RatKit: A Repeatable Automated Testing Toolkit for Agent-based Modeling and Simulation

İbrahim Çakırlar[1], Önder Gürcan[1, 2], Oğuz Dikenelli[1], Şebnem Bora[1]

[1] Ege University, Department of Computer Engineering, 35100, İzmir, Turkey
icakirlar@gmail.com,
{onder.gurcan,oguz.dikenelli,sebnem.bora}@ege.edu.tr
[2] CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems, Point Courrier 174, Gif-sur-Yvette, F-91191 France
onder.gurcan@cea.fr

Abstract. Agent-based modeling and simulation (ABMS) became an attractive and efficient way to model large-scale complex systems. However, achieving a sufficiently credible agent-based simulation (ABS) model is still difficult due to weak verification, validation and testing (VV&T) techniques. Moreover, there is no comprehensive and integrated toolkit for VV&T of ABS models that demonstrates that inaccuracies exist and/or which reveals the existing errors in the model. Based on this observation, we designed and developed RatKit: a toolkit for ABS models to conduct VV&T. RatKit facilitates the VV&T process of ABMS by providing an integrated environment that allows repeatable and automated execution of tests. This paper presents RatKit in detail and demonstrates its effectiveness by showing its applicability on a simple well-known ABMS case study.

Keywords: Agent-based modeling and simulation, model testing, verification and validation

1 Introduction

Agent-based modeling and Simulation (ABMS) is a very multidisciplinary complex system modeling and simulation technique, which is has been used increasingly during the last decade. The multidisciplinary scope of ABMS ranges from the life sciences (e.g. Biological Networks [6], Ecology [7], social Sciences [8], Scientometrics [9] to Large-scale Complex Adaptive COmmunicatiOn Networks and environmentS (CACOONS) [10] such as Wireless Sensor Networks, Peer-to-Peer networks, and the Internet of Things (IoT)). While in some domains ABMS is used for understanding complex phenomena, in other domains it is used to design complex systems. However, whatever the objective is, in all of these domains large sets of agents interacting locally give rise to bottom-up collective behaviors. The collective behaviors of agents, whether emergent or not [11], depend on the local competences, the local perceptions and the partial knowledge of agents as well as the global parameter values of the simulation run. A slight difference in any of these properties (whether intentional or not) may result in totally different collective behaviors. Such a consequence leads either to a misunderstanding of the

complex system under study or a bad complex system design. In this sense, right design and implementation of ABS simulation models is becoming more important to increase reliability.

Besides, despite all ABMS platforms are developed by computer scientists, the users of these platforms (i.e. The developers of ABMS models) are more heterogeneous. Depending on the application domains, they can be (1) computer scientists that are building ABS models for their domains, (2) non-computer scientists that are building models for their domains or (3) computer scientists that are working closely with non-computer scientists. On the one hand, non-computer scientist modelers are experts in their domains (i.e. Domain experts) and are said to be capable of building the right models. However, translating these models into their corresponding software models (i.e. ABS models) can sometimes be problematic and open to mistakes. Moreover, since they have less expertise concerning software development, it is a big mystery as to whether they are building the models right or not. On the other hand, computer scientist modelers are better at building models correctly, but they usually lack the expertise to build the right models.

In the literature, the solution to the above problem is defined as the verification, validation and testing (VV&T) of simulation models. Model verification deals with "*building the model right*" while model validation deals with "*building the right model*", as stated in [1]. Model testing, on the other hand, is a general technique that can be conducted to perform verification and/or validation of models. It demonstrates that inaccuracies exist in the model or reveals the existing errors in the model. In model testing, test data or test cases are subject to the model to see if it functions properly [2]. As [12] points out traditional techniques for VV&T cannot be transferred easily to ABS. There are some efforts [13,14, 15, 16, 17], but these studies do not directly deal with model testing processes and focus on late validation and verification. As well, there are few proposed model testing frameworks to conduct validation and verification throughout the model testing process [15,18,19]. Among them, [18] proposed an integrated and automated testing framework, but unfortunately this framework not easy to use for non-computer scientists.

Based on the above observations, our desire is to develop an automated and integrated testing framework for ABSs in order to facilitate the model testing process for all types of modelers. Towards this objective, we took the generic testing framework proposed by Gürcan et al. [18] and improved it one step further by taking into account the requirements of testing frameworks for

ABMS. Previously, testing requirements for ABMS are defined and testing levels of ABMS that can be subject of the model testing process are clarified in our previous studies [18,22]. We also revise our multi-level testing categorization by keeping in sight the requirements of agent-based simulation testing frameworks.

