Engineering Applications of Artificial Intelligence 26 (2013) 2128–2142

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

Designing multi-agent unit tests using systematic test design patterns-(extended version)☆

Mohamed A. Khamis [a,*], Khaled Nagi [b]

[a] Department of Computer Science and Engineering, Egypt-Japan University of Science and Technology (E-JUST), New Borg El-Arab City, Alexandria 21934, Egypt
[b] Department of Computer and Systems Engineering, Faculty of Engineering, Alexandria University, Egypt

ARTICLE INFO

Article history:
Received 21 December 2012
Received in revised form
3 March 2013
Accepted 18 April 2013
Available online 20 May 2013

Keywords:
Multi-agent unit tests
Test-driven development
Mock agent
Agent social design patterns
Code generation
Code coverage

ABSTRACT

Software agents are the basic building blocks in many software systems especially those based on artificial intelligence methods, e.g., reinforcement learning based multi-agent systems (MASs). However, testing software agents is considered a challenging problem. This is due to the special characteristics of agents which include its autonomy, distributed nature, intelligence, and heterogeneous communication protocols. Following the test-driven development (TDD) paradigm, we present a framework that allows MAS developers to write test scenarios that test each agent individually. The framework relies on the concepts of building *mock agents* and testing common *agent interaction design patterns*. We analyze the most common agent interaction patterns including *pair* and *mediation* patterns in order to provide stereotype implementation for their corresponding test cases. These implementations serve as test building blocks and are provided as a set of ready-for-reuse components in our repository. This way, the developer can concentrate on testing the business logic itself and spare him/her the burden of implementing tests for the underlying agent interaction patterns. Our framework is based on standard components such as the JADE agent platform, the JUnit framework, and the eclipse plug-in architecture. In this paper, we present in details *the design and function* of the framework. We demonstrate how we can use the proposed framework to define more *stereotypes* in the code repository and provide a detailed *analysis of the code coverage* for our designed stereotype test code implementations.

© 2013 Elsevier Ltd. All rights reserved.

1. Introduction

Nowadays, software agents are being used intensively in many real-world applications especially the on-line control applications which affect everyone's daily life, such as urban traffic signal control systems (e.g., Khamis and Gomaa, 2012). Agents interact in a concurrent, asynchronous and decentralized manner (Huget and Demazeau, 2004), thus MAS are complex systems (Jennings, 2001). In addition, the behaviors of agents are non-deterministic since there is a difficulty to determine a priori all interactions of an agent during its execution. Consequently, software agents are difficult to debug and test. As mentioned in Timm et al. (2006), there are five approaches for testing MASs, those are: testing, runtime monitoring, static analysis, model checking, and theorem proofing. In the recent years, much effort has been made to identify common interactions between agents and define their design patterns (Tahara et al., 1999; Kolp et al., 2002, 2005). However, defining standard test design patterns for these interactions have not yet been implemented. In this paper, we extend the original work presented in Nagi and Khamis (2011) that specifically deals with the *testing* approach. Our aim is to let the developer concentrate on testing the business logic of his/her MAS rather than the underlying framework and the different interactions between the participating agents. Thus, spare the developer the burden of implementing tests for the underlying agent interaction patterns.

In Caire et al. (2004), the authors proposed the PASSI MAS testing tool. This work presented diagrammatic notations to allow the developers to move easily from the design phase towards the implementation phase of the MASs. In addition, the authors proposed pattern reuse, deployment of MASs, and testing activities; a database of agents/tasks patterns that could be tested. An issue that needs to be addressed is testing the MAS at the society level where a group of different agents interact and their social behavior has to be evaluated. In Wang and Zhu (2012), the authors proposed a specification-based test automation framework through a tool called CATest for testing MAS. The correctness of agents behaviors are automatically checked against formal specifications. In Padgham et al. (2013), the authors presented a

☆In this paper, we extend the work published in Proceedings of the 5th International Conference on Intelligent Systems and Agents (ISA 2011), International Association for Development of the Information Society (IADIS).

* Corresponding author. Tel.: +20 3 309 4075, +20 100 638 2428 (mobile).
E-mail addresses: mohamed.khamis@ejust.edu.eg, mohamed.abdelaziz.khamis@gmail.com (M.A. Khamis).

0952-1976/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.engappai.2013.04.009

model-based oracle generation method for unit testing belief–desire–intention (BDI) agents.

In Nguyen et al. (2011), the authors provided a reference framework with a classification of MAS testing levels (such as unit, agent, integration, system, and acceptance); examples of unit testing level are: Tiryaki et al. (2007), Zhang et al. (2008), and Ekinci et al. (2009), whereas examples of agent testing level are: Coelho et al. (2006), Nguyen et al. (2008b), and Gómez-Sanz et al. (2009).

In Tiryaki et al. (2007), the authors proposed a test-driven MAS development approach that supports iterative and incremental MAS construction. In addition, they also introduced a testing framework called SUnit which supports the proposed approach by extending the JUnit framework. This framework allows the developers to write automated tests for agent behaviors and interactions between agents. The framework also includes the necessary mock agents to model the organizational aspects of the MAS.

