Open access • Journal Article • DOI:10.1057/JOS.2012.26

A generic testing framework for agent-based simulation models — Source link

Önder Gürcan, Oguz Dikenelli, Carole Bernon

Institutions: Ege University

Published on: 25 Jan 2013 - Journal of Simulation (Palgrave Macmillan UK)

Topics: Verification and validation of computer simulation models, Verification and validation, Discrete event simulation and Scenario testing

A generic testing framework for agent-based simulation models

Önder Gürcan, Oguz Dikenelli, Carole Bernon

A generic testing framework for agent-based simulation models

Ö Gürcan[1,2]*, O Dikenelli[1] and C Bernon[2]

[1]*Ege University, Izmir, Turkey;* [2]*Toulouse III University, Toulouse, France*

Agent-based modelling and simulation (ABMS) had an increasing attention during the last decade. However, the weak validation and verification of agent-based simulation models makes ABMS hard to trust. There is no comprehensive tool set for verification and validation of agent-based simulation models, which demonstrates that inaccuracies exist and/or reveals the existing errors in the model. Moreover, on the practical side, many ABMS frameworks are in use. In this sense, we designed and developed a generic testing framework for agent-based simulation models to conduct validation and verification of models. This paper presents our testing framework in detail and demonstrates its effectiveness by showing its applicability on a realistic agent-based simulation case study.

Keywords: agent-based modelling and simulation; model testing; verification and validation

1. Introduction

Verification, validation and testing (VV&T) of simulation models is one of the main dimensions of simulation research. Model validation deals with building the *right model*, on the other hand, model verification deals with building the *model right*, as stated in Balci (1994). Model testing is a general technique that can be conducted to perform validation and/or verification of models. Model testing demonstrates that inaccuracies exist in the model or reveals the existing errors in the model. In model testing, test data or test cases are subjected to the model to see if it functions properly (Balci, 1995).

Traditional techniques for VV&T (Sargent, 2005) cannot be transferred easily to agent-based simulation. There are some efforts (Terano, 2007; Klügl, 2008; Niazi *et al*, 2009; Pengfei *et al*, 2011; Railsback and Grimm, 2011), but these studies do not directly deal with model testing process and there is no proposed model testing framework to conduct validation and verification through the model testing process. On the basis of this observation, our main motivation is to build a testing framework for agent-based simulation models in order to facilitate the model testing process. Such a testing framework should focus on testing the implementation of the agent-based simulation models, since they are mostly specified by their implementation unlike other multi-agent system (MAS) models. Apparently, increasing the confidence of agent-based simulation models with model testing will contribute to transforming agent-based modelling and simulation (ABMS) from a potential modelling revolution (Bankes, 2002) to an actual modelling revolution with real-life implications.

Naturally, one has to define all the model testing requirements of ABMS to be able to develop a model testing framework. To define these requirements, we first identify the basic elements of ABMS that can be subject of a model testing process. Then, we use a generic model testing process (Balci, 1994) and elaborate on the requirements of the model testing framework when it is used throughout this process. Finally, we categorize requirements of model testing of ABMS into *micro-*, *meso-* and *macro-levels* by an inspiration from ABMS applications in sociology domain (Troitzsch, 1996). These levels describe a system considering its size, its characteristics and an inclusion relation with other systems or subsystems. In this categorization, the *micro-level* takes the basic elements individually and defines the framework requirements from the perspective of each basic element. The *meso-level* considers a group of basic elements and assumes that such a group has a well-defined model that needs to be validated. Hence, the meso-level defines model testing requirements of such groups. And the *macro-level* considers the systems as a whole.

After having defined the requirements of the framework, a conceptual model that includes the conceptual elements to satisfy them is proposed. These elements are specified and brougth together to conduct the model testing of any ABMS application. Then, a generic architecture is introduced, which realizes the conceptual elements. This architecture is extensible in a sense that new functionalities based on domain requirements might be easily included. Also, on the practical side, since there are many agent-based simulation frameworks in use (Nikolai and Madey, 2009), the proposed architecture is generic enough to be customized for different frameworks.

**Correspondence:* *Ö Gürcan, Computer Engineering Department, Ege University, Universite cad, Izmir, Bornova 35100, Turkey.*
E-mail: onder.gurcan@ege.edu.tr

This paper is organized as follows. The next section defines the testing requirements for ABMS. Section 3 then describes the generic agent-based simulation testing framework we propose. A case study that shows the effectiveness of the proposed framework is studied in Section 4. After discussing the proposal in Section 5, Sections 6 and 7 conclude the paper with an insight into some future work.

2. Testing requirements for agent-based simulation models

This section deals with the testing requirements for agent-based simulation models by first identifying its basic elements.

2.1. Basic elements

The basic elements of agent-based simulations are agents, the simulated environment and the simulation environment (Klügl *et al*, 2005). *Agents* are active entities that try to fulfill their goals by interacting with other agents and/or simulated environments in which they are situated. They behave autonomously depending on their knowledge base. Moreover, during an agent-based simulation, new agents may enter the system and/or some agents may also disappear.

A *simulated environment* contains agents and non-agent entities of the simulation model. This environment can also carry some global state variables that affect all the agents situated in it and can have its own dynamics like the creation of a new agent. In an agent-based simulation model, there must be at least one simulated environment. However, there may also be various simulated environments with various properties depending on the requirements and the complexity of the model. Apart from explicitly specified behaviours of these model elements (agent and simulated environments), higher level behaviours can emerge from autonomous agent behaviours and model element interactions (agent-to-agent interactions and agent-to-simulated environment interactions). As well as agent-to-agent interactions, a small change in the simulated environment can also dramatically change the nature, and even the occurence, of high-level behaviours (Polack *et al*, 2010). As a result, simulated environments are as important as agents in order to reach the purpose of the simulation study.

The *simulation environment* (or infrastructure), on the other hand, is an environment for executing agent-based simulation models. Independent from a particular model, it controls the specific simulation time advance and provides message passing facilities or directory services. Unlike the other basic elements, the simulation environment is unique for every simulation model and does not affect the higher level behaviours. However, it is not possible to trust totally the simulation environment. In this sense, replicating the simulation model on different simulation environments is proposed as a solution in some studies (Sansores and Pavon, 2005; Wilensky and Rand, 2007).

2.2. Model testing

The basic elements are developed and brought together following a development process to produce a simulation model (Klügl, 2009). The overall simulation model is also verified and validated in parallel with the development process. Our aim is to develop a generic testing framework to conduct model testing in agent-based simulations. In general, model testing requires the execution of the model under test and evaluating this model based on its observed execution behaviour. Similarly, in the simulation domain this approach is defined as *dynamic validation, verification and testing (VV&T) technique* (see the classification of Balci in (Balci, 1995)). According to Balci, dynamic VV&T techniques are conducted in three steps: *model instrumentation*, *model execution* and *model evaluation*. Below, we interpret those three steps in terms of model testing of agent-based simulations to be able to capture the requirements for the intended testing framework:

1. *Observation points for the programmed or experimental model are defined (model instrumentation)*. An observation point is a probe to the executable model for the purpose of collecting information about model behaviour (Balci, 1995). Model testing requires observation of the system under test using points of observation—this is a strong design constraint on the test application and an important testability criterion (Utting and Legeard, 2007). In this sense, a model element is said to be *testable* if it is possible to define observation points on that element. From the perspective of ABMS, *agents* and *simulated environments* might be testable when it is possible to define observation points for them. The *simulation environment*, on the other hand, is not a testable element. However, it can be used to facilitate the testing process.
2. *The model is executed.* As stated above, in agent-based simulations, model execution is handled by the simulation environment. During model execution, a model testing framework can use the features of the simulation environment (if any) to collect information through the observation points.
3. *The model output(s) obtained from the observation point(s) are evaluated.* Thus, for evaluating the model outputs, a model testing framework should provide the required evaluation mechanisms. Observed outputs are evaluated by using reference data. Reference data could be either empirical (data collected by observing the real world), a statistical mean of several empirical data, or they can be defined by the developer according to the specification of the model.

However, execution-based software testing is usually carried out at different levels (Burnstein, 2003) where at each level there are specific testing requirements and goals. Thus, apart from the model testing framework requirements given in this subsection, to be able to design a well-structured

testing framework, we also need to identify the testing requirements of testable elements in terms of testing levels. In the following subsection, model testing levels for ABMS are described and an orderly progression of these levels is given.

2.3. Levels of testing

In traditional testing literature major phases of testing are unit testing, integration testing, system testing and some type of acceptance testing (Burnstein, 2003). Since the nature of MAS demands different testing strategies, the MAS community interprets testing levels as unit, agent, integration (or group), system (or society) and acceptance (Nguyen *et al*, 2011). They consider unit testing as testing all units that make up an agent and they see agent testing as the integration of these units. Their integration testing considers integration of agents and their interactions with their environments, and system testing considers a MAS running at a target operating environment. The last level, acceptance testing, tests the MAS in the customer's execution environment.

However, the nature of ABMS is also slightly different from the nature of MAS. Thus ABMS demands different testing strategies. Unlike MAS, the developers of ABMS are not just computer scientists and software engineers. There is a wide variety of application domains of ABMS from neuroscience (see, eg, Gürcan *et al*, 2012) to ecology (see, eg, Grimm *et al*, 2005), from social sciences (see, eg, Epstein, 2007) to economy (see, eg, Windrum *et al*, 2007), etc and each domain's experts are trying to build their simulation models by themselves. In this sense, the terminology used in testing of ABMS needs to be more understandable and familiar for the experts of these domains. Morever, the multi-level[1] nature of these domains has already been recognized long time ago (Ghosh, 1986) and consecutively currently there is plenty of work about multi-level simulations on these domains. From the ABMS perspective, this was first realized by Uhrmacher and Swartout (2003). They stated that agent-based simulation models describe systems at two levels of organization: *micro-level* and *macro-level*. However, in sociology the distinction between these levels is comparatively well established (Troitzsch, 1996). The *micro-level* considers the model elements individually and their interactions from their perspectives, while the *macro-level* considers the model elements as one element, and focuses on the properties of this element resulting from the activities at the *micro-level*. The same year, in organizational behaviour domain, House *et al* (1995) proposed the *meso-level* as a framework for the integration of *micro-* and *macro-levels*.

[1]The term *level* and the term *scale* are often used interchangeably. Here, the term *level* is chosen since it situates the described system considering its size, its characteristic evolution time or an inclusion relation with other systems or subsystems. However, the term *scale* refers to a dimension of analysis in which the system of interest can be measured.

The necessity of validating model elements at *micro-* and *macro-levels* in simulation studies was first recognized by Robinson (1997). He defined micro-check of the model as *white-box validation* and macro-check of the operation of the model as *black-box validation*. The *white-box validation* examines whether each element of the model and its structure represents the real world (or the artificial world defined by the developer) with sufficient accuracy. The *black-box validation*, on the other hand, deals with the relationships between the inputs and the outputs of the model, ignoring the elements within this model. However, the *micro-* and the *macro-levels* are not sufficient enough for testing agent-based simulation models since they are pretty large and complex (indeed, many modellers introduce an intermediate level to reduce such complexity). As House *et al* (1995) stated: *micro-* and *macro-*processes cannot be considered separately and then added up to understand behaviour of organizations. In this sense, the *macro-level* emergent behaviours of agent-based simulation models are highly dependent on the behaviour of the groups or sub-societies of the elements. Thus, an intermediate testing level (*meso-level*) to test model elements as a group or sub-society is needed in order to increase the confidence. A group or sub-society consists of model elements that are related, for example, they may cooperate to support a required *macro-level* behaviour of the complete system.