2 Requirements of Agent Based Simulation Testing Frameworks

VV&T leads the simulation model development to increase understanding of the potential of models and to decide when to believe a model, and when not to, and to interpret and to use the model's results. However, it should be noted that VV&T is not a silver bullet. VV&T also has some limitations and constraints. Apparently, one intending to design a testing framework should take into consideration the requirements below.

- **Integration with the Simulation Environment**: A testing tool for ABMS should be integrated into the simulation environment or pluggable. Since it should behave like a simulation engine, interpret the model outputs and execute the testing criteria corresponding to the evaluation rules.
- ***Multilevel Testing:*** Due to the multilevel nature of ABMS [25] and experiences reported in the testing literature [26], obviously a testing framework dedicated to ABMS should support multilevel testing as discussed [18,22,24]. Such an approach provides control over the degree of detail during simulation model development.
- ***Automated Testing:*** Automation of testing is the capability of to execute simulation model tests, for all levels and types, together and individually. Especially for simulation models with a high degree of complexity, automated testing [21] provides the degree of confidence without any special effort.
- ***Monitoring the Model without Any Intervention:*** Monitoring the simulation models, the behaviors of agents, or occurrence of special or unexpected cases are the main expectations for model testing. In most of the testing effort [15,18] testing models intervene in the scheduling of agents, agent behaviors or the simulated environment. Controlling the scheduling of agents or agent behaviors in testing is not the same as execution of the simulation model. This effort is the adaptation of the simulation model to the test scenarios.

- ***Parameter Tuning:*** Simulation models contain model parameters. These parameters are the key values for the simulation and affect the simulation behaviors. Such a testing tool should provide parameter tuning capability [5] to the model testers for finding appropriate parameter values, showing the domino effect between model parameters, testing the variety of parameter values, drawing the boundaries for the parameter value set and testing parameter sensitivity etc.
- **Presenting Model Outputs/Results**: Model results are the only output for the model under execution. Evaluation of the results corresponding to the testing needs is the subject of testing [2]. Model results are not only final outputs. Here, it means the observations that are captured at any time during the test execution. An observation can be value of an agent attribute, value of the environment parameters or resources.
- **Visualization Support [15]:** VV&T of ABMS do not only focus on quantitative methods. In this sense, the model under test is tested against all possible parameter values. Model testers can monitor the behaviors of agents or a group of agents with different conditions without any extra effort. However, visualization is not only visualizing the simulation execution, but also presenting model outputs or summarizing them. Drawing a graphical representation of observation history should help model testers to review simulation execution or the behaviors of test scenario agents.
- **Logging:** Logging [15] is presenting a history of test execution to the model tester. Some of the situations not considered in a test case can be determined with the help of logs. Reviewing test logs help model testers to monitor the model behaviors. Logging should be optional and should support different log levels.
- **Testing Simulation Models Should Easy-to-Use:** Testing proposals [13,14, 15, 16, and 17] for ABS is hard-to-use and extra effort is required. VV&T is difficult enough for simulation developers because of its nature. Therefore, it should be identical to the model development to address all model testers especially non-computer scientists.

It's inevitable that such a testing framework for ABMS should support these requirements. Towards this objective, we designed and developed RatKit for ABMS to facilitate the model testing process taking into consideration ABMS audience requirements and expectations.

3 RatKit: A Repeatable Automated Testing Toolkit for ABMS

RatKit (Repeatable Automated Testing toolKIT) is a testing toolkit to facilitate model testing. Testing requires the execution of the model under test as stated in [18]. In this context, each specific model designed for testing is called *Test Scenario*. Each Test Scenario is defined for specific purpose(s) and includes the required test cases, activities, sequences, and observations. Observations are collected by the *Test Environment* during the execution of the test scenario. The *Test Agent* is responsible for evaluating these assertions according to the collected observations in order to check if these testable elements [18] behave as expected or not.

3.1 RatKit Architecture

The architectural UML model of the RatKit is given in Fig. 1. RatKit uses Junit[2] testing infrastructure for all testing purposes like assertions, test runners, etc. *RatKitRunner* is the main class for the architecture and the Junit test runner for simulation tests. When a test class is annotated by the annotation *@RunWith*(*RatKitRunner*.class) all test methods of the test class are evaluated by the TestAgent. RatKit toolkit is implemented for the Repast simulation environment [23] (RatKit4Repast[1]).

