ISBN 978-83-60810-22-4

Towards a Generic Testing Framework for Agent-Based Simulation Models

Önder Gürcan*†, Oğuz Dikenelli*
*Ege University
Computer Engineering Department
Universite cad. 35100
Bornova, Izmir, Turkey
E-mail: name.surname@ege.edu.tr

Carole Bernon†
†Toulouse III University
Institut de Recherche en Informatique de Toulouse (IRIT)
118 route de Narbonne
31062 Toulouse Cedex 9, France
Email: name.surname@irit.fr

Abstract—**Agent-based modeling 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 which reveals the existing errors in the model. Moreover, on the practical side, there are many ABMS frameworks 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.**

I. Introduction

VERIFICATION and validation 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 [1]. Model testing is a general technique which 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 [2].

There are many works about verification and validation of agent-based simulations [3]–[6]. However, 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. Based on this observation, our main motivation is to build a testing framework for agent-based simulation models to facilitate the application of the model testing.

Naturally, one has to define the whole model testing requirements of ABMS to be able to develop a model testing framework. To define these requirements, we first define basic elements of ABMS that can be subject of model testing process. Then, we use a generic model testing process [1] and elaborate on the requirements of the model testing framework when it is used throughout this process. Finally, we categorize ABMS's model testing requirements in micro- and macro-levels by an inspiration from ABMS applications in sociology domain [7]. In this categorization, the micro-level takes basic elements individually and defines the framework requirements from the basic element's perspective. On the other hand, the macro-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 macro-level defines model testing requirements of such groups.

After having defined the requirements of the framework, a conceptual architecture which includes some generic conceptual elements to satisfy these requirements of the framework is proposed. These elements are specified and brougth together to conduct the model testing of any ABMS application. Then, a software 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 [8], the proposed architecture is generic enough to be customized for different frameworks.

This paper is organized as follows. The next section defines the testing requirements for ABMS. Section III 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 IV. After discussing the proposal in Section V, Sections VI and VII conclude the paper with an insight to some future work.

II. Testing Requirements for Agent-based Simulation Models

The basic elements of agent-based simulations are agents, the simulated environment and the simulation environment [9]. *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 or some agents may also disappear.

A *simulated environment* contains non-agent entities of the simulation model and agents of that environment. 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. As well as explicitly specified behaviours of these 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).

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.

The basic elements are developed and brought together following a development process to produce a simulation model [10]. 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, 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* according to Balci's classification [2]. According to Balci, dynamic VV&T techniques are conducted in three steps. 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. 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.

Apart from the above requirements, to be able to design a well structured testing framework, we also need to identify the testing requirements of testable elements in detail. As we stated in the first step, the testable elements of agent-based simulation models are *agents* and *simulated environments*. In this sense, both of these elements need to be verified and validated. However, what we also require is a means of verifying and validating hypotheses about how interactions and behaviours of these elements at different abstraction levels are related to each other. As Uhrmacher et al. [11] stated, agent-based simulation models describe systems at two levels of organization: *micro-level* and *macro-level*. In sociology, the distinction between these levels is comparatively well established [7]. The *micro-level* considers the model elements individually and their interactions from their perspectives. However, 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*.

In the following subsections, depending on the characteristics of agent-based simulations, we derive both the *micro-* and *macro-level* testing requirements of agent-based simulation models.

A. Micro-level Testing Requirements

In this level, the testing requirements of the basic elements alone and interactions from their perspective are considered.

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 workload for agents (number of behaviours scheduled at a specific time).

B. Macro-level Testing Requirements

In this level, the testing requirements of the elements of agent-based simulations as groups or sub-societies is considered. The aim is to test the expected collective properties as a whole:

- Testing the organization of the agents (how they are situated in a simulation environment or who is interacting with who) during their lifetime.

- Testing if a group of basic elements exibit the same *macro-level* behaviour with different initial conditions. This macro-level behaviour could be either an *emergent* behaviour or a *non-emergent* behaviour.
- Testing if 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 *macro-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.).

