International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2013): 6.14 | Impact Factor (2013): 4.438

Methodologies and Technique for Software Agent

Kiran[1], Vijay[2]

[1]Department of Computer Science & Engineering, Meri College of Engineering.& Technology, Sampla, Bhadurgarh, Haryana-(India)

[2]B. Tech (IT), M.Tech (CSE), Department of Computer Science & Engineering, Sampla, Rohtak, Haryana-(India)

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 demand in particular to support systematic verification/validation and automated test generation and execution. This work deals with two major research problems: the lack of a structured testing process in engineering software agents and the need of adequate testing techniques to tackle the nature of software agents, e.g., being autonomous, decentralized, collaborative. 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. It encompasses the development process from the early requirements analysis until the deployment. We investigated how to derive test arte facts, i.e. inputs, scenarios, and so on, from agent requirements specification and design, and use these arte facts to refine the analysis and design in order to detect problems early. More importantly, they are executed afterwards to and defects in the implementation and build confidence in the operation of the agents under development. 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. These include goal-oriented, ontology-based, random, and evolutionary generation techniques. 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 and evaluating them. All together, the generation, evaluation, and monitoring techniques form a bigger picture: our novel continuous testing method. In this method, test execution can proceed 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: Application, ECAT, Goal Types, Testing Suit, Testing Types, Mas

1. Introduction

Software agents, with their peculiar properties, e.g., (semi-)autonomy, adaptivity, are key technologies to meet modern business needs, e.g., world- wide 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 Multi Agent System (MAS) has grown into a very active area, and interestingly they are receiving more industrial attention as well.

As these systems 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. This calls for an investigation of suit- able software engineering frameworks, including requirements engineering, architecture, and testing techniques, to provide adequate software development processes and supporting tools.

Testing of software agents and MAS is a challenging task because these systems are distributed, autonomous, and deliberative. They operate in an open world, which requires context awareness. 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 (Bergenti et al. 2004), but also to test. In particular, the very specific nature of software agents, which are designed to be autonomous, proactive, collaborative, and ultimately intelligent, makes it difficult to apply existing software testing techniques to them. For instance, agents operate asynchronously and in parallel, which challenges testing and de- bugging. 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. Moreover, agents can be programmed to learn; so successive tests with the same test data may give different results (Rouf2002).

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

2. Methodology

This Paper 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, (Dasani 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:

Descriptions
Goals that represent stakeholder objectives and requirements to- wards the system to-be. This type of goal is mainly identified at the early requirements phase of Tropos. 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. goals that require the agents of the system to-be to cooperate or share tasks, or goals that are related to emergent properties resulting from interactions. This type of goal can be called also as group goal. Goals that belong to or are assigned to particular agents. This type of goal appears when designing agent. Let's go back to our motivating example in Section 3.3. Goals shown in

3. Test Suit Derivation

This section introduces in details guidelines to derive test suites according to the proposed **V** process model. The guidelines contain four parts, as illustrated in Figure 3.6. First, we discuss how to derive test suites for acceptance test from organizational and system goals. Second, we discuss how system, collaborative, and agent goals are used to create system test suites. Next, as we move on in the development process to the agent interaction and capability design, we show how to exploit collaborative and agent goals to create integration test suites. Finally, we discuss in depth how to create test suites for agent plans, goals, and agents themselves. Examples are given in each part to illustrate the derivation. In addition, we also discuss when the derivations take place, when test suites are executed, and goal-oriented test adequacy at each test level

Acceptance test
Acceptance test suite derivation takes place at the Late Requirements phase, in parallel with the system analysis. At this stage, we have identified: actors, actors' goals, and dependencies between actors. Actors in the organizational setting include stakeholders, identified at Early Requirements phase, and system actors. Stakeholder actors present their intentions to the system actors by goal dependencies: they delegate goals to the system actors. In general, these goals represent users' objectives and intentions with regard to the system-to-be, so the fulfillment of these goals is a pivotal benchmark to the system acceptance. Thus, we will use them as foundations for acceptance test suites.

System Test
The transition from Late Requirements to Architectural Design phase consists of identifying agents that realize the specified system actors, assigning system actors' goals (called system goals) to agents goals, and projecting system actors' dependencies to agents dependencies and interactions. At this stage, apart from the arte facts (actors, goal models) obtained from the Late Requirements phase, there are agents, their goals, roles, collaborative goals, agents' dependencies for goals, resources, the dependencies between agents and the environment, regulations, constraints, and so forth. System test suites should consider and make use of these arte facts. System tests suite derivation takes place in parallel with architectural design. Similar to acceptance test suite derivation where we take stake- holder actors' goals as foundation concepts, we use system actors' goals as foundations to create system test suites as they provide the system-level objectives and requirements. When the system as a whole is built so that the system actors' goals (including functional hard goals and quality soft- goals) are fulfilled, it is ready to be passed to the customer for acceptance test

Integration Test
The aim of integration testing is to make sure that agents work together correctly - sharing tasks and resources - to achieve collaborative or agent goals. To obtain this objective, we consider dependencies between agents for collaborative goals and dependencies between agents and resources. In fact, these dependencies are sources that lead to interactions, i.e. agent- agent and agent-environment interactions. We can use them to derive test suites that exercise these dependencies and then evaluate the result of the interactions.

Agent Testing
An agent is composed 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 verify this integration. Agent-level test suites have a strong relation with test suites created for testing agent goals. Because, first of all, in most cases, testing a goal involves testing one or a number of plans, testing a plan involves events, percepts, and resources. So to some extent, testing a goal triggers some integration of plans, events, and so on. Hence, test suites derived to test agent goals are also effective to test the agent integration.

