Toward testing self-organizations in multi-embedded-agent systems

Arthur Baudet[1], Oum-El-Kheir Aktouf[1], Annabelle Mercier[1], and Jean-Paul Jamont[1]

Univ. Grenoble Alpes, Grenoble INP, LCIS, Valence, France
arthur.baudet@grenoble-inp.org
{oum-el-kheir.aktouf,annabelle.mercier,jean-paul.jamont}@lcis.grenoble-inp.fr

Abstract. This paper presents a testing approach for validating global adaptation in multi-embedded-agent systems. Those systems are gaining increasing attention due to their high adaptability and resilience. They differ from software multi-agent systems because embedded agents have additional constraints, like energy management that software agents don't. Those constraints and other specificities, like the tight link with the physical environment, require the use of specific methods and tools for testing these systems. The proposed approach aims at validating at run-time the adaptation of those systems when the entities composing them, the agents, are able to change their global behaviors with self-organization processes. Self-organization processes are not specific to multi-agent systems but in their case, they allow agents to change their organization, i.e.their way of interacting, at runtime. The proposed approach and tool are designed to support lifelong monitoring of multi-embedded-agent systems. In such systems, agents have self-organization behaviors resulting in complex and ever adapting systems, which are challenging to test and monitor.

Keywords: Embedded multi-agent systems · Run-time validation · Multi wireless agent communication · Self-organization testing

1 Introduction

To tackle the problem of over increasing complexity, software engineers are using new and innovative ways of thinking, with more distributed data and/or computation. In this context, agent oriented software engineering is gaining increasing attention thanks to some interesting characteristics of multi-agents systems, like high fault-tolerance, flexibility and capacity to adapt to the environment or previous experiences.

There are many definitions of multi-embedded-agent systems (MEAS) or multi-agents systems as they are used in many different scientific fields [14]. From the software engineering point of view [10, 13, 23] MEAS are seen as systems revolving around the cooperation of physical autonomous entities called agents. Those agents use cognitive characteristics, like reactivity or proactivity, and

cooperation to achieve their goals, known as the local goals. While they are aware of their local goals, agents are not fully aware of the complete system they belong to, and especially the system's goal, known as the global objective. Such systems allow simple or low capability agents to work on simple tasks but achieve, as a whole system, a much more complex objective. A particularity of a multi-agent system, embedded or not, is the absence of a central entity coordinating the agents, so that the system-level decisions are distributed among the agents. The differences due to the embedded feature are related to constraints like energy management, safety management, or other issues related to mobility, communications and integrity of the agents in a physical environment [2]. Those constraints need to be considered in the testing phase. As a consequence, MEAS are not considered as a subpart of multi-agent systems but as a different kind of systems, at least from the testing point of view.

In this context, we are interested in testing the achievement of the global objective, which could be considered as be acceptance testing, from the software engineering perspective. Testing in MEAS is quite challenging because MEAS are asynchronous complex[1] systems of intelligent agents [19]. Very few existing works focus on MEAS, most of them focus only on MAS independently from the implementation of the system under test (SuT). In the following, we will only consider works with hypothesis on the SuT that do not conflict with the constraints added by the embedded feature of MEAS.

Testing the achievement of the global objective is to test if the agents, when working together, produce the expected global output. As there is no central control over the cooperation of the agents, the cooperation is structured by an organization. This organization is known to every agent and is implemented inside the agents. Testing the global objective can then be done by assessing that the organization is correct. In this paper, we will be focusing on the self-organization behaviors. This kind of behavior is the capacity of the agents to create or change the organizations they are in during the execution [5]. It gives the agents the ability to adapt the whole system so it can fit at the best its specifications in a changing environment. Therefore, the organization depends on the agents' autonomy, which increases the difficulty to test it.

Next section introduces challenges of testing self-organization in MEAS. Then a review of existing testing techniques applied to this problem is provided in Section 2. In Section 3 we provide the adaptation of a MAS testing method to a MEAS through a real life example, and we conclude with main learned aspects and some perspectives for future work in Section 4.

2 Review of testing methods for self-organizations

From a testing point of view, MEAS are different from other distributed systems due to the autonomy of the agents. These are intelligent entities which are hard to test [17], mainly due to some cognitive features such as the ability to

[1] in the number of variables considered and inter-dependencies between agents

autonomously make decisions in situations that were not planed. Furthermore, the whole decision process is distributed among them, making the determination and the control of the whole system's behavior very hard, not to say unfeasible.

