2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) | 978-1-6654-1455-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/PDP52278.2021.00018

A Case Study of Run-Time Testing of Self Organizations in Multi-Embedded-Agent Systems

Arthur Baudet, Oum-El-Kheir Aktouf, Annabelle Mercier and Jean-Paul Jamont
Univ. Grenoble Alpes, Grenoble INP, LCIS, Valence France
Email: {arthur.baudet,oum-el-kheir.aktouf,annabelle.mercier,jean-paul.jamont}@lcis.grenoble-inp.fr

***Abstract*—We develop a testing approach for validating global adaptation in self-organizing multi-embedded-agent systems. These systems are complex and ever adapting, thus challenging to test. We propose an approach aiming at validating at run-time the adaptation of these systems when the entities composing them, the agents, use self-organization processes. These processes allow agents to change their organization, *i.e.*, their way of interacting, at run-time. Thus, run-time testing is needed to validate self-organization. We propose a decentralized self-organization run-time testing algorithm and we apply it, through simulation, to the Multi-Wireless-Agent Communication model.**

I. Introduction

There are many definitions of multi-agent systems (MAS) and multi-embedded-agent systems (MEAS) as they are used in many different application fields [1], [2]. From the engineering perspective [2], [3], a MAS is a complex system with more than two agents, which collaborate to achieve a global behavior. Each agent has a level of autonomy and achieves its own goal. The autonomy relates to proactivity, reactivity and ability of an agent to negotiating, convincing or being convinced by other agents. In MAS and MEAS, there is generally no central entity coordinating the agents. Consequently, the system-level decisions are distributed among agents. In a MEAS, agents are embedded systems. The embedded feature adds constraints like energy management, safety management, *etc.* [4]. These constraints need to be considered in the testing phase and also at run-time to take into account the system's dynamic behavior. As a case study, we chose the Multi-Wireless-Agent Communication (MWAC) model, which includes a self-organization process. During previous studies, we identified an error in its self-organization process which wasn't detected at design time. This was a perfect opportunity to validate a new self-organization testing approach at runtime. Testing in MEAS is quite challenging because MEAS are asynchronous complex[1] systems of intelligent agents [5].

The paper is organized as follows. In the next section we introduce related work. In Section III we describe the Multi-Wireless-Agent Communication model, a model including an identified faulty self-organization process. In Section IV, we present our approach and its application to our use case. We detail our experimentation and obtained results in Section V. Finally, we conclude this paper in Section VI.

[1]with regards to the number of variables considered and inter-dependencies between agents

II. Review of organization and self-organization testing methods in MAS

Validating a multi-agent system encompasses the validation of the agents, the agents interactions among themselves and with their environment, and the global behavior of the MAS.

To tackle the complexity of MAS, many models have been developed. The AEIO decomposition model shows clearly the main components of a MAS, which are [6]: **A**gent concerns the agent's knowledge, model and architecture; **E**nvironment deals with interactions between the MAS and its environment; **I**nteraction includes inter-agent communication features (languages, protocols...); **O**rganization allows to manage agent groups in organizations determined according to their roles. Based on this model, existing MAS testing works have been classified according to agent (A) testing works and collective features (I, O) testing works [4]. Collective features testing targets both agent interactions and organizations. Self-organization algorithms bring adaptability to the organization level. Thus, testing self-organization algorithms requires testing organizations and adaptation mechanisms at the same time. They can be seen as testing at both A and O levels in the AEIO model. Simulations are usually used to verify the self-organization process. MAS and self-organization algorithms simulations at design time are designed to determine if the MAS is designed correctly [7]. Some formal works [8] model MAS and self-organization algorithms as mathematical objects for being formally tested. These can be considered as pre-analysis approaches, yet, it seems unrealistic to model or simulate all the possible changes in the SuT environment. This is why, other works such as [9] offer tools to verify, at run-time, in an environment close to the deployment one, the behavior of the SuT. Nevertheless, run-time validation does not prevent errors from occurring and some works [10] try to support the system to prevent it from mutating in wrongful ways. A recent systematic literature review and mapping of verification in MAS [11] has shown the importance of addressing run-time verification approaches, including run-time testing, to cope with run-time features of agents and MAS (openness, reactivity and proactivity). These features may lead to unexpected behavior, so-called emergent behavior. We based our contribution on a run-time validation approach of MEAS.

III. System under test specification and modeling

The MWAC [12] model provides a self-organization process for managing communication in wireless multi-embedded

978-1-6654-1455-5/21/$31.00 ©2021 IEEE
DOI 10.1109/PDP52278.2021.00018

agent systems. Examples of implementations of MWAC can be found in [13] and [14]. In this model, self-organization is described as the ability given to each agent to choose its role wrt the roles of the agents next to it (cf. Section III-A).

A. Definition of the Multi-Agent-Wireless Communication model

MWAC defines three roles for the agents. A representative (r) manages and forwards messages of agents that are directly connected to it. It sends messages to its neighbors and responds to message forwarding requests. A link (l) enables message exchange between the representative agents 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 perform any message transfer task, unless it is the first sender or the final receiver of a message.

We define $n_t \in \mathbb{N}$, $t \in \mathbb{T} \equiv \mathbb{N}$ the ordered set of dates and $A_t = \{a_i \mid i \in [\![1, n]\!]\}$ the set of agents at a given time t.

Definition III.1 (MAS)**.** A MAS can be modeled as an undirected graph $G_t = (A_t, \omega_t)$ whose vertices correspond to the agents and edges represent the connections between agents. This graph is called the *organization graph*.

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

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

Remark. Let $t \in \mathbb{T},\ \forall\ a, b \in A_t$ a adjacent to $b \iff a \in V_t(b) \iff b \in V_t(a)$

Agents set their roles after analyzing their neighborhood and applying the Definition III.4.

