and even creativity degree in how it tries to transform goals into action tasks [18].

Software agents have (among others) the following properties:

1. Reactivity: agents are able to sense contextual changes and react appropriately;
2. Pro-activity: agents are autonomous, so they are able to select which actions to take in order to reach their goals in given situations;
3. Social ability: that is, agents are interacting entities, which cooperate, share knowledge, or compete for goal achievement

A multi-agent system (MAS) is a computational context in which individual software agents interact with each other, in a collaborative (using message passing) or competitive manner, and sometimes autonomously trying to attain their individual goals, accessing resources and services of the context, and occasionally producing results for the entities that initiated those software agents [25]. The agents interact in a concurrent, asynchronous and decentralized manner [21] hence MAS turn out to be complex systems [23]. Consequently, they are difficult to debug and test.

Due to those peculiar characteristics of agents and MAS as a whole, testing them is a challenging task that should address the following issues. (Some of them were stated in [37]):

Distributed/asynchronous: Agents operate concurrently and asynchronously. An agent might have to wait for other agents to fulfill its intended goals. An agent might work correctly when it operates alone but incorrectly when put into a community of agents or vice versa. MAS testing tools must have a global view over all distributed agents in addition to local knowledge about individual agents, in order to check whether the whole system operate accordingly to the specifications. In addition, all the issues related to testing distributed systems are applied in testing software agent and MAS as well, for example problems with controllability and observability [6].

Autonomous: Agents are autonomous. The same test inputs may result in different behaviors at different executions, since agents might modify their knowledge base between two executions, or they may learn from previous inputs, resulting in different decisions made in similar situations.

Message passing: Agents communicate through message passing. Traditional testing techniques, involving method invocation, cannot be directly applied.

Environmental and normative factors: Context and conventions (norms, rules, and laws) are important factors that govern or influence the agents' behaviors. Different contextual settings may affect the test results. Occasionally, a context gives means for agents to communicate or itself is a test input.

Scaled agents: In some particular cases, agents could be seen as scaled in that they provide no or little observable primitives to the outside world, resulting in limited access to the internal agents' state and knowledge. An example could be an open MAS that allows third-party agents to come in and access to the resources of the MAS, how do we assure that the third-party agents with limited knowledge about their intentions behave properly?

The agent oriented methodologies provide a platform for making MAS abstract, generalize, dynamic and autonomous. However, many methodologies like MASE, Prometheus, and Tropos do exist for the agent oriented framework but on contrary to it the *testing techniques* for the methodologies are not clearly supported [10].

A. Test Levels

Over the last years, the view of testing has evolved, and testing is no longer seen as a step which starts only after the implementation phase is finished. Software testing is now seen as a whole process that filters in the development and maintenance activities. Thus, each development phase and maintenance phase should have a corresponding test level. Figure 2 shows V model in which the correspondence between development process phases and test levels are highlighted [28].

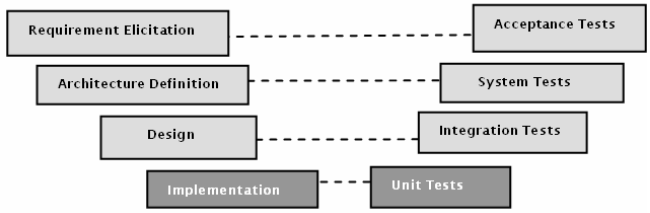

Figure 2. V model

Work in testing software agents and MAS can be classified into different testing levels: unit, agent, integration, system, and acceptance. Here we use general terminologies rather than using specific ones used in the community like group, society. Group and society, as called elsewhere, are equivalent to integration and system, respectively. The testing objectives, subjects to test, and activities of each level are described as follows:

- Unit testing tests all units that make up an agent, including blocks of code, implementation of agent units like goals, plans, knowledge base, reasoning engine, rules specification, and so on; make sure that they work as designed.
- Agent testing tests the integration of the different modules inside an agent; test agents' capabilities to fulfill their goals and to sense and effect the environment.
- Integration or Group testing tests the interaction of agents, communication protocol and semantics, interaction of agents with the environment, integration of agents with shared resources, regulations enforcement; Observe emergent properties, collective behaviors; make sure that a group of agents and environmental resources work correctly together.

