TEST SUITE GENERATION PROCESS FOR AGENT TESTING

HOUHAMDI ZINA

Software Engineering Department, Al-Zaytoonah University
P.O. Box 130, 11733 Amman, Jordan
z_houhamdi@yahoo.fr

Abstract

Software agents are a promising technology for today's complex, distributed systems. Methodologies and techniques that address testing and reliability of multi agent systems are increasingly demanded, in particular to support automated test case generation and execution. In this paper, we introduce a novel approach for goal-oriented software agent testing. It specifies a testing process that complements the goal oriented methodology *Tropos* and reinforces the mutual relationship between goal analysis and testing. Furthermore, it defines a structured and comprehensive agent test suite generation process by providing a systematic way of deriving test cases from goal analysis.

Keywords: Testing; Goal-Oriented Testing Methodology; Unit Testing; Agent testing, Test Case Generation.

1. Introduction

MAS are increasingly taking over operations and controls in enterprise management, automated vehicles, and financing systems, assurances that these complex systems operate properly need to be given to their owners and their users [Nguyen *et al.* (2008)]. This calls for an investigation of suitable software engineering frameworks, including requirements engineering, architecture, and testing techniques, to provide adequate software development processes and supporting tools.

There are several reasons for the increase of the difficulty degree of multi-agent systems testing and debugging:

- 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, which means that it is not ensured that two executions of the systems will lead to the same state, even if the same input is used. As a consequence, looking for a particular error can be difficult if it is not possible to reproduce it each time [Huget and Demazeau (2004)].

As a result, testing software agents seeks for new testing techniques dealing with their peculiar nature. The techniques need to be effective and adequate to evaluate agent's autonomous behaviors and build confidence in them.

Testing a single agent is different from testing a community of agents. When testing a single agent a developer is more interested in the functionality of one agent and whether the agent operates for a set of messages, contextual inputs and error conditions. But, when testing a community of agents, the tester is interested in whether the agents operate together, are coordinated, and if message passing between them is correct [Gatti and Staa (2006)].

Several AOSE methodologies have been proposed [Henderson-Sellers and Giorgini (2005)]. In terms of testing and verification, while some consider specification-based formal verification [Dardenne *et al.* (1993); Fuxman *et al.* (2004); Perini *et al.* (2003)], other borrow Object-Oriented (OO) testing techniques, taking advantage of a mapping of agent-oriented abstractions into OO constructs [Cossentino (2005); Pavon *et al.* (2005)]. To the best of our knowledge, a *structured testing process* for AOSE methodologies is still absent.

In this paper, we propose a structured testing process that exploits the link between requirements and test cases following the V Model. We describe the proposed approach with reference to the *Tropos* software development methodology [Mylopoulos and Castro (2000)] and consider MAS as the target implementation technology.

The remainder of the paper is organized as follows. Section 2 recalls basic elements of the *Tropos* methodology and introduces related works. Section 3 discusses the proposed approach, an agent testing process and test suite generation. An illustrative example is used. Similar works are presented in section 5. Finally, Section 6 concludes our work and discusses future research directions.

2. Background and Related Works

2.1. *Tropos*

Tropos is an AOSE methodology that covers the whole software development process. *Tropos* is based on two key ideas. First, the notion of agent and all related mentalistic notions (for instance goals and plans) are used in all phases of software development, from early analysis down to the actual implementation. Second, *Tropos* covers also the very early phases of requirements analysis, thus allowing for a deeper understanding of the operational context of the software, and of the interactions kind that should occur between software and human agents. Tropos is structured in five main phases, namely [Morandini *et al*. (2008)]:

(1). *Early requirements* analysis that focuses on the understanding of the existing organizational setting where the expected system will be introduced;
(2). *Late requirements* that deals with the analysis of the expected system;
(3). *Architectural design* that defines the system's global architecture in terms of subsystems;
(4). *Detailed design* that specifies the system agents micro-level;
(5). *Implementation* that concerns code generation according to the detailed design specifications.