In this sense, we propose *micro-*, *meso-* and *macro-level* testing for ABMS as major testing phases. Since generally domain experts are developing their own agent-based simulation models, we see acceptance test as an activity performed at each level and we include unit testing as a subphase of *micro-level* testing. It is worthwhile noting that the chosen levels are not intended to be comprehensive. They rather provide a useful framework to systematically organize the testing requirements. In the following subsections, depending on the characteristics of agent-based simulations, we define the *micro-*, *meso-* and *macro-level* testing requirements of agent-based simulation models. The testing objectives, subjects to test and activities of each level are described progressively.

Micro-level testing. In this level, the testing requirements of the basic elements alone and interactions from their perspective are considered. The principal goal for *micro-level* testing is to ensure that each individual testable element is functioning according to its specification. In other words the aim is to detect functional and structural defects in a testable element.
In this sense, a *micro-level* test may require the following:

- Testing building blocks of agents like behaviours, knowledge base and so forth and their integration inside agents.
- Testing building blocks of simulated environments like non-agent entities, services and so forth and their integration inside simulated environments.

- Testing the outputs of agents during their lifetime. An output can be a log entry, a message to another agent or to the simulated environment.
- Testing if an agent achieves something (reaching a state or adapting something) in a considerable amount of time (or before and/or after the occurrence of some specific events) with different initial conditions.
- Testing the interactions between basic elements, communication protocols and semantics.
- Testing the quality properties of agents, such as their workload (number of behaviours scheduled at a specific time).

The testable elements should be tested by an independent tester (someone different from the developer) if possible.

Meso-level testing. The *meso-level*, settled between the *micro-* and *macro-levels*, deals with the model elements of an intermediate level. Thus, the testing requirements of the elements of agent-based simulations as groups or sub-societies are considered. With a few minor exceptions, the *meso-level* tests should only be performed on elements that have been reviewed and successfully passed the *micro-level* testing. This level has two major goals: (1) to detect defects that occur on the communication protocols of testable elements and (2) to assemble the individual elements into working sub-societies and finally into a complete system that is ready for the *macro-level* test. This process is driven by assembly of the elements into cooperating groups (the elements that may work together). The cooperating groups of elements are tested as a whole and then combined into higher-level groups.

- There is some simple testing of communication protocols of the elements from their perspective in *micro-level*. However, communication protocols are more adequately tested during the *meso-level* testing when each element is finally connected to a full and working implementation of those communication protocols.
- Testing the organization of the agents (how they are situated in a simulation environment or who is interacting with who) during their lifetime. In this sense, the well-known *K-means* algorithm (MacQueen, 1967) can be used in order to discover and assess interacting groupings of model elements[2] as in Serrano *et al* (2009).
- Testing whether a group of basic elements exhibits the same *long-term* behaviour (which could be *emergent* or not) with different initial conditions.
- Testing whether a group of basic elements is capable of producing some known output data for a given set of input data.
- Testing the timing requirements of the *meso-level* behaviours of a group of basic elements.
- Testing the workload for the system as a whole (number of agents, number of behaviours scheduled, number of interactions etc).

When the *meso-level* tests are completed, an agent-based simulation model has been assembled and its major sub-societies have been tested. At this point, the developers/testers begin to test the system as a whole.

Macro-level testing. The *macro-level* tests are performed after all elements and sub-societies have been created and tested (after the *micro-* and the *meso-level* tests are performed). In this level, thorough end-to-end testing of complete, integrated simulation models from an end-user's perspective is performed. The scope of the *macro-level* testing is different from the *meso-level* one. Rather than configuring and running relatively controlled, focused tests, the *macro-level* tests have a broader perspective. The main goal is to test the expected functionality as a whole. The other goals are to evaluate performance, usability, reliability and other quality-related requirements to increase the confidence of the simulation model.
In this sense, a *macro-level* test may require the following:

- Testing whether the overall system is capable of producing some known output data for a given set of legal input data.[3]
- Testing whether the overall system is capable of remaining available for a given set of illegal input data.
- Testing whether the overall system is capable of producing some known output within given time constraints.
- Testing whether the overall system exhibits the same *long-term* behaviour (which could be *emergent* or not) with different initial conditions.[4]
- Testing the workload for the system as a whole (number of agents, number of behaviours scheduled, number of interactions etc).
- Testing the significance of the simulated data with respect to reference data. This can be done by various data comparison techniques such as cross-correlation analysis, coherence analysis, goodness of fit tests etc.
- The communication protocols are tested in *micro-* and *meso-level* testing levels from individual and group perspectives. However, having a correct execution of protocols does not imply the overall system is behaving correctly. Hence, an agent can execute protocols and still insist on collaborating with wrong agents. To detect such situations, some post-mortem analysis might be required as suggested by Serrano *et al* (2009). To be able to

[2]This algorithm arranges data points into clusters and it locates a centroid in each cluster This centroid is the point at which the distance from the rest of the points of the clusters is on average minimum.

[3]Law (2007) defines a simulation as a numerical technique that takes input data and creates output data based upon a model of a system.

[4]For example, Wolfram (1994) defines four classifications into which different Cellular Automata systems can be placed based on their *long-term* behaviours. The first one is *evolving to a homogeneous state*, which means that changes to the initial state have no impact on the final state.

conduct such an analysis, large amount of data should be collected (and sorted) and intelligent data analysis techniques must be performed.
- Stress testing of the overall system with a load that causes it to allocate its resources in maximum amounts. The objective of this test is to try to break the system by finding the circumstances under which it will crash (Burnstein, 2003).
- Testing the robustness to parameter alterations of the overall system, in order to fully trust the results of the simulation runs.

These tests may require many resources and long test times. Thus, they must be performed by a team of testers (or the entire development team).

In this section we have presented the testing requirements of agent-based simulation models at different levels of abstraction. It should be noted that each testing level (1) focuses on a specific level of abstraction of the agent-based model, (2) has a set of specific objectives, (3) is useful for revealing different types of defects, and (4) is useful for evaluating certain functional and quality attributes of the model.

3. The generic agent-based simulation testing framework

To be able to satisfy the aforementioned requirements and to perform testing effectively, developers/testers need an automated testing tool that supports model instrumentation. In other words, the tool should allow defining observation points for each testable element both individually and as a group. Morever, this tool has to support collecting information from these observation points while the model is executed. And apparently, it has to provide evaluation mechanisms for the assessment of the collected information.

3.1. The conceptual model

We designed a generic testing framework that provides special mechanisms for model testing of ABMS. As we mentioned before, testing requires the execution of the model under test. In this context, each specific model designed for testing is called a *Test Scenario*. A *Test Scenario* contains at least one *Model Element* under test (depending on the level and the need), one special agent to conduct the testing process (the *Tester Agent*), the other required *Model Elements*, the data sources these elements make use of and a special simulated environment (the *Test Environment*) that contains all these elements (see Figure 1). It can also include one or more fake elements (elements that behave like real elements) to facilitate the testing process. Each *Test Scenario* is defined for specific requirement(s) and includes the required test cases, activities, and their sequences and observation requirements. For executing *Test Scenario*s, we designed another concept called *Scenario Executer*. The *Scenario Executer* is able to execute each *Test Scenario* with different initial conditions for predefined durations.

The *Tester Agent* is responsible for instrumenting the testable elements, collecting information from them and evaluating these information in order to check if these testable elements behave as expected. For the evaluation of different conditions, the *Tester Agent* uses a set of *Assertions*. The *Tester Agent* is able to access every basic element during the execution of a *Test Scenario*. However, none of these basic elements are aware of it. Therefore it does not affect the way the other elements of the scenario behave. To be able to supply this feature, we designed a special *Simulated Environment* called *Test Environment*. All the *Model Elements* of the scenario, including the *Tester Agent*, are situated in this environment. However, apart from the *Tester Agent*, none of the other elements are aware of the *Test Environment*.

Another special mechanism introduced is the usage of special elements called *Fake Agent*s and *Fake Environment*s to facilitate the testing process. They are especially useful when a real element is impractical or impossible to incorporate into a scenario execution. They allow developers to discover whether the element(s) being tested respond(s) appropriately to the wide variety of states such element(s) may be in. For example, for a *micro-level* test aiming at testing the interaction protocol of a model element, there is no need to use the real implementation of the other model elements, since the aim is to focus on the interaction protocol. In this sense, *Fake Agent*s mimic the behaviour of real agents in controlled ways and they simply send pre-arranged messages and return pre-arranged responses. Likewise, *Fake Environment*s mimic the behaviour of real simulated environments in controlled ways and they are used for testing agents independently from their simulated environments. Although the term 'mock' can also be used in testing in MASs literature (Coelho *et al*, 2006), we preferred using the term 'fake' rather than 'mock' for describing the non-real elements, since there is also a distinction between 'fake' and 'mock' objects in object-oriented programing. Fakes are the simpler of the two, simply implementing the same interface as the objects that they represent and returning pre-arranged responses (Feathers, 2004). Thus a fake object merely provides a set of method stubs. Mocks, on the other hand, do a little more: their method implementations contain assertions of their own.

Furthermore, model elements may use *Data Generators* that generate data for the corresponding model element when needed. *Data generators* can be simple tools that output a fixed set of pre-defined patterns or they can be complex tools that use statistical patterns to generate data (Burnstein, 2003)[5].

The objective of this framework is to facilitate the model testing process. In model testing, as mentioned above, the inputs and the outputs of the systems are known. However,

[5]Burnstein (2003) refers *data generators* as *load generators* since *load generators* are aimed at being used in system-level tests. However, we use the term *data generator* since it can be used in all levels.

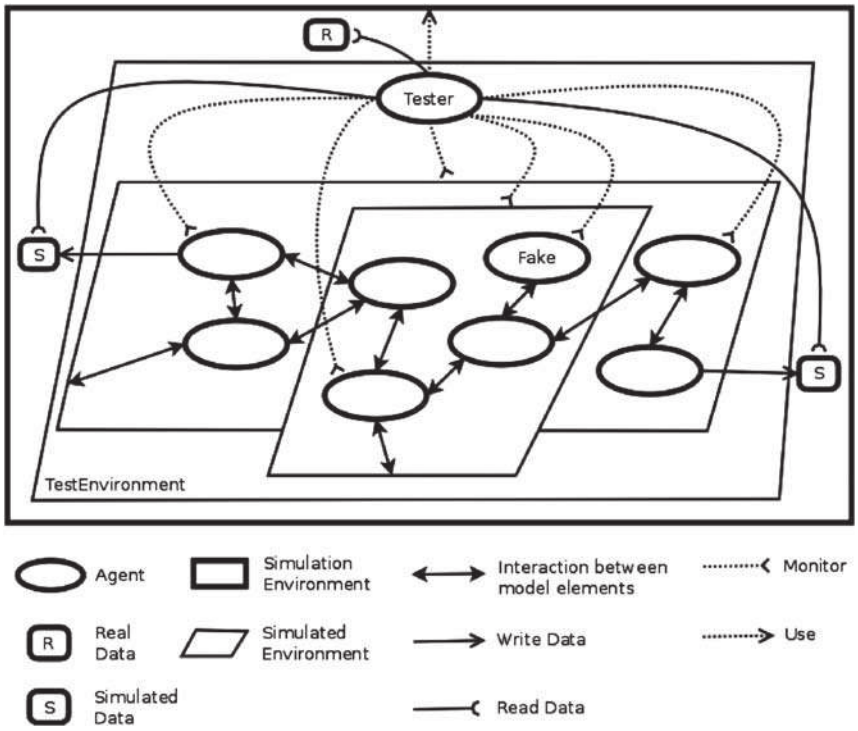

Figure 1 An illustrative example for a test scenario. As represented in the figure, the basic ingredients for test scenarios are: the tester agent, fake agents, the basic elements of agent-based simulation models (agents, simulated environment and simulation environments) and the data they use/produce. The *Tester Agent* is able to collect information from all these elements. A test scenario is executed by a scenario executor that is not shown in this figure.

it is not always practical to evaluate the output with computer programs. It can be time consuming as well as hard to implement. It is also a common practice to ask domain experts about the system whether the model and/or its behaviour are reasonable. This process is defined as 'face validity' by Sargent (2005). Face validity also includes validating graphically values of various performance measures as the model is running. Moreover, one may also want to test the display settings of the *Simulation Environment* visually (such as the size of the space, and whether the space wraps in either the horizontal or vertical dimension) (Railsback and Grimm, 2011). In this sense, we also included a visual testing mechanism in this generic testing framework. Basically, the *Tester Agent* is able to plot a visual output to the developer/tester and asks him/her to validate or invalidate this visual output.