- System or Society testing tests the MAS as a system running at the target operating environment; test the expected emergent and macroscopic properties of the system as a whole; test the quality properties that the intended system must reach, such as adaptation, openness, fault tolerance, performance.
- Acceptance testing tests the MAS in the customer's execution environment and verifies that it meets stakeholder goals, with the participation of stakeholders.

B. MAS Testing Problems

Defining a structured testing process for software agents and MAS: Currently, AOSE methodologies have been interesting principally on requirement analysis, design, and implementation; limited attention was given to validation and verification, as in Formal Tropos [11, 14]. A structured testing process that complements analysis and design is still absent. This problem is determinant because without detailed and systematic guidelines, the development cost may increase in terms of effort and productivity.

1. They have their own reasons for engaging in proactive behaviors that might differ from a user's concrete expectation, yet are still appropriate.
2. The same test input can give different results in different executions.
3. Agents cooperate with other agents, so they may run correctly by themselves but incorrectly in a community or vice versa.
4. Moreover, agents can be programmed to learn; so successive tests with the same test data may give different results.

As a conclusion, defining adequate and effective techniques to test software agents is, thus, a key problem in agent development.

IV. A Survey of Testing Multi-Agent Systems

There is very brief written work that describes agents software testing. The remainder of this section surveys recent and active work on testing software agents and MAS, with respect to previous categories. This classification is intended only to facilitate easily understand the research work in the field. It is also interesting to notice that this classification is incomplete in the sense that some work addresses testing in more than one level, but we put them in the level they principally focus.

A. Unit Testing

Unit testing approach calls attention to the test of the smallest building blocks of the MAS: the agents. Its essential idea is to check if each agent in isolation respects its specifications under normal and abnormal conditions. Unit testing needs to make sure that all units that are parts of an agent, like goals, plans, knowledge base, reasoning engine, rules specification, and even blocks of code work as designed. Effort has been spent on some particular elements, such as goals, plans. Nevertheless, a complete approach addressing unit testing in AOSE still opens room for research. An analogy of expected results can be those of unit testing research in the object-oriented development. At the unit level,

1. Zhang et al. [42] introduced a model based testing framework using the design models of the Prometheus agent development methodology [31]. Different from traditional software systems, units in agent systems are more complex in the way that they are triggered and executed. For instance, plans are triggered by events. The framework focuses on testing agent plans (units) and mechanisms for generating suitable test cases and for determining the order in which the units are to be tested.
2. Ekinci et al. [13] claimed that agent goals are the smallest testable units in MAS and proposed to test these units by means of test goals. Each test goal is conceptually decomposed into three sub-goals: setup (prepare the system), goal under test (perform actions related to the goal), and assertion goal (check goal satisfaction). The first and last goal prepares pre-conditions and check post-conditions while testing the goal under test, respectively. Moreover, they introduce a testing tool, called as SEAUnit that provides necessary infrastructure to support proposed approach.

B. Agent testing

At the agent level we have to test the integration of the different modules inside an agent, test agents' capabilities to achieve their goals and to sense and effect the context. There is several works in agent testing level.

1. Agile PASSI [7] proposes a framework to support tests of single agents. They develop a test suite specifically for agent verification. Test plans are prepared before the coding phase in according with specifications and the AgentFactory tool is also able of generating driver and stub agents for speeding up the test of a specific agent. Despite proposing valuable ideas concerning MAS potential levels of tests, PASSI testing approach is poorly documented and does not offer techniques to help developers in the low level design of unit test cases.
2. Lam and Barber [26] proposed a semi-automated process for comprehending software agent behaviors. The approach imitates what a human user (can be a tester) does in software comprehension: building and refining a knowledge base about the behaviors of agents, and using it to verify and explain behaviors of agents at runtime. Although the work did not deal with other problems in testing, like the generation and execution of test cases, the way it evaluates agent behaviors is interesting and relevant for testing software agents.
3. Nunez et al. [30] introduced a formal framework to specify the behavior of 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. Two testing methodologies were proposed

to check whether an implementation of a specified agent behaves as expected (i.e., conformance testing). In their active testing approach, they used for each agent under test a 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. On the other hand, the authors also proposed to use passive testing in which the agents under test were observed only, not stimulated like in active testing. Invalid traces, if any, are then identified thanks to the formal specifications of the agents.