Self-organizations share those difficulties because the organizations result from the autonomy of the agents. They also add challenges like the error masking. Indeed, errors can be hidden from the tester if the system adapts when agents fail. Not only agents are capable of adapting but the way they globally behave is also adapted, increasing the number of states to consider when testing MEAS [7].

Very few works have been done to test specifically MEAS and even less for testing self-organization in MEAS. In this section we will present the main methods used to test self-organizations in MAS and we'll discuss to what extent they can be applied to MEAS.

2.1 Formal methods

Formal methods offer systemic ways to check if the self-organization process will work as intended by studying models of the system and its specifications. These methods mainly comprise model-checking, model-based testing and simulation-based testing.

Model checking is one of the most widespread techniques [8, 16, 21, 20]. It consists of modelling the system into a known model and using mathematical tools to prove that the system is correct, based on the model.

Model-based testing methods [15] use modelled specifications to automatically generate test cases with large coverage.

Simulation methods [4, 6, 18] use modelled system and specifications to run the modelled system through the modelled specifications and analyze results to ensure that the system will behave as expected.

2.2 Run-time validation methods

Run-time validation methods aim to complete the work done at design time assuming that it is really hard to anticipate every external stimuli and mutation of the system. Moreover, they offer tools to help the supervision process through monitoring the system, either by only presenting a more usable view of the system [22] or analyzing the system to highlight possible errors [12, 3]. Nevertheless, those systems do not prevent errors from occurring and some work like in [1] try to support the system to prevent it from mutating in wrongful ways.

Generally, formal methods are widely used to guide the developers at design time. However, it is also relevant to add run-time testing to ensure that the system will not misbehave at run-time since MEAS have strong constraints on safety, integrity, and it would require a very complex model to validate those constraints for every possible events during the design phase.

Moreover, automating the validation process at run-time is relevant in our case because we aim at providing a tool for software testers whether they have the mathematical knowledge to use formal methods or not. We do not look for

a fully automated system since it will make the testing process complex. Also, we will have to validate the testing approach so keeping it as simple as possible is helping not creating a dependency of complex systems, needing to be able to to test another complex system...

3 Testing Multi Wireless Agent Communication

In this section, we develop a run-time system validation method and we show how to apply it to a real life example. To do so, we use the Multi Wireless Agent Communication model (MWAC) described below.

3.1 Definitions

MWAC [11] provides a routing solution for wireless MEAS where no infrastructure for communication exists. Its objective is simple: giving every agent a way to communicate with every other agent. This solution relies on the execution of a routing protocol on a specific organization. This organization is done at run-time via a self-organization mechanism. In our study, we only consider the organization and the self-organization processes.

MWAC defines three roles that can be assigned to the agent. A *representative (r)* manages and routes the messages of nodes that are directly connected to it. To achieve this task, it broadcasts, relays, and responds to route search requests. A *link (l)* enables message exchange between the representative nodes that are directly connected to it. A *simple member (s)* communicates only with the representative node to which it is directly connected. It does not have any routing task to ensure, unless it is the first sender or the final receiver of a message.

Definition 1 (MAS). *Let n be the number of agents, $A_t = (a_i)_{i\in[\![1,n]\!]}$ be the set of agents of the MAS at time $t \in \mathbb{T} \equiv \mathbb{N}$ (the ordered set of dates), a MAS can be modeled as an undirected graph $G_t = (A_t, \omega\_t)$ whose vertices correspond to the agents and the edges represent the connections between agents.*

This graph is called the organizational graph.

Definition 2 (Neighborhood). *Let $t \in \mathbb{T}$ and $a \in A_t$, the neighborhood of a is*

$$V_t(a) = \{a_i \mid i \in [\![1, n]\!], \{a, a_i\} \in \omega\}$$

Remark 1. Suppose $t \in \mathbb{T}$ $then forall$ $a_0, a_1 \in A_t$ a_0 adjacent to $a_1 \iff a_0 \in V(a_1) \iff a_1 \in V(a_0)$

Agents determine their roles after analyzing their neighborhood. MWAC uses a specialization of the agents to select the eligible links.

