Inf Syst Front (2014) 16:169–182
DOI 10.1007/s10796-013-9438-5

Beast methodology

An agile testing methodology for multi-agent systems based on behaviour driven development

Álvaro Carrera · Carlos A. Iglesias · Mercedes Garijo

Published online: 18 July 2013

Abstract This paper presents a testing methodology to apply Behaviour Driven Development (BDD) techniques while developing Multi-Agent Systems (MASs), termed BEhavioural Agent Simple Testing (BEAST) Methodology. This methodology is supported by the open source framework (BEAST Tool) which automatically generates test cases skeletons from BDD scenarios specifications. The developed framework allows the testing of MASs based on JADE or JADEX platforms. In addition, this framework offers a set of configurable Mock Agents with the aim of being able to execute tests while the MAS is under development. The BEAST Methodology presents transparent traceability from user requirements to test cases. Thus, the stakeholders can be aware of the project status. The methodology and the associated tool have been validated in the development of a MAS for fault diagnosis in FTTH (Fiber To The Home) networks. The results have been measured in quantifiable way obtaining a reduction of the tests implementation time.

Keywords Test · Behaviour-driven development · Multi-agent systems · Mock-agents · Agile · Methodology

Á. Carrera (✉) · C. A. Iglesias · M. Garijo
Departamento de Ingeniería de Sistemas Telemáticos,
Universidad Politécnica de Madrid,
Av. Complutense 30, 28040, Madrid, Spain
e-mail: a.carrera@dit.upm.es

C. A. Iglesias
e-mail: cif@dit.upm.es

M. Garijo
e-mail: mga@dit.upm.es
URL: http://www.gsi.dit.upm.es

1 Introduction

Understanding stakeholders requirements and fulfilling their desired functionality is considered as the most important aspect for a software project to be considered successful (Agarwal and Rathod 2006). Thus, requirements engineering plays a key role in the development process. The main challenges of requirements engineering are (Marnewick et al. 2011): (i) improving the communication between the stakeholders and the implementation team and (ii) understanding the problem.

Nevertheless, the process of eliciting requirements and communicating them is still an issue and some authors consider it the *next bottleneck to be removed from the software development process* (Adzic 2009). The main reasons for this communication gap between stakeholders and the development team are that (Adzic 2009) (i) imperative requirements are very easy to misunderstand; (ii) even the obvious aspects are not so obvious and can be misinterpreted and (iii) requirements are overspecified, since they are expressed as a solution, and focus on what to do and not why, not allowing the development team whether discuss if those requirements are the best way to achieve stakeholders' expectations.

The context of this article was a research project contracted by the company Telefónica R&D. They requested us to develop a multi-agent system for fault diagnosis in their network. From a software engineering point of view, the main challenges were: (i) they required managing the project using the SCRUM Agile Methodology (Schwaber and Sutherland 2009), (ii) the project involved integration with a wide range of external systems and the emulation of faulty behaviour of network transmission and (iii) the development team was composed of students with different timetables, so they were not

working together most of the time. After the first release, the main problems we encountered were communication problems between the development team and the customer (expert network engineers), communication problems within the development team, where agents were being developed in parallel, and lack of automation in the unit testing process, which involved to test physical connections with a manual and very time consuming process.

After analysing several Agent Oriented Software Engineering (AOSE) proposals based on agile principles (Clynch and Collier 2007; García-Magariño et al. 2009), we have not found any proposal which covers *acceptance tests* and provides a good starting point for its application in an agile context. Thus, this research aims at bridging the gap between acceptance testing and AOSE. The key motivation of this paper is to explore to what extent acceptance testing can benefit MAS development, in order to provide support in the development of MAS in agile environments. This brought us to identify the need for an agile testing methodology for MAS.

The rest of the article is structured as follows. First, Section 2 discusses related work in the research field of agile acceptance testing. Section 3 describes a testing methodology for MAS based on BDD techniques. Section 4 provides an overview of the open source tool that supports the proposed methodology. Section 5 presents a worked example of the application of the methodology and the tool. Section 6 provides an evaluation of the benefits of the proposed approach. Finally, Section 7 presents some concluding remarks and discusses potential future work.

2 Related work

In order to bridge the communication gap between developers and stakeholders, the agile movement has proposed to shift the focus of requirements gathering. Instead of following a contractual approach where the requirements documents is the most important goal, they put emphasis on improving the communication among all the stakeholders and developers to have a common understanding of these requirements. Many approaches have been explored for requirements gathering, such as the use of ontologies for modelling the user requirements (Sun et al. 2010) or the definition of UML models for capturing quality requirements (Guerra-García et al. 2013). Moreover, given that requirements will have inconsistencies and gaps (Adzic 2011), it has been proposed to anticipate the detection of these problems by checking the requirements as soon as possible, even before the system is developed. In this line, Martin and Melnik formulated the equivalence hypothesis: "As formality increases, tests and requirements become indistinguishable. At the limit, tests and requirements are equivalent" (Martin and Melnik 2008).

As a result, they have proposed a practice so called *agile acceptance testing*, whose purpose is improving communication by using real-world examples for discussion and specifications of the expected behaviour at the same time, which is called Specification by Example (SBE). Different authors have proposed to express the examples in a tabular form (Acceptance Test Driven Development (ATDD)[1] with Fit test framework (Mugridge and Cunningham 2005)) or as scenarios (BDD (North 2007) with tools such as JBehave (North 2011) or Cucumber (Wynne and Hellesøy 2008)). In this way, requirements are expressed as acceptance tests, and these tests are automated. When an agile methodology is followed, acceptance tests can be checked in an automated way during each iteration, and thus, requirements can be progressively improved. Most frameworks provide a straight forward transition from acceptance tests to functional tests based on tools such as the xUnit family (Hamill 2004). Agile acceptance testing complements Test Driven Developoment (TDD) practices, and it can be seen as a natural extension of TDD practices, which have become mainstream in among software developers. In this way, software project management can be based not only on estimations but on the results of acceptance and functional tests. In addition, these practices facilitate to maintain requirements (i.e. acceptance tests) updated along the project lifespan.

In the multi-agent field, there have been several efforts in the testing of final systems. MAS testing present several challenges (Nguyen 2009), given that agents are distributed, autonomous and it is interesting not its individual behaviour but the emergent behaviour of the multi-agent system that arises from the interaction among the individual behaviours. A good literature review of MAS testing can be found in (Nguyen 2009; Nguyen et al. 2011; Houhamdi 2011). To the best of our knowledge, there is no previous work dealing explicitly with acceptance testing in AOSE. Thangarajah et al. (2011) propose to extend the scenarios of the Prometheus Methodology in order to be able to do testing of scenarios as part of requirements or acceptance testing. The work describes also a novel technique for integrating agent simulation in the testing process. Nevertheless, their proposal of acceptance tests seems targeted at technical users, given than the scenarios are described in terms such as percepts, goals and actions. Nguyen et al. (2010) propose an extension of the Tropos Methodology by defining a testing framework that takes into account the strong link between requirements and test cases. They distinguish external and internal testing, but they focused on the internal one. External testing produces acceptance tests

[1] A literate review of ATDD can be found in Haugset and Hanssen (2008).

for being validated by project stakeholders, while internal testing produces system and agent tests for being verified by developers.