4. Coelho et al. [8] proposed a framework for unit testing of MAS based on the use of Mock Agents. Even though they called it unit testing but their work focused on testing roles of agents at agent level according to our classification. Mock agents that simulate real agents in communicating with the agent under test were implemented manually; each corresponds to one agent role. Sharing the inspiration from JUnit [15] with Coelho et al. [8], Tiryaki et al. [40] proposed a test-driven MAS development approach that supported iterative and incremental MAS construction. A testing framework called SUnit, which was built on top of JUnit and Seagent [12], was developed to support the approach. The framework allows writing tests for agent behaviors and interactions between agents.
5. Gomez-Sanz et al. [16] introduced advances in testing and debugging made in the INGENIAS methodology [33]. The meta-model of INGENIAS has been extended with concepts for defining tests to incorporate the declaration of testing, i.e., tests and test packages. The code generation facilities are augmented to produce JUnit-based test case and suite skeletons based on these definitions with respect to debugging and it is the developer's task to modify them as needed. The work also provided facilities to access mental states of individual agents to check them at runtime. The system is integrated with ACLAnalyzer [4], a data mining facility for capturing agent communication and exploring them with different graphical representations.
6. Houhamdi [18] introduces a suite test derivation approach for Agent testing that takes goal-oriented requirements analysis artifact as the core elements for test case derivation. The proposed process has been illustrated with respect to the *Tropos* development process. It provides systematic guidance to generate test suites from agent detailed design. These test suites, on the one hand, can be used to refine goal analysis and to detect problems early in the development process. On the other hand, they are executed afterwards to test the achievement of the goals from which they were derived.

C. Integration Testing

Integration testing test the interaction of agents, communication protocol and semantics, interaction of agents with the context, integration of agents with shared resources, regulations enforcement; observe emergent properties; make sure that a group of agents and environmental resources work correctly together.

Only a few of methodologies define an explicit verification process by proposing a verification phase based on model checking to support automatic verification of inter-agent communications. Only some iterative methodologies propose incremental testing processes with supporting tools. At the integration level, effort has been put in agent interaction to verify dialogue semantics and workflows.

1. Agile [24] defines a testing phase based on JUnit test framework [15]. In order to use this tool, designed for OO testing, in MAS testing context, they needed to implement a sequential agent platform, used strictly during tests, which simulates asynchronous message-passing. Having to execute unit tests in an environment different from the production environment results in a set of tests that does not explore the hidden places for failures caused by the timing conditions inherent in real asynchronous applications.
2. The ACLAnalyser [4] tool runs on the JADE [39] platform. It intercepts all messages exchanged among agents and stores them in a relational database. This approach exploits clustering techniques to build agent interaction graphs that support the detection of missed communication between agents that are expected to interact, unbalanced execution configurations, overhead data exchanged between agents. This tool has been enhanced with data mining techniques to process results of the execution of large scale MAS [5].
3. Padgham et al. [32] use design artifacts (e.g., agent interaction protocols and plan specification) to provide automatic identification of the source of errors detected at run-time. A central debugging agent is added to a MAS to monitor the agent conversations. It receives a carbon copy of each message exchanged between agents, during a specific conversation. Interaction protocol specifications corresponding to the conversation are fired and then analyzed to detect automatically erroneous conditions.
4. Also at the integration level but pursuing a deontic approach, Rodrigues et al. [36] proposed to exploit social conventions, i.e. norms, rules, that prescribe permissions, obligations, and/or prohibitions of agents in an open MAS to integration test. Information available in the specifications of these conventions gives rise to a number of types of assertions, such as time to live, role, cardinality, and so on. During test execution a special agent called Report Agent will observe events and messages in order to generate analysis report afterwards.
5. Ekinci et al. [13] view integration testing of MAS rather abstract. They considered *system goals* as the source cause for integration and use them as driving

criteria. They apply the same approach for testing agent goals (unit according to their view) to test these goals. They define the concept of test goal. This concept represents the group of tests needed in order to check if the system goal is achieved correctly.