The next subsection explains the architecture of our generic testing framework.

3.2. The architectural model

The architectural UML model[6] of the generic testing framework is given in Figure 2. When loaded, `ScenarioExecuter` first initializes the given test scenario by using the generic simulation runner interface (`SimulationRunner`) that builds the scenario by using a builder (`ScenarioBuilder`). `ScenarioExecuter` uses its `getScenarioDir()` method to retrieve the name of the directory in which the required files of the scenario are located. After, `ScenarioExecuter` executes the test scenario with different parameters by sweeping the provided file until the defined limit for the test scenario is reached. To do so, the `ScenarioExecuter` class provides an `executeTestScenario()` method that enables executing the same test scenario with different initial conditions for different pre-defined durations. The runner of the agent-based simulation framework is responsible for loading `ScenarioBuilder`, which builds the scenario by constructing the required model elements. It thus builds `TestEnvironment` and `TesterAgent` internally by using the `buildTestElements()` method. Other model test elements (the `SimulatedEnvironment` and the `Agent` elements)[7], on the other hand, are built externally by using the provided stub method `buildElements()`.

`TesterAgent` is able to access all basic elements in order to make model instrumentation. For accessing the simulated

[6]The Unified Modeling Language (UML) is a standard that can be used 'for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes'. Thus, the UML is a visual language that can be used to create software architectures.

[7]We do not address implementation issues on how to apply these concepts in practice, as this is highly dependent upon the simulation framework used and the objective of the simulation study.

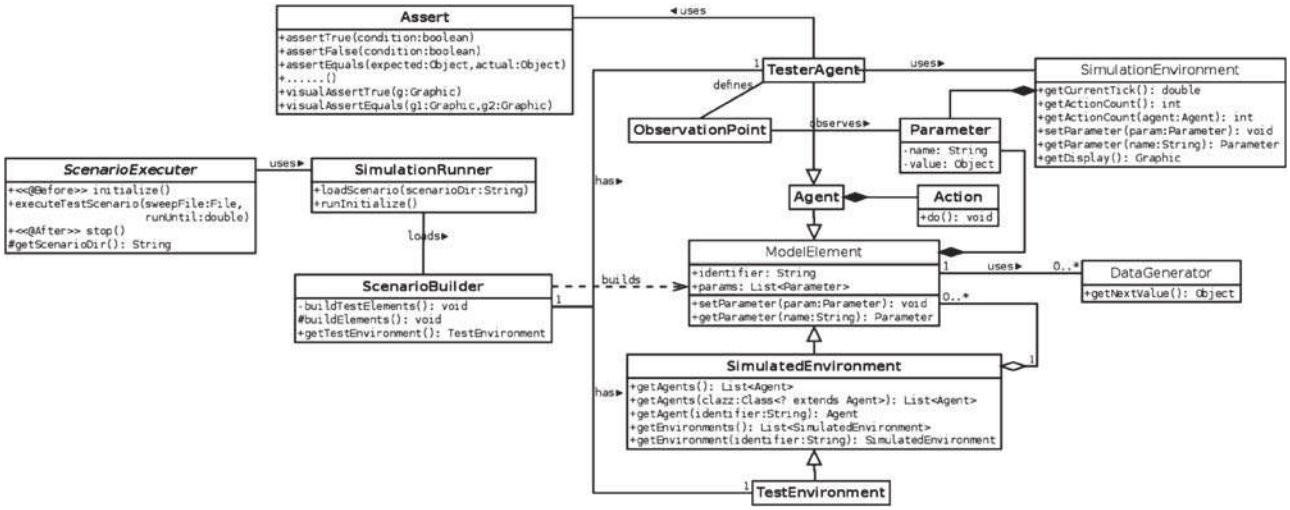

Figure 2 The architectural UML model for the generic testing framework.

environments and the agents, it uses `TestEnvironment` and for accessing the simulation infrastructure it uses a special interface (`SimulationEnvironment`) that provides utility methods to gather information about the ongoing scenario execution. For example, it can get the current value of the simulation clock (`getCurrentTick()`), get the number of actions scheduled at specific time points (`getActionCount()`)[8] and so on. `TesterAgent` is responsible for managing the testing process in a temporal manner. Basically, it monitors the agents and the simulated environments through the observation points (`ObservationPoint`) and performs assertions (using the methods provided by `Assert`) depending on the expected behaviour of the agent-based model under test. However, if the ABMS framework provides pre-defined features for defining observation points, it is not necessary to use the `ObservationPoint` concept in the concrete model testing framework. Since `TesterAgent` itself is also an agent, all these aforementioned mechanisms can be defined as agent actions (`Action`) that can be executed at specific time points during the testing process. It can monitor and keep track of the states of all the elements of the test scenario, or the messages exchanged between them during the scenario execution. As a result, `TesterAgent` is able to test the model at specific time points by using instant or collected data, and when there is a specific change in the model (when an event occurs). If all the assertions pass until the specified time limit for the test, the test is said to be *successful*, otherwise the test is said to be *failed*.

Fake agents can be defined by using the same interface (`Agent`) as the real agents they mimic, allowing a real agent to remain unaware of whether it is interacting with a real agent or a fake agent. Similarly, fake environments can also be defined by using the same interface (`SimulatedEnvironment`) as the real interfaces they mimic.

Data generators are defined by the `DataGenerator` interface. They are responsible for generating data that can be retrieved by using the `getNextValue()` method step by step.

All assertion methods are defined in `Assertion`, including visual ones. These methods basically check whether a given condition is true or not. The assertions can also be visual in order to conduct visual tests. In this case, they take `Graphic` parameters. A `Graphic` parameter can be generated by the developer/tester or it can be retrieved from `SimulationEnvironment` by using the `getDisplay()` method. This method returns the current diplay from `SimulationEnvironment`.

3.3. Implementation

The generic framework defines only the required generic elements. For implementation, some of these elements can be removed or combined, or some new elements can be added depending on the architectural design of the simulation environment. The generic framework does not affect the software architecture of the simulation environment, it is pluggable. Rather, it uses the constructs provided by the simulation environment to specialize itself for that framework.

The generic testing framework has been successfully implemented for Repast Symphony 2.0 Beta[9] (Figure 3) and MASON Version 15[10] (Figure 4).

Repast implementation. Repast is an agent-based simulation framework written in Java (North *et al*, 2006).

[8]Since many agent-based simulators use a global scheduler, such information can be retrieved from the scheduler of the simulation infrastructure.

[9]http://repast.sourceforge.net/, latest accessed 13 July 2012.
[10]http://cs.gmu.edu/eclab/projects/mason/, latest accessed 13 July 2012.

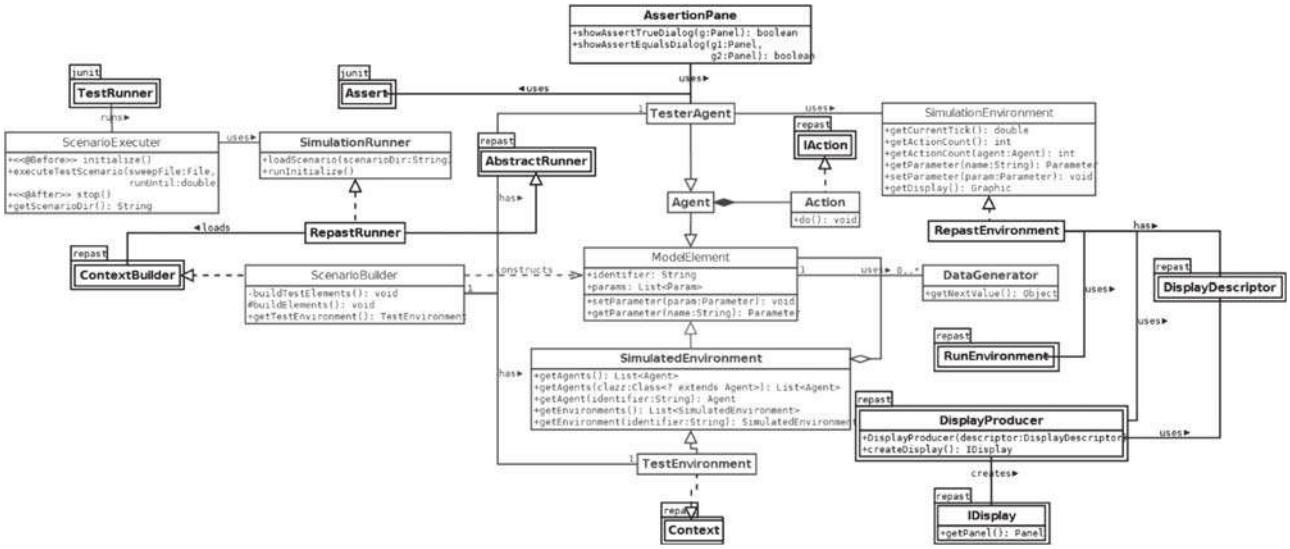

Figure 3 The UML class model for the repast implementation of the generic testing framework.

It provides pre-defined classes for building agent-based simulation models as well as for accessing the Repast simulation infrastructure during runtime. Since Repast is written in Java, the implementation of the framework is based on the Junit[11] testing framework, which is a simple framework to write repeatable tests for Java applications. Basically, the test runner of JUnit (`TestRunner`) runs test cases and prints a trace as the tests are executed followed by a summary at the end. Using the JUnit infrastructure, the scenario executer (`ScenarioExecuter`) is defined as a test case of JUnit. Consequently, by using the existing mechanisms and graphical user interfaces of JUnit, test scenarios of Repast can easily be executed. Then, a simulation runner (`RepastRunner`) is defined by extending the `AbstractRunner` class provided by Repast. Since Repast uses the `ContextBuilder` interface for building simulations, our ScenarioBuilder implements this interface. Then, a class for representing the Repast simulation infrastructure (`RepastEnvironment`) is defined. This class uses the methods provided by the `RunEnvironment` class of Repast for accessing the Repast simulation infrastructure as defined in the `SimulationEnvironment` interface, apart from the `getDisplay()` method. For getting the display from the Repast infrastructure, RepastEnvironment creates a `DisplayProducer` by using a `DisplayDescriptor`. Then by using this `DisplayProducer`, `RepastEnvironment` creates a display (`IDisplay`) and returns its panel (`getPanel()`).

`TestEnvironment` is made real by implementing the `Context` interface provided by Repast, since it is the core concept and object in Repast that provides a data structure to organize model elements. The `ObservationPoint` concept is removed here, since Repast provides a special mechanism called Watcher that can be used for model instrumentation. Basically, a Watcher allows an agent to be notified of a state change in another agent and it schedules an event to occur as a result. The watcher is set up using an annotation @Watch. Finally, the actions of agents are implemented as a subclass of `IAction` provided by Repast.

