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.

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

then writing test cases based on that list. The pseudo-code presented in Coelho et al. (2006) sketches this approach:

1. **For the** agent to be tested
   1.1. List the set of roles that it plays
2. **For each** role played by the AUT
   2.1. List the set of other interacting roles
3. **For each** interacting role
   3.1. Implement in the Mock Agent a *plan* that codifies a *successful scenario*.
   3.2. List possible *exceptional scenarios* that the Mock Agent can participate in.
   3.3. Implement in the Mock Agent an extra plan that codifies each *exceptional scenario*.

The test-cases designer should apply this procedure on each agent in the MAS under study (or at least for the subset of agents that are responsible on the most important tasks). By the end of this procedure, a *mock agent* will compromise the expected interactions (under successful and exceptional scenarios) of an agent interacting with the AUT. The *mock agent* plan codifies the logic of the test. Each test case starts both the AUT and the corresponding *mock agents* and waits to be notified from the *agent manager* that the interaction between the agents has been finished to ask the *mock agents* whether or not the AUT acted as expected.

2.2. Agent social design patterns

As mentioned in Tahara et al. (1999), the concept of design patterns can reduce the development costs and signify the software reuse in MASs. A software pattern describes a recurring problem and proposes its solution (Kendall et al., 1998). This may include conceptual, architectural, or design problems. In addition, the proper utilization of design patterns leads to a successful implementation of MASs (Hayden et al., 1999). Multi-agent design patterns present a set of standard interactions and guide MAS designers. For example, in Castro et al. (2002), the authors proposed the Tropos requirements-driven framework based on a set of design patterns, namely *social patterns* (Kolp et al., 2002) that focus on the social and intentional aspects that are recurring in multi-agent and cooperative systems.

The framework presented in Kolp et al. (2005) has classified the social patterns into two categories: *pair* patterns and *mediation* patterns. Pair patterns (such as booking, call-for-proposals, subscription, or bidding) describe direct interactions between negotiating agents. Mediation patterns (such as monitor, broker, matchmaker, mediator, embassy, or wrapper) describe intermediary agents that help other agents reach an agreement on services exchange. In our work, we consider each of these design patterns and design the suitable testing scenarios following the methodology mentioned in Section 2.1.

3. Mock agent test cases design framework

A brief sketch of the framework was presented in Nagi and Khamis (2011). In the following subsections, we describe the proposed framework in details. We adopt the *agile software engineering* approach (Beck and Andres, 2004) that addresses testing continuously within the implementation process. The basic idea is the ability of developers to formulate *test-cases* during or even before implementation, which are automatically executed on demand. This procedure is also known as "Test-driven Development" (Beck, 2002). We focus on the *unit test level* where the implementation is tested by *unit test* during the coding.

3.1. Component interaction

The interaction between the different components of the proposed framework is illustrated by the collaboration diagram sketched in Fig. 1 that represents the integration between the *workflow* proposed in Coelho et al. (2006) and the MAS unit test cases designer proposed in this paper.

Test case design is done through a set of steps (as depicted in Fig. 1). Initially, the mock agent designer chooses which design pattern his/her AUT follows from the plug-in repository. Accordingly, the necessary files employed in the test scenario will be generated that include: the mock agents java source files, the associated resource files, and the AUT test cases. The test suite instantiates the test case class and sets up the test environment. After that, the framework creates the agent platform. During the test case execution, the framework creates a mock agent to every role that interacts with the AUT. Next, the test case creates the AUT and asks the agent manager for notification when the interaction between the AUT and the mock agent is complete. The AUT and the mock agent begin to interact (through the messages exchanged according to the interaction protocol). This process allows the mock agent to perform the test codified in its testing plan.

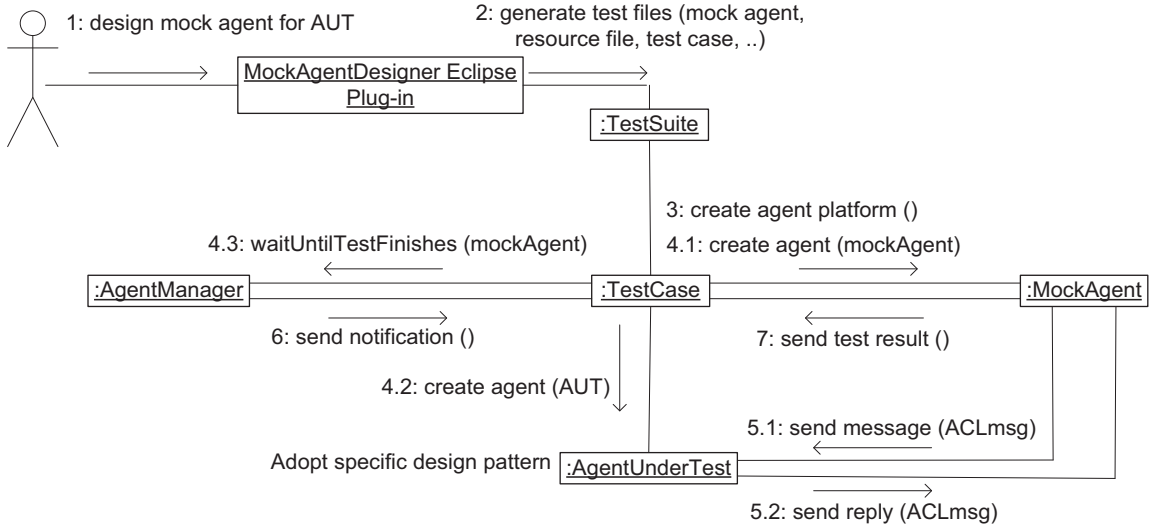

Fig. 1. Collaboration diagram of the components of the proposed framework.

When the mock agent concludes its plan, the agent manager notifies the framework that the interaction between the mock agent and the AUT has been concluded. Finally, the test case records whether or not the AUT acted as expected in response to the mock agent interaction. In scenarios, where the mock agent is the one who initiates the interaction process (i.e., test driver), the framework will create the mock agent after the AUT creation. Also, in scenarios, where the AUT needs to interact with more than one mock agent, the framework will create two types of mock agents; the first type is the simple mock agents that only send simple messages for the integrity of the interaction protocol, and the second type is the more sophisticated mock agents that validate the order and the content of the exchanged messages.

3.2. System design

The design of the proposed framework is illustrated in Fig. 2 that resembles with the main modules of the extended JUnit framework presented in Coelho et al. (2006). The JUnit framework includes six main components: *Test*, *TestRunner*, *TestResult*, *TestSuite*, *TestCase*, and *AssertionFailedError*. The *Test* interface contains two abstract methods; *countTestCases()* for counting the number of test cases that will be run by this test, and *run(TestResult result)* for running a test and collecting its result in a *TestResult* instance. The *TestRunner* class is used to execute the *Test*. The test results are collected in a *TestResult* instance. The tests can be grouped in one *TestSuite* class that runs a collection of test cases. A *TestCase* defines the fixture to run multiple tests. Finally, the *AssertionFailedError* is thrown when an assertion failed.

The exploited JADE sub-framework includes four main components: *Agent*, *ContainerController*, *AgentController*, and *Behaviour*. The *Agent* class is the common super class for user-defined software agents. It provides methods to perform basic agent tasks, such as message passing, both uni-cast and multi-cast with optional pattern matching (i.e., receiving a message based on fields matching with the sent message). The class *ContainerController* is a proxy class, allowing access to a JADE agent container and the *AgentController* is a proxy class, allowing access to a JADE agent. The *Behaviour* class is a super class for the user-defined JADE behaviours.

The *MASUnitTesting* framework contains five components: *JADETestCase*, *AgentManager*, *JADEMockAgent*, *TestResultReporter*, and *ReplyReceptionFailed*. The *JADETestCase* extends the *TestCase* class from the *JUnit* framework. It is the super class of all AUT test cases and contains two main methods *createEnvironment()* and *createAgent()*. The *createEnvironment()* method is responsible for creating the JADE environment that will be active during the execution of the test scenario. Each test method will be able to include agents in such environment by calling the *createAgent()* method.

The *JADEMockAgent* class implements three main methods: *sendMessage()* which triggers the mock agent test interaction, *receiveMessage()* that performs assertions concerning the received message (e.g., whether the message was received within a specific timeout, or if it obeys a pre-defined format), and *prepareMessageResult()* that prepare the test result with any failure occurred in the test scenario. During