2.2. *Goal types and test types*

This section presents different goal types and testing types. The relationships between goal types and testing levels are presented with reference to *Tropos* development process.

2.2.1. Testing Type

There are four types of testing: *Agent testing*, *Integration testing*, *System testing* and *Acceptance testing* [Nguyen *et al.* (2008)]. The objectives and scope of each type is described as follows:

- *Agent testing*. The smallest unit of testing in agent-oriented programming is an agent. Testing a single agent consists of testing its inner functionality and agent's capabilities to fulfill its goals and to sense and effect the environment.
- *Integration testing*. Integration testing make sure that a group of agents and environmental resources work correctly together which involves checking an agent works properly with the agents that have been integrated before it and with the "future" agents that are in the course of *Agent testing* or that are not ready to be integrated.
- *System testing*. Agents may operate correctly when they run alone but incorrectly when they are put together. System testing involves making sure all agents in the system work together as intended.
- *Acceptance testing*. Test the MAS in the customer execution environment and verify that it meets the stakeholder goals, with the participation of stakeholders.

2.2.2. Goal Type

Goals can be classified according to different perspectives or criteria. For instance, goals can be classified into perform goals, achieve goals, and maintain goals according to the agent's attitude toward goals [Huget and Demazeau (2004)]. Other goal types are also discussed elsewhere e.g. *KAOS* [Dardenne *et al.* (1993)]. In this paper, since we are interested in separating individual agent's behavior from community behavior induced by goal delegation in *Tropos*, we consider two types of goal:

- *Delegated goal:* this goal type is delegated to one agent (dependee) by another agent (depender). This goal type often leads to interactions between the two agents: The depender demands (by sending requests to) the dependee to fulfill the goal.
- *Own goal*: This goal type requires responsibility of its owner; however, the owner agent does not necessarily run within its boundary, i.e. it can involve interactions with other agents as well.

One can reason about assigned goals of the *own* type of a single agent to come out with *agent testing* level. That is, based on these goals, developers could figure out which plans or behaviors of the agent, i.e. functionality, to test. Since *integration testing* and *system testing* involve making sure the operation of agents together in the system, the goals of type *delegated* and those goals of type *own* that involve agents interactions are good starting points for these testing types.

2.3. *Goal-oriented testing*

The V Model is a representation of the system development process, which extends the traditional waterfall model. The left branch of the V represents the specification stream, and the right branch of the V represents the testing stream where the systems are being tested (against the specifications defined on the left branch). One of the advantages of the V model is that it describes not only construction stream but also testing stream (unit test, integration test, acceptance test) and the mutual relationships between them (Fig. 1).

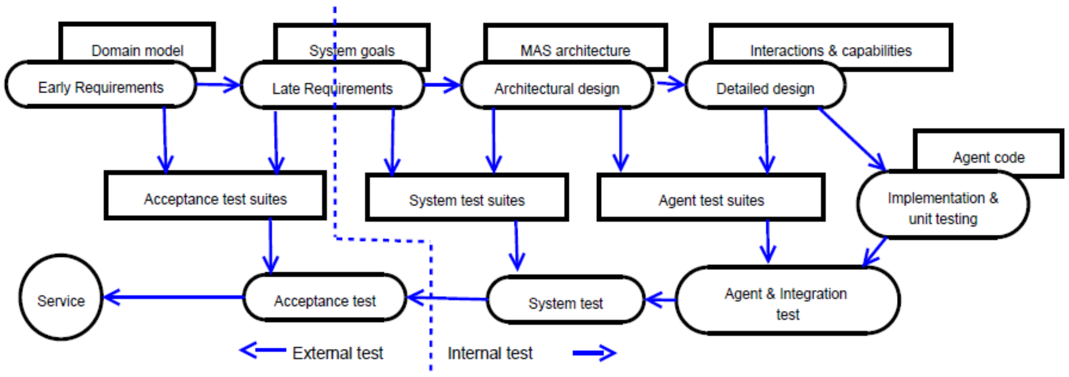

Fig. 1. V-model of goal-oriented testing.

