Behaviour Driven Development for Multi-Agent Systems

Álvaro Carrera, Jorge J. Solitario, and Carlos A. Iglesias
Dpto. de Ingeniería de Sistemas Telemáticos
Universidad Politécnica de Madrid, UPM
Madrid, Spain
{*a.carrera,jjsolitario,cif*}*@dit.upm.es*

Abstract

This paper presents a testing methodology to apply Behaviour Driven Development (BDD) techniques while developing Multi-Agent Systems (MAS), so called BEhavioural Agent Simple Testing (BEAST) methodology. It is supported by the developed open source framework (BEAST Tool) which automatically generates test cases skeletons from BDD scenarios specifications. The developed framework allows testing MASs based on JADE or JADEX platforms and offers a set of configurable Mock Agents which allow the execution of tests while the system is under development. BEAST tool has been validated in the development of a MAS for fault diagnosis in FTTH (Fiber To The Home) networks.

Keywords: agent, test, simple, management, development, BDD, multi-agent systems, MAS, mock-agents, JADE, JADEX.

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 [4]. Thus, requirements engineering plays a key role in the development process. The main challenges of requirements engineering are [17]: (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* [2]. The main reasons for this communication gap between stakeholders and the development team are that [2] (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.

In order to bridge this communication gap, 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. Moreover, given that requirements will have inconsistencies and gaps [3], 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" [18].

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 [19]) or as scenarios (BDD [25] with tools such as JBehave [24] or Cucumber [30]). 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 [14]. Agile acceptance testing complements Test Driven Delevopoment (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 multi-agent systems. Multi-agent systems testing present several challenges [20], 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 [20, 21, 16]. To the best of our knowledge, there is no previous work dealing explicitly with acceptance testing in Agent Oriented Software Engineering (AOSE). Thangarajah et al [28] 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 for based on percepts, goals and actions. Nguyen et al. [22] 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. External testing produces acceptance tests for being validated by project stakeholders, while internal testing produces system and agent tests for being verified by developers. They focused on internal testing.

[1]A literate review of ATDD can be found in [15].

The research context of this article was a research project contracted by the company Telefonica. They requested us to develop a multi-agent system for fault diagnosing in their network. From a software engineering point of view, the main challenges were: (i) they required managing the project using the agile SCRUM methodology [26], (ii) the project involved integration with a wide range of external systems and the emulation 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 AOSE proposals based on agile principles [8, 12], 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 organised as follows. First, we propose a testing methodology for MAS based on BDD techniques in Sect. 2 which is supported by an open source tool. Sect. 3 presents a worked example of the application of the methodology and the tool. Then, Sect. 4 presents an evaluation of the benefits of the proposed approach. Finally, Sect. 5 presents some concluding remarks.

2 BEAST Methodology

In order to cover the problems identified in Sect. 1, we should identify which requirements should have the testing methodology. First, our primary concern is that it should help in improving the communication between the stakeholders and the development team. In addition, it should help to improve the communication between 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 (so-called *sprints* in SCRUM terminology). 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 consists of four phases (Fig. 1): *System Behaviour Specification*, *MAS*