In order to write tests in Repast, the developer/tester first needs to extend `ScenarioBuilder` to define the elements of the test scenario and the initial parameters. Then TesterAgent needs to be designed together with its monitoring and testing actions for the testing process. For performing assertions, `TesterAgent` uses the Assert class provided by JUnit. However, the assertion methods provided by JUnit do not allow making visual assertions. To provide this ability, a dialogue window[12] (`AssertionPane`) that asks for validation of a given visual graphic (or a comparison of two visual graphics) is implemented (there are just two buttons: *validate* and *invalidate*). The methods of `AssertionPane` return true when the developer/tester presses the *validate* button and false when the he/she presses the *invalidate* button. Then this return value can be controlled by using the `assertTrue()` method provided by the Assert class. Finally, `ScenarioExecuter` should be extended for defining the different initial conditions and time limits for each scenario execution.

MASON implementation. MASON is an extensible multi-agent simulation toolkit in Java (Luke *et al*, 2005). It provides pre-defined classes for building agent-based simulation models as well as for accessing the MASON simulation infrastructure during runtime. Since MASON is

[11]JUnit, http://www.junit.org/

[12]A *Dialog* window (in Java) is an independent subwindow meant to carry temporary notice apart from the main Swing Application Window. Most *Dialogs* present an error message or warning to a user, but *Dialogs* can present images, directory trees, or just about anything compatible with the main Swing Application that manages them.

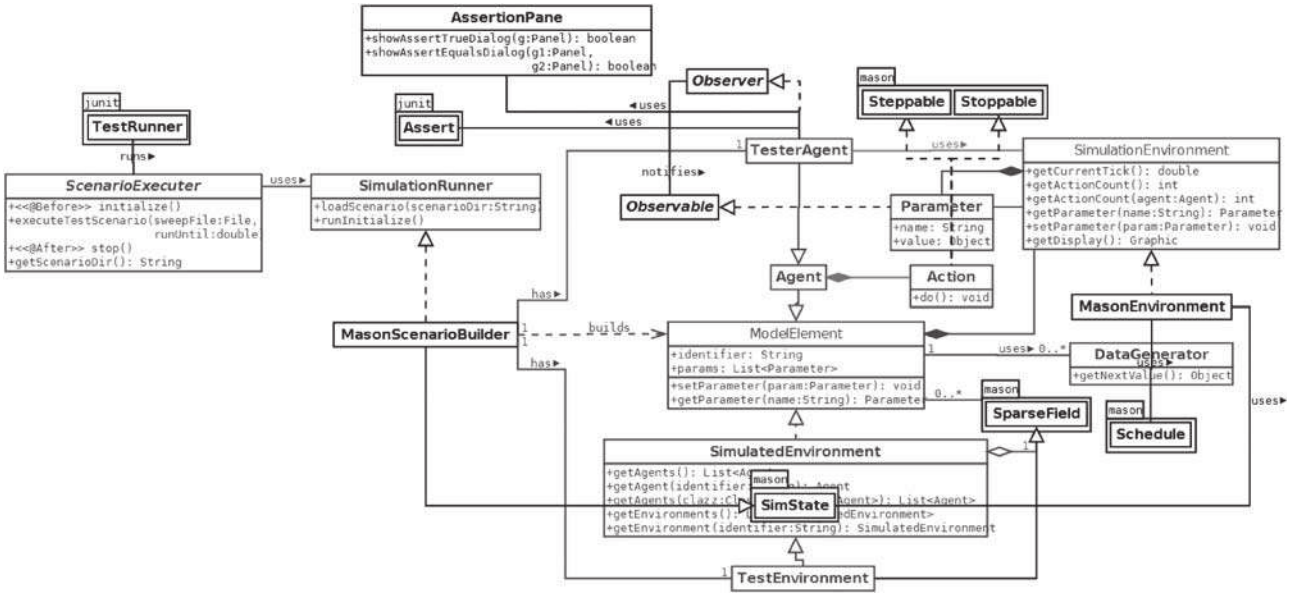

Figure 4 The UML class model for the MASON implementation of the generic testing framework. For model instrumentation, the Observer design pattern (Larman, 2004) is used. In this sense, all observable (testable) elements are registered to `TesterAgent` by `MasonScenarioBuilder`.

also written in Java, like Repast, its scenario executer is implemented in a similar manner. Afterwards, first, the `MasonScenarioBuilder` class is defined for running and building simulation models by extending the `SimState` class provided by MASON. Then, a class for representing the MASON simulation infrastructure (`MasonEnvironment`) is defined. This class uses the methods provided by the `SimState` and the `Schedule` classes of MASON for accessing the MASON simulation infrastructure as defined in the `SimulationEnvironment` interface. And after, `TestEnvironment` is realized by extending the `SparseField` class provided by MASON, since it is the core concept and object in MASON that provides a data structure to organize model elements. Finally, the actions of agents are defined by implementing `Steppable` and `Stoppable` provided by MASON.

Writing tests in MASON is quite similar to writing tests in Repast. The developer/tester first needs to extend `MASONScenarioBuilder` to define the elements of the test scenario and the initial parameters. During this definition process, the `TesterAgent` should be registered to the observable elements. Then the `TesterAgent` needs to be designed together with its monitoring and testing actions for the testing process. Finally, `ScenarioExecuter` should be extended for defining the different initial conditions and time limits for each scenario execution.

4. Case study: agent-based simulation of synaptic connectivity

To demonstrate the effectiveness of our testing framework, we show its applicability on a *micro-level*, a *meso-level* and a *macro-level* testing example. For the case study, we have chosen one of our ongoing agent-based simulation projects. In this project, we are developing a self-organized agent-based simulation model for exploration of synaptic connectivity of the human nervous system (Gürcan *et al*, 2010). All the tests of this project are conducted by the Repast implementation of the testing framework, and the initial results of this project have just been published (Gürcan *et al*, 2012).

In an organism, the nervous system is a network of specialized cells (including neurons) that communicate information about the organism and its surroundings. A *neuron* is an excitable cell in the nervous system that processes and transmits information by electrochemical signalling through links called axons. Neurons emit spikes when their *membrane potential* crosses a certain threshold (*firing threshold*). When this threshold is crossed, a spike is delivered to the other neurons through the axons of that neuron. The very end of an axon, which makes a junction to the other neuron, is called a synapse. When a spike transmitted by a neuron through one of its axons reaches a synapse, this latter transmits the spike to the other neuron (*post-synaptic neuron*) after a certain amount of time (depending on the length of the axon), which is called an *axonal delay*. After emitting the spike, the neuron membrane potential is reset to a certain lower value (*resting membrane potential*). According to their activation, neurons are of two types: (1) if a neuron is a resting one, it emits a spike when the total synaptic input is sufficient to exceed the firing threshold, or (2) if a neuron is a tonic firing one (eg, motoneurons, proprospinal neurons), it emits a spike when the membrane potential constantly rises to the firing threshold (Figure 5).

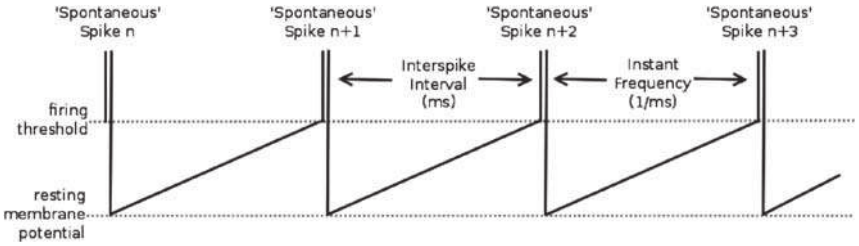

Figure 5 Tonic firing of a neuron. During tonic firing, a neuron's membrane potential continuously rises to the firing threshold and makes the neuron fire spontaneous spikes. The time interval between consecutive spikes are called inter-spike intervals (ISI).

To study synaptic connectivity in human subjects, it has been customary to use stimulus evoked changes in the activity of one or more motor units[13] in response to stimulation of a set of peripheral afferents or cortico-spinal fibers (reflex pathways[14]). These effects are often assessed by compiling a peristimulus frequency-gram (PSF) that plots the instantaneous discharge frequency values against the time of the stimulus (Türker and Powers, 2005). Figure 6 is an example of a PSF diagram for the human soleus muscle[15] single motor unit discharge frequencies. Here, the time of stimulus is represented as time 0 and the effect of the situmulus is apparently seen from the change of the frequency values after the stimulus.

The ability to record the motor activity in human subjects has provided a wealth of information about the neural control of motoneurons (Türker and Miles, 1991). Besides, the reflex pathways of motor units are less complex and involves less neurons compared to the cortical pathways in the brain. Thus, in our project we are focused on simulating the synaptic connectivity of reflex pathways. We developed and brought together the basic elements of our agent-based simulation model. To design the self-organized dynamics of the simulation model, the adaptive multi-agent systems (AMAS) theory (Capera *et al*, 2003) is used. According to the AMAS theory, agents constantly try to help to the most critical agent in order to be cooperative. Thanks to this cooperation ability, the agent-based model self-organizes in an acceptable neural pathway. An acceptable neural pathway is an artificial neural network, composed of realistic neuron agents, whose *macro-level* behaviour is very similar to the behaviour of the real reflex pathway. To verify and validate the model, various test scenarios for *micro-*, *meso-* and *macro-levels* are designed and implemented.

In order to demonstrate how the testing framework can be used, for each level one testing scenario is chosen.

[13]Motor units are composed of one or more motoneurons and all of the muscle fibres they innervate.

[14]A reflex pathway is a type of neural pathway involved in the mediation of a reflex. Reflexes are involuntary reactions that occur in response to stimuli. They often bypass the brain altogether, allowing them to occur very quickly, although the brain receives information about the reflex as it happens.

[15]The soleus is a powerful muscle in the back part of the lower leg (the calf). It runs from just below the knee to the heel, and is involved in standing and walking.

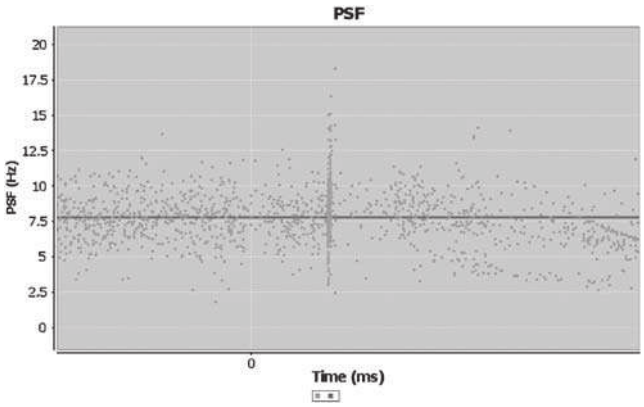

Figure 6 An example peristimulus frequencygram (PSF diagram) for the soleus muscle single motor unit pathway. The frequency values before the stimulation (time 0) show the normal behaviour of the motor unit. However, there are significant changes after the stimulation.

4.1. Micro-level testing example: tonic firing of a motoneuron

In this scenario, the aim is to test one of the *micro-level* behaviours of motoneurons: the constant emission of spikes (since motoneurons are tonically active). For tonic firing, Motoneuron agents use the experimental data recorded from single motor units of human subjects in Ege University labs.[16] Thus, the expected tonic firing behaviour of this agent is to generate spikes similar to the real motoneurons.

Figure 7 is an illustrative diagram for the selected test scenario. The basic element under test is the Motoneuron agent. In order to be able to test this *micro-level* behaviour, the Motoneuron agent is connected to a FakeNeuron agent with a synaptic link. The FakeNeuron agent imitates a resting neuron and it is just responsible for receiving the incoming spikes. The synaptic link is responsible for conducting a given spike to the FakeNeuron agent after a predefined axonal delay. During the scenario execution, the Motoneuron agent constantly emits spontaneous spikes and these spikes are be delivered to the FakeNeuron agent. Each

[16]Ege University Center for Brain Research, http://www.eubam.ege.edu.tr/.