Tropos guides the software engineers in building a conceptual model, which is incrementally refined and extended, from an early requirements model to system design artifacts and then to code, according to the upper branch of the V.

In next section, we present in depth a structured testing process model and we discuss how to derive systematically test cases from goal models by proposing a test suites generation process for agent plans, goals, and agents themselves.

3. Unit and Agent Test Suite Derivation

Unit testing test all units composing an agent, including blocks of code, implementation of agent units like goals, plans, knowledge base, reasoning engine, rules specification, and etc.; 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 affect the context. In the rest of this section we discuss essentially plan, goal, and agent testing.

To illustrate our approach, we introduce a multi-agent system that is made of several cleaner agents working at a public garden. This software could be deployed on a physical platform composed of a set of moving robots. Robots are in charge of keeping the garden clean and system agents have to collaborate to optimize their work and be pleasant with the visitors. To reach these softgoals, three other sub-goals need to be fulfilled: G1: keep the garden clean, G2: Team work and G3: be polite (Fig. 2). There could be more goals that the stakeholder wants to achieve, but we consider only these goals to keep the example understandable and simple.

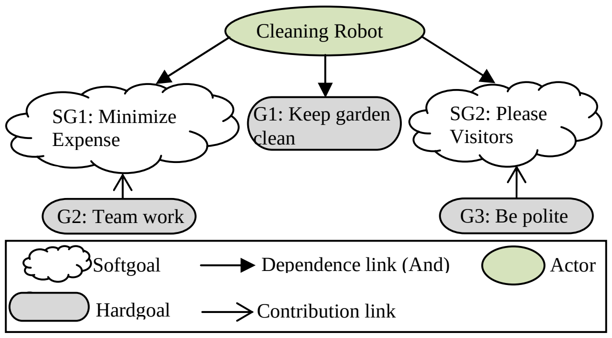

Fig. 2. Late requirements for cleaning Robot.

Following the guidelines, system actors become visible in the MAS architectural design. In this example, system actors are the cleaner agents. Goals of the system G1, G2, G3 are delegated to the agents. The internal architectural design of the cleaner agent is described in Fig. 3 which shows the architectural design of the cleaner agent.