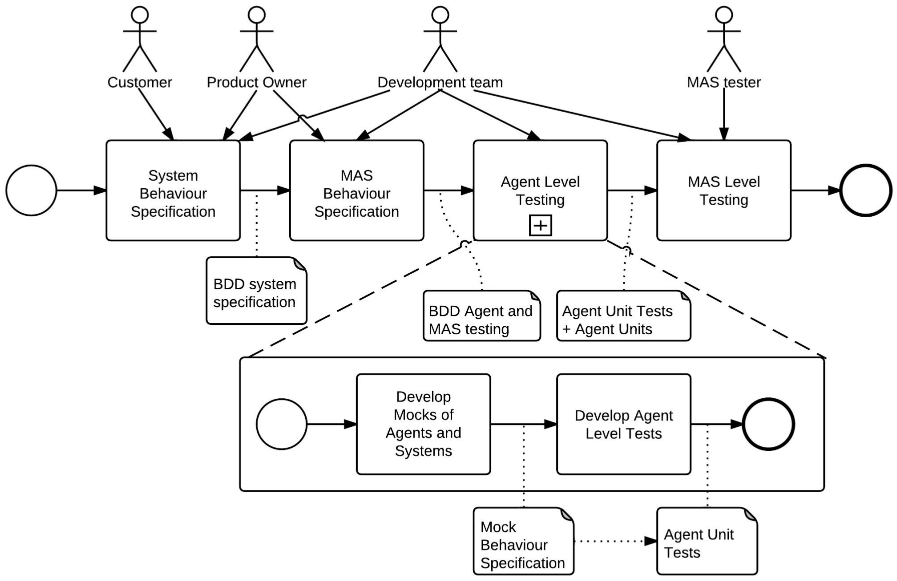

Customer
Product Owner
Development team
MAS tester
System Behaviour Specification
MAS Behaviour Specification
Agent Level Testing
MAS Level Testing
BDD system specification
BDD Agent and MAS testing
Agent Unit Tests + Agent Units
Develop Mocks of Agents and Systems
Develop Agent Level Tests
Mock Behaviour Specification
Agent Unit Tests

Figure 1: Beast Testing Methodology

Behaviour Specification, *Agent level testing* and *MAS level testing*, which are applied in each agile iteration.

2.1 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 [25]. System behaviours are derived from the business outcomes the system intends to produce. These business outcomes should be prioritized by the stakeholders. Then, business outcomes are drilled down to feature sets. 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. These scenarios are the basis of acceptance tests. Instead of using plain natural language, BDD proposes the usage of textual templates (Tables 2 and 3). These templates should be instantiated by the pertinent concepts. These concepts are part of the *ubiquitous language* [11] 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.

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

Figure 2: User Story template

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

Figure 3: Scenario template [25]

2.2 MAS Behaviour Specification

This phase has the goal of architecturing the multi-agent system. Based on the features identified in the previous phase, the new features are realised with the MAS system. This can involve adding, removing or modifying behaviours developed in the previous iterations. 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.* 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.

This phase could be skipped and system behaviours could be directly translated into agent unit tests (Sect. 2.3). 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 provide better insight for developers about them.

2.3 Agent level testing

Based on 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 (Fig. 1): (i) developing mocks of the external systems an agent interacts with and (ii) developing the unit tests of every agent.

The first step (Sect. 2.3.1) requires to simulate the intended behaviour of the external systems according to the scenarios described in the previous phase. In our methodology, an external system includes other agents different from the one we are developing.

The second step (Sect. 2.3.2) provides mapping rules from the specifications developed in the previous phases to executable code.

2.3.1 Mock development

There have been several research works developing the concept of using mock testing for agent unit testing. Coelho et al. [9] proposed a framework for unit testing of MAS based on the use of mock agents on top of the multiagent framework JADE [5]. 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. [29] proposed the framework *SUnit* on top of the multiagent framework *Seagent* [10]. They extended *JUnit* testing in order to cope with agent plan structures testing. Zhang [31] generated automatically mock agents from design diagrams developed within the Prometheus methodology.

We propose to use a mock testing technique for simulating external systems, being agents or any other system. In addition, we need that the framework is valid for different MAS platforms, such as JADEX [7] and JADE [5]. In addition, the mocking framework should allow an easy configuration of the mock objects (or agents), with patterns such as *when(< some input >).thenReturn(< some answer >)*. After analysing available mocking frameworks, we have selected *Mockito* [1] 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.

Given that we are interested in simulating the behaviour of agents, we have created several classes which provide a simple FIPA interface. In particular, we provide these three agent classes:

- *ResponderMockAgent:* mock agent that replies to predetermined incoming messages.
- *MediatorMockAgent:* mock agent that sends a message to a third agent based on predefined filters.
- *ListenerMockAgent:* mock agent that just receives messages.