3 BEAST methodology

To cover the problems identified in Section 1, we should identify which requirements should have the testing methodology. First, our primary concern is that the methodology should help in improving the communication between the stakeholders and the development team, as well as the communication among the development team. Another requirement comes from the overall methodology: it should be compatible and suitable for its application in combination with agile techniques. Finally, it should not be tied to a specific MAS tool, and it should be feasible to integrate with other MAS environments with low effort.

The BEAST Methodology is intended to be used in agile environments, with special focus on providing traceability of stakeholder requirements. With this end, requirements are automated as *acceptance tests*, which are linked with MAS testing. The main benefit of this approach is that it improves the understanding of the real advance of the project from the stakeholders perspective, and, moreover, it provides a good basis for reviewing the objectives of each iteration. As a result, requirements negotiation and specification can be done in an iterative way, and can be adapted to the improved understanding of the desired system by both stakeholders and development team.

BEAST Methodology consists of four phases: *System Behaviour Specification*, *MAS Behaviour Specification*, *Agent level testing* and *MAS level testing*, which are applied in each agile iteration.

Figure 1 depicts these steps and the actors that appear in each one of them. During the first step, the expected behaviour of the system is specified by the customer, the product owner and, at least, one member of the development team, as it is described in the SCRUM (Schwaber and Sutherland 2009) Methodology. This specification is done following the BDD technique (see Section 3.1). Once the BDD system specification is available, the product owner and the designer of the development team must translate the system specification into agent behaviours specification (see Section 3.2). The output of this step is a set of BDD requirements for the MAS that are implemented and tested in the following step of the methodology (see Section 3.3). During the agent level testing phase, the methodology proposes the use of mocking techniques to replace other agents which are not developed yet or external systems that are not available during the development phase. Once the behaviour of all agents have been tested, the *MAS level testing* phase has two purposes. First of all, once agents have been developed, integration testing can be done replacing mocks by the real systems. Second, *emergent features* should be validated in the developed scenario. Simulation techniques can complement this phase to simulate different system configurations.

This article is focused on the first three steps of the general methodology. *MAS level testing* will be addressed in an article in progress. Nonetheless, we would like to point out that once mocks objects have been replaced by the real entities they emulate, business requirements can be tested on the real system. Thus, acceptance testing is straight forward,

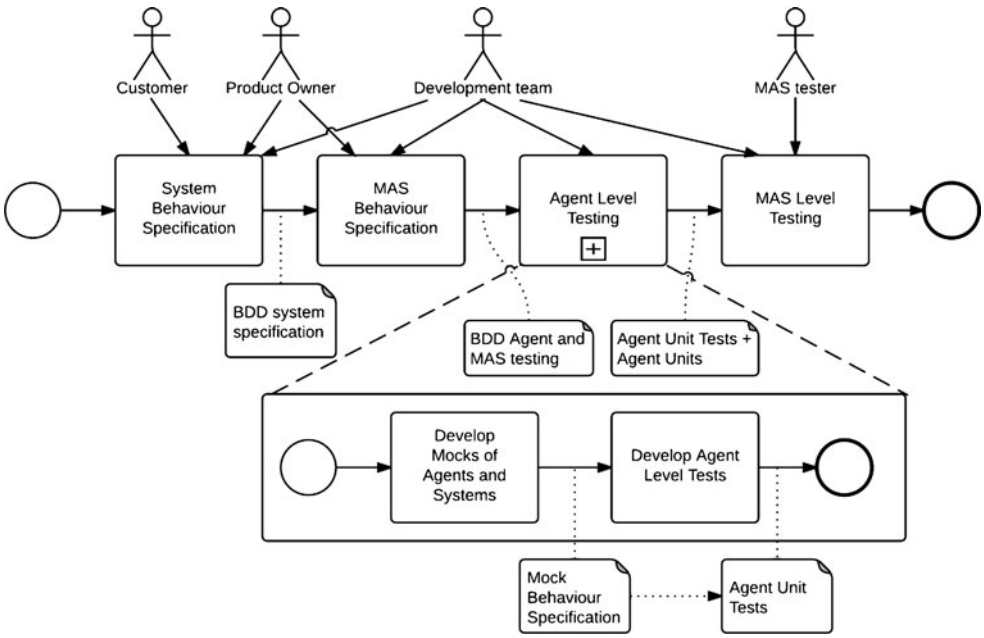

Fig. 1 Beast testing methodology

and the expectations of the stakeholders can be checked without discussing about ambiguities or omissions in the requirements document. The main benefit of BDD techniques is the continuous validation of user requirements in each iteration. This helps to refine iteratively requirements based on the current project advance and available resources.

Finally, Section 3.4 shows the mapping rules that ensure the traceability of the test cases with the project requirements. These mapping rules connect the outputs of every step of the methodology with the input of the next one.

3.1 System behaviour specification

The *System Behaviour Specification* phase aims at providing a communication bridge between the project stakeholders and the development team during requirements gathering. This phase follows the BDD technique (North 2007). System behaviours are derived from the business outcomes that the system intends to produce. These business outcomes should be prioritized by the stakeholders. Then, business outcomes are drilled down to feature sets. The feature sets decompose a business outcome into a set of abstract features, which show what should be done to achieve a business outcome. These feature sets are the result of discussions between stakeholders and developers. Features are described using *User Stories*. Then, *User Stories* are described in scenarios for each particular instantiation of a *User Story*. In other words, the scenarios exemplify a *User Story* to cover all possible variations of the presented feature. Thus, these scenarios are the basis of acceptance tests.

Instead of using plain natural language, BDD proposes the usage of textual templates. Figure 2 presents the template for a *User Story*. This template presents a feature, i.e. a requirement, of the system and the benefit that this feature has from the point of view of a specific role, such as a final client or a system administrator. Figure 3 presents the template for a scenario. A set of scenarios must exemplify a *User Story* giving specific situations to well understand the feature and to test if the system meets the requirement. These templates should be instantiated by the pertinent concepts. These concepts are part of the *ubiquitous language* (Evans 2004) which establishes the common terminology used by stakeholders and developers. Thus, these terms will be used in the implementation, helping in reducing the gap between technical and business terminology.

3.2 MAS behaviour specification

This phase has the goal of architecturing the multi-agent system specifying all agent roles and the interaction among them. Based on the features identified in the previous phase,

```
[Story title] - description
As a [Role]
I want to [Feature]
So that [Benefit]
```

Fig. 2 User story template (North 2007)

the new features are realised with the MAS system. In order to maintain traceability and improve communication within the development team, we have found useful to use the same approach than in the previous phase for specifying the MAS behaviour. Thus, *business benefits* are described by *features* which are assigned to *agent roles*. The templates presented in Figs. 2 and 3 are used by the MAS designer in this phase to create *Agent Stories*. These *Agent Stories* describes the expected behaviour of an agent given a context and an event to achieve a concrete goal. The described scenarios are translated into test cases in the following phase of the methodology. These scenarios must represent all behaviours of an agent, both reactive and proactive. As the proposed methodology is focused on tests acceptability, no restriction is added for designing the MAS using any design methodology. Thus, methodologies, such as Ingenias (Pavon et al. 2005), Prometheus (Padgham and Winikoff 2003), MAS-CommonKADS (Iglesias et al. 1998) or Gaia (Wooldridge et al. 2000), can be used to design and/or develop the MAS. In other words, the methodology presented in this paper is an agile testing methodology to ensure the communication among stakeholders and developers and no design or implementation restrictions are proposed. So, the MAS designer has the responsibility to translate *User Stories* to *Agent Stories* describing all agent roles in the system and all their behaviours using any MAS design methodology.