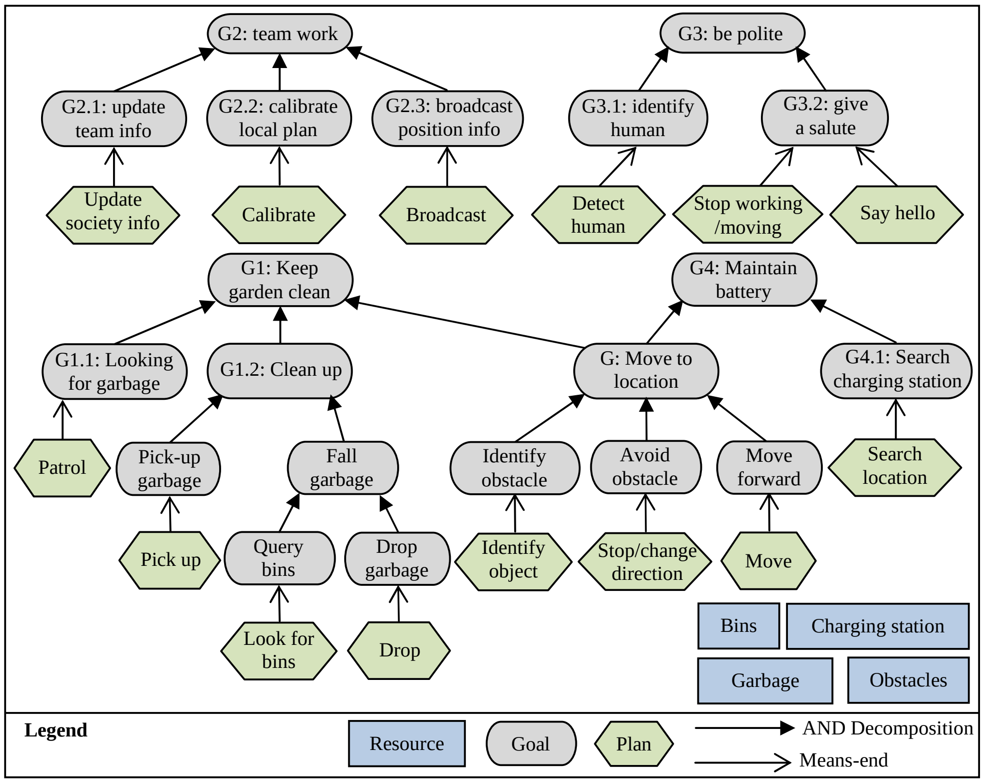

Fig. 3. Cleaner agent architecture.

A number of goals and plans (tasks) are affected to the agent. At the highest level there are 4 root goals: G1: keep the garden clean, G2: team work, G3: be polite and G4: maintain battery. G1, G2, G3 are delegated from the system, while G4 is the agent own goal to keep the agent alive. These goals are, then, decomposed into sub-goals. For instance, G4: maintain battery is AND-decomposed into two sub-goals G4.1: search charging station, and G: move to location. AND decomposition requires all sub-goals to be accomplished to obtain the satisfaction of their root goal. Finally, we add Plans to the cleaner agent design in order to achieve hardgoals.

3.1. *Plan testing*

Zhang et al. have presented different aspects linked to plans and events testing [Zhang *et al.* (2007)]. Albeit presented in the Prometheus methodology context [Padgham and Winikoff (2004)], those aspects can be used in our approach too, because both *Prometheus* and *Tropos* use the Belief-Desire-Intention agent software architecture [Rao and Georgeff (1995)]. However, plans are means to reach goals; plans are triggered when goals are selected.

Therefore, to test a plan, we require creating test suites such that they satisfy all the pre-conditions of its end goal and pre-conditions of the plan itself. These conditions, among others, contain corresponding events that eventually activate the plan. Afterwards, we have to evaluate the plan execution and its ulterior tasks.

As for plan testing criteria, plan execution can be evaluated by using its goal's state. For example, if the goal state is maintained or achieved as a consequence of the plan execution, we can deduce that the implemented plan succeeds the test.

Test suite derivation for plans takes place at the detailed design phase. For each single plan, we need to create a test suite that contains a set of Unit Test Cases (UTC) to test the plan with different inputs. In our illustrative example, there are 14 test suites in total. The associated test suite for the plan Move is described in Table 1**.**

Table 1. Test suite for plan Move.

Test Case	Scenario	Criteria
UTC1	There is an event that requires the cleaner agent to move from position A(1,2) to position B(5,3), no obstacle is in the middle of the two points	The cleaner agent moves straight from A to B
UTC2	Between A(1,2) and B(5,3), there is a static obstacle at point C(3,2)	The cleaner agent moves close to C, identifies the obstacle, avoids C before going to B
UTC3	The agent is requested to move from (1,1) to (2,-1)	The cleaner agent moves to the boundary nearest to (2,-1)

3.2. *Goal testing*

Goals are states of affair, and one must do something in order to reach his/her goals. A very natural way of testing the goal achievement is to verify one's behavior with respect to the goal. In the same manner, to test a goal we have to check what the agent does to achieve the goal [Nguyen *et al.* (2008)].

The agent internal design consists of goal decomposition trees. For example, Fig. 3 depicts the design of the cleaner agent, consisting of five trees associated with four root goals: G2: team work, G4: maintain battery, G1: keep garden clean, G3: be polite. The root goals fulfillment is estimated based on the achievement of their sub-goals and the relationships between the root goals and the sub-goals, and same procedure with the intermediate goals inside the trees. The fulfillment of the leaf goals of the trees is evaluated using their relationships with the means plans. These relationships are called basic relationships.

The principal basic relationships are showed in Fig. 4. These include:

(1). *Means-End* between a plan and a hardgoal. If there is a *Means-End* relationship between goal *g* and plan *p*, we conclude that *g* is achieved when *p* is executed successfully.

