Software Testing In Small

Ms. J Cecilia Vikashini
M.Phil Research Scholar
Department of MCA
Sankara College of Science and Commerce
Coimbatore

Ms. M. Rajalakshmi
Assistant Professor
Department of MCA
Sankara college of Science and Commerce
Coimbatore

Abstract:

Software agents and multi-agent systems are a promising technology for today's complex, distributed systems. Methodologies and techniques that address testing and reliability of these systems are increasingly demanded, in particular to support systematic verification/validation and automated test generation and execution. To address the first problem, we proposed a goal-oriented testing methodology, aiming at defining a systematic and comprehensive testing process for engineering software agents. Concerning the second problem, the peculiar properties of software agents make testing them troublesome. We developed a number of techniques to generate test cases, automatically or semi-automatically. Our experiments have shown that each technique has different strength. For instance, while the random technique is effective in revealing crashes or exceptions, the ontology-based one is strong in detecting communication faults. The combination of these techniques can help to detect different types of fault, making software agents more reliable. We also investigated approaches to monitoring agent behaviors and evaluating them. Altogether, the generation, evaluation, and monitoring techniques form a bigger picture: our

Industries

novel continuous testing method. In this method, test execution can proceed unattended and independently of any other human-intensive activity; test cases are generated or evolved continuously using the proposed generation techniques; test results are observed and evaluated by our monitoring and evaluation approaches to give feedbacks to the generation step. The aim of continuous testing is to exercise and stress the agents under test as much as possible, the final goal being the possibility to reveal yet unknown faults. We applied a case study to illustrate the proposed methodology and performed three experiments to evaluate the performance of the proposed techniques. The obtained results are promising.

Keywords: Software agent testing, goal-oriented testing methodology, multiagent systems, Agent-oriented software engineering.

1.INTRODUCTION:

The increasing use of Internet as the backbone for all interconnected services and devices makes software systems highly complex and virtually

unlimited in scale. These systems often involve variety of users and heterogeneous platforms. They are evolved continuously to meet the changes of business and technology. In some circumstances, they need to be autonomous and adaptive for dealing with such changes.

Software agents, with their peculiar properties, e.g., (semi-)autonomy, adaptively, are key technologies to meet modern business needs, e.g., worldwide computing, ubiquitous computing, networked enterprises. They offer also an effective conceptual paradigm to model such complex systems. In fact, research on the development of software agents and MultiAgent System (MAS) has grown into a very active area, and interestingly they are receiving more industrial attention as well.

Software testing is important, but unfortunately there are many problems related with it. Testing is complicated and takes a lot of time. In general, complicated tasks that are performed by humans are prone to incompleteness and errors. This can be improved by automation, but unfortunately automation is difficult and requires changes in the software development process in order to be applicable efficiently. In addition, exhaustive testing is usually not possible, so the decision of which tests to use out of an infinite choice is difficult. The same difficulty applies to creating an automated mechanism that makes this choice. There is significant effort in the research community to solve these problems. As is often the case in software engineering, however, there is a wide gap between what techniques are proposed and applied by researchers and what is actually used in industry.

Work in testing software agents and MAS can be classified into different testing levels: unit, agent, integration, system, and acceptance. Here we employ general terminologies rather than using specific ones used in the community, e.g., 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.

Unit: Test 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 forth; make sure that they work as designed.

Agent: Test 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: Test 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: Test 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: Test The Mas In The Customer's Execution Environment And verify that it meets

stakeholder goals, with the participation of stakeholders.

The rest of this section surveys recent and active work on testing software agents and MAS, with respect to these categories. This classification is intended only to help easily understand the research work in the field. It is also worthwhile noticing that this classification is not complete in the sense that some work addresses testing in more than one level, but we put them in the level they mainly focus.

2. RELATED WORK:

While this research field is becoming more mature, there is an emerging need for detailed guidelines during the development process. This is considered a crucial step towards the adoption of Agent-Oriented Software Engineering (AOSE) methodology by industry. A number of methodologies (Perini 2009, Henderson-Sellers and Giorgini 2005) have been proposed so far. While some work considered specification-based formal verification (e.g., Formal Tropos (Fuxman et al. 2004) and (Dardenne et al. 1993)), others relied on object-oriented techniques, taking advantage of a mapping of agent-oriented abstractions into Object-oriented constructs, UML for instance. 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 agent-oriented methodologies to be widely applicable.