III. The Generic Agent-based Simulation Testing Framework

A. The Conceptual Model

To be able to satify the testing requirements for agent-based simulation models, developers first need a tool that supports model instrumentation. In other words, the tool should allow defining observation points for each testable element 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.

In this sense, 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, we call each specific model designed for testing 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 (*Test agent*), the other required model elements, the data sources in which these elements make use of and a special simulated environment (*Test environment*) that contains all these elements (see Figure 1). It can also include one or more fake elements to facilitate the testing process. Each *Test Scenario* is defined for specific requirement(s) which includes the required test cases, activities and their sequences, and observation requirements. For executing *Test Scenario*s, we designed another mechanism called *Scenario Executer*. *Scenario Executer* is able to execute each *Test Scenario* with different initial conditions for pre-defined durations.

Test agent is responsible for instrumenting the testable elements, collecting information from them and evaluating these information in order to check if they behave as expected. *Test agent* can access every model element during the execution of a *Test Scenario*. However, none of the model elements are aware of it. So 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 *Test agent*, are situated in this environment. However, apart from *Test agent*, none of the other elements are aware of *Test environment*.

Another special mechanism introduced is the usage of special elements called *Fake agent*s and *Fake environment*s

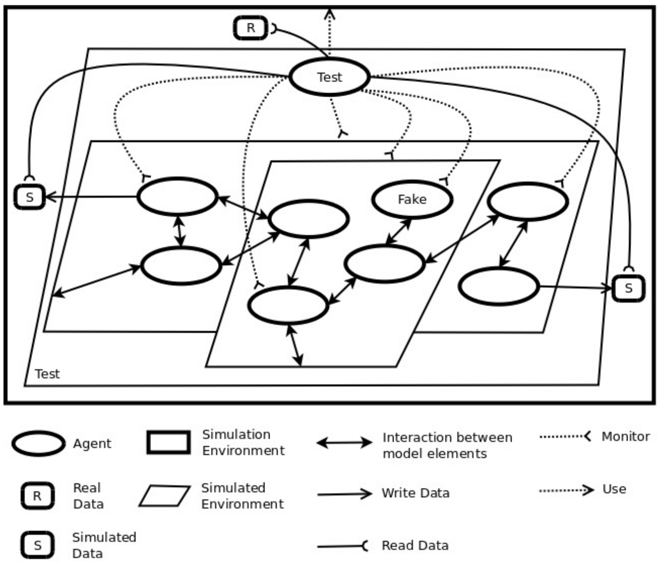

Fig. 1. An illustrative example for a test scenario. As represented in the figure the basic ingredients for test scenarios are: the Test agent, fake agents, the basic elements of agent-based simulation models (agents, simulated environment and simulation environments) and the data they use/produce. The Test agent is able to collect information from all these elements.

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 elements 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 multi-agent systems literature [12], 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 object that they represent and returning pre-arranged responses [13]. 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.

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

B. The Internal Architecture

The internal architecture of the generic testing framework is given in Figure 2. This framework is based on the JUnit[1]