(2). *Contribution+* between a plan and a softgoal. If goal *g* contributes positively to softgoal *sg* then we deduce that *sg* is partially satisfied when *g* is reached.

(3). *Contribution-* between a plan and a softgoal. If goal *g* contributes negatively to softgoal *sg* then we can say *sg* is partially unsatisfied when *g* is achieved.

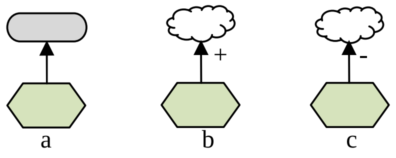

Fig. 4. Basic relationships. (a): Means-End; (b): Contribution+; (c): Contribution-

When applying the *Tropos* methodology, we can deduce how goals can be achieved by looking at their relationships with other goals and plans. According to the goal's relationships, we can verify the fulfillment of the goal. In order to test these relationships, the execution of the plan corresponding to a goal is triggered and checked using constraints and assertions on the intended behavior.

Developers derive test suites from goal diagrams by starting from the relationships associated with each goal. Each relationship gives raise to a corresponding test suite which consists of a set of test cases that are used to check goal fulfillment (called positive test cases) and counter-fulfillment (called negative test cases). Positive test cases are needed at checking the agent capability to reach a given goal; negative test cases, on the other hand, are used to ensure that agent under test behave appropriately when it fails to fulfill a given goal.

The test suite generation procedure can be described as follows: for each leaf goal, we find out means plans from the goal basics relationships. The goal achievement and possible plan execution scenarios are then identified. Finally, a test suite should be created for the leaf goal. A test suite consists of a set of test cases, where each test case corresponds to one possible scenario execution (Fig. 5).

For the intermediate goals (not leaf goal), test suites are generated by examining all relationships that conduct to the target goals. This concludes at analyzing all basic relationships and reusing test suites derived for them. When the outputs of these test suites are found, we can reason about the fulfillment of the intermediate goals using the decomposition and/or contribution analysis.

For example, to test the goal G2: team work of our cleaner agent, we have to analyze its tree decomposition into 3 sub-goals; from there, we have to test 3 basic relationships between the sub-goals and their corresponding plans. Because this is simply an AND-decomposition, if 3 test suites derived for these 3 basic relationships are succeed, then the goal G2 is passed; otherwise the goal is failed.