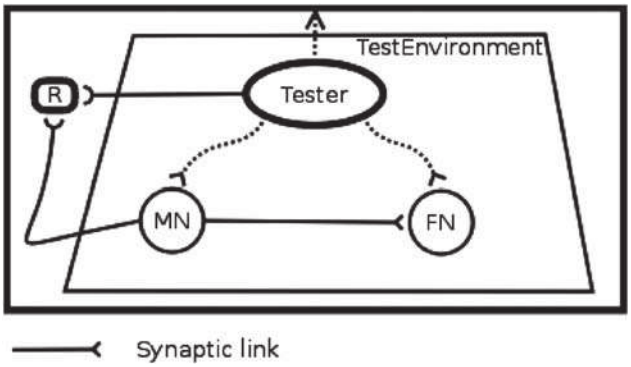

Figure 7 An illustrative diagram for the 'tonic firing of a motoneuron' micro-level testing scenario. In this scenario, there is a motoneuron (MN) that fires constantly by using the reference data R. MN is connected to a fake neuron (FN) through a synaptic link. During the test, the Tester agent both monitors the firing behaviours of MN and FN. Then using the monitored data, the Tester agent checks if the firing behaviour of MN is acceptable according to R.

time the FakeNeuron agent receives a spike, its membrane potential rises a little for a while and then goes back to the resting membrane potential. In order to test the tonic firing behaviour of the Motoneuron agent, the Tester agent observes the activity of both the Motoneuron agent and the FakeNeuron agent for the given amount of time (for each scenario execution this amount may differ). At the end of this time limit, the Tester agent conducts tests using the information it collected during the scenario execution.

For implementing this scenario, first a test builder (`TonicFiringScenario`) needs to be created by extending the `ScenarioBuilder` class (Algorithm 1). Within this class, the construction of the basic elements of the test scenario (the Motoneuron agent and the FakeNeuron agent) is done. Then, the Tester agent (`TonicFiringTester`) is implemented together with its behaviours by extending the `Tester Agent` class for the testing process (Algorithm 2 and Algorithm 3). As shown in Algorithm 2, `TonicFiring Tester` monitors the activities of the Motoneuron and the FakeNeuron agents by observing their membrane potentials. For defining the observation points, the watch mechanism (the @Watch annotation) provided by the Repast infrastructure is used (Algorithm 2, lines 17–21 and 27–31). The resting membrane potential is defined as –55mV and the firing threshold is defined as –45mV in our simulation model, based on many intracellular studies of tonically active motoneurons (eg Calvin and Schwindt, 1972; Schwindt and Crill, 1982). Thus, when the membrane potential of the Motoneuron agent becomes higher than –45mV, the Tester agent records the time of occurrence in a list to keep track of the activities of the Motoneuron agent during the simulation (`monitorMotoneuronActivity()`). Likewise, when the membrane potential of the FakeNeuron agent becomes higher than –55 mV, the Tester agent records the time of occurrence to keep track of the activity of FakeNeuron (`monitorFakeNeuronActivity()`).

Algorithm 1 Source code for the TonicFiringScenario class.

```
package test.microlevel.tonicfiring;                                    1
import motorunit.*;                                                      2
public class TonicFiringScenario                                        3
                                    extends ScenarioBuilder{             4
    private RunningNeuron motorNeuron;                                   5
    private FakeNeuron fakeNeuron;                                       6
    private TestEnvironment testEnv;                                     7
                                                                         8
    public static String EXP_MOTOR_ACTIVITY =                            9
                ".\data\real_motoneuron_activity.txt";                   10
    @Override                                                            11
    protected void buildElements() {                                     12
       testEnv = getTestEnvironment();                                   13
       motorNeuron = new RunningNeuron("Motoneuron",                     14
                                        EXP_MOTOR_ACTIVITY);             15
       testEnv.add(motorNeuron);                                         16
       fakeNeuron = new FakeNeuron("FakeNeuron");                        17
       testEnv.add(fakeNeuron)                                           18
       double axonalDelay = 10.0;                                        19
       motorNeuron.makeSynapseWith(fakeNeuron,                           20
                                              axonalDelay);              21
    }                                                                    22
}                                                                        23
```

Algorithm 2 Source code for the TonicFiringTester class. Summarized for a better representation of the model instrumentation.

```
package test.microlevel.tonicfiring;                                    1
...                                                                      2
import java.util.*;                                                      3
import repast.simphony.*;                                                4
...                                                                      5
public class TonicFiringTester extends TesterAgent {                    6
  ...                                                                    7
  private Vector<Double> mnSpikes;                                       8
  private Vector<Double> fnActivities;                                   9
                                                                         10
  public TonicFiringTester() {                                           11
    ...                                                                  12
    mnSpikes = new Vector<Double>();                                     13
    fnActivities = new Vector<Double>();                                 14
  }                                                                      15
                                                                         16
  @Watch(watcheeClassName="motorunit.RunningNeuron",                     17
  watcheeFieldNames = "membranePotential",                               18
  query = "colocated",                                                   19
  triggerCondition = "$watchee.getPotential()>-45",                      20
  whenToTrigger = WatcherTriggerSchedule.IMMEDIATE)                      21
  public void monitorMotorneuronActivity() {                             22
    double tick=RepastEnvironment.getCurrentTick();                      23
    mnSpikes.add(tick);                                                  24
  }                                                                      25
                                                                         26
  @Watch(watcheeClassName="motorunit.FakeNeuron",                        27
  watcheeFieldNames = "membranePotential",                               28
  query = "colocated",                                                   29
  triggerCondition = "$watchee.getPotential()>-55",                      30
  whenToTrigger = WatcherTriggerSchedule.IMMEDIATE)                      31
  public void monitorFakeNeuronActivity() {                              32
    double tick=RepastEnvironment.getCurrentTick();                      33
    fnActivities.add(tick);                                              34
  }                                                                      35
  ...                                                                    36
}                                                                        37
```

Algorithm 3 Source code for the TonicFiringTester class. Summarized for a better representation of the test cases.

```
1  package test.microlevel.tonicfiring;
2  import static org.junit.Assert.*
3  import java.util.*;
4  import repast.simphony.*;
5  import motorunit.*;
6  import umontreal.iro.lecuyer.randvar.
7                                  RandomVariateGen;
8  public class TonicFiringTester extends TesterAgent {
9    private RunningNeuron motorNeuron;
10   private FakeNeuron fakeNeuron;
11   ...
12   public TonicFiringTester() {
13     motorNeuron = (RunningNeuron)
14         getTestEnvironment().getAgent("Motorneuron");
15     fakeNeuron = (FakeNeuron)
16         getTestEnvironment().getAgent("FakeNeuron");
17     ...
18   }
19   ...
20   @ScheduledMethod(start = ScheduleParameters.END)
21   public void testTonicFiringOfMotorNeuron() {
22     // motorneuron has to generate some spikes.
23     assertTrue(motorNeuronSpikes.size() > 0);
24
25     Vector<Double> isiV = ISIDistribution.
26      getInstance().calculateISI(motorNeuronSpikes);
27     RandomVariateGen rvgSim =
28         RandomVariateGenFactory.getGenerator(isiV);
29     assertNotNull(rvgSim);
30     RandomVariateGen rvgExp =
31                           motorNeuron.getRandomGen();
32     assertNotNull(rvgExp);
33     // distribution should be the same
34     assertEquals(rvgExp.getClass().getName(),
35                         rvgSim.getClass().getName());
36
37     Distribution dExp = rvgExp.getDistribution();
38     Distribution dSim = rvgSim.getDistribution();
39     // alpha parameters should be close
40     double alphaExp = dExp.getParams()[0];
41     double alphaSim = dSim.getParams()[0];
42     assertEquals(alphaExp, alphaSim, 0.1);
43     // gamma parameters should be close
44     double gammaExp = dExp.getParams()[1];
45     double gammaSim = dSim.getParams()[1];
46     assertEquals(gammaExp, gammaSim, 0.1);
47   }
48   @ScheduledMethod(start = ScheduleParameters.END)
49   public void testConductionOfSpikes() {
50     for (int i = 0; i < fnActivities.size(); i++) {
51       double axonalDelay = fnActivities.get(i)
52         - mnSpikes.get(i);
53       assertEquals(10.0, axonalDelay, 0.0);
54     }
55   }
56 }
```

As shown in Algorithm 3, `TonicFiringTester` executes two actions for testing the *micro-level* behaviour of the Motoneuron agent at the end of each scenario execution (`ScheduleParameters.END`)[17]. For defining the test cases, the schedule mechanism (the @ScheduleMethod annotation) provided by the Repast infrastructure is used (Algorithm 3, lines 20 and 48). One of the test cases (`testTonicFiringOfMotorNeuron()`) checks whether the generated spikes of the Motoneuron agent have similar characteristics with the real data or not. In order to be able to test the tonic firing of the Motoneuron agent, this test case first tests if the Motoneuron agent generated some spikes (Algorithm 3, line 23). And after, it tests if the simulated data generated by the Motoneuron agent have similar statistical characteristics: they should represent the same statistical distribution whose alpha and gamma parameters are nearly the same. To do so, first the class names of both distributions are compared (Algorithm 3, lines 34 and 35), then the *alpha* and *gamma* parameters of both distributions are, respectively, compared (Algorithm 3, lines 42 and 46). The second test case (`testConductionOfSpikes()`) is designed to test if the spikes generated by the Motoneuron agent are properly received by the FakeNeuron agent. To do so, this test examines if all the delays between the consecutive activities of the Motoneuron agent and the FakeNeuron agent are exactly the same and are equal to 10.0 (Algorithm 3, line 53).

[17]The time for the end of the scenario execution may change at each execution, according to the values given by the developer for ScenarioExecuter. See Algorithm 8.

Algorithm 4 Source code for the TonicFiringExecuter class

```
1  package test.microlevel.tonicfiring;
2  import org.junit.Test;
3  public class TonicFiringExecuter
4                              extends ScenarioExecuter {
5    @Test
6    public void tonicFiringTestScenario()
7                                        throws Exception {
8      executeTestScenario(null, 2000001);
9      executeTestScenario(null, 4000001);
10   }
11 }
```

Finally, in order to execute the test scenario (with various criteria), the basic class that the JUnit runner will use (`TonicFiringExecuter`) is implemented by extending the `ScenarioExecuter` class (Algorithm 4). In this class, to execute this test scenario with different time limits, the `executeTestScenario()` method is called twice with different `runUntil` parameters (Algorithm 4, lines 8 and 9).

4.2. Meso-level testing example: creation of a new synapse for helping an inactive neuron

In this scenario, one *meso-level* behaviour of self-organizing neurons is considered: the creation of a new synapse for helping an inactive neuron. The test scenario is composed of a group of cooperative neuron agents. The aim of the test is to evaluate if these agents behave cooperatively for helping each other. However, before delving into this scenario, it should be noted that the *micro-level* tests concerning basic agent interactions have already been performed and passed.