Mock agents allow the specification of the simulated behaviour using Mockito constructions. Here follows an example.

```
when(mockAgent.processMessage(
  eq(``REQUEST''),
  eq(``VoD Loss Rate'')))
  .thenReturn(``INFORM'',``Loss Rate=0.2'')
```

2.3.2 Agent testing

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. Thus, *JBehave* has been extended with this purpose. Mapping rules [27] provide a standard mapping from scenarios to test code. In JBehave, 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. This Java annotation text provides the name for the test method.

In BEAST, a *stakeholder user story* (obtained in *System Behaviour Specification* phase) is a file that contains *agent stories* (obtained in *MAS Behaviour Specification* phase). These *agent stories* are files which contain a set of *scenarios*. BEAST Tool translates a "Given/When/Then" scenario to a test case class which extends JBehave JUnitStory class and contains three key methods which facilitates the connection to the MAS platform:

- *setUp* method. The "Given" scenario condition previously defined in natural language is translated into Java. This method typically initialises agents and configures their state (goals, beliefs, . . .) as well as initialises the environment.
- *launch* method. The "When" scenario condition is implemented in Java. This method generates and schedules the trigger event to start the test.
- *verify* method. The "Then" scenario condition is checked here. The expected states (goals, beliefs, etc.) are checked in this method once test execution is over.

In order to provide MAS platform independence, a testing interface selector has been defined, so called *PlatformSelector* (see Fig. 4). This selector provides the proper platform access according to the MAS framework specified in the configuration file. This access consists of three interfaces that should be implemented in order to integrate a MAS platform:

- *Connector* interface provides an abstract interface to agent managing functions (launch platform, start an agent, etc.).
- *Messenger* interface declares methods for sending and receiving messages from the platform.
- *Agent Introspector* interface provides access to the agent model (such as goals, beliefs, etc.).

These interfaces have been implemented for JADE 4.0 [5] and JADEX 2.0 [7]. A requirement for integrating an agent platform is that it provides methods for external interaction. For example, the integration of JADEX 0.96 was too complex to be carried out because of the lack of these interfaces.

2.4 MAS level testing

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.

Currently we do not provide specific facilities for this phase and will be developed as future work. Nonetheless, we would like to point out that once mocks objects have been replaced by the real entities they emulate, business requirements

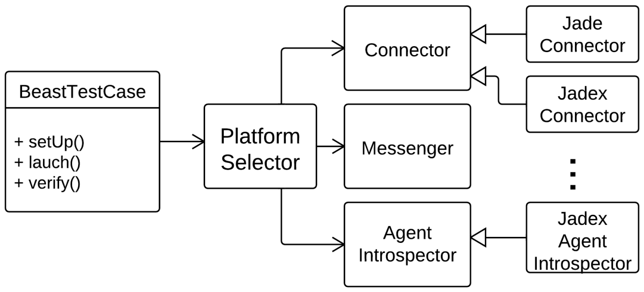


Figure 4: Beast Test Case

can be tested on the real system. Thus, acceptance testing is straight forward, and stakeholders' expectations 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 for refining iteratively requirements based on the current project advance and available resources.

3 Case Study: MAS for fault diagnosis on Fiber To The Home (FTTH) scenario

To properly frame our proposed BEAST Technique, a network management project was chosen to examplify the use of BEAST. In this example, our stakeholder is a network operator company which wants a tool to reduce the management cost of FTTH networks.

The first step of the project is to write a high level project proposal and to explore different possible approaches to solve the problem. The result of this phase is that the solution that best fits the problem is a MAS architecture. The project and BEAST Methodology, assisted by the BEAST Tool, is used in the project. For exemplification purposes, it is exposed how tests can be implemented in JADEX [7] framework.

So, one of the next tasks that the development team has to do is to arrange a meeting with the stakeholder to specify a set of requirements. In this meeting, the main requirements of the stakeholder are written in BDD format (see Sect. 2). Table 1 shows an example of one gathered requirement.