As previously, *features* can be obtained in different contexts which are described as scenarios, which can involve one or more agent roles in the case of *cooperative scenarios*. In the case of *emergent features* coming from emergent behaviour, they will be only verified when the full system has been developed. This kind of emergent behaviour will be specified at MAS level in the agent stories, instead of for a particular agent role.

```
Scenario [Scenario name]
Given [Context]
And [Some more contexts] ...
When [Event]
Then [Outcome]
And [Some more outcomes] ...
```

Fig. 3 Scenario template (North 2007)

This phase could be skipped and system behaviours could be directly translated into agent unit tests (Section 3.3). In fact, our first version of the framework did not include this step. Nevertheless, we have found it very useful in order to make explicit how stakeholders specifications are translated into MAS requirements, and to provide better insight for developers about them.

3.3 Agent level testing

Based on the requirements obtained during the previous phase, agents are designed and developed. For this purpose, any of the available AOSE methodologies can be used for modelling and implementing agents. Since we are focused on testing aspects, this phase has two main steps (see Fig. 1): (i) developing mocks of the external systems that an agent interacts with and (ii) developing the unit tests of every agent. Note that an agent that is being tested is denoted as Agent Under Test (AUT).

The first step requires to simulate the intended behaviour of the external systems according to the scenarios. These scenarios are described in the previous phase using mocks. The second phase implements the tests configuring the developed mocks.

There have been several research works developing the concept of using mock testing for agent unit testing. Coelho et al. (2006) proposed a framework for unit testing of MAS based on the use of Mock Agents on top of the multiagent framework JADE (Bellifemine et al. 2007). They proposed to develop one Mock Agent per interacting agent role. Mock Agents were programmed using script-based plans which collect the messages that should be interchanged in the testing scenarios. Tiryaki et al. (2007) proposed the framework *SUnit* on top of the multiagent framework *Seagent* (Dikenelli et al. 2005). They extended *JUnit* testing in order to cope with agent plan structures testing. Zhang et al. (2011) generated automatically Mock Agents from design diagrams developed within the Prometheus Methodology (Padgham and Winikoff 2003).

We propose to use a mock testing technique for simulating external systems, being agents or any other system. As we are interested in simulating the behaviour of agents, we have defined several types of Mock Agents which provide a simple FIPA interface. Three basic mock patterns have been defined: mock that simulates answering messages (*ResponderMockAgent*), mock that simulates receiving messages without providing an answer (*ListenerMockAgent*) and mock that receives a message from one agent and sends a new message to a different agent (*MediatorMockAgent*). Figure 4 depicts the interaction among the proposed Mock Agents, the Test Case and the AUT.

The *ResponderMockAgent* has been designed to reply incoming messages with predetermined ones. This can be used to simulate external services or agents, that have to connect to those services. So, an AUT can interact with this type of Mock Agent to get external information in the same way as the final MAS. The *MediatorMockAgent* has been designed to act as a filter of messages. In other words, this Mock Agent receives messages from an AUT and sends a different message to other AUT. So, this type of Mock Agent can be used as processes that have to perform some actions with the information enveloped in the first message and have to inform to another agent. Finally, the *ListenerMockAgent* has been designed as a mailbox. The Mock Agent can be used to check if the content of a message that is sent by an AUT.

The proposed BEAST Methodology defines these three types of agents as they cover the most general communications among agents in a MAS. However, other Mock Agents can be designed if it is a need for the project requirements. For example, the proposed *ResponderMockAgent* could query a database to reply the message with real data that would be used by the AUT.

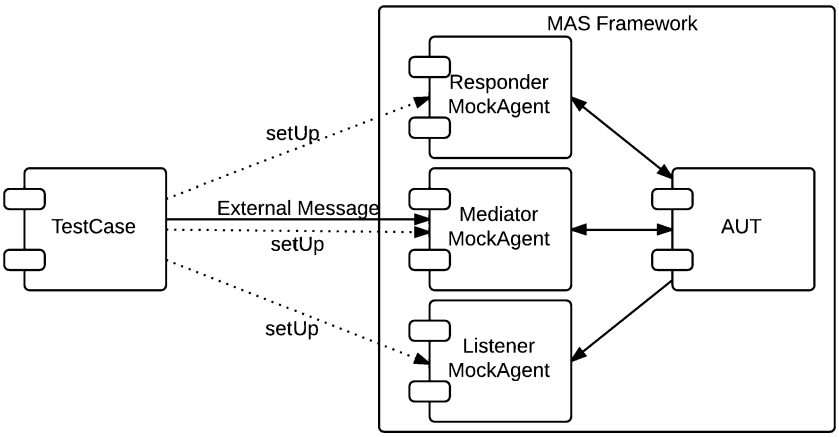

Fig. 4 BEAST mock agents

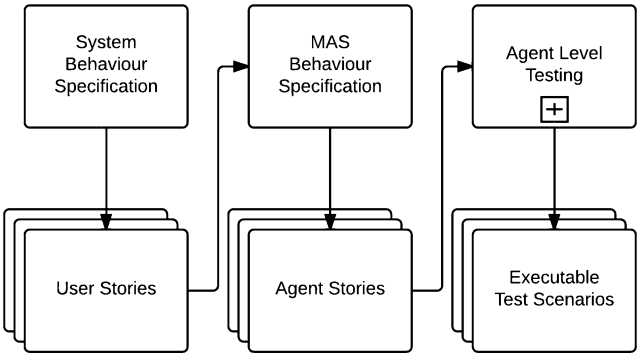

Fig. 5 Outcomes of BEAST phases

3.4 Providing traceability

The traceability from user requirements to executable tests is one of the keys of success in any software project (Almeida et al. 2007). To ensure this traceability, BEAST Methodology proposes a set of mapping rules that connect the outcomes of all phases of the methodology. Figure 5 shows that the *User Stories* obtained in the *System Behaviour Specification* phase are used as input in the *MAS Behaviour Specification* phase. In this phase, an *User Story* is translated to one or more *Agent Stories*. Both *User Stories* and *Agent Stories* follow the same template format (see Figs. 2 and 3). Finally, the scenarios of the *Agent Stories* are implemented to test the developed MAS. Thus, a *User Story* is break down in *Agent Stories*. An *Agent Story* is composed by a set of scenarios that are implemented as test cases. So, the stakeholder knows automatically which requirement is not fulfilled when a test fails.

Following the *JUnit* framework, both *User* and *Agent Stories* can be tested at once using *TestSuites* to execute all test cases related with. A *TestSuite* is a collection of test cases to show if a software has a specified behaviour. So, the set of scenarios which compose a story are joint in *TestSuite* to check if a story feature is satisfied. Figure 6 shows an example of traceability in BEAST Methodology.

4 BEAST tool

To support the methodology detailed in Section 3, an open source tool has been developed and hosted in a *Github* repository, named BEAST Tool.[2] This tool aims at providing assistance in the application of the BEAST Methodology. The tool translates story and scenario templates (see Fig. 2 and 3 respectively) into Java templates integrated with an extended version of JBehave framework (North 2011). The tool build process has been automated with Apache Maven (Foundation TAS 2011). One of the design principles of BEAST Tool has been that it must be valid for different MAS platforms. By now, the current version of the tool supports JADEX (Braubach et al. 2005) and JADE (Bellifemine et al. 2007) frameworks, but it can be extended to other frameworks with low effort as it is explained below in Section 4.3.