At the agent level, Gomez-Sanz et al. (2008) introduced advances in testing and debugging made in the INGENIAS methodology (Pavon et al. 2005). The meta-model of INGENIAS has been extended to incorporate the declaration of testing, 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.

At the system level of testing MAS, one has to test the expected emergent and macroscopic properties and/or the expected qualities of the system as a whole. Some initial effort has been devoting to the validation of macroscopic behaviors of MAS. Sudeikat and Renz (2008) 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.

At the unit level, Zhang et al. (2007) introduced a model based testing framework using the design models of the Prometheus agent development methodology (Padgham and Winiko 2002). Different from traditional software systems, units in agent systems are more complex in the way that they are triggered and executed.

Methodology:

This section presents the proposed methodology. We discuss different goal types, testing types, a testing process model. The relationships between goal types and testing levels are presented with reference to the process. Finally, we discuss how to derive systematically test cases from goal models.

Goal types:

Different perspectives give different goal classifications. For instance, (Dastani et al. 2006) classify agent goals in agent programming into three categories, namely perform, achieve, and maintain, according to the agent's attitude toward them. We use a general perspective on goals, but not from a specific subject (e.g., agent), to classify them based on the Tropos software engineering process. Goals are classified into the following types according to the different phases of the process:

Stakeholder: goals that represent stakeholder objectives and requirements towards the system to-be. This type of goal is mainly identified at the early requirements phase of Tropos.

System: goals that represent system-level objectives or qualities that the system to-be has to reach or provide. For instance, goals that are related to performance, openness of the system as a whole are system goals. This type of goal is mainly specified at the late requirements phase of Tropos.

Agent: goals that belong to or are assigned to particular agents. This type of goal appears when designing agents.

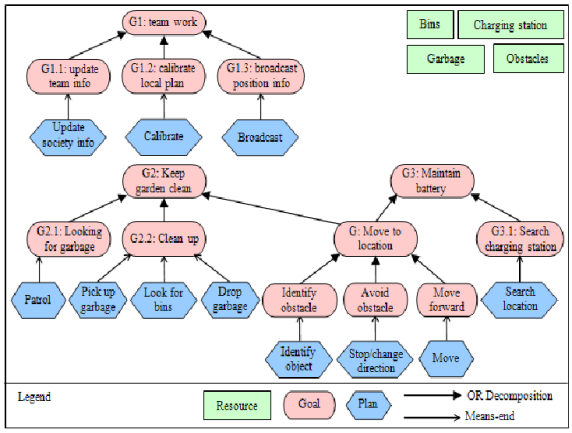


Figure 1 : Architecture of Mr. Cleaners

Testing levels:

We propose to divide the MAS testing process into different levels to better focus on the specific problems that may occur at each level. The five testing levels being proposed are: unit, agent, integration, system, and acceptance. Details are as follows:

Unit: test code units and modules that make up agents like goals, plans, beliefs, sensors, reasoning engine, and so on.

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

Integration: test 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; make sure that a group of agents and environmental resources work correctly together.

System: test the MAS as a system running at the target operating environment; test for quality properties that the intended system must reach, such as adaptation, openness, fault-tolerance, performance.

Acceptance: test the MAS in the customer execution environment and verify that it meets the stakeholder goals, with the participation of stakeholders.

A process model for goal-oriented testing:

The V-Model (Development Standards for IT Systems of the Federal Republic of Germany

2005) proposes a system development process, which defines a parallel flow of testing activities with respect to construction activities. The upper branch of the V (see Figure.5, turn this figure on end to see the V) represents the construction activities, and the lower branch of the V represents the testing flow where the application is tested against the artifacts defined on the upper-branch. The main trait of the V-model is that it represents explicitly the mutual relationships between construction artifacts and testing artifacts.