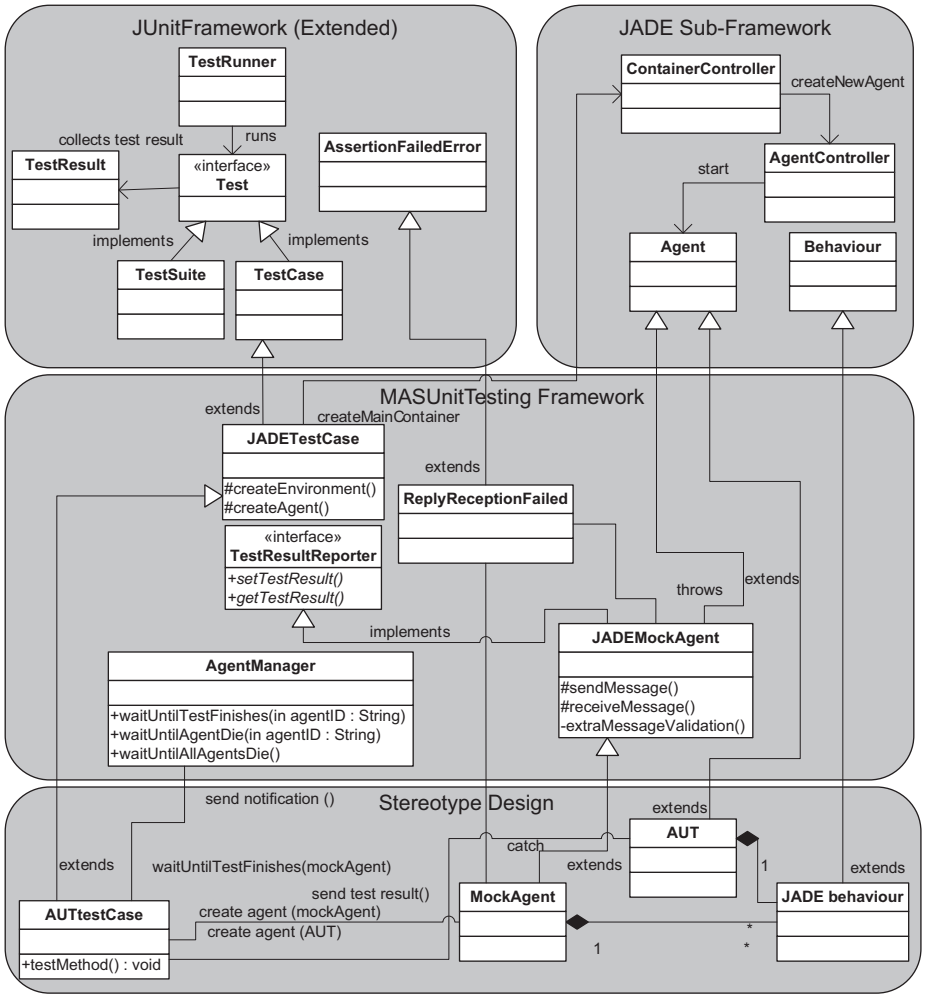

Fig. 2. UML class diagram of the MockAgentDesigner framework.

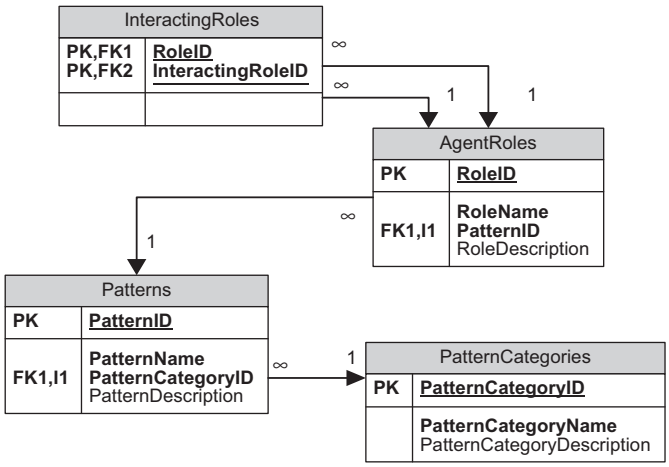

Fig. 3. Entity Relationship Diagram (ERD) of the MockAgentDesigner framework.

execution, the *JADEMockAgent* needs to report the test result (success or failure) to the AUT *test case*, which in counterpart, will be in charge of examining the test result. For reporting the result of the test, the *JADEMockAgent* class implements the *TestResultReporter* interface that has two methods: *setTestResult(TestResult testResult)* and *getTestResult()*. The test case stereotype design is depicted in Section 3.3.

3.3. Test cases stereotypes design and definition

In our proposed framework, we offer an extendible set of pattern categories. Every pattern category contains a set of agent interaction design patterns. In each of these design patterns, a set of agent roles can be defined. Every agent role has a set of interacting roles, and every interacting role can interact with more than one agent role. Fig. 3 represents the Entity Relationship Diagram (ERD) of the proposed mock agent designer framework.

Any stereotype contains three main components: the AUT, the mock agents, and the AUT test cases. As mentioned earlier, the AUT is the agent whose behavior is verified by a set of test cases. A mock agent has a testing plan (represented by one or more JADE behaviors) to test the AUT. These behaviors allow the mock agent to send, receive, and check the content of the messages received from the AUT; the *receive()* method throws an instance of the *ReplyReceptionFailed* class in case of an exceptional scenario.

Both of the successful and exceptional scenarios can be created by changing the parameters of the messages sent from the *MockAgent* to the AUT (e.g., setting the resource title by "other-service-title" in the call-for-proposals design pattern for an exceptional scenario simulation). The *AUTtestCase* is a subclass of the *JADETestCase* class. Inside this test case, we implement a *testMethod()* that creates an instance of the AUT and an instance of the *MockAgent* that interacts with this AUT. After that, the *testMethod()* asks the *AgentManager* to be notified when the interaction between the AUT and the *MockAgent* finishes in order to ask the *MockAgent* whether or not the AUT acted as expected (test result). Each *AUTtestCase* will be able to include agents in the created JADE environment by calling the *createAgent()* method that exists in the parent *JADETestCase*. We describe the way to define the test cases stereotypes using some examples in Section 4.

4. Defining test case stereotypes with the eclipse plug-in

4.1. Current repository contents

In our work, we designed and implemented the mock agents for *ten* interaction design patterns that can be classified into two main categories (Kolp et al., 2005): (1) *pair* patterns such as booking (Do et al., 2003b), call-for-proposals (Coelho et al., 2006), subscription (Do et al., 2003a), and bidding (FIPA, 2000), which describe direct interactions between negotiating agents, (2) *mediation* patterns such as monitor (Kolp et al., 2005), broker (Do et al., 2003a), matchmaker (Silva et al., 2005), mediator (Kolp et al., 2005), embassy (Hayden et al., 1999), and wrapper (Kolp et al., 2005), which describe intermediary agents that help other agents to reach an agreement on services exchange. In the following two subsections, we briefly list all use cases for the sake of completeness. In Section 5, we present the results of code coverage when running all test cases stereotypes for all the agents in these design patterns.

4.2. Pair patterns

In this paper, we provide only a detailed sample for designing test patterns for one of the *pair* patterns category that is the *booking* design pattern. The test cases of the rest of design patterns in this category are stated briefly with the main successful scenario only.

4.2.1. Booking pattern

In the *booking* pattern (Do et al., 2003b), a client sends a reservation request containing the characteristics of a resource to a service-provider. The service-provider may alternatively answer with a refusal, a waiting-list-proposal or the resource-proposal (when there is such a resource that satisfies the sent characteristics). When the client accepts a waiting-list-proposal, it sends a waiting-list-time-out to the service-provider. By the time-out deadline, the service-provider must send a refusal or a resource-proposal to the client. A resource that is not available becomes available when some client cancels its reservation.

Two roles are identified: the *client* and the *service-provider agents*. In this section, we elaborate only on the booking-client role, which would be the AUT. Thus, the mock agent would be the service-provider agent. The interaction begins with sending the request-reservation message to the mock service-provider agent. The mock agent simulates a check for availability of the resource in the catalog and assumes its reservation, thus issuing a waiting-list-proposal message back to the AUT client agent. The AUT normal behavior is to accept the proposal and sends a waiting-list-deadline. The mock agent simulates a re-check for availability, assumes the readiness of the resource, checks that the deadline has not been reached and sends a resource-reservation-price-proposal with an unbeatable price (e.g., 0,–€).

The AUT should accept the price-proposal and sends a notification to the mock agent. The mock agent simulates a check on the availability of the resource and assumes it is not reserved by another client. As a result, the mock agent sends an inform-reservation-completed message to the AUT in order to avoid any timeouts from the side of the AUT. In the exceptional scenarios, all successful assumptions previously made by the mock agent will be reverted, one at a time. Thus, the first exceptional behavior is illustrated in Fig. 4a in which the mock agent sends a resource-unavailable message and responds immediately after receiving the reservation-request message. The second exceptional scenario is illustrated in Fig. 4b in which the mock agent sends a reservation-failure message upon receiving the accept-reservation-price-proposal message from the AUT. This analysis is carried on till we define *five* exceptional scenarios based on the successful one.

In order to complete the test suite for this interaction pattern, the test case stereotype of reversing the roles of the AUT and the mock agent is done as well in our repository.

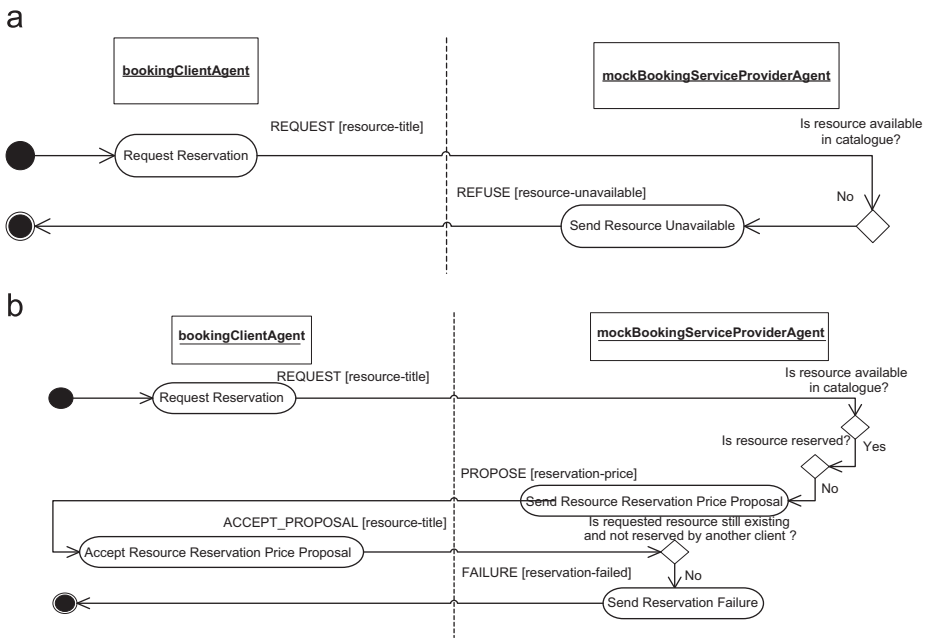

Fig. 4. (a) Sequence diagram for the first exceptional scenario: testing booking client agent. (b) Sequence diagram for the second exceptional scenario: testing booking client agent.