The initial setting of the test scenario is shown in Figure 8a. There are three neuron agents: Neuron-1, Neuron-2 and Neuron-3. Neuron-1 has a synapse with Neuron-2 and Neuron-2 has a synapse with Neuron-3. Although the synapse between Neuron-1 and Neuron-2 is

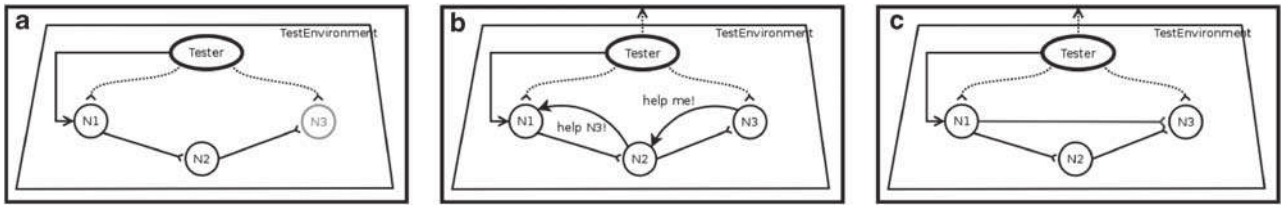

Figure 8 An illustrative diagram that shows the creation of a new synapse by a second level neighbour for helping an inactive neuron. Initially three neuron agents are considered: Neuron-1 (N1), Neuron-2 (N2) and Neuron-3 (N3). Although the synapse between Neuron-1 and Neuron-2 is strong enough to activate (fire) Neuron-2, the synapse between Neuron-2 and Neuron-3 is not strong enough to activate Neuron-3. In this sense, the expected behaviour of this sub-society is to create a synapse between Neuron-1 and Neuron-3.

Algorithm 5 Source code for the NewSynapseCreationSce-nario class.

```
1  package test.mesolevel.newsynapsecreation;
2  import motorunit.*;
3  public class NewSynapseCreationScenario
4                              extends ScenarioBuilder{
5      private Neuron n1, n2, n3;
6      private TestEnvironment testEnv;
7
8      @Override
9      protected void buildElements() {
10         testEnv = getTestEnvironment();
11         n1 = new Neuron("Neuron-1");
12         testEnv.add(n1);
13         n2 = new Neuron("Neuron-2");
14         testEnv.add(n2);
15         n3 = new Neuron("Neuron-3");
16         testEnv.add(n3);
17         double axonalDelay = 10.0;
18         n1.makeSynapseWith(n2, axonalDelay);
19         n2.makeSynapseWith(n3, axonalDelay);
20     }
21 }
```

strong enough to activate (fire) Neuron-2, the other synapse is not strong enough to activate Neuron-3.

The developer/tester designed this scenario to verify that the expected behaviour of this sub-society is to create a synapse between Neuron-1 and Neuron-3. When this scenario is executed, Neuron-3 begins to continuously ask for some help from its direct neighbour (Neuron-2). However, since Neuron-2 has already a synaptic link with Neuron-3 and is unable to create another synapse for helping Neuron-3, Neuron-2 forwards the help call to the most reasonable neighbour neuron agent, which is from its point of view, Neuron-1 (Figure 8b). If Neuron-1 receives too many help calls from Neuron-3, in order to help this latter, it creates a synaptic link between them as shown in Figure 8c.

For implementing this scenario, first a test builder (`NewSynapseCreationScenario`) needs to be created by extending the `ScenarioBuilder` class (Algorithm 5). Within this class, the construction of the basic elements of the test scenario (neuron agents) is done. Then, the Tester agent (`NewSynapseCreationTester`) is implemented together with its behaviours by extending the `TesterAgent` class for the testing process (Algorithm 6).

`NewSynapseCreationTester` first examines the initial organization of the scenario (`testInitialOrganization()`). There must be three neurons with the following organizational characteristics: Neuron-1, which has no pre-synapse and one post-synapse, Neuron-2, which has one pre-synapse and one post-synapse, and Neuron-3, which has one pre-synapse and no post-synapse. After a pre-defined amount of time, given by the developer/tester, `NewSynapseCreation Tester` tests whether there is a new synapse in the organization or not (`testCreationOfANewSynapse()`). To do so, it checks if Neuron-1 has one more post-synapse and if Neuron-3 has one more pre-synapse.

Finally, in order to execute the test scenario the executor class that the JUnit runner will use (`NewSynapseCreation Executer`) is implemented by extending the `Scenario Executer` class (Algorithm 7).

4.3. Macro-level testing example: self-organization of reflex pathways

After the *micro-* and the *meso-level* tests are performed for our project, we developed the *macro-level* testing scenarios. One of these scenarios is about producing the known output data for a given legal input data. In this scenario, the aim is to test, given experimental data for a reflex pathway (legal input data), whether the developed agent-based model self-organizes in an acceptable neural network and produces legal output data. In this sense, to be able to test this *macro-level* behaviour, the significance of the simulated data with respect to the reference data is evaluated by cross-correlation analysis.

The initial setting of the test scenario is shown in Figure 9. Initially there are one WiringViewer agent, two Neuron agents and one Muscle agent. The AfferentNeuron agent has a synapse with the Motoneuron agent, and the Motoneuron agent has a synapse with the Muscle agent. The Motoneuron agent is a tonic firing neuron agent that represents a single motor unit.

When this scenario is executed, the Motoneuron agent begins to continuously generate spikes. Meanwhile, the

Algorithm 6 Source code for the NewSynapseCreationTester class.

```
package test.mesolevel.newsynapsecreation;
...
import java.util.*;
import repast.simphony.*;
...
public class NewSynapseCreationTester
                                    extends TesterAgent {
  private Neuron n1, n2, n3;
  private TestEnvironment testEnv;

  public NewSynapseCreationTester() {
    this.testEnv = getTestEnvironment();
    this.n1 = (Neuron) testEnv.getAgent("Neuron-1");
    this.n2 = (Neuron) testEnv.getAgent("Neuron-2");
    this.n3 = (Neuron) testEnv.getAgent("Neuron-3");
  }

  @ScheduledMethod(start = 1,
       priority = ScheduleParameters.LAST_PRIORITY)
  public void testInitialOrganization() {
    List<Agent> agents =
                    testEnv.getAgents(Neuron.class);
    assertEquals(3, agents.size());
    // Neuron-1
    assertEquals(0, n1.getPreSynapses().size());
    assertEquals(1, n1.getPostSynapses().size());
    // Neuron-2
    assertEquals(1, n2.getPreSynapses().size());
    assertEquals(1, n2.getPostSynapses().size());
    // Neuron-3
    assertEquals(1, n3.getPreSynapses().size());
    assertEquals(0, n3.getPostSynapses().size());
  }

  @ScheduledMethod(start = ScheduleParameters.END,
       priority = ScheduleParameters.LAST_PRIORITY)
  public void testCreationOfANewSynapse() {
    List<Agent> agents =
                    testEnv.getAgents(Neuron.class);
    assertEquals(3, agents.size());
    // Neuron-1
    assertEquals(0, n1.getPreSynapses().size());
    assertEquals(2, n1.getPostSynapses().size());
    // Neuron-2
    assertEquals(1, n2.getPreSynapses().size());
    assertEquals(1, n2.getPostSynapses().size());
    // Neuron-3
    assertEquals(2, n3.getPreSynapses().size());
    assertEquals(0, n3.getPostSynapses().size());
  }
}
```

WiringViewer agent periodically innervates the Afferent-Neuron agent and monitors the discharges of the Motoneuron agent through the Musle agent (just like in the biological experimental setting). When the AfferentNeuron agent is innervated, it generates and sends spikes to the Motoneuron agent. These spikes evoke changes on the behaviour of the Motoneuron agent. According to the monitored discharges, the WiringViewer agent calculates an instant frequency and compares it to the reference instant frequencies (Figure 6) observed at the same point of time. Then the WiringViewer agent sends feedbacks to the Motoneuron agent about its firing behaviour. The Motoneuron agent evaluates these feedbacks and propagates them to other neurons if it cannot tackle the problem itself. As a result of this mechanism, the system self-organizes in a network (by creating new neurons and synapses) until the

Algorithm 7 Source code for the NewSynapseExecuter class.

```
package test.mesolevel.newsynapsecreation;
import org.junit.Test;
public class NewSynapseCreationExecuter
                              extends ScenarioExecuter {
  @Test
  public void newSynapseCreationTestScenario()
                                        throws Exception {
    executeTestScenario(null, 2000001);
  }
}
```

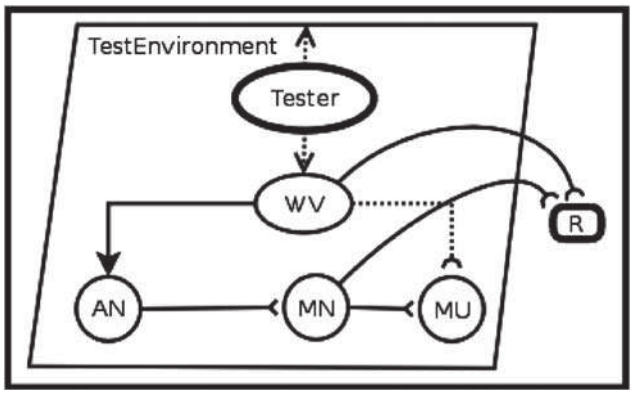

Figure 9 An illustrative diagram for the 'self-organization of reflex pathways' *macro-level* testing scenario. Initially there are four agents: an Afferent-Neuron agent (AN), a Motoneuron agent (MN), a Muscle agent (MU) and a WiringViewer agent (WV). The Motoneuron agent uses the reference experimental data (R) to calibrate its tonic firing behaviour. The WiringViewer uses the same data to compare the simulated behaviour of the system to the real system.

desired motoneuron discharging behaviour (the *macro-level* behaviour) is achieved. At the end of the scenario execution, the Tester agent gathers the stimulation data and the motoneuron discharges data from the WiringViewer agent and conducts tests using these data.

For implementing this scenario, first a test builder (`SoleusPathwayScenario`) needs to be created by extending the `ScenarioBuilder` class (Algorithm 8). Within this class, the construction of the basic elements of the test scenario is done. Then, the Tester agent (`SoleusPathwayTester`) is implemented together with its behaviours by extending the `TesterAgent` class for the testing process (Algorithm 9).

After a pre-defined amount of time, the simulation ends. Then `SoleusPathwayTester` analyses the results in order to ensure that the generated network is realistic. To calculate the relationship between the real behaviour and the simulated behaviour of the system, the Tester agent gets both reference and simulated data from the WiringViewer agent, performs a Pearson-correlation analysis between the experimental reference PSF-CUSUM and the simulated

Algorithm 8 Source code for the SoleusPathwayScenario class.

```
package test.macrolevel.solues;
import motorunit.*;
public class SoleusPathwayScenario
                              extends ScenarioBuilder{
    private WiringViewer wiringViewer;
    private Neuron afferent;
    private RunningNeuron motorNeuron;
    private Muscle muscle;
    private TestEnvironment testEnv;
    public static String EXP_SOLEUS_TRIGGER =
              "./data/real_soleus_trigger.txt";
    public static String EXP_SOLEUS_MNDISCHARGE =
              "./data/real_soleus_msdischarge.txt";

    @Override
    protected void buildElements() {
        afferent =
              new Neuron(null, "AfferentNeuron", true);
        testEnv.add(afferent);
        DataGenerator isiGen =
                    new InterspikeIntervalGenerator(
                                    EXP_SOLEUS_TRIGGER,
                                  EXP_SOLEUS_MNDISCHARGE);
        motorNeuron = new RunningNeuron(
                    null, "MotorNeuron", isiGenerator);
        testEnv.add(motoNeuron);
        muscle = new Muscle();
        testEnv.add(muscle);
        afferent.makeSynapseWith(motorNeuron);
        motorNeuron.makeSynapseWith(muscle);
        wiringViewer = new WiringViewer(afferent,
                                motorNeuron,
                                EXP_SOLEUS_TRIGGER,
                                EXP_SOLEUS_MNDISCHARGE);
        testEnv.add(wiringViewer);
    }
}
```

Algorithm 9 Source code for the SoleusPathwayTester class.

```
package test.mesolevel.soleus;
...
import java.util.*;
import repast.simphony.*;
...
public class SoleusPathwayTester
                                  extends TesterAgent {
  private WiringViewer Wv;
  private TestEnvironment testEnv;
  private List<Double> refStimuli, refDischarge;
  private List<Double> simStimuli, simDischarge;

  public NewSynapseCreationTester() {
    testEnv = getTestEnvironment();
    wiringViewer = (WiringViewer)
                    testEnv.getAgent("WiringViewer");
  }


 @ScheduledMethod(start = ScheduleParameters.END,
       priority = ScheduleParameters.LAST_PRIORITY)
 public void testCreationOfANewSynapse() {
   refStimuli = WV.getReferenceStimuli();
   refDischarge = WV.getReferenceDischarges();
   CUSUM refCUSUM =
                new CUSUM(refStimuli, refDischarge);

   double until = 2000000;
   simStimuli = WV.getSimulatedStimuli(until);
   simDischarge = WV.getSimulatedDischarges(until);
   CUSUM simCUSUM =
                new CUSUM(simStimuli, simDischarge);

   double correlation = 0.0;
   correlation = Correlation.getPearsonCorrelation(
                    refCUSUM, simCUSUM, 0.0, 200.0);

   assertTrue(correlation >= 0.90);
 }
}
```

PSF-CUSUM at time 0.0 and at time 200.0 (Figure 10). This analysis shows the degree of functional equivalence between the simulated network and the experimental reference network. The result of this function may vary from –1.0 to 1.0. These time values are chosen because reflex pathways are relatively short and the effects of stimulation after 200.0ms are said to be effects coming from cortical pathways. Therefore, the times considered are between 0ms (stimulation) and 200.0ms (last plausible effect). According to this test, the acceptable correlation is set to 0.90 by the human tester, so the Tester agent checks whether the correlation is greater than 0.90 or not.