4. ECAT Testing Framework

We build a testing frame work called eCAT (stand for Environment for Continuous Agent Testing) to support the GOST methodology and different testing techniques . The framework consists of the TA, monitoring agent network, and tools for test case specification, graphical visualization, continuous execution, and fault reporting.. It consists of threemain components: Test Suite Editor, allowing human testers to derive test cases from goal analysis diagrams; TA, capable to generate automatically new test cases and to execute them on a MAS; and

Monitoring Agents, that monitor communication among agents, including the TA, and all events happening in the execution environments in order to trace and report errors. Remote monitoring agents are deployed with the environments of the agents under test, transparently to them, in order to avoid possible side effects. All the remote monitoring agents are under the control of the Central monitoring agent, which is located at the same host as the TA. The monitoring agents overhear agent interactions, events, and constraint violations taking place in the environments, providing a global view of what is going on during testing and helping the TA evaluate test results.

Generation & Execution Tool
Four test cases generation techniques are equipped to eCAT: Goal-oriented, Ontology-based,Random, and volutionary.

Goal-Oriented
Goal-oriented test cases generation is a part of the GOST methodology presented in 3 that integrates testing into Tropos, providing a systematic way of deriving test cases from Tropos output arte facts. eCAT can take these arte facts as inputs to generate test case skeletons that are aimed at testing goal fulfillment. Specific test inputs (i.e. message content), and expected outcome are partially generated from plan design (e.g., UML activity or sequence diagrams) and are then completed manually by testers.

Ontology-Based
eCAT takes advantage of agent interaction ontologies, which define the semantics of agent interactions, in order to generate automatically both valid and invalid test inputs, to provide guidance in the exploration of the input space, and to obtain a test oracle against which to validate the test outputs.

Random
eCAT is capable of generating random test cases. First, the TA selects a communication protocol among those provided by the agent platform, e.g., FIPA Interaction Protocol (FIPA 2002b). Then, messages are randomly generated and sent to the agents under test. The message format is that prescribed by the agent environment of choice (such as the FIPA ACLMessage (FIPA 2002a)), while the content is constrained by a domain data model. Such a model prescribes the range and the structure of the data that are produced randomly, either in terms of generation rules or in the (simpler) form of sets of admissible data that are sampled randomly.

Evolutionary
Evolutionary algorithms guided by mutation or quality-function-based fitness are implemented in eCAT, allowing it to evolve test cases during test execution. Based on monitoring data from the current execution, the TA can evolve the existing test cases (current population) to be more challenging ones for the next execution. All the above-mentioned techniques can be used in the continuous test execution mechanism of eCAT. Testing process is seen as a loop of generating, executing and monitoring, evaluating, evolving (only in evolutionary technique), then go back to generating. This continuous process makes it possible to test software agents extensively and automatically.

Monitoring Tool
eCAT contains a network of monitoring agents: the remote monitoring agents that side in agent platforms guard for events, violations, interactions happened at platform level during testing, while the central agent incorporates monitoring data from all the remote agents, makes the avail- able for evaluating test results and reporting. Multiple agent platforms that are used for testing can be located at a same host (i.e. computer) or at geographically different hosts thank to the monitoring network.

Application
The artificial environment is a square area, A. In the area A there can be obstacles, dustbins, waste, and charging stations located randomly. We define an environmental setting as a particular configuration of A, in which numbers of obstacles, dustbins, waste, and charging stations are located at particular locations. Different settings pose different levels of difficulty in which the cleaner agent must operate.

1. Explore location of important objects;
2. Look for waste and bring it to the closest bin;
3. Maintain battery charge, with sufficient re-charging;
4. Avoid obstacles by changing course when necessary;
5. Exhibit alacrity by finding the shortest path to reach a specific location, while avoiding obstacles on the way.
6. Exhibit safely by stopping gracefully when movement becomes impossible or battery charge level is too low.

5. Conclusion

The conclusion of the paper is focusion on the increasing use of Internet as the backbone for all interconnected ser- vices 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.

Now, as software agents with built-in autonomy are increasingly taking over control and management activities, such as in automated vehicles or e-commerce systems, testing these systems to make sure that they behave properly becomes crucial. This calls for an investigation of suitable soft- ware engineering frameworks, testing in particular, to build high quality and dependable software agents and MAS.

Testing software agents and MAS has been receiving much effort from several active research groups. However, there are still many open issues for research. A complete and comprehensive testing process for software agents and MAS is absent. We need adequate approaches to judge autonomous behaviours, to evaluate agents that have their own goals.

References

[1] Adrion, W. R., Branstad, M. A. and Cherniavsky, J. C.: 1982, Validation, verification, and testing of computer software, ACM Comput. Surv.**14**(2), 159-192
[2] Beizer, B.: 1990, Software Testing Techniques (2nd ed.), Van Nostrand
[3] Bergenti, F., Gleizes, M.-P. and Zambonelli, F. (eds): 2004,Methodolo-
[4] gies and Software Engineering for Agent Systems : The Agent-Oriented Software Engineering Handbook, Springer.
[5] Bordini, R. H., Wooldridge, M. and H¨ ubner, J. F.: 2007, Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology), John Wiley & Sons
[6] Bourque, P. and Dupuis, R. (eds): 2004, Guide to the Software Engineering Body of Knowledge: 2004 Edition, IEEE
[7] Cacciari, L. and Rafiq, O.: 1999, Controllability and observability in dis-tributed testing, Information and Software Technology **41**(11-12), 767- 780