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

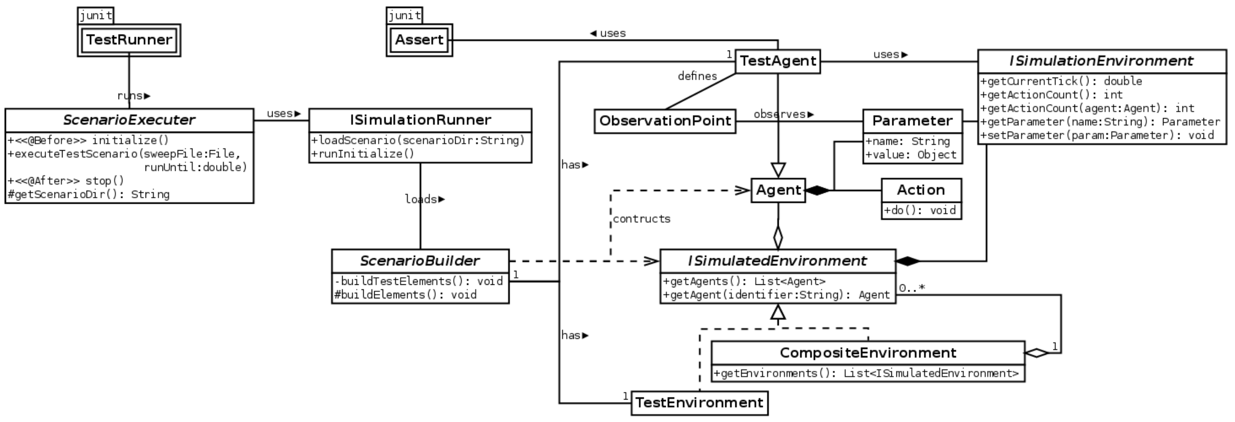
junit
TestRunner
runs▸
ScenarioExecuter
+<<@Before>> initialize()
+executeTestScenario(sweepFile:File, runUntil:double)
+<<@After>> stop()
#getScenarioDir(): String
uses▸
ISimulationRunner
+loadScenario(scenarioDir:String)
+runInitialize()
loads▸
ScenarioBuilder
-buildTestElements(): void
#buildElements(): void
junit
Assert
◂uses
TestAgent
defines
ObservationPoint
observes▸
Parameter
+name: String
+value: Object
uses▸
ISimulationEnvironment
+getCurrentTick(): double
+getActionCount(): int
+getActionCount(agent:Agent): int
+getParameter(name:String): Parameter
+setParameter(param:Parameter): void
has▸
Agent
Action
+do(): void
contructs
ISimulatedEnvironment
+getAgents(): List<Agent>
+getAgent(identifier:String): Agent
0..*
CompositeEnvironment
+getEnvironments(): List<ISimulatedEnvironment>
has▸
TestEnvironment
1

Fig. 2. The conceptual UML model for the generic testing framework.

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 JUnit infrastructure, we defined our scenario executer (`ScenarioExecuter`) as a test case of JUnit. Consequently, by using the existing mechanisms and graphical user interfaces of JUnit, test scenarios can easily be executed.

When loaded by the JUnit test runner, `ScenarioExecuter` first initializes the given test scenario by using the generic simulation runner interface (`ISimulationRunner`) 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, the `ScenarioExecuter` executes the test scenario with different parameters by sweeping the provided file until the defined limit for the test scenario. To do so, `ScenarioExecuter` class provides an `executeTestScenario()` method that enables executing the same test scenario with different initial conditions for different predefined durations. The runner of the agent-based simulation framework is responsible for loading `ScenarioBuilder`. `ScenarioBuilder` builds the scenario by constructing required model elements. It builds the `TestEnvironment` and the `TestAgent` internally by using `buildTestElements()` method. Other model test elements (the simulated environments (`ISimulatedEnvironment`) and the agents (`Agent`))[2] , on the other hand, are built externally by using the provided stub method `buildElements()`.

`TestAgent` is able to access all basic elements in order to make model instrumentation. For accessing the simulated environments and the agents, it uses the `TestEnvironment` and for accessing the simulation infrastructure it uses a special interface (`ISimulationEnvironment`) 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()`)[3] and so on. `TestAgent` 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 `Assert`) depending on the expected behaviour of the agent-based model under test. However, if the ABMS framework provides predefined features for defining observation points, it is not necessary to use the `ObservationPoint` concept in the concrete model testing framework. Since `TestAgent` 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, `TestAgent` 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 (`ISimulatedEnvironment`) as the real interfaces they mimic.

C. Implementation

The generic testing framework has been successfully implemented for Repast (Figure 3). Repast is an agent-based simulation framework written in Java [14]. It provides predefined classes for building agent-based simulation models

[2]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.