6. Nguyen et al. [29] propose using ontologies extracted from MAS under test and a set of OCL constraints, which act as a test oracle. Having as input a representation of the ontologies used, the idea is to construct an agent able to deliver messages whose content is inspired by these ontologies. The resulting behaviors are regarded as correct using the input set of OCL constraints: if the message content satisfies the constraints, the message is correct. The procedure is support by *eCAT*, a software tool.

7. Houhamdi and Athamena [19] introduced a novel approach for goal-oriented software integration testing. They propose a test suite derivation approach for integration testing that takes goal-oriented requirements analysis artifact for test case derivation. They have discussed how to derive test suites for integration test from architectural and detailed design of the system goals. These test suites can be used to observe emergent properties resulting from agent interactions and make sure that a group of agents and contextual resources work correctly together. This approach defines a structured and comprehensive integration test suite derivation process for engineering software agents by providing a systematic way of deriving test cases from goal analysis.

D. System and Acceptance Testing

System testing tests; test for quality properties, such as adaptation, openness, fault-tolerance, performance.

At the system level of testing MAS, one has to test the MAS as a system running at the target operating environment; test the expected emergent and macroscopic properties and/or the expected qualities that the intended system as whole must reach. Some initial effort has been devoting to the validation of macroscopic behaviors of MAS.

- Sudeikat and Renz [38] proposed to use the system dynamics modeling notions for the validation of MAS. These allow describing the intended, macroscopic observable behaviors that originate from structures of cyclic causalities. System simulations are then used to measure system state values in order to examine whether causalities are observable.

- Houhamdi and Athamena [20] introduced a suite test derivation approach for system testing that takes goal-oriented requirements analysis artifact as the core elements for test case derivation. The proposed process has been illustrated with respect to the *Tropos* development process. It provides systematic guidance to generate test suites from modeling artifacts produced along with the development process. They have discussed how to derive test suites for system test from late requirement and architectural design. These test suites, on the one hand, can be used to refine goal analysis and to detect problems early in the development process. On the other hand, they are executed afterwards to test the achievement of the goals from which they were derived.

Acceptance testing tests the MAS in the customer execution environment and verifies that it meets the stakeholder goals, with the participation of stakeholders. To the best of our knowledge, there is no work dealing explicitly with testing MAS at the acceptance level, currently. In fact, agent, integration, and system test harnesses can be reused in acceptance test, providing execution facilities. However, as testing objectives of acceptance test differ from those of the lower levels, evaluation metrics at this level, such as metrics for openness, fault-tolerance and adaptivity, demand for further research.

V. CONCLUSION

In summary, most of the existing research work on testing software agent and MAS focuses mainly on agent and integration level. Basic issues of testing software agents like message passing, distributed/asynchronous have been considered; testing frameworks have been proposed to facilitate testing process. And yet, there is still much room for further investigations, for instance:

- A complete and comprehensive testing process for software agents and MAS.

- Testing MAS at system and acceptance level: how do the developers and the end-users build confidence in autonomous agents?

- Test inputs definition and generation to deal with open and dynamic nature of software agents and MAS.

- Test oracles, how to judge an autonomous behavior? How to evaluate agents that have their own goals from human tester's subjective perspectives?

- Testing emergent properties at macroscopic system level: how to judge if an emergent property is correct? How to check the mutual relationship between macroscopic and agent behaviors?

- Deriving metrics to assess the qualities of the MAS under test, such as safety, efficiency, and openness.

- Reducing/removing side effects in test execution and monitoring because introducing new entities in the system, e.g., mock agents tester agents, and monitoring agent as in many approaches, can influence the behavior of the agents under test and the performance of the system as a whole.

REFERENCES

[1] K. Beck, “Test Driven Development: By Example”, Addison-Wesley Longman Publishing Co., Boston, USA, 2005.

[2] B. Beizer, “Software Testing Techniques”, 2nd edition, Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[3] F. Bergenti, M. Gleizes, and F. Zambonelli, “Methodologies and Software Engineering for Agent Systems”, The Agent-Oriented Software Engineering Handbook, Springer, Vol. 11, 2004.

[4] J. Botia, A. Lopez-Acosta, and G. Skarmeta, "ACLAnalyser: A tool for debugging multi-agent systems", Proceeding of the 16th European Conference on Artificial Intelligence, pp. 967-968, IOS Press 2004.