The BEAST Tool is structured as follows. A reader (or parser) package is used to manage the translation of user and agent stories to Java code as it is shown in Section 4.1. As *JBehave* (North 2011) is a framework to apply BDD in Java software development, it has been extended to apply BDD on MAS development and to design the BEAST Test Case model shown in Section 4.2. Finally, the adaptation of the tool to the MAS platforms is defined in Sections 4.3 and 4.4.

4.1 Story parser

There are two different types of stories in the proposed methodology (Section 3): *User Story* and *Agent Story*.

Following the BEAST Methodology, the first step is the *System Behaviour Specification* when the stakeholders and the development team (or at least one or two people of the development team) defines a set of *User Stories* in BDD format (see Figs. 2 and 3). Then, these *User Stories* are processed with the parser included in BEAST Tool and a *TestSuite* is created for every *User Story*.

Once the *User Stories* are defined, the designer defines a set of *Agent Stories* that must fulfill the requirements specified in the *User Stories*. These *Agent Stories* contain the specification of all behaviours of any system agent. Then, the parser is used again to generate a new *TestSuite* per *Agent Story* and a set of BEAST Test Case templates (one per scenario). The parser is configured to generate BEAST Test Case templates or not depending on what type of story is being parsed (see Fig. 7).

To define which *Agent Story* is related with an *User Story*, a manual matching process must be carried out by the designer. During this process, the designer must edit the *User Stories TestSuites* with the corresponding *Agent Stories TestSuites* or their specific scenarios. After this matching process, an *User Story* is completely traceable to the BEAST Test Cases that implement a concrete test scenario.

4.2 BEAST test case model

Our approach to agent level testing has consisted of extending JUnit framework in order to be able to test MAS systems. Mapping rules have been defined in order to provide full traceability of acceptance tests defined previously in BDD format. Thus, *JBehave* has been extended with this purpose. Mapping rules (Solis and Wang 2011) provide a

[2] http://github.com/gsi-upm/BeastTool/

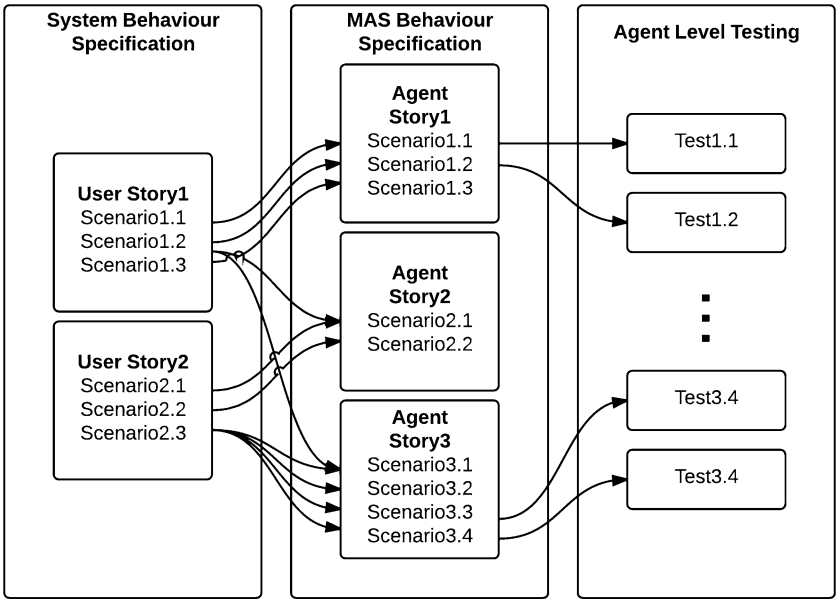

Fig. 6 Example of traceability in BEAST methodology

standard mapping from scenarios to test code. In *JBehave* framework, a *user story* is a file containing a set of *scenarios*. The name of the file is mapped onto a user story class. Each scenario step is mapped onto a test method using a Java annotation.

BEAST Tool translates a scenario (see Fig. 3) to a test case class, termed BEAST Test Case, which extends *JUnitStory* class of *JBehave* framework and contains three key methods that directly related with the three parts of a scenario ("Given-When-Then").

The three methods that a tester must implement are depicted in Fig. 8. The *setUp* method represents the "Given" scenario condition. This method typically initialises agents and configures their state (goals, beliefs, ...) as well as initialises the environment. The *launch* method represents the "When" scenario condition. This method generates and schedules the trigger event to start the test. The *verify* method represents the "Then" scenario condition. The expected states, such as goals or beliefs, are checked in this method once test execution is over.

BEAST Test Case has several methods that allows the interaction with a generic interface to interact with the MAS platform. This interface offers some facilities to prepare a concrete state of the agent. For example, external messages can be sent to the MAS platform, agents can be started and stopped, or internal information of an Agent Under Test (AUT) can be configured, such as beliefs or goals.

4.3 MAS platform interface

To provide MAS platform independence, three different interfaces have been defined to interact with the MAS platform from a BEAST Test Case. Each of them is responsible of different aspects on the platform management. The first one, *Connector* interface, provides an abstract interface to agent managing functions, such as launch platform or start an agent. The second one, *Messenger* interface, declares methods for sending and receiving messages to or from the platform respectively. Finally, the third one, *Agent Introspector* interface, provides access to the agent model, such as goals and beliefs.

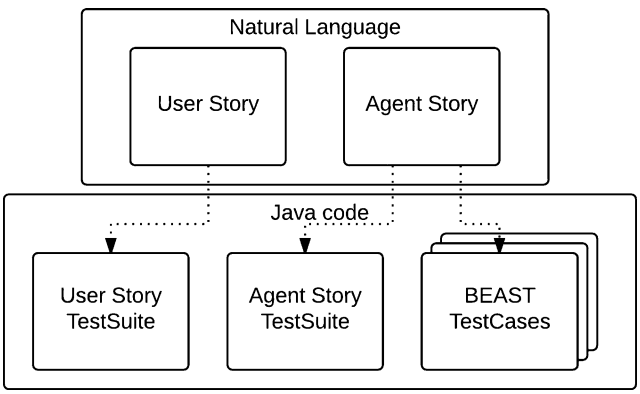

Fig. 7 Java classes generated in the parsing process

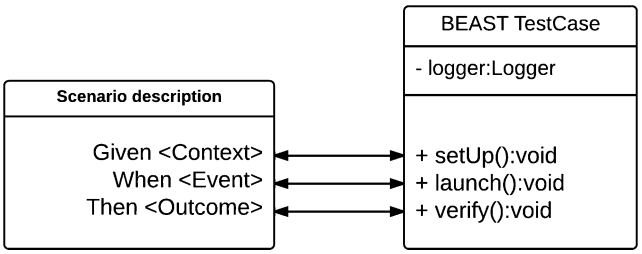

Fig. 8 BEAST test Case class

To integrate BEAST Tool with any MAS platform, these three interfaces must be implemented to get access to their agents. In the current version of BEAST Tool, JADE 4.0 (Bellifemine et al. 2007) and JADEX 2.0 (Braubach et al. 2005) are completely integrated.

To make easier the generation of BEAST Test Case classes (see Section 4.1), a testing interface selector has been defined, so called *PlatformSelector*. This selector provides the proper platform access from a BEAST Test Case to the MAS platform as shown in Fig. 9.

4.4 Mock definition

BEAST Methodology proposes three basic mock patterns for messaging (see Section 3.3). As these agents have to execute in the MAS platform, they have been implemented for JADE 4.0 and JADEX 2.0 with mock behaviours. So, a BEAST Mock Agent is an agent that contains mock plans or mock behaviours and can be configured and started from a BEAST Test Case.