RatKitRunner first initializes the given test scenario for each test method and creates test scenario elements using *RatKitScenarioLoader*. *RatKitScenarioLoader* creates the necessary test scenario files like scenario, dataset, data gatherer and parameter files corresponding to the defined test method parameters. RatKit provides test developers defining parametric, periodic and repeatable test executions with the *@RatKitTest* annotation. *RatKitScenarioLoader* evaluates the defined parameters for the test scenario and decides the type of test execution. Each RatKitParameter definition corresponds to a simulation model parameter. *RatKitParameter* values can be constant, number, value iterations like 0 to 100, or a list of values. *RatKitParameterSweeper*

---

[1] RatKit4Repast http://code.google.com/p/ratkit (Accessed: March 2014).

[2] JUnit. http://www.junit.org (Accessed: March 2014).

evaluates these parameter definitions and triggers the *RatKitRunner* for parametric/periodic test scenario executions.

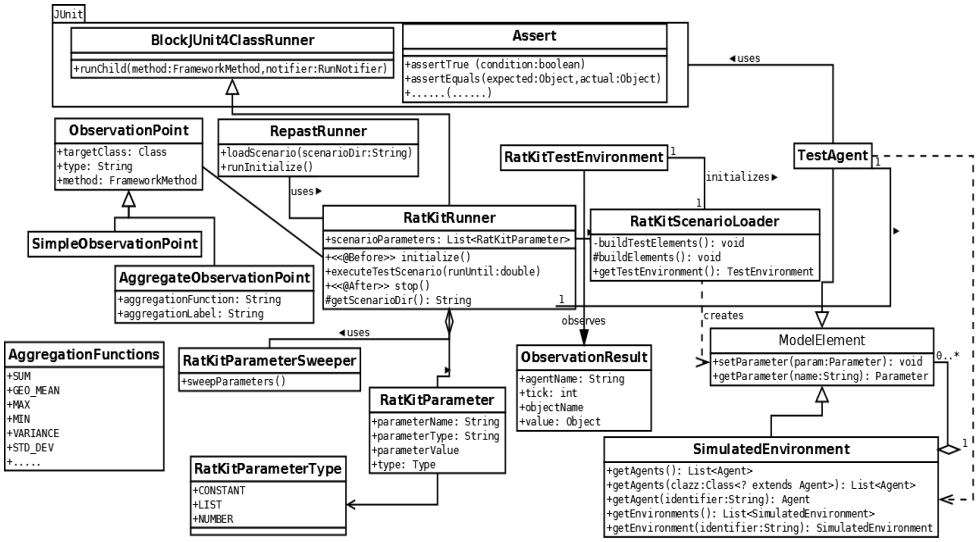

Fig. 1. : RatKit Architecture (UML Class Model)

The test scenario is a sub model, except that it contains fake agents in order to achieve required behaviors of the real simulation model. For simulation tests, each test, changeable corresponding to the testing levels, should have at least one observation point definition. TestAgent executes the assertions corresponding to the observation results. There are two types of observation definitions: *SimpleObservationPoint* and *AggregateObservationPoint*. *SimpleObservationPoint* definitions provide gathering model element properties; a property of an Agent type or an environmental variable. *AggregateObservationPoint* definitions provide summarized results for the model under testing using aggregate functions (count, max, min etc.). Each observation point definition is handled by the *RatKitRunner* and presents it to the *TestAgent* during the execution of the test cases as an *ObservationResult*. Each observation result is time stamped, when it's observed and by whom (agent identifier) if required. *RatKitTestEnvironment* holds the current observation history, a map of the observation results gathered during the execution, and presents to the TestAgent. According to the test execution behavior of the de-

veloper, *TestAgent* executes evaluations (assertions) corresponding to the observations.

4 Case Study: Predator Prey

In this section, we demonstrate the effectiveness of RatKit and its applicability on a well-known case study: Predator Prey. We use a model of wolf-sheep predation [4] that is intentionally simple as an introductory tutorial. While the example is not intended to show real VV&T phenomenon, the model's complexity is high enough to illustrate developing simulation tests.