[5] J. Botia, J. Gomez-Sanz, and J. Pavon, "Intelligent Data Analysis or the Verification of Multi-Agent Systems Interactions", 7th International Conference of Intelligent Data Engineering and Automated Learning, Burgos, Spain, September 20-23, pp. 1207-1214, 2006.

[6] L. Cacciari, and O. Rafiq, "Controllability and observability in distributed testing", Information and Software Technology, Vol. 41, 11-12, pp. 767-780. 1999.

[7] G. Caire, M. Cossentino, A. Negri, and A. Poggi, "Multi-agent systems implementation and testing", Proceedings of the 7th European Meeting on Cybernetics and Systems Research - EMCSR2004, Vienna, Austrian Society for Cybernetic Studies, pp. 14-16, 2004.

[8] R. Coelho, U. Kulesza, A. Staa, and C. Lucena, "Unit testing in multi-agent systems using mock agents and aspects", Proceedings of the international workshop on Software engineering for large-scale multi-agent systems, ACM Press, New York, pp. 83–90, 2006.

[9] M. Cossentino, "From Requirements to Code with PASSI Methodology", In Vijayan Sugumaran (Ed.), Intelligent Information Technologies: Concepts, Methodologies, Tools, and Applications, USA, 2008.

[10] K.H. Dam, and M. Winikoff, "Comparing Agent-Oriented Methodologies", 5th International Bi-Conference Workshop, AOIS 2003 at AAMAS 2003, Melbourne, Australia, July 14, pp. 78-93, 2003.

[11] A. Dardenne, A. Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition", Science of Computer Programming 20(1-2), pp.3-50, 1993.

[12] O. Dikenelli, R. Erdur, and O. Gumus, "Seagent: a platform for developing semantic web based multi agent systems", AAMAS'05 Proceedings of the fourth International Joint Conference on Autonomous agents and multi-agent systems, ACM Press, New York, pp. 1271–1272, 2005.

[13] E. Ekinci, M. Tiryaki, O. Cetin, and O. Dikenelli, "Goal-Oriented Agent Testing Revisited", Proceeding of the 9th International Workshop on Agent-Oriented Software Engineering, pp. 85-96, 2008.

[14] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, and M. Roveri, "Specifying and analyzing early requirements in *Tropos*", Requirement Engineering, Springer Link, Vol. 9, 2, pp. 132-150, 2004.

[15] E. Gamma, and K. Beck, "JUnit: A Regression Testing Framework", http://www.junit.org, 2000.

[16] J. Gomez-Sanz, J. Botia, E. Serrano, and J. Pavon, "Testing and Debugging of MAS Interactions with INGENIAS", Agent-Oriented Software Engineering IX, Springer, Berlin, pp. 199-212, 2009.

[17] B. Henderson-Sellers, and P. Giorgini, "Agent-Oriented Methodologies", Proceedings of the 4th International Workshop on Software Engineering for Large-Scale Multi-Agent Systems - SELMAS'05, Idea Group Incorporation, 2005.

[18] Z. Houhamdi, "Test Suite Generation Process for Agent Testing", Indian Journal of Computer Science and Engineering, Vol. 2, 2, 2011.

[19] Z. Houhamdi, and B. Athamena, "Structured Integration Test suite Generation Process for Multi-Agent System", Journal of Computer Science, Vol. 7, 5, 2011.

[20] Z. Houhamdi, and B. Athamena, "Structured System Test Suite Generation Process for Multi-Agent System", International Journal on Computer Science and Engineering, Vol.3, 4, pp.1681-1688, 2011.

[21] M. Huget, and Y. Demazeau, "Evaluating multi agent systems: a record/replay approach", Intelligent Agent Technology, IAT 2004, Proceedings IEEE/WIC/ACM International Conference, pp. 536 – 539, 2004.

[22] I. Sommerville, "Software Engineering", 9th edition, Addison Wesley, 2011.

[23] N.R. Jennings, "An Agent-Based Approach for Building Complex Software Systems", Communications of the ACM, Vol. 44, 4, pp. 35-41, 2001.