After analysing available mocking frameworks, we have selected *Mockito* (Mockito Project 2012) framework, because of its easiness to be learnt, its popularity and its wide coverage of mocking functionalities. Thus, we have extended *Mockito* in order to be able to use it in MAS environments. In addition, the mocking framework allows an easy configuration of the mock objects (or agents), with patterns such as *when(< some input >).thenReturn(< some answer >)*. Mock Agents allow the specification of the simulated behaviour using *Mockito* constructions. Here follows an example.

```
when(mockAgent.processMessage(
  eq("REQUEST"),
  eq("ConnectionLossRate")))
  .thenReturn("INFORM", "LossRate = 0.2");
```

Using this type of constructions, a tester can simulate the behaviour of Mock Agents as complex as required. The tester can consider a Mock Agent as a *black box*, i.e. the inputs and the outputs are known but the internal process is unknown. Thus, the use of Mock Agents is not restricted to the messaging. BEAST Methodology presents only three Mock Agents for messaging (see Section 3.3) because those agents are completely generic models and can be used in any MAS development. But, the tester can implement other specific Mock Agents for a concrete project to make easier and faster its work as this agents can be configured with a few lines of code.

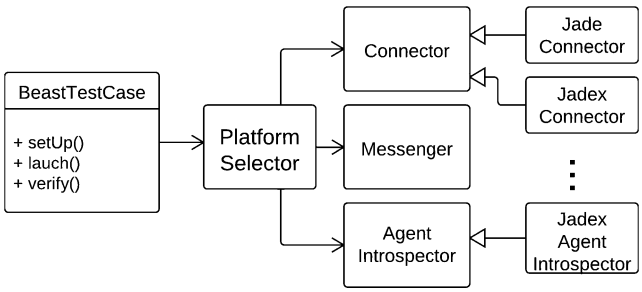

Fig. 9 Beast test case

5 Case study: MAS for fault diagnosis on FTTH scenario

To properly frame the proposed BEAST Technique, a network management project has been chosen as case study. In this example, the stakeholder is a network operator company which wants a tool to reduce the management cost of FTTH networks. The first task of the project was to write a high level project proposal and to explore different possible approaches to solve the problem. The result of this phase was that the solution that best fits the problem is a MAS architecture. SCRUM Agile Methodology (Schwaber and Sutherland 2009) and BEAST Methodology, supported by the BEAST Tool, was used to manage the progress of the project.

For exemplification purposes, it is exposed how tests can be implemented in JADEX (Braubach et al. 2005) framework.

Thus, one of the next tasks that the development team had to do was to arrange a meeting with the stakeholder to specify a set of requirements. These requirements written in BDD format (see Section 3) as *User Stories*. Table 1 shows an example of one gathered requirement.

Note that the stakeholder does not know anything about the solution, in this case, a Multi-Agent System (MAS). So, the written requirements, or *User Stories*, do not refer at all to the final agents. The translation from these

Table 1 Example of user story

Story: Time-to-repair cut down
As a operator network,
I want to have a system to diagnose root cause of faults
So that time-to-repair is below the SLA with the customer.
Scenario: System diagnoses a QoS decreasing failure
Given a user that has a Video On Demand (VoD) service connected through an FTTH access network and the user requests a film from the streaming server,
When loss rate is higher to 1%, latency is higher to 150ms or jitter is higher to 30ms,
Then the system must diagnose the root cause of fault is 'Damaged Fibre', 'Inadequate Bandwidth' or 'Damaged Splitter'.

requirements to agents is done by the designer who writes the *Agent Stories* based on the *User Stories*, as shown in Section 3.4.

The designed solution had to work in a Fiber To The Home (FTTH) scenario that is composed by a set of specific devices. In an FTTH network, the optical fiber reaches the boundary of the living space, such as a box on the outside wall of a home. In these networks, there are some passive elements, such as splitters or fibers, and active elements, such as Optical Network Terminal (ONT), Optical Line Termination (OLT) or ethernet routers. Figure 10 depicts a standard structure of an FTTH network. This network architecture usually delivers triple-play services directly from the central office of the operators. Furthermore, the final system should deal with devices from different vendors and different access protocols, which was an issue in the project.

To improve the resource assignment, several developers were assigned only to implement the access to the FTTH devices, such as ONT or OLT, to collect information. Then, the rest of the developers had to implement the agents to meet the *Agent Stories*. As some key elements of the final system were developing in parallel, the agents had to be tested using Mock Agents. The final MAS was too complex to be shown in this section, so a simplified scenario is presented to exemplify the use of the methodology. We are going to focus only on the *Agent Stories* exposed in Table 2. These *Agent Stories* define the behaviour of a *Diagnosis Agent* that must be able to diagnose the root cause of fault and it is directly related with the *User Story* shown in Table 1.

Every story has one scenario to exemplify the requirement defined in its story. The first story defines the goal that the agent has to perform a diagnostic process as soon as possible for the devices under its supervision. The first scenario exemplifies that requirement with the diagnosis of a concrete type of device. This scenario is an example of how to test a reactive behaviour, since the agent has a trigger to start a new goal and it must achieve it as soon as possible. The second story defines the goal of fulfil the conditions contracted with the final user in the Service Level Agreement (SLA). The second scenario specifies the goal of satisfying a concrete time restriction contracted with the customer. This scenario is an example of how to test a proactive behaviour, since the agent tries continuously achieve a goal if the conditions are met.

Table 2 Examples of agent stories

Story: Diagnosis process triggered by a symptom
As a Diagnosis Agent,
I want to process a FIPA-INFORM message with a detected symptom,
So that the system under my supervision is diagnosed as soon as possible.
Scenario: Diagnosis Agent diagnoses Damaged Splitter
Given a VoD streaming session,
When a 'high loss rate' symptom is received from a Probe Agent
And two or more geographically close users have loss rate higher to 1%,
Then the Diagnosis Agent must infer that the root cause of the problem is 'Damaged Splitter'.
Story: SLA fulfilment
As a Diagnosis Agent,
I want to report issue status before a given deadline,
So that I achieve my goal of fulfilling SLA restrictions.
Scenario: Diagnosis Agent meets the SLA
Given an SLA is contracted with a customer
And the Diagnosis Agent is aware of SLA commitments,
When that customer is current on payments
And any diagnosis is in progress,
Then the Diagnosis Agent must give a response in time that fulfils the SLA time restrictions.