4.2.2. *Call-for-proposals pattern*

In the *call-for-proposals* pattern (Coelho et al., 2006); as soon as a seller agent joins the environment, it registers itself in a service-directory (which is part of the JADE framework) as a "service-seller" and starts to wait for "service-buying" requests. When a buyer agent joins the environment, it initially looks for the agents already registered in the service-directory as "service-sellers". After that, it sends a call-for-proposals message to all the agents registered as "service-sellers". When the seller agent receives a call-for-proposals message from a buyer, it searches in its catalog for the requested service. If it is available, the seller agent sends a "propose" message in reply to the call-for-proposals message, whose content is the service-price.

If on the other hand, the seller agent does not have the service in its catalog; it will send a "refuse" message informing the buyer agent that the service is not available. The buyer agent receives all proposals/refusals from seller agents and chooses the agent with the best offer. Then, it sends the chosen seller a "purchase" message. When the seller agent receives a "purchase" message it removes the service from the catalog and sends an "inform" message to notify the buyer agent that the service sale was completed. However, if for any reason the service is no more available in the catalog the seller agent sends a "failure" message informing the buyer agent that the requested service is no more available. If the buyer agent receives a message indicating that the sale was completed, the agent can terminate. Otherwise, it will re-execute its plan and try to buy the service again from some other agent.

In this section, we elaborate only on the buyer role, which would be the AUT. Thus, the mock agent would be the seller agent. Fig. 5 illustrates the sequence diagram of the successful scenario in which the call-for-proposals interaction completes successfully.

As in the previous use case, in the exceptional scenarios, all successful assumptions previously made by the mock agent will be reverted, one at a time.

4.2.3. *Subscription pattern*

The *subscription* pattern (Do et al., 2003a) involves a "yellow page" agent and a number of service-providers. The providers advertize their services by subscribing to the "yellow page". A provider that no longer wishes to be advertized can request to be unsubscribed. The "yellow page" agent allows agents to publish one or more services they provide so that the other agents can find and successively exploit them. The "yellow page" agent uses the underlying "service-directory" which is part of the JADE Framework.

In this section, we elaborate only on the "yellow page" role, which would be the AUT. Thus, the mock agent would be the service-provider agent. Fig. 6 illustrates the sequence diagram of the successful scenario in which the subscription interaction completes successfully.

4.2.4. *Bidding pattern*

The bidding pattern is in a great deal similar to the standard FIPA Contract Net Interaction Protocol (FIPA, 2000). In the *bidding* pattern (a.k.a. English auction), the auctioneer seeks to find the market price of a good by initially proposing a price below that of the supposed market value and then gradually raising the price. Each time the price is announced, the auctioneer waits to see if any buyers will signal their willingness to pay the suggested price. The auctioneer issues a new call-for-bids to the confirmed bidder agents with an incremented price than the best accepted proposed price. The auction continues until no buyers are prepared to pay the proposed price, at which point the auction ends. If the last price that was accepted by a buyer exceeds the auctioneer (privately known) reservation price, the good is sold to that buyer for the agreed price. If the last accepted price is less than the reservation price, the good is not sold.

In this section, we elaborate only on the auctioneer role, which would be the AUT. Thus, the mock agent would be the bidder agent. Fig. 7 illustrates the sequence diagram of the successful scenario in which the bidding interaction completes successfully.

4.3. *Mediation patterns*

In this paper, we provide only a detailed sample for designing test patterns for one of the *mediation* patterns category that is the *broker* design pattern. Similarly, the test cases of the rest design

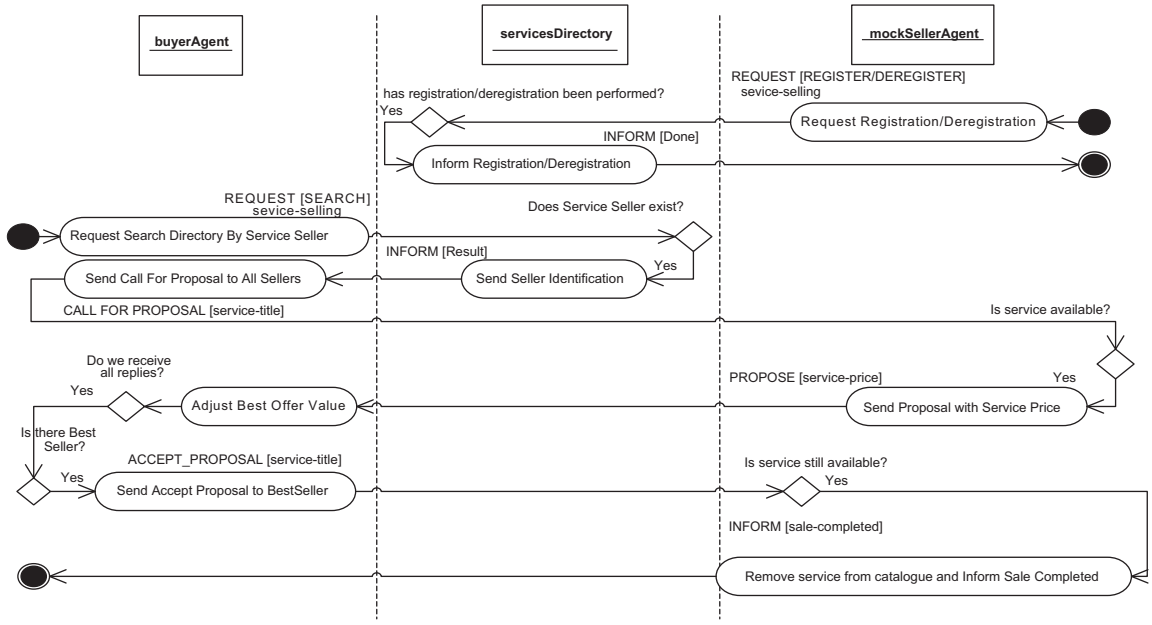

Fig. 5. Sequence diagram for a successful scenario: testing call-for-proposals buyer agent.

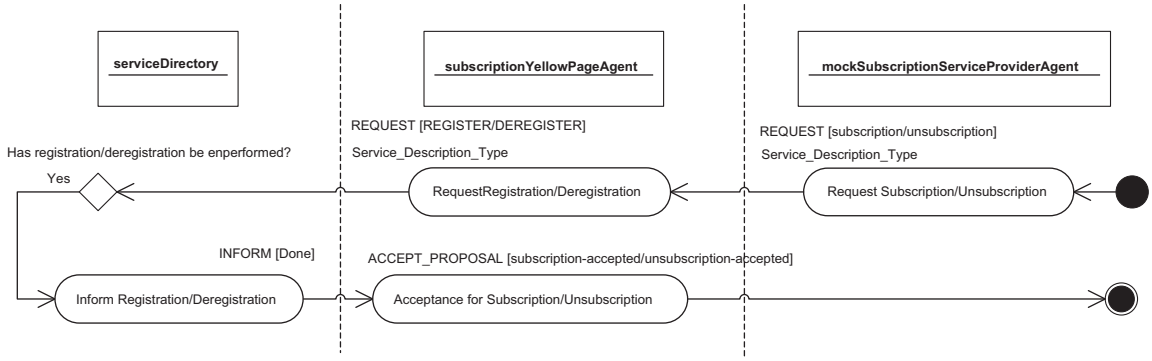

Fig. 6. Sequence diagram for a successful scenario: testing subscription yellow page agent.

patterns in this category are stated briefly with the main successful scenario only.

4.3.1. Broker pattern

In the *broker* pattern (Do et al., 2003a), the broker agent is an arbiter and intermediates the access to the services of a service-provider agent to satisfy the request of a client. The client sends a service-request containing the characteristics of the service it wishes to obtain from the broker. The broker may alternatively answer with a refusal or an acceptance. In the case of an acceptance, the broker sends a call-for-proposal to the registered service-providers. The call-for-proposals pattern is then applied to model the interaction between the broker and the service-providers. The service-provider either fails or achieves the requested service. The broker then informs the client about this result by sending an inform-failure-service-request or an inform-service-price messages, respectively.

Three roles are identified in this interaction pattern: the *client*, the *service-provider* and the *broker* agents. In this section, we first elaborate on the client role, which would be the AUT. Thus, the mock agent would be the broker agent. After that, we elaborate on the broker role, which would be the AUT, and the mock agent would be the service-provider (note that in the case of the AUT broker agent, there is another mock agent for the other interacting role that is the mock client agent). Fig. 8 illustrates the sequence diagram of the successful scenario in which the broker interaction completes successfully. The interaction begins with sending the request-service-provider message to the mock broker agent. The mock agent simulates a check for availability of the service-description-type and sends a service-providers-existence message back to the AUT client agent. The AUT should confirm the service-providers-existence and send a confirmation to the mock agent. Immediately, the mock agent sends an inform-service-price message to the AUT.

As in the previous use cases, in the exceptional scenarios, all successful assumptions previously made by the mock agent will be reverted, one at a time. Thus, the first exceptional behavior is illustrated in Fig. 9a in which the mock agent sends a service-description-type-unavailable message and responds immediately after receiving the service-provider-request message. The second exceptional scenario is illustrated in Fig. 9b in which the mock agent sends a service-failure message upon receiving the service-request-acceptance-confirmation message from the AUT.