Finally, in order to execute the test scenario the base class that the JUnit runner will use (`SoleusPathwayExecuter`) is implemented by extending the `ScenarioExecuter` class (Algorithm 10).

5. Related work

Although there is a considerable amount of work about testing in MASs in the literature (for a review see Nguyen *et al*, 2011)[18], there is not much work on model testing in ABMS.

Here we need to make a distinction between ABMS and MAS in terms of testing. In ABMS, since it is simulation, there is always a global simulation clock that makes monitoring and controlling the actions and the events of the model elements easier. However in MAS, there is no global clock and all the elements execute in real-time. The local time recorded in each element is potentially different. Therefore a testing framework for MAS needs additional mechanisms to be able to properly sort the different events independently from the computer local clock. A way of achieving this independence is, for example, by using logical clocks (Lamport, 1978); this was used in Serrano and Botia (2009).

In terms of ABMS, there is more work on validation. Some of these works are related to the definition of conceptual processes for ABMS validation. Klügl (2008), for instance, proposes a validation process for agent-based simulation models combining face validation, sensitivity analysis, calibration and statistical validation. She then defines the main problem in validation as the missing availability of empirical data. Cooley and Solano (2011) describe the use of

[18]In this article, testing methods and techniques with respect to the MAS properties they are able to address are extensively reviewed. But none of the reviewed studies focus on ABMS.

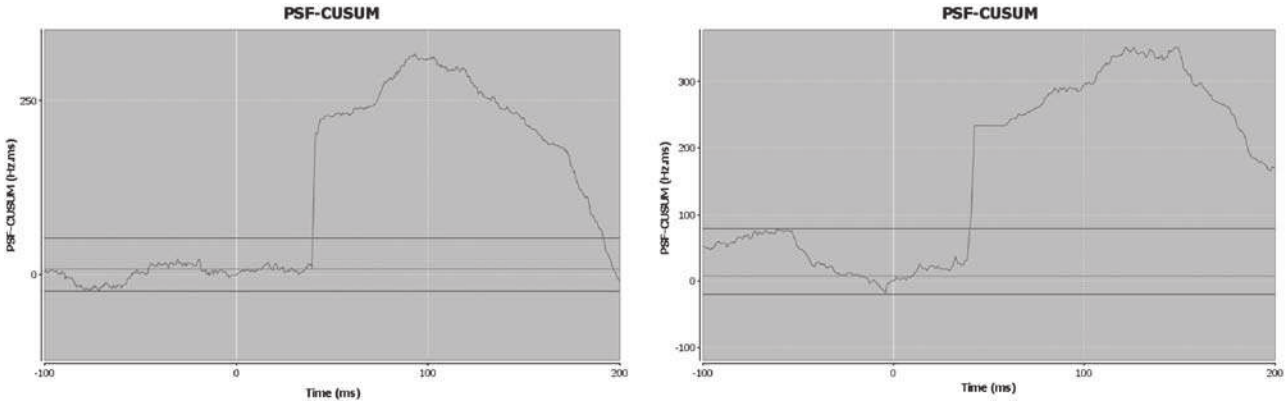

Figure 10 PSF-CUSUM diagrams for both reference and simulated data. The PSF-CUSUM of reference data (on the top) is calculated by using the soleus muscle single motor unit PSF values given in Figure 6. The PSF-CUSUM of simulated data (in the bottom) is calculated by using the motoneuron agent's latest discharge.

Algorithm 10 Source code for the SoleusPathwayExecuter class.

```
1  package test.mesolevel.soleuspathway;
2  import org.junit.Test;
3  public class SoleusPathwayExecuter
4                              extends ScenarioExecuter {
5      @Test
6      public void soleusPathwayTestScenario()
7                                          throws Exception {
8          executeTestScenario(null, 2000001);
9      }
10 }
```

validation methods in model building. They discuss the stages of simulating an agent-based simulation model and present six specific validation approaches. However, they do not define the connection between the basic elements of agent-based simulation models and their approach. These works define validation techniques and their usage within a validation process for ABMS. In other words, they are more focused on 'how' agent-based simulation models could be validated, rather than what should be validated. On the contrary, our approach focuses on 'what' should be validated in ABMS by clearly defining testing (verification and/or validation) levels and requirements. Moreover, we propose a generic framework for the automated execution of these requirements defined at each level. As a result, the proposed framework can also be used in any validation process.

Railsback and Grimm have a recent study (Railsback and Grimm, 2011) about testing agent-based simulation models. In this study, they define 10 important techniques for finding and fixing software errors: syntax checking, visual testing, print statements, spot tests with 'agent monitors', stress tests, test procedures, test programmes, code analysis, statistical analysis of file output and independent reimplementation of submodels. However, the approaches they propose are far from automating the model testing process, since they do not have an architectural perspective about how these solutions could be integratively constructed and conducted. Moreover, some of their solutions are not generic and depend on the NetLogo simulation framework (Sklar, 2007). They do not give any idea about how these solutions can be applied to different simulation frameworks in a generic manner. Apart from that, they are also aware of the deficiency of conducting tests on the *micro-* and the *macro-levels* (they call these levels the *individual-* and the *system-level,* respectively) and they claim that tests on the *submodel-level* should also be performed in order to control uncertainty. They define a submodel as an independent model that has its own distinct inputs and outputs. As a result of this definition, their *submodel-level* tests are black-box tests. However, to better control uncertainty, one also needs to make white-box tests in this level[19].

Some other works focus on data validation in agent-based simulation models. As defined by Robinson (1997), data validation is needed to determine whether the data used are sufficently accurate. Windrum *et al* (2007) examine a set of methodological problems in the emprical validation of agent-based economics models. Pengfei *et al* (2011) propose validation of agent-based simulation through human computation as a means of collecting large amounts of contextual behavioural data. Data validation is a critical subprocess for ABMS validation. In our case, data validation is a process that should be conducted before model testing of ABMS.

[19]The *meso-level* testing example given in this paper (Section 4.2) is a good example of white-box testing. In this test, the behaviour of a sub-society in order to handle a problem in certain conditions is tested by controlling its internal organization. The same sub-society, in another condition, may behave differently to handle the same problem. For example, rather than creating a new synapse, the sub-society may create a new neuron agent. This kind of solution may also satisfy this sub-society and make it produce the same output. However, the internal organization will be totally different. Therefore, only testing inputs and outputs in order to control uncertainty is not enough.

In the literature only a few works focus on designing and implementing tools for model testing of ABMS. Niazi *et al* (2009) present a technique that allows for flexible validation of agent-based simulation models. They use an overlay on the top of agent-based simulation models that contains various types of agents that report the generation of any extraordinary values or violations of invariants and/or reports the activities of agents during simulation. Our approach is similar to theirs in the sense that they use special agents where the agents undergo tests they are not aware of. However, instead of using various types of special agents, we only use one. Actually, we use a single agent for testing, since at every test our aim is to test one single use case of the system (Beck, 2003). Besides, they define an architecture but since they begin without defining the requirements it is not quite possible to understand 'what' they are testing. Montanola-Sales *et al* (2011) present the verification and validation of an agent-based demographic simulation model implemented using a parallel demographic simulation tool (Yades) using white-box validation methods described by Pidd (2004). In this sense, Montanola-Sales *et al* divide their model into smaller components and test the correctness of each component.

None of the tools and practical works in the literature are well-structured (their authors do not give internal details) and they are not pluggable to any existing ABMS framework. Unlike our framework, they are more suitable for manual testing, rather than automated testing. These works also do not take into account all basic elements of agent-based simulation models (agents, simulated environments and the simulation environment). We actually think that all these elements need to be involved in the model testing process since they are main ingredients of agent-based simulation models.

6. Conclusions

This body of work presents the design of a novel generic framework for the automated model testing of agent-based simulation models. The basic elements for testing are identified as agents and simulated environments. For testing each use case for these elements, a test scenario needs to be designed. In our active testing approach, for each test scenario, there is a special agent that observes the model elements under test, and executes tests that check whether these elements perform the desired behaviours or not. The framework also provides generic interfaces for accessing both the simulation environment and the simulated environments. Moreover, the framework allows for visual test in order to increase the confidence.

To demonstrate the applicability of the framework, it was implemented for two well-known agent-based simulation frameworks written in Java: Repast and MASON. For model instrumentation in Repast, the @Watch annotation provided by Repast is used. However, for model instrumentation in MASON, since it does not provide any mechanism facilitating the definition of observation points, the Observer design pattern (Larman, 2004) is used. Besides, to show the suitability of the proposed generic framework in case of the adoption of frameworks written in other languages, implementation for other frameworks is being planned.

Moreover, we integrated these implementations into JUnit, which is a framework to write repeatable tests for Java applications. Such an integration facilitates the automated execution of test scenarios for ABMS. As a result, performing regression tests for an agent-based simulation model becomes feasible. Apart from that, such an automated execution also provides a tool that enables *empirical calibration*. There are three well-known empirical validation approaches (Windrum *et al*, 2007): the indirect calibration approach, the Werker-Brenner approach and the history-friendly approach. The indirect calibration approach, as its name suggests, first performs validation, and then indirectly calibrates the model by focusing on the parameters that are consistent with output validation (Dosi *et al*, 2006). Such a process can be easily performed once the output validation tests for the model have been implemented by re-executing all the test scenarios after each parameter modification. The Werker-Brenner approach (Werker and Brenner, 2004) and the history-friendly approach (Malerba *et al*, 1999), on the other hand, first focus on calibration and then perform validation. These processes can also be performed by using the proposed framework but they cannot benefit from the regression test capability of the tool.

7. Future work

As a future work, we are planning to focus on testing of more complex simulated environment(s). In the current version, the `SimulatedEnvironment` interface worked well for our case study and other experimental systems. But, testing of more complex simulated environments may require more work to face particular modelling and implementation situations such as management of time and space in the simulation, considering, for instance, the complexity of a robotic simulated environment as described in Helleboogh *et al* (2007). In this sense, we first want to adapt the proposed generic framework to the GAMA platform (Taillandier *et al*, 2012), which has ability to use complex GIS data as an environment for the agents. Then, we plan to conduct more complex case studies to improve our environment interface.