We extend the example model of the Repast Simphony [23] wolf-sheep predation as a demonstration of the toolkit's capabilities. This model represents a simple variation of predator prey behavior using three agent types: wolf, sheep, and grass. Both the wolves and sheep move randomly on a grid, and lose energy. The wolves and sheep need to feed in order to replenish their energy, and they will die once their energy level reaches zero. Wolves prey on sheep and may eat them if the two are located in the same spatial position. Sheep may similarly eat grass if the sheep is located on a patch that contains living grass. Reproduction is modeled by a random process that creates a child from the parent, divides the energy of the parent agent in half, and assigns the energy equal to the parent and child.

```
public class WolfSheepScenarioBuilder extends PredatorPreyScenarioBuilder {
   @Override
protected void createAgents() {
Wolf wolf = this.getEnvironment().createFakeWolf("wolf1");
getContext().add(wolf);
Sheep sheep = this.getEnvironment().createFakeSheep("sheep1");
getContext().add(sheep);
Grid patch = (Grid) getContext().getProjection("Simple Grid");
      patch.moveTo(sheep, 20, 30);
      patch.moveTo(wolf, 20, 30);
   }
}
```

Fig. 2. Test Scenario Definition

In the case study, all of the possible test scenarios are implemented corresponding to our testing levels. Because of page limits we only present a meso-level test: a wolf agent prey on a sheep agent. The definition of the test scenario is shown in the Fig. 2. WolfSheepInteractionScenarioBuilder class defines the test scenario. In the scenario, there are two fake agents [18]:

FakeSheep, FakeWolf. These agent classes are extended from original agent classes to prevent random movement of the real agent classes. The real purpose of the test scenario is to test the interaction between wolf and sheep agent in the same spatial position. Therefore, both of the scenario agents are located in the same (20, 30) position. We expect in the first tick (runUntil=1.0 defines the test method execution time in the header of the test method in Fig. 3) of the test execution the wolf agent will prey on the sheep agent in the same spatial position.

The test method of the test scenario is shown in Fig. 3. Case study test cases are defined by the *wolfEatSheep* method which is annotated by the @RatKitTest annotation. The test method annotation includes the definitions of test scenario, execution parameters, simulation model parameters and observation points. In our test scenario, *sheepgainfromfood*, *wolfgainfromfood*, *wolfreproduce*, *sheepreproduce* are model parameters. These are required parameters for the initialization of agent instances and also parameter values which affect agent's behaviors in the simulated environment. *Sheepgainfromfood*, *wolfreproduce*, *sheepreproduce* are constant type parameters. And *wolfgainfromfood* parameter type is defined as the number (type= NUMBER). In the execution of simulation tests, the parameter value will be increased from 5 to 10 by the RatKit infrastructure.

In this scenario, we intend to test the wolf agent to see whether it gains energy and the sheep agent dies. Firstly, we need to monitor the energy value of the wolf agent. For this reason, we define a simple observation target SimpleAgent class instance (we want to monitor all wolves in the simulated environment) by collecting the values of the "getEnergy" method with the identifier "getLabel" value. Observation results are presented to the developers with an identifier and a time value (in which tick observation result is gathered). In test cases, we need to separate which result belongs to which agent. In the definition of test scenarios, we define the identifiers of the agents like "*wolf1*" and "*sheep1*".

Another purpose of the test case to test whether the sheep agent has died (removed from the simulated environment). For this reason, we define an aggregate observation point for counting the sheep agent instances in the environment for each tick of the simulation run. The aggregate observation

point targets the Sheep agents by using the "*count*" aggregate function which is named as "*sheep_count*".

```
@RunWith(RatKitRunner.class)
public class WolfSheepInteractionTest {
@RatKitTest(runUntil = 1, scenarioBuilderClass= WolfSheepScenarioBuild-
er.class,parameters =
{@RatKitParameter(parameterName="sheepgainfromfood",parameterValue = "5",
parameterType = ParameterTypes.DOUBLE),
@RatKitParameter(parameterName="wolfgainfromfood", from = "5.0", to = "10.0",
step = "1.0", parameterType =DOUBLE, type= NUMBER),
@RatKitParameter(parameterName="wolfreproduce",parameterValue="0",
parameterType =DOUBLE),
@RatKitParameter(parameterName="sheepreproduce", parameterValue =
"0",parameterType=DOUBLE)},
observationPoints = {@ObservationPoint(targetClass=SimpleAgent.class,
method="getEnergy",label = "getLabel"),
@ObservationPoint(targetClass = Sheep.class,function=COUNT,
type=AGGREGATE,label = "sheep_count")})
public void wolfEatSheep() throws Exception {
RatKitTestEnvironment env=RatKitTestEnvironment.getInstance();
double initialEnergy=(Double) env.getSimpleObservation( "wolf1", 0.0,"getEnergy");
double energy=(Double)env.getSimpleObservation("wolf1", 1.0,"getEnergy");
// assert that wolf consume the sheep agent...
assertTrue(energy > initialEnergy);
double gain = (Double) RunEnviron-
ment.getInstance().getParameters().getValue("wolfgainfromfood");
// assert that wolf loses energy after each tick
Assert.assertEquals(initialEnergy - 1 + gain, energy);
// get count value in the initial scenario
int initialSheepCount = (Integer) env.getAggregateObservation(0.0,"sheep_count");
Assert.assertEquals(1, initialSheepCount);
int finalSheepCount = (Integer) env.getAggregateObservation(1.0,"sheep_count");
Assert.assertEquals(0, finalSheepCount);
}
```