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

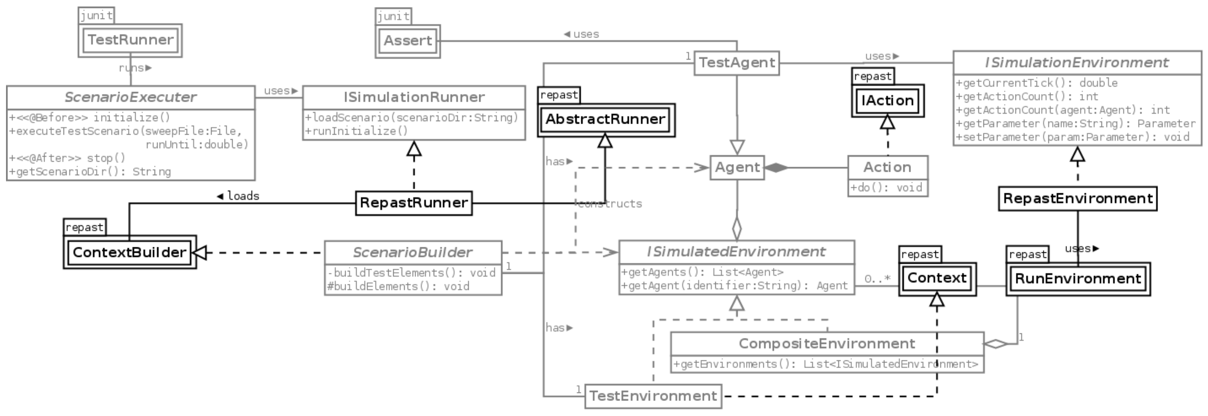

Fig. 3. The UML class model for the Repast implementation of the generic testing framework. The `ObservationPoint` concept is removed here, since Repast provides a speacial mechanism for defining observation points.

as well as for accessing the Repast simulation infrastructure during run time. For the implementation, first 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 `RunEnvironment` class of Repast for accessing the Repast simulation infrastructure as defined in `ISimulationEnvironment` interface. And after, `TestEnvironment` is realized by implementing `Context` interface provided by Repast, since it is the core concept and object in Repast that provides a data structure to organize model elements. Finally, the actions of agents are implemented as a subclass of `IAction` provided by Repast.

D. Usage

The developer first needs to extend `ScenarioBuilder` to define the elements of the test scenario and the initial parameters. Then the `TestAgent` 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.

IV. Case Study: Tonic Firing of a Motoneuron

To demonstrate the effectiveness of our testing framework, we show its applicability on a *micro-level* testing example[4]. For the case study, we have chosen a test scenario from one of our ongoing projects. In this project, we are developing a self-organized agent-based simulation model for exploration of synaptic connectivity of human nervous system [15].

The nervous system is a network of specialized cells that communicate information about organism's surroundings and itself. It is composed of neurons and other specialized cells that aid in the function of the neurons. A *neuron* is an excitable cell in the nervous system that processes and transmits information by electrochemical signalling. Neurons emit spikes when their *membrane potential* crosses a certain threshold (*firing threshold*). 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 firing one (e.g., motoneurons, proprospinal neurons), it emits a spike when the membrane potential constantly rises to the firing threshold (Figure 4). A spike is delivered to the other neurons through its axons. When a spike transmitted by a pre-synaptic neuron through one of its axons reaches a synapse, it transmits the spike to the post-synaptic neuron after a certain amount of time (depending on the length of the axon) which is called an axonal delay. 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[5] in response to stimulation of a set of peripheral afferents or cortico-spinal fibers. Besides, the ability to record motor activity in human subjects has provided a wealth of information about the neural control of motoneurons [16]. Thus, in our project we are focused on simulation of motor units. We developed and brought together the basic elements of our agent-based simulation model. Meanwhile, to

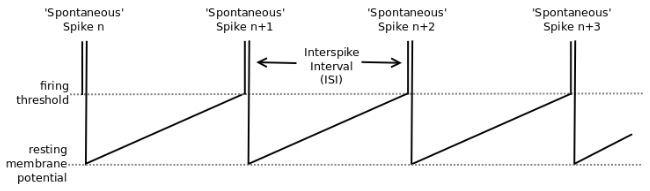

Fig. 4. Tonic firing of a neuron. During tonic firing, a neuron's membrane potential continuously rises to the firing threshold and makes the neuron fire spontaneous spikes. The time interval between consecutive spikes are called inter-spike intervals (ISI).