Notice here that the stakeholder does not know anything about the solution, in this case, a Multi-Agent Systems (MAS). So, the written requirements do not refer at all to the final agents. These requirements are the *acceptance tests* of the project.

Once the developers have the requirements, they can write even more test cases in BDD format. Let us suppose at this point that the development team has identified two agent roles (Probe and Diagnosis) for implementing the requested behaviour as shown in Table 1).

The first role, *Probe Agent*, is responsible for monitoring the service quality,

```
As a operator network,
I want to have a system to diagnose faults root cause
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
  a 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'.
```

Table 1: Stakeholder Requirements Example

```
As a Diagnosis Agent,
I want to diagnose failures when I receive a symptom
So that I am able to report the diagnosis result to other agents.

Scenario: Diagnosis Agent diagnoses Damaged Splitter
Given a 'high loss rate' symptom is received from a Probe Agent,
When two or more geographically close users have loss rate higher to 1%,
Then the Diagnosis Agent must diagnose the root cause of the
  problem is 'Damaged Splitter'.
```

Table 2: Developers Test Cases

while the second role, *Diagnosis Agent*, is responsible for diagnosing faults. These two roles will be implemented in a distributed fashion. Thus, the previous *acceptance tests* is further refined and, as a result, several detailed test cases are obtained, shown in Table 2).

These tests can be further refined and more specific test cases can be defined for obtaining a suitable testing coverage of the desired agent behaviour.

At this stage, the developer has not developed yet *Probe* and *Expert* agents. Since they are required for implementing the aforementioned tests, a mocking facility of BEAST Tool will be used. As previously introduced, BEAST Tool includes several Mock patterns. In this case, the *Mediator Mock Agent* is suitable for implementing both *Probe* and *Expert* agents. Thus, this mock is configured for sending symptoms and network information, respectively.

about the network status around the location of the user. Table 3 shows a sample of an implementation of the *setUp* method in the BEAST test case class.

```
public void setUp() {
 startAgent("DiagnosisAgent","DiagnosisAgent.agent.xml");
 AgentBehaviour mockBeh =  mock(AgentBehaviour.class);
 when(mockBeh.processMessage(eq(INFORM),
      eq("Generate High Loss Rate Symptom")))
  .thenReturn("DiagnosisAgent", INFORM, "Loss rate=0.15");
 MockConfiguration mockConf = new MockConfiguration();
 mockConf.addBehaviour(mockBeh);
 MockManager.startMockJadexAgent("ProbeMockAgent","MediatorMock.agent.xml",
   mockConf,this);
}
```

Table 3: setUp method implementation

```
public void launch() {
 sendMessageToAgent("ProbeMockAgent", INFORM, TRIGGER_EVENT);
 setExecutionTime(2000);// Waiting time in milliseconds
}
```

Table 4: launch method implementation

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

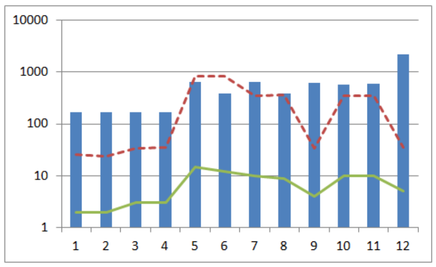
Table 5: verify method implementation

In a similar way, the other two methods of the test class are programmed. The *launch* method of the test case generates a trigger event that initiates the test and fixes the test duration(see Table 4). Then, the *verify* method consists of checking the expected status of the agent under test (see Table 5).

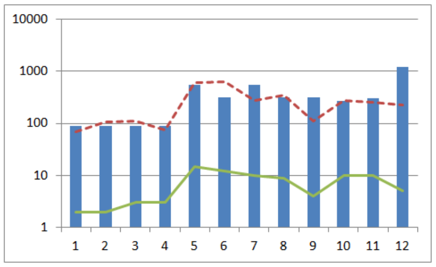
After this test coding task, tests can be launched using standard development tools, since it is a standard JUnit test.