Definition 3 (Role). *Let $\mathcal{R} = \{r, l, s\}$ the set of roles, we define $role : A_t \to \mathcal{R}$ as being the function assigning a role to an agent.*

Definition 4 (Group organization correctness). *The organization inside the groups is considered correct if and only if the following properties are met:*

Let $t \in \mathbb{T}$,
1. $\forall\ a \in A_t\ role(a) = s \iff \exists!\ b \in V(a)\ role(b) = r$

This organization allows the characterization of groups composed by a representative agent together with the agents of its neighborhood. It is worth noting that:

1. each representative agent determines a unique group;
2. each link agent belongs to at least two groups;
3. every simple member belongs to a unique group.

The management of messages is assigned to representative agents that communicate using link agents. An example of this organization is given in Fig. 1.

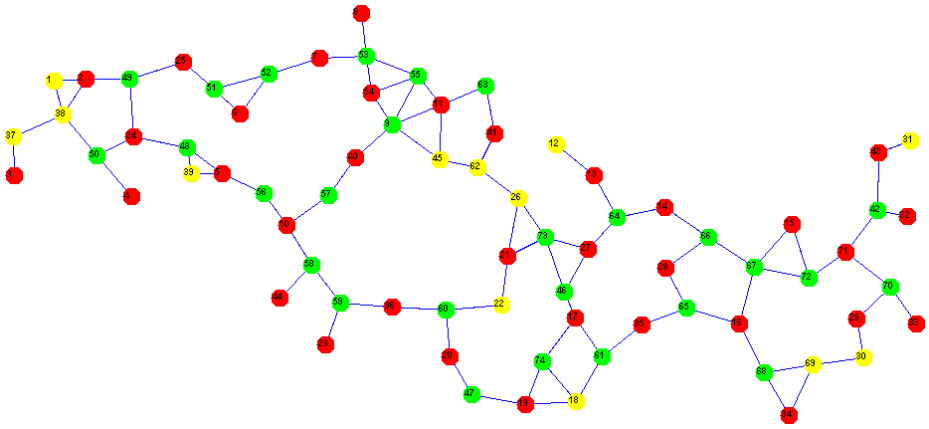

Fig. 1. Simulation of an MEAS executing MWAC
Red = representative, Green = Link, Yellow = Simple member

Note 1. When an agent changes role, it broadcasts its new role and groups to its neighborhood.

3.2 Approach Validation

Validation of the self-organization is done in three steps, where each step depends on the validation of the previous one:

1. Stability validation
2. Groups validation
3. Connectivity validation

Stability issues have to be resolved first. The organization can not be fixed just once, it needs to be re-organized, for example to allow agents with low energy level to be replaced. However, the monitoring system should wait for a given time (which depends on the specification) to allow the system to first organize itself or re-organize. This time represents the time given to the system to re-organize. Should it exceed this time, an error will be sent to the operator.

Groups validation is done through the verification of the three properties of Definition 4.

To study connectivity, let's first define an undirected graph $G'_t = (A'_t, \omega')$ at time $t \in \mathbb{T}$ where

$$\forall\ t \in \mathbb{T}\ A'_t = \{a_i \mid i \in [\![1, n]\!],\ role(a_i) = r\}$$
$$\omega' = \{\{a_i, a_j\} \mid i, j \in [\![1, n]\!],\ \exists\ a \in A_t, \{a_i, a\}, \{a_j, a\} \in \omega,\ role(a) = l\}$$

The graph represents the groups in the organization. Hence, if the graph is connected, the groups are all connected. If not, there is an error in the organization. An example of organization modelling is given in Fig. 2. This graph is called the *groups graph*.

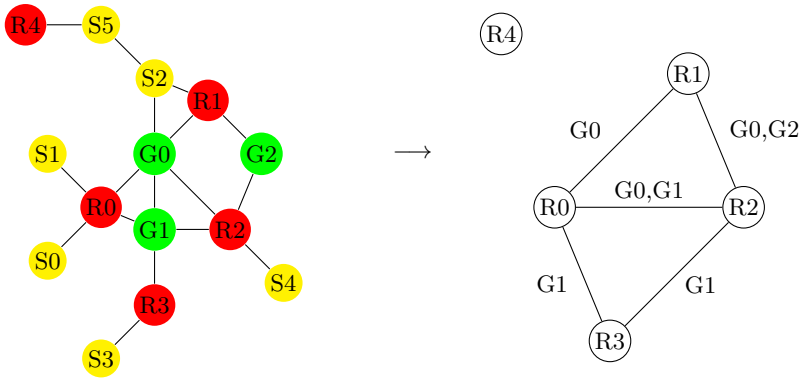

Fig. 2. Example of organization modelling
Red = representative, Green = Link, Yellow = Simple member

The validation approach As the SuT is a MEAS, several constraints, mainly due to the embedded dimension, have been considered during the design of the validation approach:

C0: The SuT has a finite amount of energy, so the validation approach must not to put any strains on it.
C1: The SuT is physically distributed, so the validation tool needs to be able to monitor agents on a possibly wide area.
C2: The SuT can have a huge number of agents, so the validation tool needs to be easily scalable.