Fig. 3. Test Method Definition

In that test method body firstly an instance of the *RatKitTestEnvironment* class is initialized. This class is responsible for presenting observation results to the developers. To test whether the wolf agent gains energy from the eating behavior that is executed in the first tick (tick value is 1.0), we need to get observation results of the "getEnergy" observation results of the initial and final ticks. The wolf agent is created with an initial energy; its energy is decreased by one for each simulation tick and in the first tick the wolf agent gains energy by eating the sheep agent. These values are evaluated based on the model parameter "*wolfgainfood*" value.

Another purpose of the test method is comparing the initial and the final sheep count in the environment. For this reason, we get the "*sheep_count*" observation results of the initial and the final tick value. And we expect here

that there are no sheep in the final tick of the simulation run. The result of the single execution of the test scenario is shown in the Fig. 4. The scenario parameters are attached to the Junit results in order to for visually monitor the effects of the model parameters in the test execution.

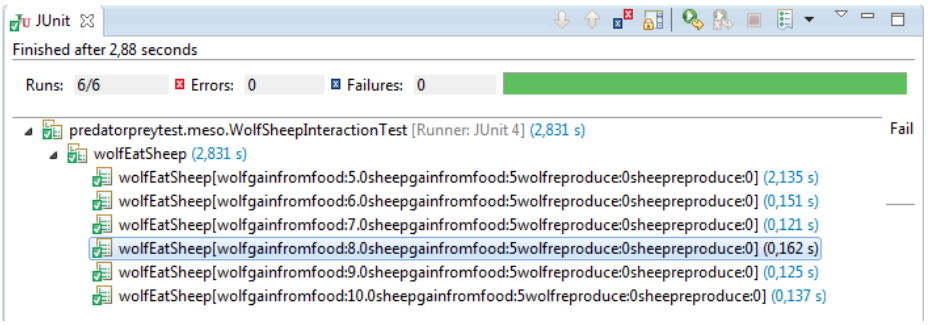

Fig. 4. Test Results

5 Related Work

There has been little work that specifically addresses testing of ABSs and also simulation models.

MASTER is proposed by Wright et al in [19], is a simulation model testing framework for ABSs and compatible with the MASON ABS development environment. MASTER is an external validation tool that provides defining acceptance tests for simulation models. MASTER aims to detect suspicious simulation runs corresponding to the user defined assertions. The modeler defines normal situations, facts, constraints and abnormal situations for the model under test; the framework monitors the simulation runs and evaluates deviations from the normal situations. MASTER is a semi-automatic testing tool and only focuses on prepared simulation models. Rather than developing credible simulation models, it focuses on final VV&T process.

VOMAS, proposed by Niazi et al. [15], is one tool for VV&T of ABMS. They propose using a group of specialized agents, agents specialized in monitoring and testing, over an overlay network to conduct the VV&T process. The agents of the overlay use defined constraints in order to detect unusual behaviors, and report violations if they occur. However, it is not clear how the constraints for the overlay agents are derived and how observations are

evaluated. And also, monitoring of the model agents is not clarified. Intervention into the simulation agents breaks the normal simulation run and VV&T gets further away from its main objective.

6 Future Works and Conclusions

This paper has introduced RatKit and its VV&T approach against ABMS. A tool supporting all needs and aforementioned requirements for VV&T targeting ABMS is an important lack. Our main motivation is filling this gap by the development of RatKit.

Currently using RatKit, users can define simulation tests according to their VV&T purposes. All of the tests are implemented by the users. However, for future work we intend to support fully automated test case generation from the test scenarios. Most of the testing requirements for the models except domain specific ones have some common points. So, automatic generation of common test cases will be supported by RatKit next versions. In this study, we defined the requirements of ABMS testing frameworks. Except visual support, we implement all of the requirements for the current study. For future work, we are studying on the visualization of simulation execution, testing and observations.