The modeling artifacts produced along the development process are:

Early Requirements model: a domain model (i.e. the organizational setting, as is)

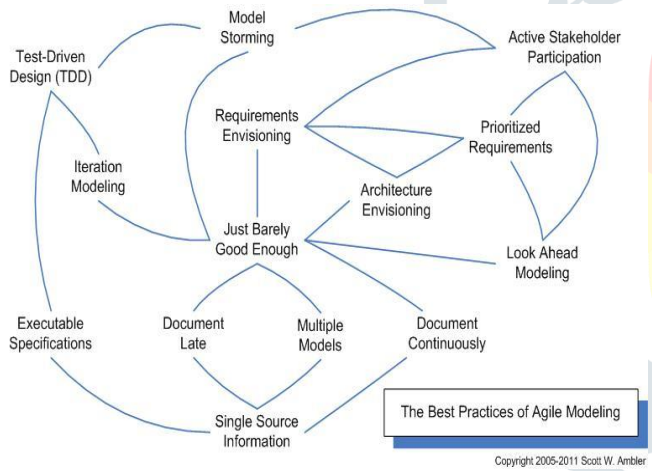


Figure 2: V process model for Goal-oriented testing

Late Requirements model: a model of the system-to-be where system requirements are modeled in terms of system goal graph

Architectural Design model: a system architecture model, specified in terms of a set of interacting software agents

Detailed Design model: a specification of software agent roles, capabilities, and interactions

Implementation artifacts: agent code and implementation documents

3. PROPOSED METHOD:

Testing techniques:

Testing can be subdivided into defining or generating test inputs and test scenarios, specifying test oracles to judge testing results, and executing test cases. The automated generation, to some extent, helps dealing with the dynamic nature of the environments where the agents under test operate. There are three types of techniques in testing:

Functional Testing:

The main quality factor in software is to meet its required functionality and behavior. The functional part of software includes the external behavior that mainly specifies all user requirements. The high level design of the software is produced so that the customer would be satisfied at an early stage of design and development. The functional testing revolves around the basic work flows and alternative flows of software.

Performance Testing:

This is one of a non-functional testing type which test performance of software under all favorable and nonfavorable conditions. This includes all time related parameters like Load time, access time, run time, execution time etc. This also includes success rate, failure frequency, mean time between failures and overall reliability of software. The most popular types of testing performed in Performance testing [14] are Stress testing and Load testing.

Security Testing:

Security testing basically follows two types of approaches: a. testing software regarding software's functional mechanisms b. performing risk based approach according to attackers' mindset. Penetration Testing is a security testing in which evaluators attempt to circumvent the security features of a system based on their understanding of the system design and implementation.

Monitoring:

In testing software agent and MAS, monitoring plays an important role as it allows us to observe the operation and interaction of the agents under test. It provides necessary data to detect abnormalities in the system, such as constraint violation, communication semantics mismatching, or requirement unsatisfaction. We propose two reference architectures for monitoring agent locally, Figure 3, and globally.

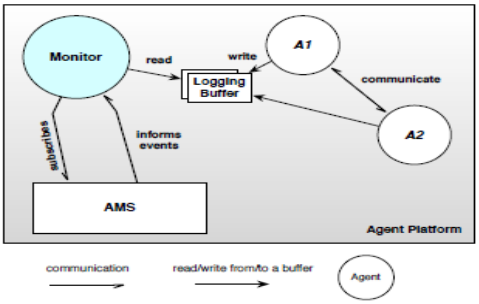


Figure 3: Reference architecture for monitoring one single platform

At the local level, in a single platform shown in Figure 4.4, a special agent named Monitor (or Monitoring Agent in other places) subscribes itself to the Agent Management System (AMS)3 (FIPA 2004) in order to be notified about all relevant events happen within the platform. These events include: an agent was born, is dead, is frozen, moves, adopts a goal, changes its beliefs, and the like. In particular, the AMS will inform the Monitor about any interactions, messages sent or received by the agents under test.