C0 makes it necessary to only listen to the agent communications and see agents as black boxes. C1 forces the monitoring system to be distributed to cover the SuT. At last, C2 requires the monitoring system to be scalable, since the organization and the group graph can be very large. A solution is to use decentralized computation, the part of the monitoring system will only know a part of the graph and cooperate to validate the whole graph but without one entity knowing the totality of each graph.

The validation approach is therefore a degraded multi-embedded-agent system where the agents are not highly adaptable and where only a fixed plane organization is used. Also, the agents only have basic behavior since they only have to detect errors and not correct them. Each agent knows only a part of the system and may or may not exchange their knowledge to validate the whole system. The cooperation between the testing agents should be minimized; thus lowering the number of messages needed to exchange and so minimizing the impact of the testing system on the SuT.

In the following, we will make the hypothesis that the monitoring system is fully covering the SuT, each agent of the SuT is a neighbor of at least one testing agent.

Stability validation To achieve this task, the testing agents do not need to cooperate. Every agent will run the algorithm below.

Algorithm 1: Stability validation

Input: d0, d1 two durations defined from the specifications
`/* d0 is the maximum duration for one (re-)organization */`
`/* d1 is the deadline for the system to be stable */`
Output: An error can be detected. The Sut can be considered stable. The Sut graph is constructed

```
1  while Testing agent is alive do
2      when first organization message is received from sut do
3          t ⟵ start_timer()
4          sut_graph ⟵ add_message_info()
5      when organization message is received from sut do
6          if t < d0 then
7              sut_graph ⟵ add_message_info()
8          else if d0 < t < d1 then
9              trigger_error()
10         else
11             t ⟵ start_timer()
12             sut_stable ⟵ false
13     when t > d0 do
14         sut_stable ⟵ true
```

Groups validation As the monitoring system relies on listening the communication inside the SuT, and the MWAC agents do not send their position, the monitoring system can not compute their neighborhood. It could be done by triangulation with the testing agents but this would increase significantly complexity of the monitoring system and generate a huge amount of messages to exchange between the monitoring agents.

Testing the algorithm of group formation needs to be done beforehand.

Connectivity validation This task is the most complex one and cooperation between testing agents is necessary. We need to define a distributed decentralized algorithm to determine if the SuT graph is connected when no agent can construct the whole graph (see Algorithm 2).

Note 2. Agent behavior is mostly reactive, so all the algorithms will run simultaneously. Consequently, the `when` statement from different algorithms may refer to the same event.

The main difficulty arises when a validation agent computes two components as being not connected. Indeed, to construct the path between the two components, the validation agent will broadcast a first request and the receiving agents will then broadcast every possible path until a path is created between the two components. This is done by sending a message containing the components and, for each known group not in the received components, if a path exists between one of the agent of the components, a message is broadcast. This message is the start of a possible path. When an agent is able to connect the two components using its knowledge and the knowledge accumulated by the previous agents, it will send a response to the first requester.

As cooperation between agents requires message exchanging, we need to define the messages structure.

Message structure The messages are formed as shown in Fig. 3.

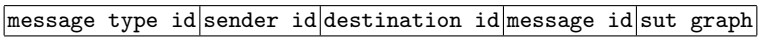

Fig. 3. Message structure

Where:

`message type id` indicates the type of the message, see next paragraph from message type description
`sender id` indicates the id of the *first* agent which sent the message
`destination id` is most of the time a broadcast id. It is used in the `response` type message (see next paragraph for message type description)
`message id` is used to differentiate messages from the same sender
`sut graph` is the known SuT graph of the sender