In order to complete the test suite for this interaction pattern, the test case stereotype of reversing the roles of the AUT and the mock agent is done as well in our repository.

Fig. 10 illustrates the sequence diagram of the successful scenario in which the broker service-provider interaction completes successfully. The interaction begins with sending the request-subscription in yellow pages message to the AUT broker agent. The broker agent replies with subscription-request-acceptance message. The broker

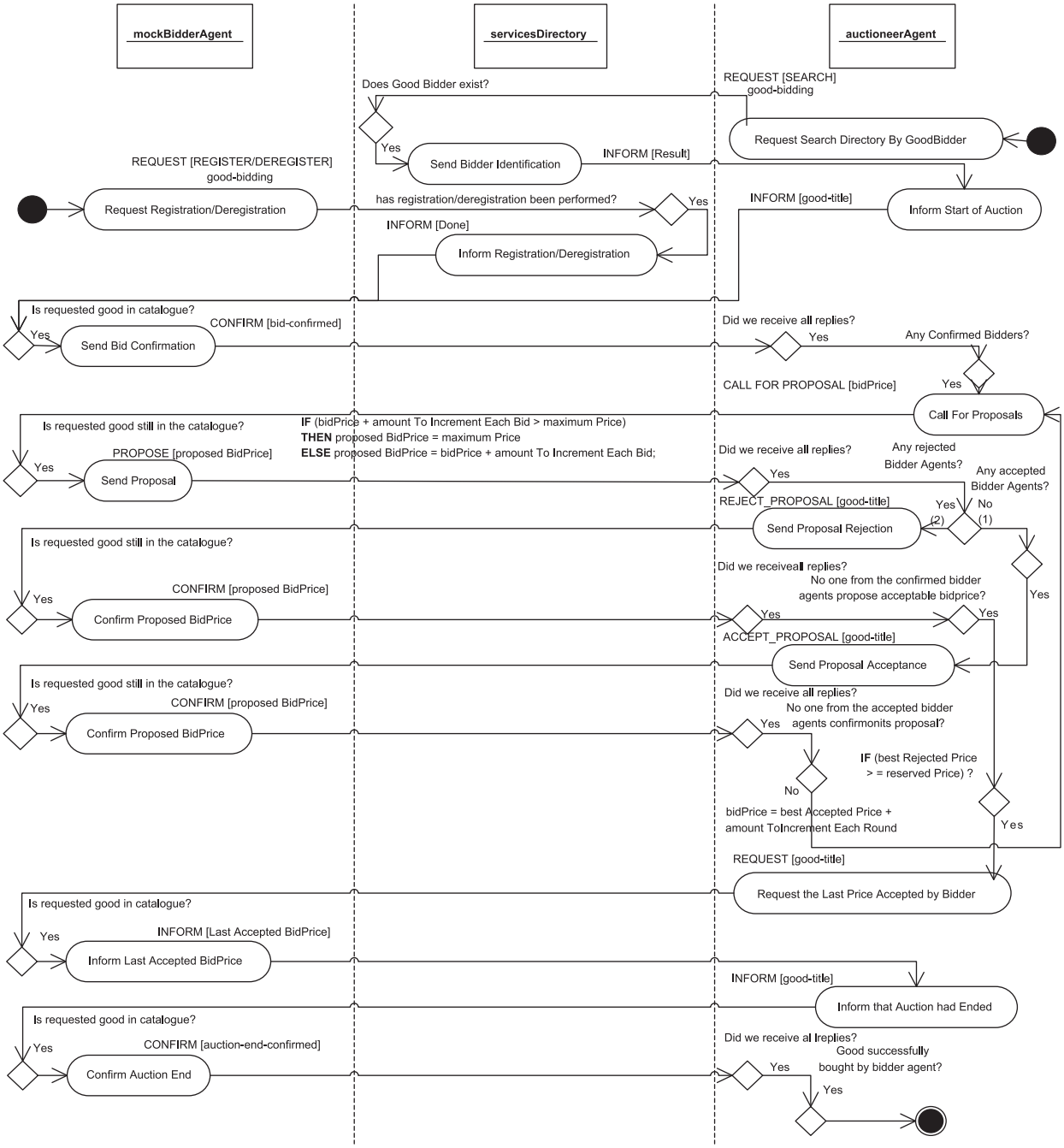

Fig. 7. Sequence diagram for a successful scenario: testing bidding auctioneer agent.

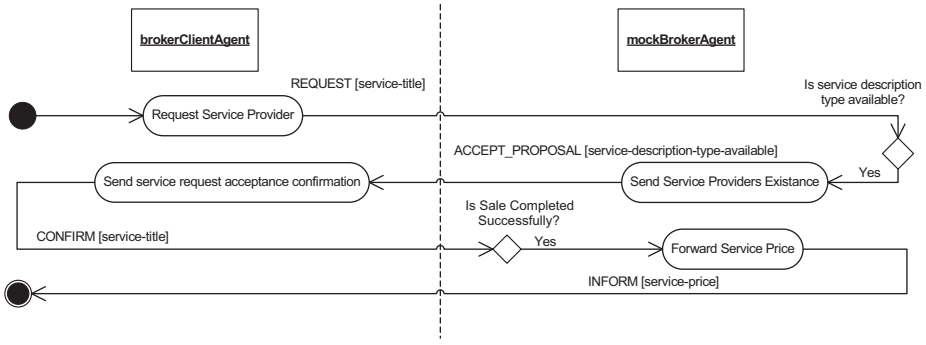

Fig. 8. Sequence diagram for a successful scenario: testing broker client agent.

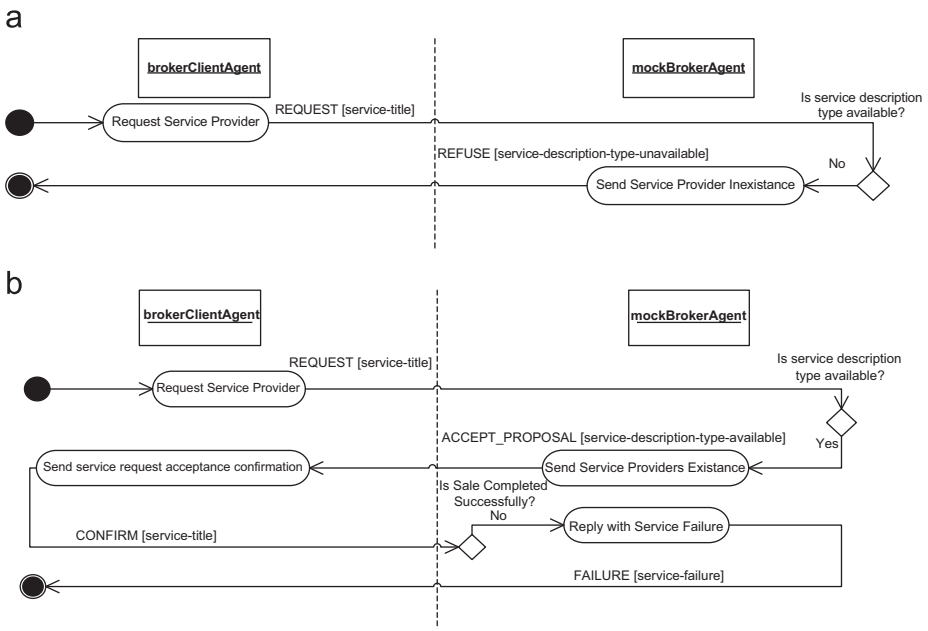

Fig. 9. (a) Sequence diagram for the first exceptional scenario: testing broker client agent. (b) Sequence diagram for the second exceptional scenario: testing broker client agent.

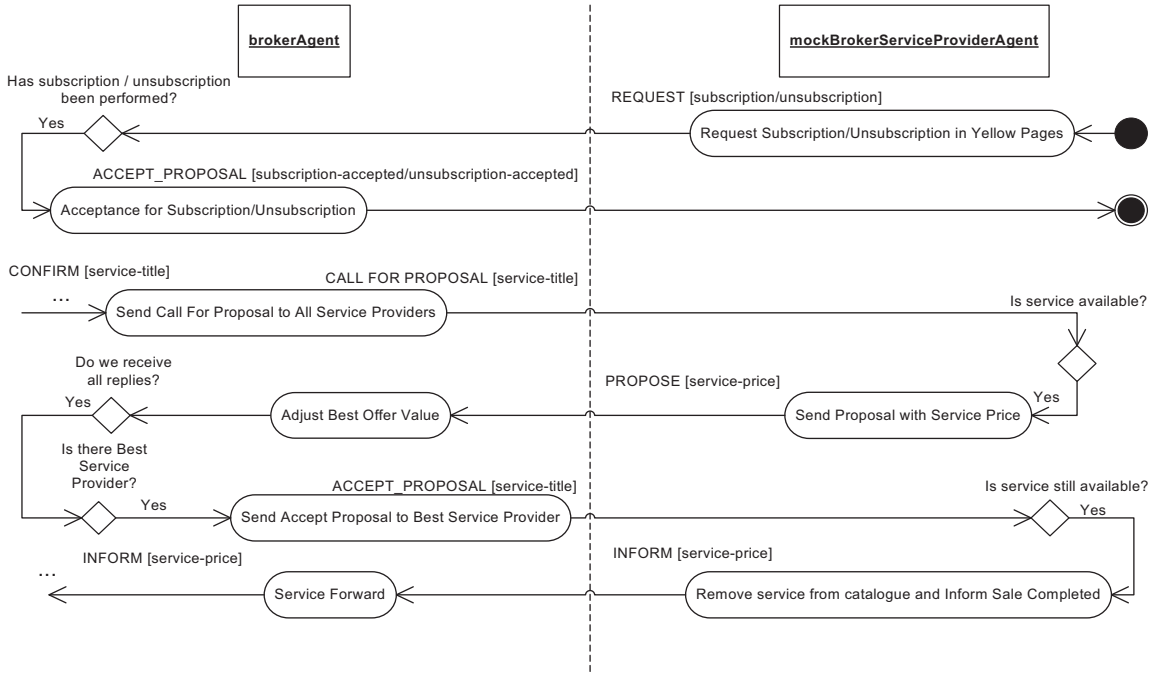

Fig. 10. Sequence diagram for a successful scenario: testing broker agent.

agent sends call-for-proposal with the service-title. The mock agent simulates a check for availability of the service and sends a proposal with the service-price message back to the AUT broker agent. The AUT will adjust the best offer value and checks whether it received all replies and whether there is best service-provider and sends an accept-proposal message to the best service-provider (that is the mock agent in this case). Immediately, the mock agent sends an inform-service-price message to the AUT.