In Coelho et al. (2007), the authors proposed the JAT framework for building and running MASs test scenarios. This framework relies on the use of aspect-oriented techniques to monitor the autonomous agents during tests and control the test input of asynchronous test cases. The proposed tool has been developed on top of the JADE development framework. In order to reduce the cost of developing a set of mock agents per test scenario, the authors had developed a generative template-based approach for mock agents, which generate the code of a mock agent from a protocol specification defined in an XML file; hence, a developer must define the communication protocol in an XML file, and the mock agent generator will generate the code of the required mock agent.

In Zhang et al. (2008), the authors enhanced the Prometheus Design Tool (PDT) to allow the automated unit testing of agents. Skeleton code can be generated from the detailed design of agents in PDT. The code generated is in the JACK agent-oriented programming language. The testing framework is based on model-based testing where the testing is based on the design models of the system. In Ekinci et al. (2009), the authors introduced a goal-oriented testing approach. The paper proposed a new concept called “test goal” for implementing unit tests; coding tests as goals provides easy refactoring from test code to source code and vice versa. The agent goals are the smallest testable units in the MASs. In addition, the authors introduced the SEAUnit testing tool which provides necessary infrastructure to support the proposed approach.

In Coelho et al. (2006), the authors proposed to test the smallest building block of the MAS, *the agent*. The aim of this approach is to verify whether each agent in isolation respects its specified tasks under *successful* and *exceptional* scenarios. A *mock agent* is a regular agent that communicates with just one other agent that is the agent under test (AUT). The plan of the mock agent is equivalent to a test script, since it defines the messages that should be sent to the AUT and asserts the messages that should be received from it.

In Nguyen et al. (2008a), the authors introduced the eCAT tool which supports deriving test cases semi-automatically from goal-based analysis diagrams, generates test inputs based on agent interaction ontology, and executes test cases automatically and continuously on MAS. The eCAT tool has been implemented as an Eclipse plug-in. It supports testing agents implemented in JADE and JADEX development frameworks.

In Nguyen et al. (2008b), the authors investigated software agents testing, and particularly the test generation automation. This approach takes the advantage of agent interaction ontologies which defines content semantic of those interactions to: (1) generate test inputs, (2) guide the exploration of the input space during generation, and (3) verify messages exchanged among agents with respect to the defined interaction ontology. The proposed approach is integrated into the eCAT testing framework.

In Gómez-Sanz et al. (2009), the authors introduced advances on the INGENIAS agent development framework towards a complete coverage of testing and debugging activities.

In our work, we design test patterns for the *ten* most famous MAS interaction design patterns found in the literature. We design and implement an *Eclipse plug-in* to enable the MAS unit test developer to generate the *mock agent*(*s*) that interact with the AUT and follow one of the identified *social design patterns* (the term ‘*social design patterns*’ is defined in details in Section 2.2). The AUT and the mock agents run within the JADE Platform (Bellifemine et al., 2005). The MAS unit test developer has the ability to add further test design patterns for an existing agent interaction pattern or for a newly identified one. By this way, the MAS developer will focus on testing the business logic of the MAS without the burden of implementing unit tests for the design pattern itself. The proposed framework provides the first implementation that usually triggers the continuous refactoring process that is typical to the TDD paradigm. The developer can use the reflection capabilities of the Eclipse SDK to reflect the changes made in the AUT directly into the generated mock agent. The *repository* consists of a set of XML and java files that represent the behavior of the different mock agents existing in the implemented design patterns. In our work, we provide implementations for a vast majority of the agent design patterns. We evaluate the code coverage by using EMMA (Roubtsov, 2006), a code coverage tool, to demonstrate that the generated test files (mock agent, associated resource files, and AUT test cases) completely cover the AUT code for the agent interaction pattern.

The remainder of this paper is organized as follows. Section 2 presents background on the MAS unit testing approach based on mock agents. Section 3 illustrates the details of the contribution through the detailed design of our framework that is used in designing mock agents-based test cases. Section 4 presents the design of the test design patterns. Section 5 analyzes the quantitative results using the EMMA code coverage tool. Finally, Section 6 presents some conclusions and directions for future work.

2. Background

2.1. MAS unit testing approach using mock agents

We adopted the MAS testing approach presented in Coelho et al. (2006). This agent unit testing approach has two main concerns: using mock agents in test case design and executing the test case. These two concerns are analyzed in Coelho et al. (2006) and are summarized below. A complete testing for a MAS is almost impossible such that all agents specifications need to be verified, thus designing a proper test-cases is a challenging task. The aim is to make the designed test cases as complete as possible by choosing the subset from all possible test cases that will likely detect the most errors. The choice of this subset is definitely constrained by time and space complexities (the test-case design paradigm of the lowest effect would be choosing a random subset from all possible test cases that would have a low chance to detect the most errors).

To the best of the authors' knowledge, the unit testing approaches for MAS proposed in the literature do not define a methodology for test-case selection. However, in the work presented in Myers (2004), the author suggests a test-case design paradigm that is based on an error-guessing technique. This technique is based on enumerating a list of possible error-prone situations and

Download English Version:

https://daneshyari.com/en/article/380754

Download Persian Version:

https://daneshyari.com/article/380754

Daneshyari.com