[4]We are unable to give a macro-level testing example because of space limitations.

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

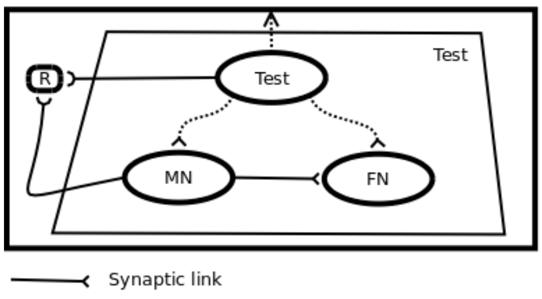

Fig. 5. An illustrative diagram for the “tonic firing of a motoneuron” case study.

verify and validate the model, we designed various test cases both for micro- and macro-levels.

In order to demonstrate how the testing framework can be used, one of our test scenarios was chosen. In this scenario, one *micro-level* behaviour of motoneurons is considered: constant emission of spikes (since they are tonically active). Figure 5 is an illustrative diagram for the selected test scenario. The basic element under test is `Motoneuron` agent (MN). For tonic firing, MN uses the realistic data recorded from a single motor unit of a human subject (R) in Ege University labs[6]. In order to be able to test this *micro-level* behaviour, MN is connected to a `FakeNeuron` agent (FN) with a synaptic link. FN 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 FN after a certain amount of time (axonal delay). During the scenario execution, MN will emit spontaneous spikes constantly. These spikes will be delivered to FN through the synaptic link. Each time FN receives a spike, its membrane potential will rise a little for a while and then go back to resting membrane potential. In order to test tonic firing behaviour of MN, *Test agent* observes the activity of both MN and FN for the given amount of time (for each scenario execution, this amount may differ). At the end of this time, it 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 `ScenarioBuilder` class (Algorithm 1). Within this class, the construction of the basic elements of the test scenario (Motoneuron agent and FakeNeuron agent) is done. Then, *Test agent* (`TonicFiringTester`) is implemented together with its behaviours by extending `TestAgent` class for the testing process (Algorithm 2 and Algorithm 3). As shown in Algorithm 2, `TonicFiringTester` monitors the activities of Motoneuron and FakeNeuron agents by observing their membrane potentials. For defining the observation points, the watch mechanism provided by the Repast infrastructure is used (by using `@Watch` annotation). The resting membrane potential is -55 and the firing threshold is -45 in our simulation model. Thus, when the membrane potential of Motoneuron becomes more than -45, *Test agent* records the time of occurrence in a list to keep track of the activities of Motoneuron during the simulation. Likewise, when the membrane potential of FakeNeuron becomes more than -55, Test agent records the time of occurrence to keep track of the activity of FakeNeuron.

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

Algorithm 1 Source code for TonicFiringScenario class.

```
package motorunit.test03;
import motorunit.*;
public class TonicFiringScenario
                              extends ScenarioBuilder{
   private RunningNeuron motorNeuron;
   private FakeNeuron fakeNeuron;

   public static String EXP_MOTOR_ACTIVITY =
              "./data/real_motoneuron_activity.txt";
   @Override
   protected void createAgents() {
      motorNeuron = new RunningNeuron("Motoneuron",
                                    EXP_MOTOR_ACTIVITY);
      fakeNeuron = new FakeNeuron("FakeNeuron");
      double axonalDelay = 10.0;
      motorNeuron.makeSynapseWith(fakeNeuron,
                                          axonalDelay);
   }
}
```