As in the previous use cases, in the exceptional scenarios, all successful assumptions previously made by the mock agent will be reverted, one at a time. Thus, the first exceptional behavior is illustrated in Fig. 11a in which the mock agent sends a service-unavailable message and responds immediately after receiving the broker call-for-proposal message. The second exceptional scenario is illustrated in Fig. 11b in which the mock agent sends a service-failure message upon receiving the accept-service-proposal message from the AUT.

In order to complete the test suite for this interaction pattern, the test case stereotype of reversing the roles of the AUT and the mock agent is done as well in our repository.

4.3.2. *Monitor pattern*

The *monitor* pattern (Kolp et al., 2005) involves at least one monitor agent, a number of subscriber agents and at least one subject

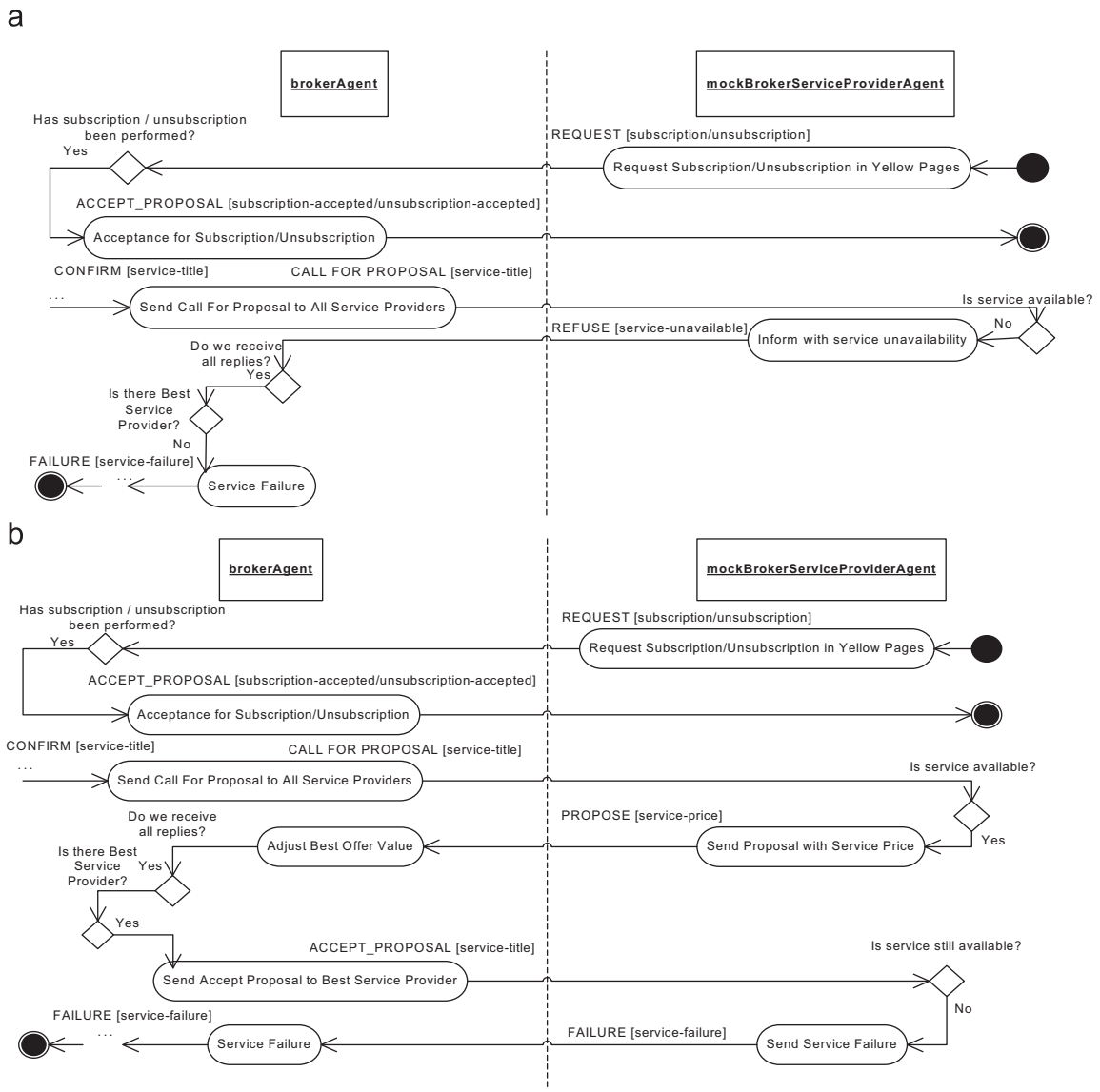

Fig. 11. (a) Sequence diagram for the first exceptional scenario: testing broker agent. (b) Sequence diagram for the second exceptional scenario: testing broker agent.

or event of interest agent. Subscribers register for receiving, from a monitor agent, notifications of the changes in the state of some subjects of their interest. The monitor accepts subscriptions, request notifications from subjects of interest, receives notifications of events and alerts subscribers with relevant events. The subject (source) agent provides notifications of state changes as requested.

In this section, we elaborate only on the monitor role, which would be the AUT. Thus, the mock agents would be the notification subscriber and the notification source agents. Fig. 12a, b illustrate the sequence diagrams of the two successful scenarios in which the monitoring interaction completes successfully.

4.3.3. Matchmaker pattern

In the *matchmaker* pattern (Silva et al., 2005), a matchmaker agent locates a provider agent corresponding to a consumer request for a service, and then hands the consumer a direct handle to the chosen provider. Contrary to the broker who directly handles all interactions between the consumer and the provider, the negotiation for some service and the actual service provision are two distinct phases.

In this section, we elaborate only on the matchmaker client role, which would be the AUT. Thus, the mock agents would be the matchmaker and the provider agents. Fig. 13a, b illustrate the sequence diagrams of the two successful scenarios in which the matchmaking interaction completes successfully.

4.3.4. Mediator pattern

In the *mediator* pattern (Kolp et al., 2005), a mediator agent mediates the interactions among agents. An initiator agent addresses the mediator agent instead of asking directly another colleague. The mediator has acquaintance models of colleagues and coordinates the cooperation between them. Each colleague has an acquaintance model of the mediator. While a broker only intermediates providers with consumers, a mediator *encapsulates* interactions and maintains models of initiators and other colleagues' behaviors over time.

In this section, we elaborate only on the mediator role, which would be the AUT. Thus, the mock agents would be the mediator client and the mediator provider agents. Fig. 14a, b illustrate the sequence diagrams of the two successful scenarios in which the mediation interaction completes successfully.

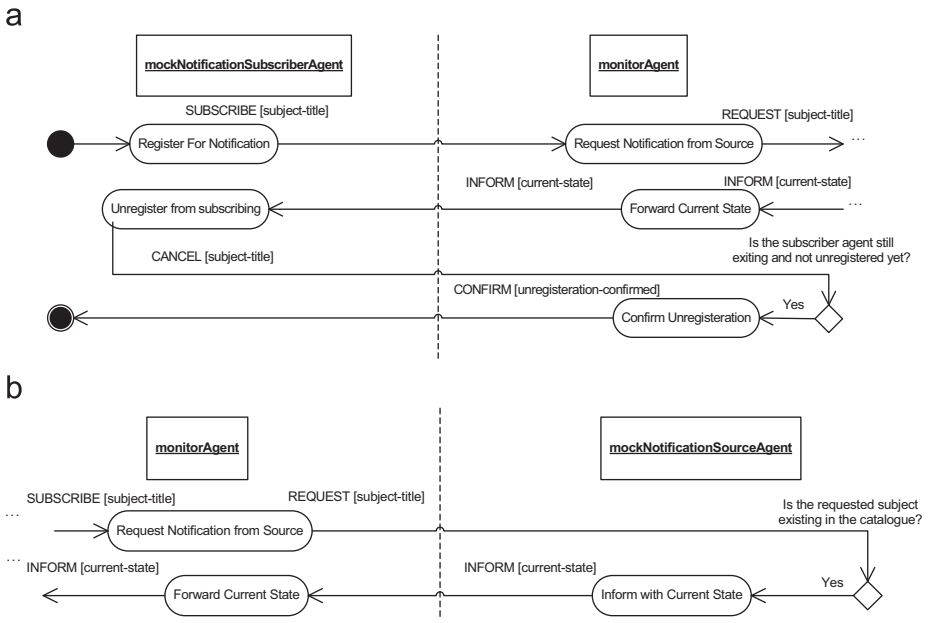

Fig. 12. (a) Sequence diagram for the first successful scenario: testing monitor agent. (b) Sequence diagram for the second successful scenario: testing monitor agent.