Algorithm 2: Connectivity validation

```
Data: d a duration
/* d is maximum duration an agent waits for a response before
   considering that there is no path between its connected
   components and triggering an error                          */
Result: An error can be detected, otherwise the organization is considered
        valid until the next re-organization
1  while Testing agent is alive do
2      when groups are valid do
3          share_sut_graph_with(neighborhood)
4      when receiving request from neighbor do
5          if is_connected(sut_graph + neighbor_graph) then
6              send_response_to(neighbor, sut_graph)
7          else
8              foreach connected agent to neighbor_graph do
9                  send_request_to(neighborhood, neighbor_graph + connected
                       agent)
10     when receiving sharing message from neighbor do
11         if not is_connected(sut_graph + neighbor_graph) then
12             send_request_to(neighborhood, sut_graph + neighbor_graph)
13             t ⟵ start_timer()
14     when receiving response from neighbor do
15         if response_destination = me and is_connected(sut_graph +
               response_graph) then
16             add_to_sut_graph(response_graph)
17             t ⟵ 0
18         else
19             send_response_to(response_destination, response_graph)
20     when t ¿ d do
21         trigger_error()
```

Message types There are four message types:

`sharing` is used to first share the SuT graph with the agent neighborhood, resulting in overlapping graphs which are easier to test.

`request` is used when the SuT graph of an agent is not connected, it will broadcast a request waiting for an other agent to respond with a path between its connected sub-graphs. The `sender id` is never changed. To know to which an agent should respond, it will have to keep the couple `sender id, message id` linked with the id of the agent which really sent the message.

`response` is used to respond to a request. The `destination id` must correspond to the requester's id so the response can be forwarded back to it.

`wait` is used to help the agents synchronize. Every agents may not reach the third step at the same time, the `wait` message is used when a request is sent

to an agent in the first two steps. It will tell the requester to wait a response without starting the timer and put the requester in waiting mode. An agent in waiting mode receiving a request to which it does not know the response will also send a `wait` message. This does not create deadlock since even if an agent never reaches the third step it will itself eventually timeout.

Note 3. To avoid the count to infinity problem [9], every received graph will be linked with the agent sending them.

Experimentation & results As first results, we were able to detect a known error in the MWAC self-organization process: as it can be seen in the Fig. 1, the group of agents 3 and 37 (left of the picture) is not connected to the other groups. This error can happen when the density of agents is low. With a hexagonal covering of the SuT shown in Fig 4 we could detect this error in several layouts and with a population ranging from low population with a layout inclined to make the error occur to randomized layouts of population up to hundreds of tested agents.

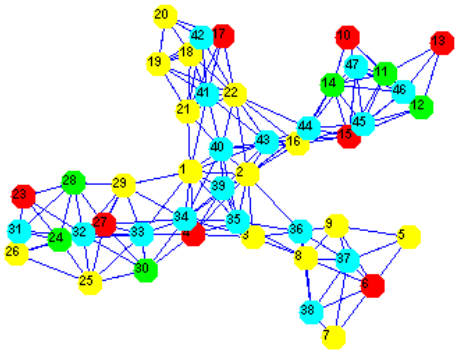

Fig. 4. Example of use of our approach
Red = representative, Green = Link, Yellow = Simple member, Cyan = Validation agent

From all tested layouts, we detected the known error but there were also false positives when the path between components observed as disconnected could not be found by the validation system even if it existed. This is due to the solution used to reduce the number of broadcast messages. When the layout and the self-organization algorithm lead to components with very few and long paths between them, the computation of those paths may be impaired by the solution used to prevent broadcast tempests, leading to the detection of an error when there is none.

A solution to this problem would be to add some routing capabilities to the validation agents to reduce the number of broadcast messages and enhance the path finding between components.

4 Conclusion and future Work

We motivated the need of specific methods to test self-organizations in embedded multi-agent systems and highlighted the differences between MEAS and software MAS. After a presentation of methods that could be adapted to embedded multi-agents, we proposed applying run-time monitoring to validate organization as they are changed by the agents. Last, we presented a case study to apply this method, the associated challenges and some encouraging results.

Future work includes working on the distributed algorithm computing whether or not the routing graph is connected, thoroughly validate the testing approach with different varieties of layouts and high population SuT. Finally, we are aiming to apply this approach to other similar MEAS.