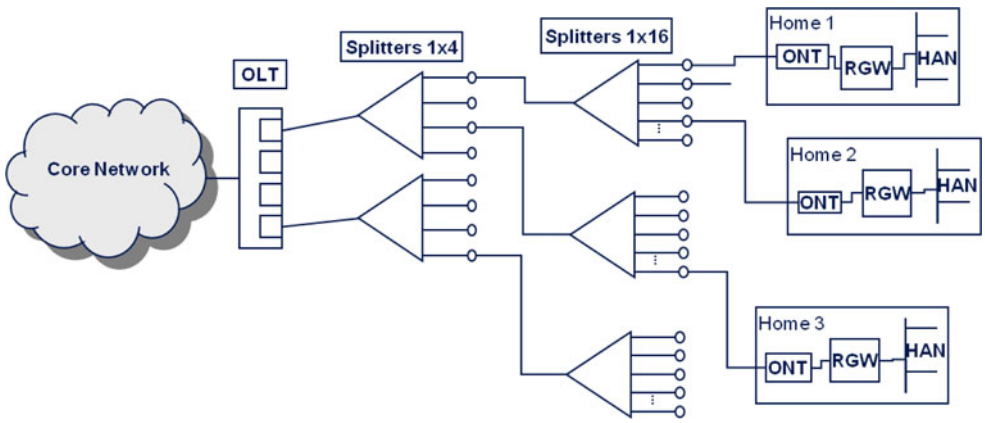

Fig. 10 Architecture of an FTTH network

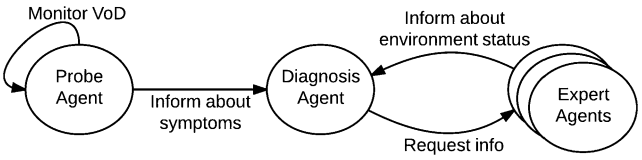

Fig. 11 Overview of agents involved in the exemplified scenario

In the following paragraphs, the first scenario is worked out to understand the proposed testing methodology. In that scenario, there are three type of agents: a *Probe Agent* responsible for monitoring Video On Demand (VoD) sessions, a *Expert Agent* to collect metrics from other lines and the *Diagnosis Agent* to request and process the available information to sum up an hypothesis. The interaction among these three types of agents is shown in Fig. 11.

As all agents were developed in parallel, the *Probe* agent and the *Expert* agents had not developed yet. Thus, the mocking facility of BEAST Tool was used. As previously introduced, BEAST Tool includes several Mock patterns (see Section 3.3). In this case, the *Mediator Mock Agent* is suitable for simulate *Probe Agent* to send symptoms to the *Diagnosis Agent* and the *Responder Mock Agent* is suitable for simulate *Expert* agents. Thus, this mock is configured for sending symptoms and network information respectively to simulate both agent roles. Finally, the *Diagnosis Agent* is the Agent Under Test (AUT) for this scenario. The scenario starts when the *Probe Mock Agent* sends a message to the *Diagnosis Agent* (AUT). Then, the *Diagnosis Agent* requests information about the status of other subscriber lines. Finally, the tester checks if the AUT has got the right hypothesis of root cause of fault.

The implementation of the scenario as test case is developed in a BEAST Test Case class. Figure 12 represents the interaction of the BEAST Test Case with the Mock Agent and the AUT in sequence diagram format. As shown in Section 4.2, there are three methods that a tester must implement to complete a test case. Table 3 shows the required code to implement the final test for agents that run on JADEX 2.0. Note that some comments and other Java code lines, such as logging lines or constants definition, have been omitted in the table to clarify the code.

Several methods, such as *startAgent*, *sendMessageToAgent* or *checkAgentsBeliefIsEqualTo*, are provided by the parent class (i.e. *BeastTTestCase* class) that interacts with the MAS Platform interface to access to the agents, as described in Section 4.3.

As SCRUM Agile Methodology (Schwaber and Sutherland 2009) had been chosen to manage the project progress, the result of these tests are shown to the stakeholder (*Product Owner* in SCRUM terminology) periodically during check progress meeting (*Sprint Reviews* in SCRUM terminology). This helps to the *Product Owner* to know the status of the project and to modify the *User Stories* to represent better the idea of the stakeholder that is not always well translated in the initial *User Stories*. Furthermore, the traceability from an *User Story* to a test case (see Section 3.4) makes easy to know which features or what test cases must be modified to fits the updated requirements.

6 Evaluation

The results of the proposed BEAST Tool have been evaluated in a quantifiable way using source code metrics. In particular, we have measured the number of test code lines required to implement tests. One of the most important benefits of developed BEAST Tool is that automatically creates a wrapper of the MAS platform and allows developers to interact with a friendly interface simplifying the implementation of tests. These metrics are strongly associated with

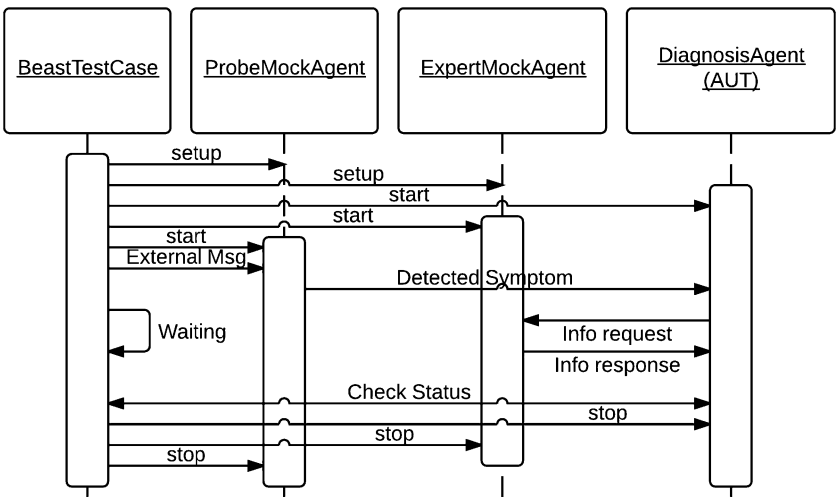

Fig. 12 Steps of the exemplified test case

Table 3 Implementation of the exemplified scenario in a BEAST Test Case

```
public class DiagnosisAgentDiagnosesDamagedSplitter extends BeastTestCase {

 public void setup(){

 // Configure Probe Mock Agent
 AgentBehaviour mockProbe = mock(AgentBehaviour.class);
 when(mockBeh.processMessage(eq(INFORM),
    eq("Generate High Loss Rate Symptom")))
  .thenReturn("DiagnosisAgent", INFORM, "Loss rate=0.15");
 MockConfiguration mockConfProbe = new MockConfiguration();
 mockConf.addBehaviour(mockConfProbe);

 // Configure Expert Mock Agent
 AgentBehaviour mockExpert = mock(AgentBehaviour.class);
 when(mockBeh.processMessage(eq(REQUEST),
    eq("Loss Rate - User Line ID: 14")))
  .thenReturn(INFORM, "Loss rate=0.09");
 MockConfiguration mockConfExpert = new MockConfiguration();
 mockConf.addBehaviour(mockConfExpert);

 // Start Diagnosis Agent
 startAgent("DiagnosisAgent","DiagnosisAgent.agent.xml");

 // Start mocks agents
 MockManager.startMockJadexAgent("ProbeMockAgent","MediatorMock.agent.xml",
  mockConfProbe,this);
 MockManager.startMockJadexAgent("ExpertMockAgent","ResponderMock.agent.xml",
  mockConfExpert,this);
 }

 public void launch() {
  sendMessageToAgent("ProbeMockAgent",INFORM,"Generate High Loss Rate Symptom");
  setExecutionTime(2000);// Waiting time in milliseconds
 }

 public void verify() {
  checkAgentsBeliefIsEqualTo("DiagnosisAgent",ROOT_CAUSE,DAMAGED_SPLITTER);
 }
}
```

the test implementation time that a developer consumes during this phase of development. The savings in number of code lines and in percentage are shown because they are quantifiable objective data, in comparison the time to develop a test that depends on the programming skills of the developer.

BEAST Tool is already adapted to test JADE (Bellifemine et al. 2007) and JADEX (Braubach et al. 2005) MASs and the evaluation process has been carried out for both platforms. To simplify the comparison, twelve different test cases have been chosen for this evaluation. These test cases are quite different among them, different Mock Agents are used, different number of agents are involved in each one of them and the interaction protocols among agents are different too.