4 Evaluation

The results of the proposed BEAST Methodology 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



(a) JADEX evaluation



(b) JADE evaluation

Figure 5: Test code lines (Y axis) per Test Case (X axis) comparison for JADEX (a) and JADE (b)

implementation of tests. These metrics are strongly associated with 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 depend on the programming skills of the developer.

BEAST Tool is already adapted to test JADE [5] and JADEX [7] MAS and the evaluation process has been carried out for both platforms. To simplify the comparison, only twelve different test cases have been chosen for this evaluation. These test cases are quite different among them, some of them use Mock Agents, different number of agents are involved in each one, etc.

Notice that graphics shown in Fig. 5(a) and in Fig. 5(b) are in logarithmic scale. In both graphics, columns represent the code lines of agents under testing and lines represent the code lines required to implement the test with (solid line) and without (dashed line) BEAST Tool. Fig. 5(a) shows the benefits of BEAST Tool in number of lines required to implement the same test using BEAST Tool and without it for JADEX. The improvement is, in average, 247.91 lines per test, i.e. a saving of 97.22%. Fig. 5(b) 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 all the stakeholders of the software project, thanks to the usage of an ubiquitous language and its formalisation using BDD templates. Moreover, this BDD technique is also used for developing MAS, which also improves communication among developers.

5 Conclusions and future work

Advances in MAS testing has been introduced in this paper using BEAST Technique and BEAST Tool. BDD perfectly fits its specification describing behaviours of a determined agent or of a set of agents. 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 Technique establishes that stakeholders must generate a set of behaviour specifications that describes the whole system. Later, 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 parts.

Other common issue in MAS development is the need of other agents to test the behaviour of an Agent Under Test (AUT). Since 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 Mocks, Mockito framework is integrated with BEAST Tool to allow the use of Mock Agents, Mock Web Services, Mock Java Objects, etc.

Besides, the use of MAS testing techniques or methodologies are commonly strongly connected to a specific MAS platform [9, 13, 23]. BEAST Tool are easily adaptable for any MAS framework. Currently, JADE and JADEX testing are supported.

For future work, we will study in depth the use of simulations for MAS Level Testing (see Sect. 2) 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, MASON framework will be explored.

We also plan to improve BEAST Tool to support other MAS frameworks, like JASON [6], 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 [19], 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.

References

[1] Mockito Framework. `http://mockito.org`. Accessed March 25, 2012.

[2] G. Adzic. *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing*. Neuri Limited, United Kingdom, 2009.

[3] G. Adzic. *Specification by Example: How Successful Teams Deliver the Right Software*. Manning Publications, 2011.

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

[5] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*, volume 5 of *Wiley Series in Agent Technology*. Wiley, 2007.

[6] R. H. Bordini, M. Wooldridge, and J. F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[7] L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex: A BDI-Agent System Combining Middleware and Reasoning. In R. Unland, M. Calisti, M. Klusch, M. Walliser, S. Brantschen, and T. Hempfling, editors, *Software Agent-Based Applications, Platforms and Development Kits*, Whitestein Series in Software Agent Technologies and Autonomic Computing, pages 143–168. Birkhäuser Basel, 2005.

[8] N. Clynch and R. Collier. Sadaam: Software agent development-an agile methodology. In *Proceedings of the Workshop of Languages, methodologies, and Development tools for multi-agent systems (LADS007), Durham, UK*, 2007.

[9] R. Coelho, U. Kulesza, A. von Staa, and C. Lucena. 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, pages 83–90, New York, NY, USA, 2006. ACM.

[10] O. Dikenelli, R. Erdur, and O. Gumus. 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*, pages 1271–1272. ACM, 2005.