References

1. Abbass, H.A., Harvey, J., Yaxley, K.: Lifelong testing of smart autonomous systems by shepherding a swarm of watchdog artificial intelligence agents. CoRR **abs/1812.08960** (2018)
2. Barnier, C., Aktouf, O., Mercier, A., Jamont, J.: Toward an embedded multi-agent system methodology and positioning on testing. In: 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) (2017). https://doi.org/10.1109/ISSREW.2017.57
3. Bulling, N., Dastani, M., Knobbout, M.: Monitoring norm violations in multi-agent systems. In: Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems (2013)
4. De Wolf, T., Holvoet, T., Samaey, G.: Engineering self-organising emergent systems with simulation-based scientific analysis. In: Proceedings of the Fourth International Workshop on Engineering Self-Organising Applications (2005)
5. Di Marzo Serugendo, G., Gleizes, M.P., Karageorgos, A.: Self-organization in multi-agent systems. The Knowledge Engineering Review **20** (2005). https://doi.org/10.1017/S0269888905000494
6. Dikenelli, O., Gürcan, Ö., Çakırlar, ., Bora, Ş.: Ratkit: A repeatable automated testing toolkit for agent-based modeling and simulation. In: the 15th International Workshop on Multi-Agent Simulation (MABS 2014), 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014) (2014)
7. Eberhardinger, B., Anders, G., Seebach, H., Siefert, F., Knapp, A., Reif, W.: An approach for isolated testing of self-organization algorithms. In: Software Engineering for Self-Adaptive Systems III. Assurances (2017). https://doi.org/10.1007/978-3-319-74183-3_7
8. El Fallah-Seghrouchni, A., Degirmenciyan Cartault, I., Marc, F.: Modelling, control and validation of multi-agent plans in dynamic context. In: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1 (2004). https://doi.org/10.1109/AAMAS.2004.175
9. Elmeleegy, K., Cox, A.L., Ng, T.S.E.: Understanding and mitigating the effects of count to infinity in ethernet networks. IEEE/ACM Trans. Netw. **17** (2009). https://doi.org/10.1109/TNET.2008.920874

10. Greenberg, M.S., Byington, J.C., Harper, D.G.: Mobile agents and security. IEEE Communications Magazine **36** (1998). https://doi.org/10.1109/35.689634
11. Hamani, N., Jamont, J., Occello, M., Ben-Yelles, C., Lagreze, A., Koudil, M.: A multi-cooperative-based approach to manage communication in wireless instrumentation systems. IEEE Systems Journal **12** (2018). https://doi.org/10.1109/JSYST.2017.2721220
12. Helsinger, A., Lazarus, R., Wright, W., Zinky, J.: Tools and techniques for performance measurement of large distributed multiagent systems. In: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (2003). https://doi.org/10.1145/860575.860711
13. Huhns, M.N., Stephens, L.M.: Multiagent systems and societies of agents. Multiagent systems: a modern approach to distributed artificial intelligence **1** (1999)
14. Jamont, J., Occello, M.: Meeting the challenges of decentralised embedded applications using multi-agent systems. IJAOSE **5**(1), 22–68 (2015). https://doi.org/10.1504/IJAOSE.2015.078435
15. Kerraoui, S., Kissoum, Y., Redjimi, M., Saker, M.: Matt: Multi agents testing tool based nets within nets. Journal of Information and Organizational Sciences **40** (2016). https://doi.org/https://doi.org/10.31341/jios.40.2.1
16. Lomuscio, A., Qu, H., Raimondi, F.: Mcmas: an open-source model checker for the verification of multi-agent systems. International Journal on Software Tools for Technology Transfer **19** (2017). https://doi.org/10.1007/s10009-015-0378-x
17. Meziane, F., Vadera, S.: Artificial intelligence applications for improved software engineering development: new prospects. IGI Global (2009)
18. Niazi, M.A., Hussain, A., Kolberg, M.: Verification & validation of agent based simulations using the VOMAS (virtual overlay multi-agent system) approach. CoRR **abs/1708.02361** (2017)
19. Rouff, C.: A test agent for testing agents and their communities. In: Proceedings, IEEE Aerospace Conference. vol. 5 (2002). https://doi.org/10.1109/AERO.2002.1035446
20. Rouff, C., Buskens, R., Pullum, L., Cui, X., Hinchey, M.: The adaptiv approach to verification of adaptive systems. In: Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering (2012). https://doi.org/10.1145/2347583.2347600
21. Samaey, G., Holvoet, T., Wolf, T.D.: Using equation-free macroscopic analysis for studying self-organising emergent solutions. In: 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (2008). https://doi.org/10.1109/SASO.2008.30
22. Tonn, J., Kaiser, S.: Asgard – a graphical monitoring tool for distributed agent infrastructures. In: Advances in Practical Applications of Agents and Multiagent Systems (2010)
23. Wooldridge, M., Jennings, N.R.: Intelligent agents: theory and practice. The Knowledge Engineering Review **10** (1995). https://doi.org/10.1017/S0269888900008122