[24] H. Knublauch, "Extreme programming of multi-agent systems", International Joint Conference on Autonomous Agent and Multi-Agent Systems, Bologna. ACM Press, pp. 704–711, 2002.

[25] J. Krupansky, "What is a software Agent?", Advancing the Science of Software Agent Technology, 2008. http://agtivity.com/agdef.htm.

[26] D. Lam, and K. Barber, "Debugging Agent Behavior in an Implemented Agent System", 2nd International Workshop, ProMAS, Springer, Berlin, pp. 104-125, 2005.

[27] P. McMinn, and M. Holcombe, "The state problem for evolutionary testing", Proceedings of the International Conference on Genetic and Evolutionary Computation, Springer, Berlin, pp. 2488-2498, 2003.

[28] G. Myers, "The Art of Software Testing", Wiley, 2nd Edition New Jersey, John Wiley & Sons, 2004.

[29] C. Nguyen, A. Perini, and P. Tonella, "Goal-oriented testing for MAS", Agent-Oriented Software Engineering VIII, Lecture Notes in Computer Science, Volume 4951, pp. 58-72, 2008.

[30] M. Nunez, I. Rodriguez, and F. Rubio, "Specification and testing of autonomous agents in e-commerce systems", Software Testing, Verification and Reliability, Vol. 15, 4, pp. 211-233, 2005.

[31] L. Padgham, and M. Winikoff, "Developing Intelligent Agent Systems: A Practical Guide", John Wiley and Sons, 2004.

[32] L. Padgham, M. Winikoff, and D. Poutakidis, "Adding debugging support to the Prometheus methodology", Engineering Applications of Artificial Intelligence, Vol. 18, 2, pp. 173-190, 2005.

[33] J. Pavon, J. Gomez-Sanz, and R. Fuentes-Fernandez, "The INGENIAS Methodology and Tools", In Agent Oriented Methodologies (eds. Henderson-Sellers and Giorgini), Idea group, pp. 236-276, 2005.

[34] A. Perini, "Agent-Oriented Software", Wiley Encyclopedia of Computer Science and Engineering, John Wiley and Sons, Chapter 1, pp. 1-11, 2008.

[35] R.S. Pressman, "Engenharia de Software", 6th edition, Rio de Janeiro, McGraw-Hill, 2002.

[36] L. Rodrigues, G. Carvalho, P. Barros, and C. Lucena, "Towards an integration test architecture for open MAS", 1st Workshop on Software Engineering for Agent-Oriented Systems/SBES. pp. 60-66. 2005.

[37] C. Rouff, "A test agent for testing agents and their communities", Aerospace Conference Proceedings IEEE, Vol. 5. pp. 5-2638, 2002.

[38] J. Sudeikat, and W. Renz, "A systemic approach to the validation of self-organizing dynamics within MAS", Proceeding of the 9th International Workshop on Agent-Oriented Software Engineering, pp. 237-248, 2008.

[39] TILAB. Java agent development framework. http://jade.tilab.com/.

[40] A. Tiryaki, S. Oztuna, O. Dikenelli, and R. Erdur, "Sunit: A unit testing framework for test driven development of multi-agent systems", AOSE'06 Proceedings of the 7th International Workshop on Agent-Oriented Software Engineering VII, Springer, Berlin, pp. 156-173, 2007.

[41] J. Wegener, "Stochastic Algorithms: Foundations and Applications", In Evolutionary Testing Techniques, Springer Berlin, Heidelberg, Chapter 9, pp. 82-94, 2005.

[42] Z. Zhang, J. Thangarajah, and L. Padgham, "Automated unit testing for agent systems", 2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering, ENASE'07, Spain, pp. 10-18, 2007.

AUTHORS PROFILE

Dr. Zina Houhamdi received the M.Sc. and PhD. degrees in Software Engineering from Annaba University in 1996 and 2004, respectively. She is currently an Associate Professor at the department of Software Engineering, Al-Zaytoonah University of Jordan. Her research interest includes Agent Oriented Software Engineering, Software Reuse, Software Testing, Goal Oriented Methodology, Software Modeling and Analysis, Formal Methods. She has published several Research Papers in referred National/ International journals. She has referred presentations in National / International Conferences and Seminars.