Besides, as we mentioned before, a testing framework leading to right design and implementation of ABS models are highly important in order to be able to increase their reliability. For another future work, we intend to define a test driven development methodology for agent based simulation and modeling. Trying to verify, validate and test the agent based simulation models after model building makes ABS development more complex. Such a test driven development methodology that is supported by a testing framework is another gap in the ABMS literature.

References

1. Balci, O. Validation, verification, and testing techniques throughout the life cycle of a simulation study. WSC'94, p 215–220, 1994
2. Balci, O. Principles and techniques of simulation validation, verification, and testing. WSC'95, p. 147–154, IEEE Comp. Soc., 1995
3. Love, G., Back, G. : Model Verification and Validation for Rapidly Developed Simulation Models: Balancing Cost and Theory, 2000
4. Wilensky, U. NetLogo Wolf Sheep Predation model. Center for Connected Learning and Computer-Based Modeling, 1997

5. Calvez, B., Hutzler, G. :Automatic Tuning of Agent-Based Models Using Genetic Algorithms, Multi-Agent-Based Simulation VI, 2006
6. Gürcan, O., Türker, K. S., Mano J., Bernon C., Dikenelli, O., Glize, P. : Mimicking Human Neuronal Pathways in Silico: an emergent model on the effective connectivity. 2013.
7. Grimm, V., Revilla, E., Berger, U., Jeltsch, W. M., Railsback, S : Pattern-oriented modeling of agent-based complex systems: Lessons from ecology. Science, 987–991, 2005
8. Epstein, J. M. Agent-based computational models and generative social science. In Generative Social Science Studies in Agent-Based Computational Modeling, 2007
9. Niazi, M.A., Hussain, A. : Agent-based Computing from Multi-agent Systems to Agent-Based Models: A Visual Survey, Springer Scientometrics, 479-499, 2011
10. Niazi, M.A. , Hussain, A. : A Novel Agent-Based Simulation Framework for Sensing in Complex Adaptive Environments Sensors Journal, IEEE (Volume:11 , Issue: 2), 2011
11. Wolf, D., Holvoet, T. : Emergence versus self-organisation: different concepts but promising when combined, Engineering Self Organising Systems: Methodologies and Applications, LNCS, volume 3464, p. 1-15, 2005
12. Sargent, R. G. Verification and validation of simulation models. WSC'05, p. 130–143, 2005
13. Terano, T. Exploring the vast parameter space of multiagent based simulation, 2007
14. Klügl, F.: A validation methodology for agent-based simulations. In Proceedings of the 2008 ACM symposium on Applied computing, SAC'08, pages 39–43, ACM, 2008
15. Niazi, M. A., Hussain, A., and Kolberg, M. Verification and Validation of Agent-Based Simulation using the VOMAS approach. volume 494, 2009
16. Pengfei, X., Lees, M., Nan, H., and T., V. V.: Validation of agent-based simulation through human computation : An example of crowd simulation pages 1–13, 2011
17. Railsback, S. F. and Grimm, V. Agent-based and Individual-based Modeling: A Practical Introduction. Princeton University Press, 2011
18. Gürcan, O., Dikenelli, O., Bernon, C. : A Generic Testing Framework for Agent-Based Simulation Models., Journal of Simulation, 2013, volume 7, pages 183-201
19. Wright, C. J., McMinn, P., Gallardo, J. : Testing Multi-Agent Based Simulations using MASTER, 2012
20. Balci, O. Golden Rules of Verification, Validation, Testing, and Certification of Modeling and Simulation Applications, *SCS M&S Magazine*, 2010
21. Beck, K. Test-Driven Development by Example, Addison Wesley - Vaseem, 2003
22. Gürcan, O., Dikenelli, O., Bernon, C. : Towards a Generic Testing Framework for Agent-Based Simulation Models. MAS&S 2011, p. 637-644, 2011,
23. North, M.J., T.R. Howe, N.T. Collier, and R.J. Vos, The Repast Simphony Runtime System, Argonne National Laboratory, 2005
24. Jean-Baptiste Soyez, Gildas Morvan, Daniel Dupont, Rochdi Merzouki: A Methodology to Engineer and Validate Dynamic Multi-level Multi-agent Based Simulations. 2012
25. Drogoul A., Edouard Amouroux E., Caillou P., GAMA: A Spatially Explicit, Multi-level, Agent-Based Modeling and Simulation Platform, 2013
26. Burstein I., Practical Software Testing, Springer, 2003.