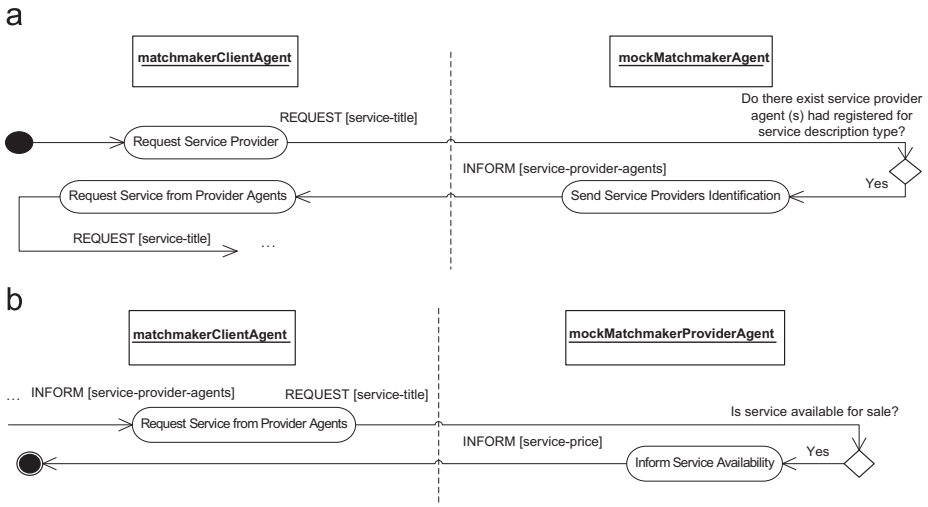

Fig. 13. (a) Sequence diagram for the first successful scenario: testing matchmaker client agent. (b) Sequence diagram for the second successful scenario: testing matchmaker client agent.

4.3.5. *Embassy pattern*

In the *embassy* pattern (Hayden et al., 1999), an embassy agent routes a service requested by a foreign agent to a local one and handle the response back. If the access to the local agent is granted, the foreign agent can submit messages to the embassy agent for translation. The content is translated in accordance to a standard ontology. Translated messages are forwarded to the target local agents. The results of the query are passed back out to the foreign agent, translated in reverse.

In this section, we elaborate only on the embassy role, which would be the AUT. Thus, the mock agents would be the foreign and the local agents. Fig. 15a, b illustrate the sequence diagrams of the two successful scenarios in which the embassy interaction completes successfully.

4.3.6. *Wrapper pattern*

The *wrapper* pattern (Kolp et al., 2005) incorporates a legacy system into a MAS. The wrapper agent interfaces the clients to the legacy system by acting as a translator between them. This ensures that communication protocols are respected and the legacy system remains decoupled from the rest of the MAS. In this section, we elaborate only on the wrapper role, which would be the AUT. Thus, the mock agents would be the wrapper client and the wrapper source agents. Fig. 16a, b illustrates the sequence diagrams of the two successful scenarios in which the wrapping interaction completes successfully.

4.4. *Using the eclipse plug-in*

Fig. 17 presents the entry point for generating stereotypes for testing agent interaction design patterns.

The mock agent designer makes the following steps in order to generate the target *mock agent* file and the associated resource file: (1) choose the required *design pattern category* that his/her AUT follows, (2) choose the required *design pattern* from the next list that will be loaded with all design patterns belonging to the chosen category, (3) choose the required *agent role* acting as the

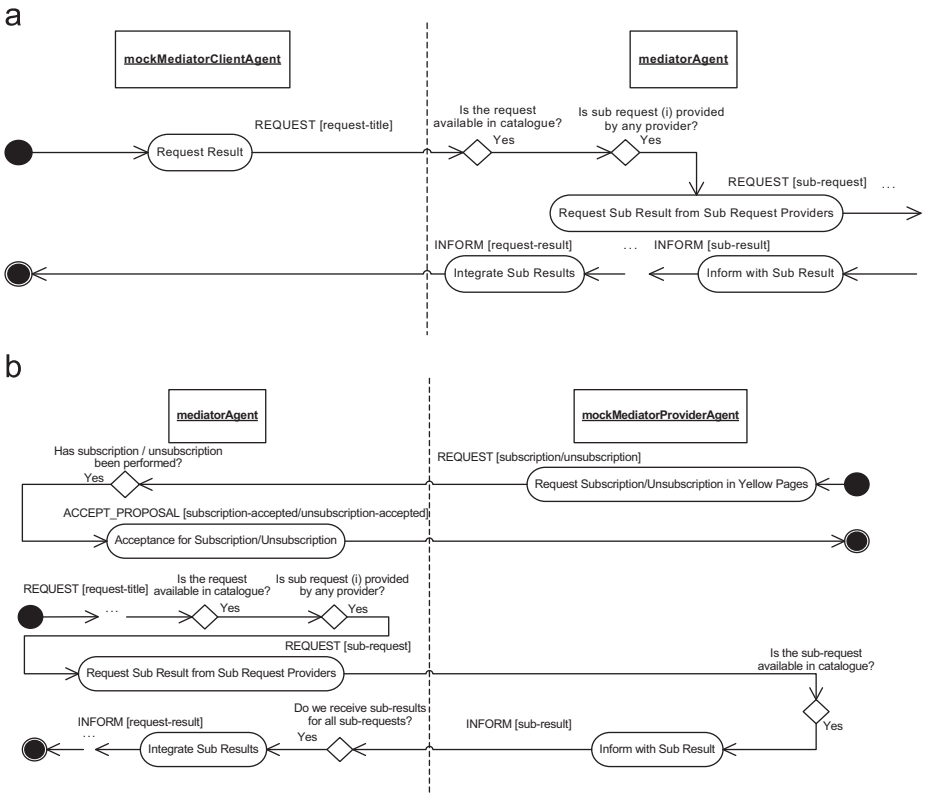

Fig. 14. (a) Sequence diagram for the first successful scenario: testing mediator agent. (b) Sequence diagram for the second successful scenario: testing mediator agent.

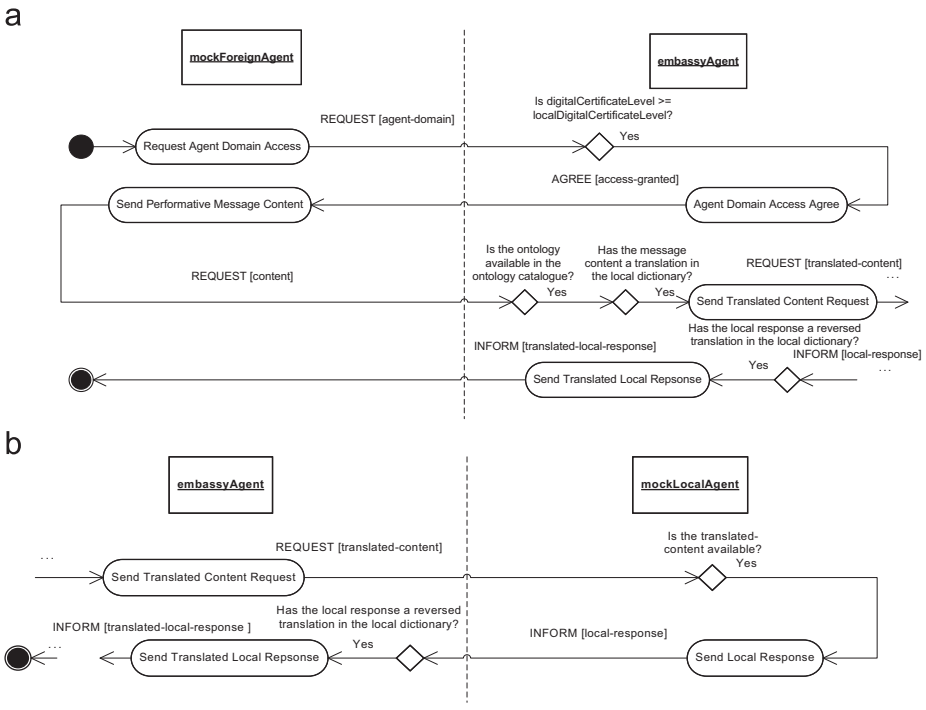

Fig. 15. (a) Sequence diagram for the first successful scenario: testing embassy agent. (b) Sequence diagram for the second successful scenario: testing embassy agent.

AUT from the third list that will be loaded with all the existing roles in the chosen *design pattern*, (4) choose the required *interacting role* acting as the *mock agent* from the fourth list that will be loaded with all the roles interacting with the chosen AUT role in the chosen *design pattern*, (5) choose the *test scenario type* whether being a *successful* or an *exceptional* scenario, (6) browse for the package where the generated *mock agent* files will reside into, (7) click "Edit Test Design Pattern" if he/she prefers to edit the *design pattern parameters* for either the *successful* scenario or the *exceptional* one, and finally (8) click the "Generate" button in order to get the required *mock agent* files generated in the chosen package.