Note that the vertical axis of the graphics shown in Fig. 13a and b are in logarithmic scale. In both graphics, columns represent the code lines of AUT and the lines represent the code lines required to implement the test with (solid line) and without (dashed line) BEAST Tool. Figure 13a shows the benefits of BEAST Tool in number of code lines required to implement the same test using

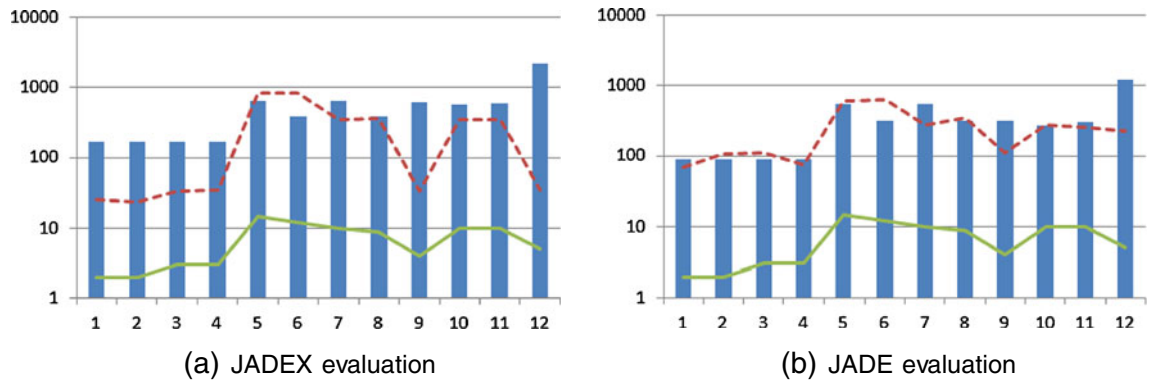

Fig. 13 Test code lines (Y axis) per Test Case (X axis) comparison for JADEX (**a**) and JADE (**b**)

BEAST Tool and without it for JADEX. The improvement is, in average, 247.91 lines per test, i.e. a saving of 97.22 %. Figure 13b shows the same comparison for JADE with similar test cases. The improvement in this case is, in average, 262,08 lines per test, i.e. a saving of 97,36 %.

Nevertheless, the main advantages of the BEAST approach do not come from the saving in coding tasks. The main benefit of our approach is the significant increase in communication between the stakeholders of the software project and the development team, thanks to the usage of an ubiquitous language and its formalisation using BDD templates. The traceability from user requirements to the executable tests are a key point to know which tests must be executed to know if the system meets a concrete requirement. Thus, the stakeholder can check easily the status of the project at each iteration.

7 Conclusions and future work

This article has proposed an agile testing methodology for Multi-Agent Systems based on BDD, termed BEAST Methodology, and a support tool, called BEAST Tool.

The main conclusion of this research is that the BDD approach has been suitable for its application in MAS development. Furthermore, the use of BDD facilitates the communication between stakeholders and designers or developers which, usually, it is a gap between both of them. To solve this problem, BEAST Methodology establishes that stakeholders must generate a set of behaviour specifications that describes the whole system. Later, the MAS designers must generate the set of agent behaviour specifications that fits the solution of the problem. These behaviours in BDD format are translated automatically with BEAST Tool to *JUnit* test cases. During this process, text plain in natural language is always available to facilitate the specification compression and communication between both stakeholders and development team.

Other common issue in MAS development is the need of other agents to test the behaviour of an AUT. As these agents are not developed yet, BEAST uses Mock Agents to allow developers to ensure the correct behaviour of an AUT. To add flexibility to mocking technique, *Mockito* (Mockito Project 2012) framework has been integrated with BEAST Tool to allow the use of the facilities of the mocking framework, such as mock agents, mock web services or mock Java objects.

Besides, the use of MAS testing techniques or methodologies are commonly strongly connected to a specific MAS platform or design methodology (Coelho et al. 2006; Gómez-Sanz et al. 2009; Nguyen et al. 2008). BEAST Tool is easily adaptable for MAS frameworks as there is a clear interface between the tool and the MAS platform. Currently, JADE 4.0 (Bellifemine et al. 2007) and JADEX 2.0 (Braubach et al. 2005) are supported and has no restriction about the design methodology, as BEAST Methodology deals with the specification of the system behaviours and with the tests to check if the final agents meet those requirements. The internal design of the agents that compose the MAS is not covered by the proposed testing methodology.

For future work, we will study in depth the use of simulations for MAS Level Testing (see Section 3) in order to cover all possible test like non-functional tests, for example, performance of all agents working together or the achievement of high level goals that can be only in a collaborative way. For this purpose, we plan to explore some interesting simulation techniques (Uhrmacher et al. 2009) and their application using MASON framework (Luke et al. 2005) will be explored. That framework has been chosen since our group has a wide experience using it and its integration with our developed BEAST Tool (and with JADE and JADEX) is simpler since it is written in Java. We are interested in simulate external systems to reproduce the environment in where the MAS must execute. Obviously, the MAS cannot be deployed until it is completely tested with guaranties that it works. So, the simulated environment can be used

to perform many type of tests, such as the existence of non-expected behaviour in the final system (with all agent roles working at the same time) or stress tests to ensure the performance of the developed MAS in real-simulated situations.

We also plan to improve BEAST Tool to support other MAS platforms, like JASON (Bordini et al. 2007), to maximize the scope of the developed tool.

Finally, other interesting issue is to evaluate other non-BDD approaches for system specifications provided by a stakeholder, like FIT (Mugridge and Cunningham 2005), that support the specification of test cases with concrete examples that provide real data. This first step of the methodology is really important and the capability for stakeholders to choose the format of system specifications can be a key point for a successful project.

Acknowledgments This research work is supported by the Ministry of Economy and Competitiveness under the R&D project CALISTA (TEC2012-32457). The authors want to acknowledge the cooperation of Telefónica R&D, and Javier García-Algarra, Javier García-Ordás, Beatriz Fuentes, Raquel Toribio and Andrés Sedano-Frade.

References

Adzic, G. (2009). *Bridging the communication gap: specification by example and agile acceptance testing*. UK: Neuri Limited.

Adzic, G. (2011). *Specification by example: how successful teams deliver the right software*, 1st edn. Greenwich: Manning Publications Co.

Agarwal, N., & Rathod, U. (2006). Defining success for software projects: an exploratory revelation. *International Journal of Project Management*, *24*(4), 358–370.

Almeida, J., Iacob, M.E., Eck, P. (2007). Requirements traceability in model-driven development: applying model and transformation conformance. *Information Systems Frontiers*, *9*(4), 327–342.

Bellifemine, F.L., Caire, G., Greenwood, D. (2007). *Developing multi-agent systems with JADE, Wiley series in agent technology* (Vol. 5). West Sussex:Wiley.

Bordini, R.H., Hübner, J.F., Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason* (Vol. 8). West Sussex:Wiley.

Braubach, L., Pokahr, A., Lamersdorf, W. (2005). Jadex: A BDI-agent system combining middleware and reasoning. In R. Unland, M. Calisti, M. Klusch, M. Walliser, S. Brantschen, T. Hempfling (Eds.), *Software agent-based applications, platforms and development kits. Whitestein series in software agent technologies and autonomic computing* (pp. 143–168). Basel: Birkhäuser.

Clynch, N., & Collier, R. (2007). Sadaam: software agent development-an agile methodology. In *Proceedings of the workshop of languages, methodologies, and development tools for multi-agent systems (LADS007)*. Durham.