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

```
package motorunit.test03;
...
import java.util.*;
import repast.simphony.*;
...
public class TonicFiringTester extends TestAgent {
  ...
  private Vector<Double> mnSpikes;
  private Vector<Double> fnActivities;

  public TonicFiringTester() {
    ...
    mnSpikes = new Vector<Double>();
    fnActivities = new Vector<Double>();
  }

  @Watch(watcheeClassName="motorunit.RunningNeuron",
  watcheeFieldNames = "membranePotential",
  query = "colocated",
  triggerCondition = "$watchee.getPotential()>-45",
  whenToTrigger = WatcherTriggerSchedule.IMMEDIATE)
  public void monitorMotorneuronActivity() {
    double tick=RepastEnvironment.getCurrentTick();
    mnSpikes.add(tick);
  }

  @Watch(watcheeClassName="motorunit.FakeNeuron",
  watcheeFieldNames = "membranePotential",
  query = "colocated",
  triggerCondition = "$watchee.getPotential()>-55",
  whenToTrigger = WatcherTriggerSchedule.IMMEDIATE)
  public void monitorFakeNeuronActivity() {
    double tick=RepastEnvironment.getCurrentTick();
    fnActivities.add(tick);
  }
  ...
}
```

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

```
package motorunit.test03;
import static org.junit.Assert.*
import java.util.*;
import repast.simphony.*;
import motorunit.*;
import umontreal.iro.lecuyer.randvar.
                                    RandomVariateGen;
public class TonicFiringTester extends TestAgent {
  private RunningNeuron motorNeuron;
  private FakeNeuron fakeNeuron;
  ...
  public TonicFiringTester() {
    motorNeuron = getTestEnvironment().
                            getAgent("Motorneuron");
    fakeNeuron = getTestEnvironment().
                            getAgent("FakeNeuron");
    ...
  }
  ...
  @ScheduledMethod(start = ScheduleParameters.END)
  public void testTonicFiringOfMotorNeuron() {
    // motorneuron has to generate some spikes.
    assertTrue(motorNeuronSpikes.size() > 0);

    Vector<Double> isiV = ISIDistribution.
     getInstance().calculateISI(motorNeuronSpikes);
    RandomVariateGen rvgSim =
       RandomVariateGenFactory.getGenerator(isiV);
    assertNotNull(rvgSim);
    RandomVariateGen rvgExp =
                       motorNeuron.getRandomGen();
    assertNotNull(rvgExp);
    // distribution should be the same
    assertEquals(rvgExp.getClass().getName(),
                  rvgSim.getClass().getName());

    Distribution dExp = rvgExp.getDistribution();
    Distribution dSim = rvgSim.getDistribution();
    // alpha parameters should be close
    double alphaExp = dExp.getParams()[0];
    double alphaSim = dSim.getParams()[0];
    assertEquals(alphaExp, alphaSim, 0.1);
    // gamma parameters should be close
    double gammaExp = dExp.getParams()[1];
    double gammaSim = dSim.getParams()[1];
    assertEquals(gammaExp, gammaSim, 0.1);
  }
  @ScheduledMethod(start = ScheduleParameters.END)
  public void testConductionOfSpikes() {
    for (int i = 0; i < fnActivities.size(); i++) {
      double axonalDelay = fnActivities.get(i)
        - mnSpikes.get(i);
      assertEquals(10.0, axonalDelay, 1.0);
    }
  }
}
```