Results:

We conducted testing experiments with the goal-oriented (G), coverage enhanced goal-oriented (G+), and random (R) techniques on a computer equipped with 2G RAM, processor Core 2 Duo 1.86GHz (named Host in the following). The last technique, Evol-Mutation testing, was used with the original version of Bib Finder running on the Host and 15 mutants running on 3 cluster machines (4GB RAM, 4 CPUs Xeon 3GHz). These experiments were repeated 10 times for each technique in order to measure the average time and the ability to discover faults. Each execution time is composed of execution cycles, in which test cases are run on Bib Finder and its mutants. Test cases executed in each cycle can be the same in the goal-oriented and coverage-enhanced goal-oriented techniques; but they are different in random testing. In the Evol-Mutation testing, the test cases executed in a cycle are those from the previous cycle plus one or two new test cases generated by evolution.

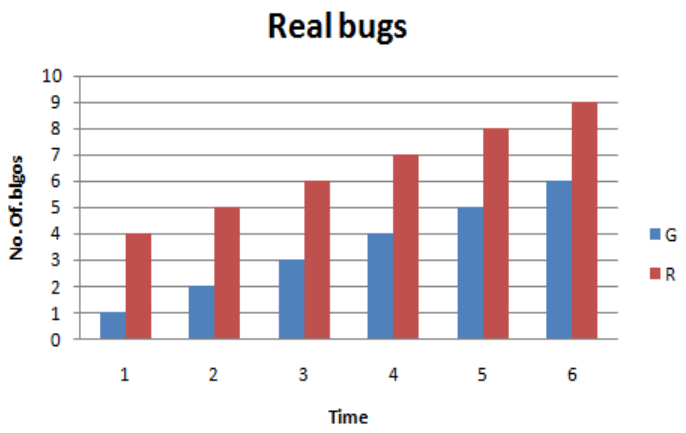


Figure 4: Real bugs revealed by cycle

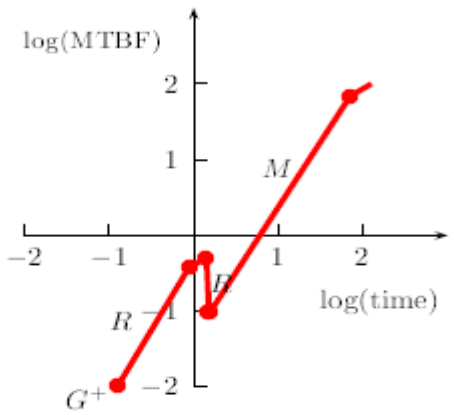
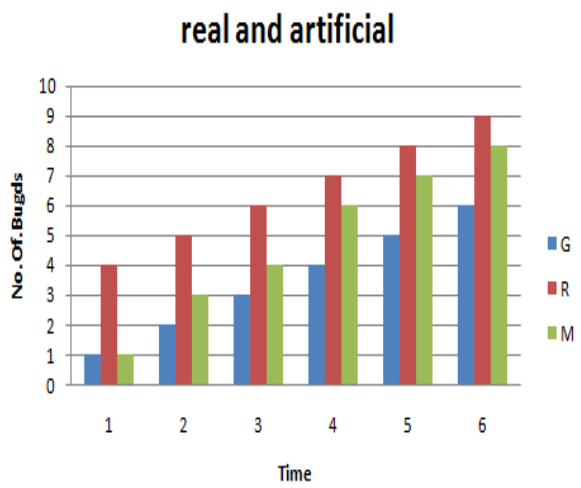


Fig 5: Total bugs revealed by cycle

Fig 6: Log-log plot of mean time

Faults are classified by the severity level (i.e. Fatal faults make agents die, Moderate faults are associated with discrepancies between implementation and specification). In the Cycle / generation column, we can find the average cycle (generation in the case of Evol-Mutation technique) when bugs were uncovered. One cycle of random testing costs less time than one of goal-oriented testing And Evol-Mutation. The number of bugs uncovered per cycle. We can notice that Random testing is quite effective in detecting fatal bugs. It actually revealed two real fatal bugs and one of them was not detected by any other technique. Goal-oriented testing revealed moderate bugs, showing that the implemented agents fail.