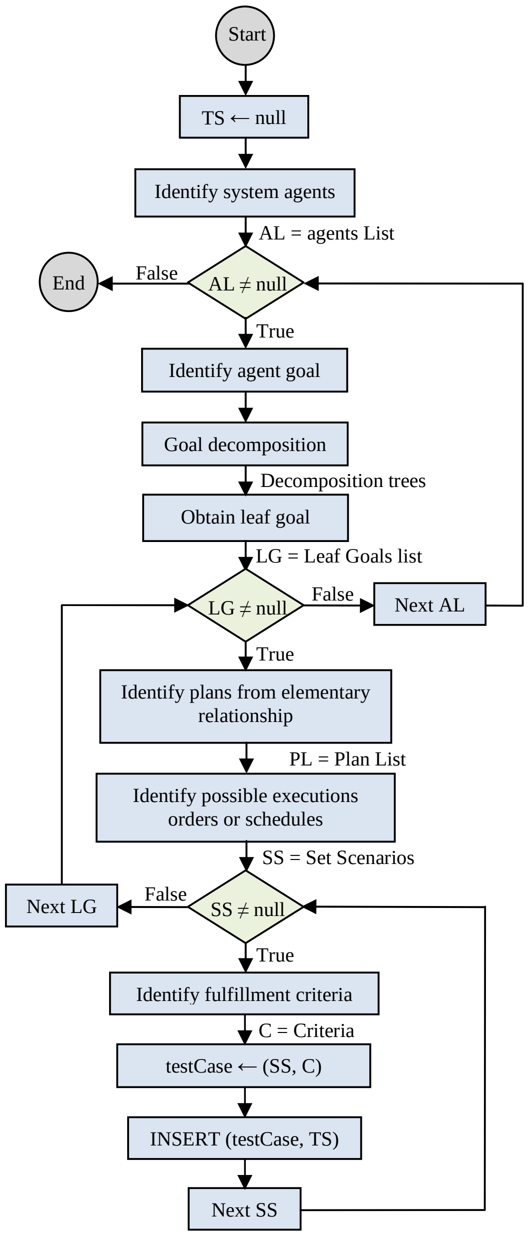

Fig. 5. Goal Test Suite derivation flowchart.

For more complicated intermediate goals, we must test all possible combination scenarios using the goal analysis and reason about the goal achievement on the basis of these scenarios and the results of the test suites derived for the related leaf goals. For example, the agent goal G1: keep garden clean of the cleaner agent. By analyzing the goal decomposition tree, which has G1 as root, 7 basic relationships are identified:

(1). Patrol → Look for garbage,
(2). Pick up → Pickup-garbage,
(3). Identify object → Identify object,
(4). Drop → Drop garbage,
(5). Move → Move forward,

(6).Stop/change direction $\rightarrow$ Avoid obstacle,
(7).Look for bin $\rightarrow$ query bin.
Each of them engenders a different test suite.

3.3. Agent testing

An agent contains set of smaller components, e.g., beliefs, goals, plans, events, reasoning module, and so forth. Testing at the agent level consists of integration testing of agent components, so one has to derive test suites to check this integration.

At agent-level, test suites have a powerful relation with test suites created for testing agent goals. Because, in general, testing a goal implies testing one or a set of plans which involves events and resources. In the same manner, testing a goal triggers some integration of plans, events, and so on. Hence, test suites derived to test agent goals are still effective to test the agent integration. Nevertheless, we need to test the integration of goals as well. Some goals have dependencies among them, such as priority or inhibition dependences; others may be maintained or achieved in parallel while sharing a resource. So we have to identify goal integration scenarios, create test suites for each, and look for integration problems such as dependency violations, deadlock, livelock, and the like.

Let's consider our motivating example once more. At the agent level, we have to derive test suites to check if the agent can execute: G1, G2, G3 and G4. Besides, we have to verify the possible conflicts among these goals. For example, at a specific moment, the cleaner agent can only move either to a recharging station, or to a bin, or to a new position for patrolling. Hence, some goal might be temporarily sacrificed in favor of another one. Furthermore, we have also to verify if collaborative goals (e.g., G2: teamwork) are fulfilled in conjunction with the other goals.

The basic test adequacy requirement for an agent is that all the agent goals must be tested. The agent should be capable to reach its own goals and act properly in the cases where its expected goal can't be achieved. This adequacy requirement may or may not be sufficient to cover the agent components, i.e. plans, events, beliefs, etc. If some are never exercised by the test suites defined to reach the basic adequacy criterion (goal coverage), more test suites have to be defined to complete agent testing.

4. Similar works

The rest of this section surveys recent and active work on testing software agents.

- Gomez-Sanz *et al*. [Gomez-Sanz *et al.* (2009)] presented advances in testing and debugging used in the INGENIAS methodology [Pavon *et al.* (2005)]. The meta-model of INGENIAS has been extended to introduce testing declaration, i.e., tests and test packages. JUnit-based test case and suite skeletons can be generated 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.
- Coelho *et al*. [Coelho *et al.* (2006)] 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 [Gamma and Beck (2000)] with Coelho, Tiryaki *et al*. [Tiryaki *et al.* (2007)] 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 [Dikenelli *et al.* (2005)] was developed to support the approach. The framework allows writing tests for agent behaviors and interactions between agents.
- Lam and Barber [Lam and Barber (2005)] 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 behaviours is interesting and relevant for testing software agents.
- Nunez *et al*. [Nunez *et al.* (2005)] 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.