As shown in Algorithm 3, `TonicFiringTester` executes two actions for testing the *micro-level* behaviour of Motoneuron at the end of each scenario execution (`ScheduleParameters.END`)[7]. For defining the test cases, the schedule mechanism provided by Repast infrastructure is used (by using `@ScheduleMethod` annotation). One of the test cases (`testTonicFiringOfMotorNeuron()`) checks whether the generated spikes of Motoneuron agent have similar characteristics with the real data or not (`testTonicFiringOfMotorNeuron()`). This test case first tests if Motoneuron agent generated some spikes. And after, it tests if the running (simulated) data generated by Motoneuron agent have similar statistical characteristics: they should represent the same statistical distribution whose parameters are nearly the same. The second test case (`testConductionOfSpikes()`) is designed to test if the spikes generated by Motoneuron agent are received by FakeNeuron agent properly. To do so, it examines if all the consecutive activities of Motoneuron agent and FakeNeuron agent have the same time difference since there is a 10.0 ms axonal delay.

[7]The time for the end of the scenario execution may change at each execution, according to the values given by the developer in `ScenarioExecuter`. See Algorithm 4.

Algorithm 4 Source code for TonicFiringExecuter class.

```
package motorunit.test03;
import org.junit.Test;
public class TonicFiringExecuter
                          extends ScenarioExecuter {
  @Test
  public void tonicFiringTestScenario()
                                  throws Exception {
    executeTestScenario(null, 2000001);
    executeTestScenario(null, 4000001);
  }
}
```

Finally, in order to execute the test scenario (with various criteria) the base class that the JUnit runner will use (`TonicFiringExecuter`) is implemented by extending the `ScenarioExecuter` class (Algorithm 4).

V. Related Work

Although there is a considerable amount of work about testing in multi-agent systems in the literature (for a review see [17]), there are not much work on testing in ABMS. Niazi et al. [5] present a technique which 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. In the sense of using special agents in which the agents under test are not aware of, their approach is similar to ours. But unlike our approach, they define various types of special agents. However, we use a single agent for testing, since at every test our aim is to test one single use case of the system [18]. Besides, they define an architecture but since they begin without defining the requirements it is not quite possible to understand what they are testing. Pengfei et al. [6] proposes validation of agent-based simulation through human computation as a means of collecting large amounts of contextual behavioural data. De Wolf et al. [19] propose an empirical analysis approach combining agent-based simulations and numerical algorithms for analyzing the global behaviour of a self-organising system.

None of the above approaches is well structured and their authors do not give internal details. These approaches also do not take into account neither simulated nor simulation environments. However, we think that both environments need to be involved in the model testing process since they are two of the main ingredients of agent-based simulation models.

VI. Conclusion & Future Work

This body of work presents the initial 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. However, these interfaces are not mature and need to be improved in order to be able to design more comprehensive test scenarios.

To demonstrate the framework's applicability, it is implemented for a well-known agent-based simulation framework called Repast. For model instrumentation, the "watch" mechanism provided by Repast is used. However, if an agent-based simulation framework does not provide such a mechanism, the developers may need to implement it. To solve this design problem, the `Observer` design pattern [20] can be used. In this sense, to show the suitability of the proposed generic framework in case of adoption of frameworks different from Repast, we are planning to implement it for other frameworks.

Since testing is meaningful when it is involved in a development methodology, as another future work, we are planning to define a test-driven process based on these 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-organising multi-agent systems. The metrics for self-organization and emergence mechanisms for achieving self-* properties are given in recent works [21] and [22]. We believe that the capabilities of our framework will be able to test and validate all the metrics given in these studies.

Acknowledgment

The authors would like to thank Kemal S. Türker and Ş. 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 French Government through the co-tutelle scholarship program.

References

[1] O. Balci, "Validation, verification, and testing techniques throughout the life cycle of a simulation study," in *Proc. of the 26th Conf. on Winter simulation*, ser. WSC'94. San Diego, CA, USA: Society for Computer Simulation International, 1994, pp. 215–220.

[2] O. Balci, "Principles and techniques of simulation validation, verification, and testing," in *Proc. of the 27th Conf. on Winter simulation*, ser. WSC'95. Arlington, VA, USA: IEEE Comp. Soc., 1995, pp. 147–154.

[3] T. Terano, "Exploring the vast parameter space of multi-agent based simulation," 2007, pp. 1–14.