Developing an agent-based simulation requires a closer working relationship between domain experts and developers, just like in *agile* development (Polack *et al*, 2010). Thus, testing is meaningful when it is involved in an agile development methodology. In this sense, we are planning to

define a test-driven process based on the testable elements and the generic framework defined in this paper. Moreover, we are also planning to show how our generic testing tool can be used for testing self-organizing MASs. The metrics for self-organization and emergence mechanisms for achieving self-*properties are given in recent works (Kaddoum *et al*, 2009 and Raibulet and Masciadri, 2009). We believe that the capabilities of our framework will be able to test and validate all the metrics given in these studies.

Acknowledgements—The authors thank Kemal S. Türker and S. Utku Yavuz from Ege University Center for Brain Research for supplying scientific data about the activity of motoneurons. The work described here was partially supported by Ege University under the BAP 10-MUH-004 project. Önder Gürcan is supported by the Turkish Scientific and Technical Research Council (TUBITAK) through a domestic PhD scholarship program (BAYG-2211) and by the French Government through the co-tutelle scholarship program.

References

Balci O (1994). Validation, verification, and testing techniques throughout the life cycle of a simulation study. In: *Proceedings of the 26th Conference on Winter simulation*, WSC'94, Society for Computer Simulation International: San Diego, CA, pp 215–220.

Balci O (1995). Principles and techniques of simulation validation, verification, and testing. In: *Proceedings of the 27th Conference on Winter simulation*, WSC' 95, IEEE Comp. Soc.: Arlington, VA, pp 147–154.

Bankes SC (2002). Agent-based modeling: A revolution? *Proceedings of the National Academy of Sciences of the United States of America* 99(3): 7199–7200.

Beck K (2003). *Test-driven Development: By Example*. Addison-Wesley: Boston.

Burnstein I (2003). *Practical Software Testing*. Springer: New York.

Calvin W and Schwindt P (1972). Steps in production of motoneuron spikes during rhytmic firing. *Journal of Neurophysiology* 35(3): 297–310.

Capera D, Georgé J, Gleizes M and Glize P (2003). The AMAS theory for complex problem solving based on self-organizing cooperative agents. In: *WETICE '03: Proceedings of the Twelfth International Workshop on Enabling Technologies*, IEEE Computer Society: Washington, DC, p 383.

Coelho R, Kulesza U, von Staa A and Lucena C (2006). Unit testing in multi-agent systems using mock agents and aspects. In: *Proceedings of the 2006 Int. Workshop on Software eng. for large-scale multi-agent systems*, SELMAS'06, ACM: New York, NY, pp 83–90.

Cooley P and Solano E (2011). Agent-based model (ABM) validation considerations. In: *Proceedings of the Third International Conference on Advances in System Simulation (SIMUL 2011)*, IARIA, Barcelona, Spain, pp 134–139.

Dosi G, Fagiolo G and Roventini A (2006). An evolutionary model of endogenous business cycles. *Computational Economics* 27(1): 3–34.

Epstein JM (2007). Agent-based computational models and generative social science. In: *Generative Social Science Studies in Agent-based Computational Modeling*, Introductory Chapters. Princeton University Press, Santa Fe, NM.

Feathers M (2004). *Working Effectively with Legacy Code*. Prentice Hall PTR: Upper Saddle River, NJ.

Ghosh S (1986). On the concept of dynamic multi-level simulation. In: *Proceedings of the 19th annual symposium on Simulation*, ANSS '86, IEEE Computer Society Press: Los Alamitos, CA, pp 201–205.

Grimm V *et al* (2005). Pattern-oriented modeling of agent-based complex systems: Lessons from ecology. *Science* 310(5750): 987–991.

Gürcan Ö *et al* (2012). Simulating human single motor units using self-organizing agents. In: *Sixth International IEEE Conference on Self-Adaptive and Self-Organizing Systems (SASO'2012)*, IEEE Computer Society: Lyon, France, pp 11–20.

Gürcan Ö, Dikenelli O and Türker KS (2010). Agent-based exploration of wiring of biological neural networks: Position paper. In: Trumph R (ed), *20th European Meeting on Cybernetics and Systems Research (EMCSR 2010*. Austrian Society for Cybernetic Studies: Vienna, Austria, EU, pp 509–514.

Helleboogh A, Vizzari G, Uhrmacher A and Michel F (2007). Modeling dynamic environments in multi-agent simulation. *Autonomous Agents and Multi-agent Systems* 14(1): 87–116.

House R, Rousseau DM and Thomas-Hunt M (1995). The meso paradigm: A framework for the integration of micro and macro organizational behavior. *Review of Organization Behavior* 17: 71–114.

Kaddoum E, Gleizes M-P, Georgé J-P and Picard G (2009). Characterizing and evaluating problem solving self-*systems (regular paper). (regular paper) In: *The First Inter. Conference on Adaptive and Self-adaptive Systems and Applications (ADAPTIVE 2009)*, Athens, Greece, 15–20 November, CPS Production—IEEE Computer Society, page (electronic medium).

Klügl F (2008). A validation methodology for agent-based simulations. In: *Proceedings of the 2008 ACM symposium on Applied computing*, SAC¡08, ACM: New York, NY, pp 39–43.

Klügl F (2009). Multiagent simulation model design strategies. In: *MAS&S @ MALLOW'09, Turin*, Vol. 494, *CEUR Workshop Proceedings*, page (on line).

Klügl F, Fehler M and Herrler R (2005). About the role of the environment in multi-agent simulations. In: Weyns D, Van Dyke Parunak H and Michel F (eds). *Environments for Multi-agent Systems*, Vol. 3374 of *LNCS*. Springer: Berlin/Heidelberg, pp 127–149.

Lamport L (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7): 558–565.

Larman C (2004). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development*, 3rd edn. Prentice Hall PTR: Upper Saddle River, NJ.

Law AM (2007). *Simulation, Modeling and Analysis,* 4th edn. McGraw Hill: New York.

Luke S *et al* (2005). MASON: A multiagent simulation environment. *Simulation* 81(7): 517–527.

MacQueen JB (1967). Some methods for classification and analysis of multivariate observations. In: Cam LML and Neyman J (eds). *Proceedinges of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, Vol 1. University of California Press, Berkeley, CA, pp 281–297.

Malerba F, Nelson R, Orsenigo L and Winter S (1999). 'History-friendly' models of industry evolution: The computer industry. *Industrial and Corporate Change* 8(1): 3–40.

Montanola-Sales C, Onggo BSS and Casanovas-Garcia J (2011). Agent-based simulation validation: A case study in demographic simulation. In: *Proceedings of the Third International Conference on Advances in System Simulation (SIMUL 2011)*, Barcelona, Spain, pp 109–115.

Nguyen C *et al* (2011). Testing in multi-agent systems. In: Gleizes M-P and Gomez-Sanz J (eds). *Agent-oriented Software Engineering X* Vol. 6038 of *Lecture Notes in Computer Science*. Springer: Berlin/Heidelberg, pp 180–190.

Niazi MA, Hussain A and Kolberg M (2009). Verification and validation of agent-based simulation using the VOMAS approach. In: *MAS&S @MALLOW'09, Turin*, Vol. 494, *CEUR Workshop Proceedings*, page (online).

Nikolai C and Madey G (2009). Tools of the trade: A survey of various agent based modeling platforms. *Journal of Artificial Societies and Social Simulation* 12(2): 2.

North M, Collier N and Vos J (2006). Experiences creating three implementations of the repast agent modeling toolkit. *ACM Transactions on Modeling and Computer Simulation* 16(1): 1–25.

Pengfei X, Lees M Nan H and Viswanthatn TV (2011). Validation of agent-based simulation through human computation: An example of crowd simulation. In: *Multi-agent-based Simulation XI* pp 1–13.

Pidd M (2004). *Computer Simulation in Management Science*. John Wiles & Sons Inc, Indianapolis, IN.

Polack FAC *et al* (2010). Reflections on the simulation of complex systems for science. In: *Proceedings of International Conference on Computational Science (ICCS'2010)*, Oxford, UK, 22–26 March, pp 276–285.

Raibulet C and Masciadri L (2009). Towards evaluation mechanisms for runtime adaptivity: From case studies to metrics. In: *Proceedings of the 2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, ComputationWorld'09, IEEE Comp. Soc: Washington, DC, pp 146–152.

Railsback SF and Grimm V (2011). *Agent-based and Individual-based Modeling: A Practical Introduction*. Princeton University Press, Princeton, NJ.

Robinson S (1997). Simulation model verification and validation: Increasing the users' confidence. In: *Proceedings of the 29th Conference on Winter Simulation*, WSC '97, IEEE Computer Society: Washington, DC, pp 53–59.

Sansores C and Pavon J (2005). Agent-based simulation replication: A model-driven architecture approach. In: *4th Mexican International Conference on Artificial Intelligence (MICAI'2005)*, Mexico, pp 244–253.

Sargent RG (2005). Verification and validation of simulation models. In: *Proceedings of the 37th Conference on Winter simulation*, WSC'05, Winter Simulation Conference, Orlando, FL, USA, 4–7 December, pp 130–143.

Schwindt P and Crill W (1982). Factors influencing motoneuron rhytmic firing: Results from a voltage-clamp study. *Journal of Neurophysiology* 48(4): 875–890.

Serrano E and Botia JA (2009). Programming multiagent systems. Chapter Infrastructure for Forensic Analysis of Multi-agent Systems, pp 168–183. Springer-Verlag: Berlin, Heidelberg.

Serrano E, Gómez-Sanz JJ, Botía JA and Pavón J (2009). Intelligent data analysis applied to debug complex software systems. *Neurocomputing* 72(13–15): 2785–2795.

Sklar E (2007). Netlogo, a multi-agent simulation environment. *Artificial Life* 13(3): 303–311.

Taillandier P, Drogoul A, Vo D and Amouroux E (2012). Gama: A simulation platform that integrates geographical information data, agent-based modeling and multi-scale control. In: Desai N, Liu A and Winikoff M (eds). *The 13th International Conference on Principles and Practices of Multi-agent Systems (PRIMA)*. Kolkata, India, 12–15 November, Lecture Notes in Computer Science, Vol.7057, Springer: Berling Heidelberg, pp 242–258.

Terano T (2007). Exploring the vast parameter space of multi-agent based simulation. In: Antunes L and Takadama K (eds). *Multi-agent Based Simulation VII*. Lecture Notes in Computer Science, Vol. 4442, Springer: Berlin Heidelberg, pp 1–14.

Troitzsch KG (1996). Multilevel simulation. In: Klaus G *et al* (eds). *Social Science Microsimulation*: Berlin: Springer-Verlag, pp 107–122.

Türker KS and Miles TS (1991). Threshold depolarization measurements in resting human motoneurons. *Journal of Neuroscience Methods* 39(1): 103–107.

Türker KS and Powers RK (2005). Black box revisited: A technique for estimating postsynaptic potentials in neurons. *Trends in Neurosciences* 28(7): 379–386.

Uhrmacher A and Swartout W (2003). *Agent-oriented Simulation*. Kluwer Academic Publishers: Norwell, MA, pp 215–239.

Utting M and Legeard B (2007). *Practical Model-based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc.: San Francisco, CA.

Werker C and Brenner T (2004). Empirical calibration of simulation models. Papers on Economics and Evolution 2004-2010, Max Planck Institute of Economics, Evolutionary Economics Group.

Wilensky U and Rand W (2007). Making models match: Replicating an agent-based model. *Journal of Artificial Societies and Social Simulation* 10(4): 2.

Windrum P, Fagiolo G and Moneta A (2007). Empirical validation of agent-based models: Alternatives and prospects. *Journal of Artificial Societies and Social Simulation* 10(2): 8.

Wolfram S (1994). *Cellular Automata and Complexity: Collected Papers* Advanced Book Program Addison-Wesley Pub. Co: Indianapolis, IN.