Bugs that are associated with a very long mean time between failures and are thus extremely hard (or impossible) to reveal in traditional testing sessions. Going back to Figure 6 we can notice that the goal-oriented and the random techniques are quite effective in the initial testing cycles, When bugs can be revealed by simple and short message sequences and the selection of the input data is not critical to expose them (i.e., there exist large equivalence classes of input data that can be used interchangeably to reveal a given fault). When remaining bugs become hard to find (last Testing cycles) goal-oriented and random testing becomes ineffective and it is only through Evol-Mutation that additional faults can be revealed.

4. CONCLUSION:

The increasing use of Internet as the backbone for all interconnected services and devices makes software systems highly complex and virtually

unlimited in scale. These systems often involve variety of users and heterogeneous platforms. They are evolved continuously in order to meet the changes of business and technology. In some circumstances, they need to be autonomous and adaptive for dealing with such changes. Software agents and MAS are considered as key enabling technologies for building such open, dynamic, and complex systems.

In evolutionary testing of agents, some research issues remain open. In our future work, we will consider multiple sets of simultaneous conflicting and competing requirements. For instance, in the cleaner agent one may want to evaluate robustness in terms of maintaining battery and avoiding obstacles. Since each requirement related to autonomy can give rise to a fitness function (or search objective), multiple requirements call for a multiobjective search technique. The multi-objective versions of evolutionary algorithms are probably suitable to deal with such situations.

REFERENCE:

1.Bhasin, Harsh.Khanna, Esha.Sudha.2014.Black Box Testing based on Requirement Analysis and Design Specifications International Journal of Computer Applications. (0975 – 8887) .18, 18(February 2014).

2.Swain, S.k.Mohapatra, D.P.Mall, R.2010.Terst Case Generation Based on Use Case and Sequence Diagram. *International journal of Software Engineering* (IJSE).3, 2.(July 2010),(289-321).

3.Sumalatha, Mary.Raju, G.2013. Object Oriented Test Case Generation Technique using Genetic Algorithms. International Journal of Computer Applications (0975-8887). 61,20 (January 2013).

4.Agent Oriented Software Pty. Ltd.n.d., JACK Agent Language, http://agent-software.com.au/jack.html.

5.Bordini, R. H., Wooldridge, M. and Hubner, J. F.: 2007, Programming Multi-Agent Systems in Agent Speak using Jason (Wiley Series in Agent Technology), John Wiley & Sons.

6.George_, M. P. and Ingrand, F. F.: 1989, Decision-making in an embedded reasoning system, International Joint Conference on Artificial Intelligence, pp. 972{978.

7.Gomez-Sanz, J. J., Bota, J., Serrano, E. and Pavon, J.: 2008, Testing and Debugging of MAS Interactions with INGENIAS, Proc. of the 9th Int. Workshop on Agent-Oriented Software Engineering.

8.Gotlieb, A., Denmat, T. and Botella, B.: 2007, Goal-oriented Test Data Generation for Pointer Programs, Information and Software Technology 49(9-10), 1030{1044.

9.Jureta, I., Faulkner, S. and Schobbens, P.-Y.: 2007, Achieving, satisfying, and excelling, Advances in Conceptual Modeling {Foundations and Applications pp. 286 295.

10. Nguyen, C. D., Perini, A. and Tonella, P.: 2007, Automated continuous testing of multi-agent systems, the fifth European Workshop on Multi-Agent Systems.

11. Nguyen, C. D., Perini, A. and Tonella, P.: 2008a, Constraint-based evolutionary testing of autonomous distributed systems, Proc. of the first Intl. Workshop on Search-Based Software Testing.

12. Nguyen, C. D., Perini, A. and Tonella, P.: 2008b, Experimental evaluation of ontology-based test generation for multi-agent systems, 9th International Workshop on Agent-Oriented Software Engineering, pp. 165176.

13. Nguyen, C. D., Perini, A. and Tonella, P.: 2008c, Ontology-based Test Generation for Multi Agent Systems, Proc. of the International Conference on Autonomous Agents and Multiagent Systems.

14. Weyns, D., Omicini, A. and Odell, J.: 2007, Environment as a first class abstraction in multiagent systems, Autonomous Agents and Multi-Agent Systems 14(1), 5{30.

15. Chauhan, Kumar, Vinod.2014.Smoke Testing-International Journal of Scientific and Research Publications4, 2 (February 2014),(2250-3153).