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 06* (pp. 83–90). New York: ACM.

Dikenelli, O., Erdur, R.C., Gumus, O. (2005). Seagent: a platform for developing semantic web based multi agent systems. In *Proceedings of the fourth international joint conference on autonomous agents and multiagent systems, AAMAS '05* (pp. 1271–1272). New York: ACM.

Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Boston:Addison-Wesley Professional.

Foundation TAS (2011). *Apache maven project*. Accessed 23 October 2012. http://maven.apache.org.

García-Magariño, I., Gómez-Rodríguez, A., Gómez-Sanz, J., González-Moreno, J. (2009). Ingenias-scrum development process for multi-agent development. In *International symposium on distributed computing and artificial intelligence 2008 (DCAI 2008)* (pp. 108–117). Springer.

Gómez-Sanz, J., Botía, J., Serrano, E., Pavón, J. (2009). Testing and debugging of MAS interactions with INGENIAS. In M. Luck, & J. Gomez-Sanz (Eds.), *Agent-oriented software engineering IX. Lecture notes in computer science* (Vol. 5386, pp. 199–212). Heidelberg: Springer.

Guerra-García, C., Caballero, I., Piattini, M. (2013). Capturing data quality requirements for web applications by means of dqwebre. *Information Systems Frontiers*, *15*(3), 433–445.

Hamill, P. (2004). *Unit test frameworks*, 1st edn. California:O'Reilly.

Haugset, B., & Hanssen, G. (2008). Automated acceptance testing: A literature review and an industrial case study. In *Agile, 2008. AGILE '08. Conference* (pp. 27–38).

Houhamdi, Z. (2011). Multi-agent system testing: a survey. *International Journal of Advanced Computer Science and Applications (IJACSA)*, *2*(6), 135–141.

Iglesias, C., Garijo, M., González, J., Velasco, J. (1998). Analysis and design of multiagent systems using mas-commonkads. In M. Singh, A. Rao, M. Wooldridge (Eds.), *Intelligent agents IV agent theories, architectures, and languages. Lecture notes in computer science* (Vol. 1365, pp. 313–327). Berlin: Springer.

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G. (2005). Mason: a multiagent simulation environment. *Simulation*, *81*(7), 517–527.

Marnewick, A., Pretorius, J.H., Pretorius, L. (2011). A perspective on human factors contributing to quality requirements: a cross-case analysis. In *2011 IEEE international conference on industrial engineering and engineering management (IEEM)* (pp. 389–393).

Martin, R., & Melnik, G. (2008). Tests and requirements, requirements and tests: a möbius strip. *IEEE Software*, *25*(1), 54–59.

Mockito Project (2012). *Mockito framework*. Accessed 25 March 2012. http://mockito.org.

Mugridge, R., & Cunningham, W. (2005). *Fit for developing software: framework for integrated tests*. Upper Saddle River, NJ:Prentice Hall.

Nguyen, C.D. (2009). *Testing techniques for software agents*. PhD thesis. International Doctorate School in Information and Communication Technologies.

Nguyen, D., Perini, A., Tonella, P. (2008). A goal-oriented software testing methodology. In M. Luck, & L. Padgham (Eds.), *Agent-oriented software engineering VIII. Lecture notes in computer science* (Vol. 4951, pp. 58–72). Berlin: Springer.

Nguyen, C.D., Perini, A., Tonella, P. (2010). Goal oriented testing for mass. *International Journal of Agent-Oriented Software Engineering*, *4*(1), 79–109.

Nguyen, C., Perini, A., Bernon, C., Pavón, J., Thangarajah, J. (2011). Testing in multi-agent systems. In M.P. Gleizes, & J. Gomez-Sanz (Eds.), *Agent-oriented software engineering X. Lecture notes in computer science* (Vol. 6038, pp. 180–190). Berlin: Springer.

North, D. (2007). *Introducing: behaviour-driven development*. Accessed 28 March 2012. http://dannorth.net/introducing-bdd.

North, D. (2011). *JBehave. A framework for behaviour driven development (BDD)*. Accessed 28 March 2012. http://jbehave.org.

Padgham, L., & Winikoff, M. (2003). Prometheus: a methodology for developing intelligent agents. In F. Giunchiglia, J. Odell, G. Weiß (Eds.), *Agent-oriented software engineering III. Lecture notes in computer science* (Vol. 2585, pp. 174–185). Berlin: Springer.

Pavon, J., Gomez-Sanz, J.J., Fuentes, R. (2005). The ingenias methodology and tools. *Agent-Oriented Methodologies*, *9*, 236–276.

Schwaber, K., & Sutherland, J. (2009). Scrum guide. *Scrum Alliance*, *19*(6), 21.

Solis, C., & Wang, X. (2011). A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO conference on software engineering and advanced applications (SEAA)* (pp. 383–387).

Sun, L., Ousmanou, K., Cross, M. (2010). An ontological modelling of user requirements for personalised information provision. *Information Systems Frontiers*, *12*(3), 337–356.

Thangarajah, J., Jayatilleke, G., Padgham, L. (2011). Scenarios for system requirements traceability and testing. In *The 10th international conference on autonomous agents and multiagent systems-international foundation for autonomous agents and multiagent systems, AAMAS '11* (Vol. 1, pp. 285–292). Richland.

Tiryaki, A., ztuna, S., Dikenelli, O., Erdur, R. (2007). SUNIT: a unit testing framework for test driven development of multi-agent systems. In L. Padgham, & F. Zambonelli (Eds.), *Agent-oriented software engineering VII. Lecture notes in computer science* (Vol. 4405, pp. 156–173). Berlin: Springer.

Uhrmacher, A.M., Uhrmacher, A., Weyns, D. (2009). *Multi-agent systems: simulation and applications*. Boca Raton, FL: CRC Press.

Wooldridge, M., Jennings, N.R., Kinny, D. (2000). The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, *3*(3), 285–312.

Wynne, M., & Hellesøy, A. (2008). *Cucumber. Behaviour driven development with elegance and joy*. Accessed 28 March 2012. http://cukes.info.

Zhang, Z., Thangarajah, J., Padgham, L. (2011). Automated testing for intelligent agent systems. In M.P. Gleizes, & J. Gomez-Sanz (Eds.), *Agent-oriented software engineering X. Lecture notes in computer science* (Vol. 6038, pp. 66–79). Berlin: Springer.

Álvaro Carrera is a PhD candidate at Universidad Politécnica de Madrid. He has worked in the industry at Telefónica R&D. His research is focussed on diagnosis systems based on multi-agent systems for telecommunication networks. He has published several papers on topics such as distributed Bayesian reasoning, testing methodologies, and agent architectures. He has taken part in several national and European projects.

Carlos A. Iglesias is an associate professor at the Universidad Politécnica de Madrid, Spain. His research interests are in multi-agent systems, service engineering, and Web engineering. Iglesias has a PhD in telecommunications engineering from the Universidad Politécnica de Madrid.

Mercedes Garijo is Associate Professor at the Universidad Politécnica de Madrid, where she received her MD and PhD in Telecommunication Engineering. She teaches at the School of Telecommunication Engineering (undergraduate, postgraduate and doctorate levels) in computer science and communications engineering. Her research interests are in telematic services engineering using techniques of software engineering and intelligent systems, especially ontologies, machine learning, and intelligent agents. She has been involved in several research and development projects funded by the EC and the Spanish government.