[11] E. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[12] I. García-Magariño, A. Gómez-Rodríguez, J. Gómez-Sanz, and J. González-Moreno. Ingenias-scrum development process for multi-agent development. In *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, pages 108–117. Springer, 2009.

[13] J. Gómez-Sanz, J. Botía, E. Serrano, and J. Pavón. Testing and Debugging of MAS Interactions with INGENIAS. In M. Luck and J. Gomez-Sanz, editors, *Agent-Oriented Software Engineering IX*, volume 5386 of *Lecture Notes in Computer Science*, pages 199–212. Springer Berlin / Heidelberg, 2009.

[14] P. Hamill. *Unit test frameworks*. O'Reilly, first edition, 2004.

[15] B. Haugset and G. Hanssen. Automated acceptance testing: A literature review and an industrial case study. In *Agile, 2008. AGILE '08. Conference*, pages 27 –38, aug. 2008.

[16] Z. Houhamdi. Multi-agent system testing: A survey. *International Journal of Advanced Computer*, 2011.

[17] A. Marnewick, J.-H. Pretorius, and L. Pretorius. A perspective on human factors contributing to quality requirements: A cross-case analysis. In *Industrial Engineering and Engineering Management (IEEM), 2011 IEEE International Conference on*, pages 389 –393, dec. 2011.

[18] R. C. Martin and G. Melnik. Tests and requirements, requirements and tests: A möbius strip. *IEEE Software*, 25(1):54–59, 2008.

[19] R. Mugridge and W. Cunningham. *Fit for developing software: framework for integrated tests*. Prentice Hall, 2005.

[20] C. D. Nguyen. *Testing Techniques for Software Agents*. PhD thesis, 2009.

[21] C. D. Nguyen, A. Perini, C. Bernon, J. Pavón, and J. Thangarajah. Testing in multi-agent systems. In *Proceedings of the 10th international conference on Agent-oriented software engineering*, AOSE'10, pages 180–190, Berlin, Heidelberg, 2011. Springer-Verlag.

[22] C. D. Nguyen, A. Perini, and P. Tonella. Goal oriented testing for mass. *Int. J. Agent-Oriented Softw. Eng.*, 4(1):79–109, Dec. 2010.

[23] D. Nguyen, A. Perini, and P. Tonella. A Goal-Oriented Software Testing Methodology. In M. Luck and L. Padgham, editors, *Agent-Oriented Software Engineering VIII*, volume 4951 of *Lecture Notes in Computer Science*, pages 58–72. Springer Berlin / Heidelberg, 2008.

[24] D. North. JBehave. A framework for Behaviour Driven Development (BDD). `http://jbehave.org`. Accessed March 28, 2012.

[25] D. North. Introducing: Behaviour-driven development. `http://dannorth.net/introducing-bdd`, 2007. Accessed March 28, 2012.

[26] K. Schwaber. Scrum development process. In *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA*, pages 117–134, 1995.

[27] C. Solis and X. Wang. A study of the characteristics of behaviour driven development. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 383 –387, 30 2011-sept. 2 2011.

[28] J. Thangarajah, G. Jayatilleke, and L. Padgham. Scenarios for system requirements traceability and testing. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '11, pages 285–292, Richland, SC, 2011. International Foundation for Autonomous Agents and Multiagent Systems.

[29] A. Tiryaki, S. Öztuna, O. Dikenelli, and R. Erdur. Sunit: A unit testing framework for test driven development of multi-agent systems. *Agent-Oriented Software Engineering VII*, pages 156–173, 2007.

[30] M. Wynne and A. Hellesy. Cucumber. Behaviour driven development with elegance and joy. `http://cukes.info`. Accessed March 28, 2012.

[31] Z. Zhang, J. Thangarajah, and L. Padgham. Automated testing for intelligent agent systems. In M.-P. Gleizes and J. Gomez-Sanz, editors, *Agent-Oriented Software Engineering X*, volume 6038 of *Lecture Notes in Computer Science*, pages 66–79. Springer Berlin / Heidelberg, 2011.