Fig. 18 represents the generated resource file *MockBuyerAgent.properties* that is associated with the generated file *MockBuyerAgent.*

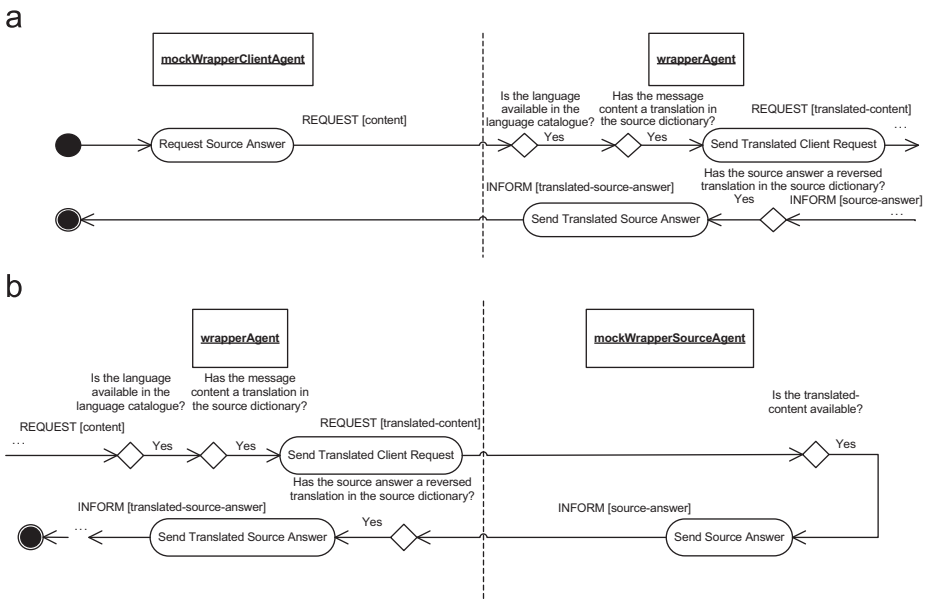

Fig. 16. (a) Sequence diagram for the first successful scenario: testing wrapper agent. (b) Sequence diagram for the first successful scenario: testing wrapper agent.

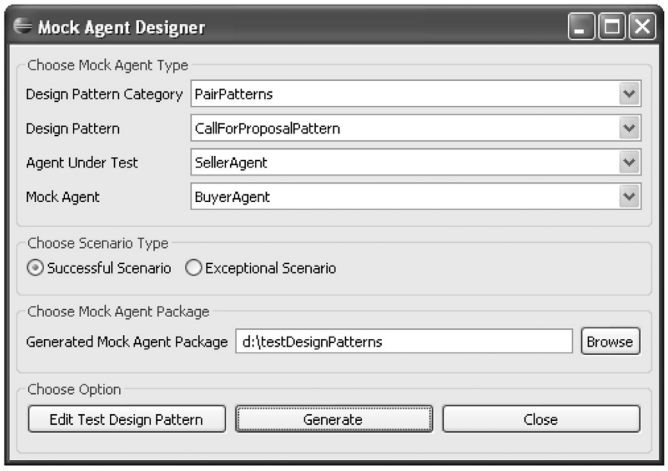

Fig. 17. The entry point of the eclipse Plug-in.

```
1 Ticker_Behaviour_Period = 60000
2 CFP_Performative = CFP
3 CFP_ConversationID = service-trade
4 PROPOSE_Performative = PROPOSE
5 ACCEPT_PROPOSAL_Performative = ACCEPT_PROPOSAL
6 ACCEPT_PROPOSAL_ConversationID = service-trade
7 ACCEPT_PROPOSAL_Content = service-title
8 INFORM_Performative = INFORM
```

Fig. 18. The generated resource file MockBuyerAgent.properties.

java that contains the values of the placeholders in the mock agent file.

Fig. 19 represents the test design pattern file *MockBuyerAgent.XML* from which the resource file *MockBuyerAgent.properties* reads the values of the placeholders in the successful scenario or a sample exceptional scenario according to the "Scenario Type" option (i.e., successful or exceptional).

The generated java source code already contains the necessary **imports** (e.g., Fig. 20) as well as the **method** definitions needed as part of the design highlighted in Section 3.2.

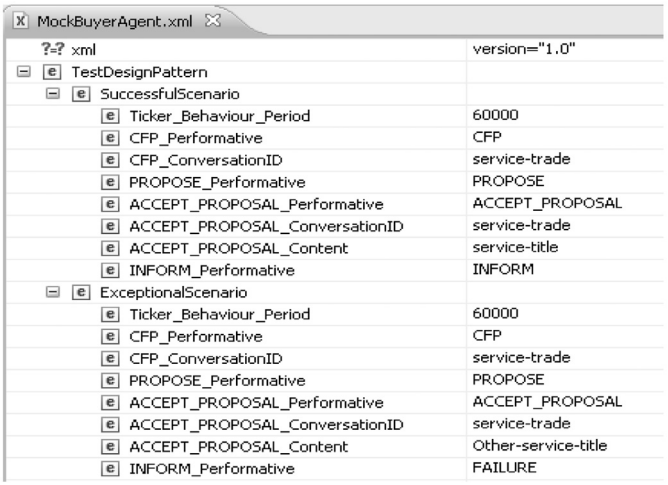

Fig. 19. The test design pattern file MockBuyerAgent.XML.

4.5. Adding further test stereotypes

In order to define further test stereotype besides the already existing ones, the first choice to the mock agent designer is to decide whether this stereotype belongs to the already existing pattern categories or a new pattern category has to be defined. If he/she decides to choose the second choice, the newly defined *design pattern category* will be appended to the corresponding list and a package with its name will be created in the following folder: ECLIPSE_HOME\plugins\MockAgentDesigner_1.0.0\testDesignPatterns.

Similar procedures can be done for defining further *design patterns*, *agent roles*, and *interacting roles*. Finally, the user can upload a new *mock agent template* or overwrite an already existing one in order to test a specific *agent role* in the newly defined *design pattern*.

5. Evaluation

In this section, we use the EMMA (Roubtsov, 2006) code coverage tool to prove that the execution of the generated test files within our

proposed framework practically provides good code coverage results. The *code coverage* refers to a software engineering technique through which the mock agent test case designer can track the quality and comprehensiveness of the test suite. This is done by determining simple metrics like the percentage of classes, methods, lines executed when the test suite ran. In Table 1, we illustrate the lines of code coverage for the 10 interaction design patterns which we implemented with our framework.

In summary, the above table indicates that all of the AUT lines of code have minimum code coverage of **89%** and an average of **96%.** EMMA also offers the possibility of highlighting lines of code that are not covered during the execution of the test suites. By investigating these lines, it appears that they occurred outside both of the normal and exceptional scenarios. These include *null arguments* passed to the AUT classes, some Java-based FIPA exceptions and *catch* exception blocks.

```
/**
 * Design Pattern Category:     PairPatterns
 * Design Pattern:              CallForProposalPattern
 * Agent Under Test:            SellerAgent
 * Mock Agent:                  MockBuyerAgent
 */
package PairPatterns.CallForProposalPattern.SellerAgent.MockBuyerAgent;

import java.util.MissingResourceException;
import java.util.ResourceBundle;

import jade.core.AID;
import jade.core.behaviours.Behaviour;
import jade.core.behaviours.TickerBehaviour;
import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;

import junit.framework.TestResult;

import MASUnitTesting.JADEMockAgent;
import MASUnitTesting.ReplyReceptionFailed;

public class MockBuyerAgent extends JADEMockAgent {
}
```

Fig. 20. Package and import structure in the generated MockBuyerAgent template

6. Conclusion and future work

In this paper, we introduce a framework for designing test cases based on a unit testing approach for MAS. Our approach relies on the use of *mock agents* for testing common *agent design patterns*. In our work, we provide implementation of *test stereotypes* for a vast majority of agent interaction design patterns. We also demonstrate how to use our toolset to implement further test stereotypes and how to add them to the available repository. The code coverage analysis reveals good values in terms of lines of code covered in the 10 *social design patterns* that we initially support. Having these blocks in a reusable repository enables MAS test case designers to concentrate on designing further tests with a good head start.

Following this systematic test case design process, our framework can enhance the performance of a large scale MAS that eventually consists of a number of agent design patterns. Through testing every agent in the MAS in isolation followed by using the reflection capabilities of the Eclipse SDK to reflect the changes made in any AUT directly into the generated mock agents, the MAS designer can have a complete test case design and implementation methodology.

As a future work, we would like to implement test stereotypes for other agent design patterns. In addition, we would like to investigate the problems that arise while working with concurrent threads such as racing and deadlocks. Traditional code coverage

Table 1
EMMA code coverage results statistics.

Pattern type	Pattern name	AUT—(Line %)	Mock Agent—(Line %)	Test Case—(Line %)
Pair	Booking pattern	Client **(98%)**	Service provider (91%)	Client (77%)
		Service provider **(98%)**	Client (95%)	Service provider (76%)
Pair	Subscription pattern	Yellow page **(100%)**	Subscription provider (95%)	Yellow page (83%)
		Subscription provider **(100%)**	Yellow page (88%)	Subscription provider (84%)
Pair	Call-for-proposals	Buyer **(95%)**	Seller (83%)	Buyer (76%)
		Seller **(89%)**	Buyer (91%)	Seller (74%)
Pair	Bidding pattern	Bidder **(95%)**	Auctioneer (96%)	Bidder (79%)
		Auctioneer **(98%)**	Bidder (90%)	Auctioneer (82%)
Mediation	Monitor pattern	Notification subscriber **(97%)**	Monitor (90%)	Notification subscriber (74%)
		Monitor **(100%)**	Notification subscriber (89%)	Monitor (78%)
			Notification source (88%)	Monitor (78%)
		Notification source **(95%)**	Monitor (96%)	Notification source (75%)
Mediation	Broker pattern	Client **(97%)**	Broker (93%)	Client (74%)
		Broker **(97%)**	Client (93%)	Broker (76%)
			Service provider (95%)	Broker (77%)
		Service provider **(98%)**	Broker (94%)	Service provider (75%)
Mediation	Matchmaker pattern	Client **(95%)**	Matchmaker (91%)	Client (78%)
			Matchmaker provider (87%)	Client (78%)
		Matchmaker **(95%)**	Client (87%)	Matchmaker (77%)
			Matchmaker provider (96%)	Matchmaker (77%)
		Matchmaker provider **(98%)**	Matchmaker (93%)	Matchmaker provider (79%)
			Client (89%)	Matchmaker provider (79%)
Mediation	Mediator pattern	Client **(91%)**	Mediator (87%)	Client (77%)
		Mediator **(94%)**	Client (86%)	Mediator (82%)
			Mediator provider (93%)	Mediator (79%)
		Mediator provider **(98%)**	Mediator (94%)	Mediator provider (77%)
Mediation	Embassy pattern	Foreign **(97%)**	Embassy (95%)	Foreign (82%)
		embassy **(97%)**	Foreign (93%)	Embassy (83%)
			Local (87%)	Embassy (84%)
		Local **(94%)**	Embassy (92%)	Local (80%)
Mediation	Wrapper pattern	Client **(96%)**	Wrapper (93%)	Client (79%)
		Wrapper **(98%)**	Client (89%)	Wrapper (82%)
			Source (87%)	Wrapper (82%)
		Source **(94%)**	Wrapper (91%)	Source (79%)

metrics, such as statement, block, and branch, fail to address test adequacy concerns introduced by concurrency.