[4] F. Klügl, "A validation methodology for agent-based simulations," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 39–43.

[5] M. A. Niazi, A. Hussain, and M. Kolberg, "Verification and Validation of Agent-Based Simulation using the VOMAS approach," in *MAS&S @ MALLOW'09, Turin*, vol. 494. CEUR Workshop Proceedings, September 2009, p. (on line).

[6] X. Pengfei, M. Lees, H. Nan, and V. V. T., "Validation of agent-based simulation through human computation : An example of crowd simulation," in *Multi-Agent-Based Simulation XI*, 2011, pp. 1–13.

[7] K. G. Troitzsch, "Multilevel simulation," in *Social Science Microsimulation*, 1995, pp. 107–122.

[8] C. Nikolai and G. Madey, "Tools of the trade: A survey of various agent based modeling platforms," *Journal of Artificial Societies and Social Simulation*, vol. 12, no. 2 2, 2009.

[9] F. Klügl, M. Fehler, and R. Herrler, "About the role of the environment in multi-agent simulations," in *Environments for Multi-Agent Systems*, ser. LNCS, D. Weyns, H. Van Dyke Parunak, and F. Michel, Eds. Springer Berlin / Heidelberg, 2005, vol. 3374, pp. 127–149.

[10] F. Klügl, "Multiagent simulation model design strategies," in *MAS&S @ MALLOW'09, Turin*, vol. 494. CEUR Workshop Proceedings, September 2009, p. (on line).

[11] A. Uhrmacher and W. Swartout, *Agent-oriented simulation*. Norwell, MA, USA: Kluwer Academic Publishers, 2003, pp. 215–239.

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

[13] M. Feathers, *Working effectively with legacy code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[14] M. North, N. Collier, and J. Vos, "Experiences creating three implementations of the Repast agent modeling toolkit," *ACM Trans. Model. Comput. Simul.*, vol. 16, no. 1, pp. 1–25, January 2006.

[15] O. Gürcan, O. Dikenelli, and K. S. Türker, "Agent-based exploration of wiring of biological neural networks: Position paper," in *20th European Meeting on Cybernetics and Systems Research (EMCSR 2010)*, R. Trumph, Ed., Vienna, Austria, EU, 2010, pp. 509–514.

[16] K. Türker and T. Miles, "Threshold depolarization measurements in resting human motoneurones," *Journal of Neuroscience Methods*, vol. 39, no. 1, pp. 103 – 107, 1991.

[17] C. Nguyen, A. Perini, C. Bernon, J. Pavon, and J. Thangarajah, "Testing in multi-agent systems," in *Agent-Oriented Software Engineering X*, ser. Lecture Notes in Computer Science, M.-P. Gleizes and J. Gomez-Sanz, Eds. Springer Berlin / Heidelberg, 2011, vol. 6038, pp. 180–190.

[18] K. Beck, *Test-driven development : by example*. Boston: Addison-Wesley, 2003.

[19] T. D. Wolf, T. Holvoet, and G. Samaey, "Engineering self-organising emergent systems with simulation-based scientific analysis," in *In: Proceedings of the Fourth International Workshop on Engineering Self-Organising Applications, Universiteit Utrecht*, 2005, pp. 146–160.

[20] C. Larman, *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development (3rd edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[21] E. Kaddoum, M.-P. Gleizes, J.-P. Georgé, and G. Picard, "Characterizing and Evaluating Problem Solving Self-* Systems (regular paper)," in *The First Inter. Conf. on Adaptive and Self-adaptive Systems and Applications (ADAPTIVE 2009), Athenes, Grece, 15/11/2009-20/11/2009*. CPS Production - IEEE Computer Society, 2009, p. (electronic medium).

[22] C. Raibulet and L. Masciadri, "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*, ser. COMPUTATIONWORLD '09. Washington, DC, USA: IEEE Comp. Soc., 2009, pp. 146–152.