Effort has been spent on some particular elements, such as goals, plans. However, fully 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. Thus, there is still much room for further investigations, for instance:

- A complete and comprehensive testing process for software agents and MAS.
- Test inputs definition and generation to deal with open and dynamic nature of software agents and MAS.
- Test criteria, how to judge an autonomous behavior? How to evaluate agents that have their own goals from human tester's subjective perspectives?
- 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.

5. Conclusion

As with the other testing levels, agent test suites are proposed at two distinctive points.

(1). *Unit testing*: 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.

(2). *Agent testing*: to test the integration of the different modules inside an agent; test agents' capabilities to fulfill their goals and to sense and effect the environment.

This paper introduced 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 modeling artifacts produced along with the development process. We have discussed how to derive test suites for system test 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.

Specifically, the proposed methodology contributes to the existing AOSE methodologies by providing:

- A testing process model, which complements the development methodology by drawing a connection between goals and test cases and
- A systematic way for deriving test cases from goal analysis.

In this paper, we have presented a structured process for Agent test case generation with reference to the *Tropos* methodology. In the future work, we will investigate other testing type like integration testing, system testing and acceptance testing.

References

[1]. Coelho R., Kulesza U., Staa A., Lucena C., (2006): 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.

[2]. Cossentino M., (2005): From requirements to code with the *PASSI* methodology. In Agent Oriented Methodologies, Hershey, PA, USA: Idea Group Publishing, Chapter IV, pp. 79-106.

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

[4]. Dikenelli O., Erdur R., Gumus O., (2005): 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.

[5]. Fuxman A., Liu L., Mylopoulos J., Pistore M., Roveri M. and Traverso P., (2004): Specifying and analyzing early requirements in *Tropos*. Requirements Engineering 9, vol. 2, pp. 132-150.

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

[7]. Gatti M. and Staa A., (2006): Testing & Debugging Multi-Agent Systems: A State of the Art Report. http://www.dbd.puc-rio.br/depto_informatica/06_04_gatti.pdf.

[8]. Gomez-Sanz J., Botia J., Serrano E. and Pavon J., (2009): Testing and Debugging of MAS Interactions with INGENIAS. Agent-oriented Software \engineering IX, Springer, Berlin, pp 199-212.

[9]. Henderson-Sellers B. and Giorgini P., (2005): Agent-Oriented Methodologies. Idea Group Inc.

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

[11]. Lam D. and Barber K., (2005): Debugging Agent Behavior in an Implemented Agent System. Second International Workshop, ProMAS 2004, Springer, Berlin, pp. 104-125.

[12]. Morandini M., Nguyen C., Perini A., Siena A. and Susi A., (2008): Tool-Supported Development with Tropos: The Conference Management System Case Study Agent-Oriented Software Engineering VIII. Lecture Notes in Computer Science, Springer 2008, Vol. 4951, pp. 182-196.

[13]. Mylopoulos J., Castro J., (2000): *Tropos*: A Framework for Requirements-Driven Software Development. Information Systems Engineering: State of the Art and Research Themes, Lecture Notes in Computer Science, Springer.

[14]. Nguyen C., Perini A. and Tonella P., (2008): Goal-Oriented Testing for MAS. International Journal of Agent-Oriented Software Engineering, LNCS 4951, pp. 58–72.

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

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

[17]. Pavon J., Gomez-Sanz J. and Fuentes-Fernandez R., (2005) : The INGENIAS Methodology and Tools. In agent oriented methodologies (eds. Henderson-Sellers and Giorgini), Idea group.

[18]. Perini A., Pistore M., Roveri M. and Susi A., (2003): Agent-oriented modeling by interleaving formal and informal specification, Agent-Oriented Software Engineering IV, 4th International Workshop, Melbourne, Australia, pp. 36-52.

[19]. Rao A. and Georgeff M., (1995): BDI-agents: from theory to practice. Proceedings of the First International Conference on Multi-agent Systems, San Francisco, pp. 312-319.

[20]. Tiryaki A., Oztuna S., Dikenelli O. and Erdur R., (2007): 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.

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