Our proposed framework can be accompanied with a theoretical model. One possible alternative is the use of Petri nets to represent the succession of events that can appear in the relation between the mock agent and the AUT. Finally, it will be interesting to integrate some design patterns in one complete MAS large scale implementation to show the usefulness of testing a large-scale application using our proposed unit testing framework.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions that helped improve this paper.

References

Beck, K., 2002. Test-Driven Development: by Example. Addison-Wesley Professional, Boston, USA.

Beck, K., Andres, C., 2004. Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, Boston, USA.

Bellifemine, F., Bergenti, F., Caire, G., Poggi, A., 2005. JADE A Java Agent Development Framework. Multi-Agent Programming, pp. 125–147.

Caire, G., Cossentino, M., Negri, A., Poggi, A., Turci, P., 2004. Multi-agent systems implementation and testing. In: Proceedings of 4th International Symposium—From Agent Theory to Agent Implementation (AT2AI-4), Vienna, Austria, Citeseer, pp.14–16.

Castro, J., Kolp, M., Mylopoulos, J., 2002. Towards requirements-driven information systems engineering: the *Tropos* project. Information systems, Vol. 27. Elsevier Science Ltd365–389.

Coelho, R., Cirilo, E., Kulesza, U., von Staa, A., Rashid, A., Lucena, C., 2007. JAT: A test automation framework for multi-agent systems. In: Proceedings of the 23th International Conference on Software Maintenance (ICSM 2007), Maison Internationale, Paris, France. October 2–5, 2007, IEEE, pp. 425–434.

Coelho, R., Kulesza, U., von Staa, A., Lucena, C., 2006. Unit testing in multi-agent systems using mock-agents and aspects. In: Proceedings of the 2006 International Workshop on Software Engineering for Large-scale Multi-agent Systems (SELMAS 2006), New York, ACM Press, pp. 83–90.

Do, T., Kolp, M., Hoang, T., Pirotte, A., 2003a. A framework for design patterns for Tropos. In: Proceedings of the 17th Brazilian Symposium on Software Engineering (SBES 2003). Maunas, Brazil, October 2003, pp. 3–343.

Do, T., Kolp, M., Pirotte, A., 2003b. Social patterns for designing multi-agent systems. In: Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering (SEKE 2003). San Francisco, USA, July 2003, pp. 103–110.

FIPA, 2000. Contract Net Interaction Protocol Specification. Foundation for Intelligent Physical Agents. ⟨http://www.fipa.org/specs/fipa00029/⟩.

Ekinci, E.E., Tiryaki, A.M., Çetin, Ö., Dikenelli, O., 2009. Goal-oriented agent testing revisited. In: Luck, M., Gomez-Sanz, J.J. (Eds.), Proceedings of the 9th International Conference on Agent-Oriented Software Engineering IX (AOSE 2008), LNCS 5386, Springer-Verlang, Berlin Heidelberg, pp. 173–186.

Gómez-Sanz, J.J., Botía, J., Serrano, E., Pavón, J., 2009. Testing and debugging of MAS interactions with INGENIAS. In: Luck, M., Gomez-Sanz, J.J. (Eds.), Proceedings of the 9th International Conference on Agent-Oriented Software Engineering IX (AOSE 2008), LNCS 5386, Springer-Verlang, Berlin Heidelberg, pp. 199–212.

Hayden, S., Carrick, C., Yang, Q., et al., 1999. Architectural design patterns for multiagent coordination. In: Proceedings of the 3rd International Conference on Autonomous Agents, Agents 99, Seattle, USA.

Huget, M., Demazeau, Y., 2004. Evaluating multiagent systems: a record/replay approach. In: Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2004), IEEE, pp. 536–539.

Jennings, N., 2001. An agent-based approach for building complex software systems. Commun. ACM 44 (4), 35–41.

Kendall, E.A., Krishna, P.V., Pathak, C.V., Suresh, C.B., 1998. Patterns of intelligent and mobile agents. In: Proceedings of the 2nd International Conference on Autonomous Agents, St. Paul, USA, ACM, pp. 92–99.

Khamis, M.A., Gomaa, W., 2012. Enhanced multiagent multi-objective reinforcement learning for urban traffic light control. In: Proceedings of the 11th International Conference on Machine Learning and Applications (ICMLA 2012), Boca Raton, Florida, USA, IEEE, pp. 586–591.

Kolp, M., Do, T., Faulkner, S., Hoang, H., 2005. Introspecting Agent-Oriented Design Patterns. In: Advances in Software Engineering and Knowledge Engineering, vol. 3. Citeseer, Omaha, USA, pp. 105–134.

Kolp, M., Giorgini, P., Mylopoulos, J., 2002. Information systems development through social structures. In: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, Ishia, Italy, ACM, pp. 183–190.

Myers, G.J., 2004. The Art of Software Testing, 2nd edition John Wiley & Sons, Inc., Hoboken, New Jersey, USA.

Nagi, K., Khamis, M.A., 2011. Designing multi-agent unit tests using systematic test design patterns. In: Proceedings of the 5th International Conference on Intelligent Systems and Agents (ISA 2011), Rome, Italy, 24–26 July, 2011, International Association for Development of the Information Society (IADIS), pp. 11–18.

Nguyen, C.D., Perini, A., Bernon, C., Pavón, J., Thangarajah, J., 2011. Testing in multi-agent systems. In: Gleizes, M.-P., Gomez-Sanz, J.J. (Eds.), Proceedings of the 10th International Conference on Agent-Oriented Software Engineering X (AOSE'10), LNCS 6038, Springer-Verlag, Berlin, Heidelberg, pp. 180–190.

Nguyen, C.D., Perini, A., Tonella, P., 2008a. eCAT: a tool for automating test cases generation and execution in testing multi-agent systems (demo paper). In: Padgham, Parkes, Müller, Parsons (Eds.), Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AA-MAS 2008), May 12–16, 2008, Estoril, Portugal, International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), pp. 1669–1670.

Nguyen, C.D., Perini, A., Tonella, P., 2008b. Ontology-based test generation for multiagent systems (short paper). In: Padgham, Parkes, Müller, Parsons (Eds.), Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AA-MAS 2008), vol. 3, May 12–16 2008, Estoril, Portugal, International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), pp. 1315–1320.

Padgham, L., Thangarajah, J., Zhang, Z., Miller, T., 2013. Model-based test oracle generation for automated unit testing of agent systems. IEEE Transactions on Software Engineering, vol. 99. IEEE Computer Society1.

Roubtsov, V., 2006. EMMA Code Coverage Tool. Available at: ⟨http://emma.sourceforge.net/reference/reference.pdf⟩, last accessed on Nov. 28, 2012.

Silva, C., Castro, J., Tedesco, P., Silva, I., 2005. Describing agent-oriented design patterns in Tropos. In: Proceedings of the 19th Brazilian Symposium on Software Engineering, Uberlandia, Minas Gerais, Brazil, Citeseer, pp. 27–78.

Tahara, Y., Ohsuga, A., Honiden, S., 1999. Agent system development method based on agent patterns. In: Proceedings of the 21st International Conference on Software Engineering, Los Angeles, California, USA, IEEE, pp. 356–367.

Timm, I., Scholz, T., Herzog, O., Krempels, K., Spaniol, O., 2006. From agents to multiagent systems, Multiagent Engineering. Springer, New York, USA35–51.

Tiryaki, A.M., Öztuna, S., Dikenelli, O., Erdur, R.C., 2007. SUnit: a unit testing framework for test driven development of multi-agent systems. In: Padgham, L., Zambonelli, F. (Eds.), Proceedings of the 7th International Conference on Agent-Oriented Software Engineering VII (AOSE 2006), LNCS 4405, Springer-Verlang, Berlin Heidelberg, pp. 156–173.

Wang, S., Zhu, H., 2012. CATest: a test automation framework for multi-agent systems. In: Proceedings of the 36th Annual Computer Software and Applications Conference (COMPSAC 2012), Izmir, Turkey, 16–20 July, 2012, IEEE, pp. 148–157.

Zhang, Z., Thangarajah, J., Padgham, L., 2008. Automated unit testing intelligent agents in PDT (demo paper). In: Padgham, Parkes, Müller, Parsons (Eds.), Proceedings of 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), May 12–16, 2008, Estoril, Portugal, International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), pp. 1673–1674.