Definition III.3 (Role)**.** We define $\forall\ t \in \mathbb{T},\ role_t : A_t \to \{r, l, s\}$ the function assigning a role to an agent.

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

Let $t \in \mathbb{T}$,

1) $\forall\ a \in A_t\ role_t(a) = r \iff \forall\ b \in V_t(a)\ role_t(b) \neq r$
2) $\forall\ a \in A_t\ role_t(a) = l \iff \exists\ b, c \in V_t(a)\ role_t(b) = r,\ role_t(c) = r,\ b \neq c$
3) $\forall\ a \in A_t\ role_t(a) = s \iff \exists!\ b \in V_t(a)\ role_t(b) = r$

When an agent changes its role, it shares its new role and groups with its neighborhood.

B. Self-organization errors in MWAC

We decided to use MWAC because during a previous study [15] we discovered a limit of the self-organization process of MWAC: its self-organization could lead to faulty organizations where two agents could not send messages to each other as their groups were not connected even if they were located in physically feasible communication range. This error may happen when the density of agents is low and is mainly due to a weak synchronization of the agents when they start running. Low density means that there are few agents within the range of each agent. Agents are constrained to choose the simple member or link roles, because their execution was seconds late, which is correct from Definition III.4 but leads to a disconnected system.

As we wanted to design a testing approach capable of detecting errors in autonomous decision making, it was an opportunity to use a MEAS model featuring an error in its self-organization process that wasn't detected at design time. Moreover, we could establish more error prone physical configurations (one is described in Section V-A) without having to bias the SuT agents. This helped us to overcome the unpredictability issue of the self-organization algorithm and to produce specific behaviors while simulating the SuT.

C. Error model

We derived our error model from this known error of the MWAC model: when correct groups are defined, they may not be all connected. All groups are connected through link agents, but it is possible for the organization to include disconnected groups even though the disconnected group is not physically disconnected. In this work, we focus on the connectivity issues of those groups and assume the groups are correct.

D. MWAC organization model

We establish a model where each representative is modeled as a vertex of an undirected graph and each link between two groups is modeled as an edge. As the system is open, agents can connect to or disconnect from it at run-time. Thus, vertices and edges are time dependent.

Let's define an undirected graph $G'_t = (A'_t, \omega'_t)$ at time $t \in \mathbb{T}$ as:

$$\begin{aligned} \forall\ t \in \mathbb{T} & \\ A'_t &= \{a \mid a \in A_t,\ role_t(a) = r\} \\ \omega'_t &= \{\{a, b\} \mid a,\ b \in A_t,\ \exists\ c \in A_t, \\ &\qquad \{a, c\}, \{c, b\} \in \omega_t,\ role_t(c) = l\} \end{aligned}$$

This graph represents the groups in the organization. If the graph is connected, each group is connected to every other group by either a link agents or by connected groups. If not, there is an error in the organization of the MEAS. We call this graph an organization graph.

IV. Description of the testing system

To implement our testing approach, we consider an underlying testing system composed of specific testing agents.

A. System specification

We have to consider several constraints, related to both MEAS and run-time testing:

- **C1:** The SuT is physically distributed;
- **C2:** The SuT should be sclalable;
- **C3:** To be as close to deployment as possible, no instrumentation should be done on the SuT;
- **C4:** To be as close to deployment as possible, the validation system should not put any strain on the SuT.

C1 forces the testing system to be physically distributed and deployed alongside the SuT. C2 requires the testing system

Phase 1: Wait for stability and construct the SuT local graph

TOrgTimeout is the time testing agents between two messages before considering the SuT stable.

1: **while** $T < TOrgTimeout$ **do**
2: **if** eavesdropped organization SuT message **then**
3: $T \leftarrow$ STARTTIMER( )
4: UPDATESUTGRAPH(SuT message)

SuT considered stable, the constructed SuT is ready to be studied

Fig. 1. Description of the phase 1 in testing agent behavior

to be scalable. C3 reduces the possibility for collecting information, the only source of information will be the overheard messages. C4 requires to use another communication medium and prohibits from masquerading testing agents as MWAC agents. The testing system should be transparent to the SuT. The testing agents do not have any cognitive behavior as they do not need to react to their environment, self-organize nor learn from their previous experience. As the system is decentralized, we expect the need for cooperation between testing agents but we intend to keep it minimal.

B. Decentralized testing algorithm

We define a cooperative behavior for the testing agents that allows to detect if the SuT organization graph is disconnected. This behavior includes for every testing agent, constructing a local graph of the SuT agents it can overhear, studying it and requesting more information to its neighbors if its local graph is disconnected.

To establish this behavior we assume that:

- **H1:** The testing system is connected (*i.e.*, it is modeled as an undirected graph where each agent is a vertex).
- **H2:** The communication medium used for exchanging messages between testing agents is reliable (no testing messages are lost).
- **H3:** The testing agents are located such that all the communications between SuT agents can be spied on.

A testing agent proceeds in two phases. During the first phase, each testing agent constructs its local graph while waiting for the SuT to stabilize. In the second phase, it checks the connectivity of its local graph and cooperates with its neighbors if necessary. Only messages related to the organization of the SuT are processed.

The first phase is straightforward and is described in Fig 1.

Testing agents assume that the SuT is stable after waiting for `TOrgTimeout` minutes after the last eavesdropped organization message. If an organization message is eavesdropped after that, it will mean that the SuT is reorganizing, and a new phase 1 will start over (see phase 2 on Fig. 2).

The behavior during the second phase may require cooperation between testing agents:

- First, each testing agent checks the connectivity of its local graph (**line 3**). If it is connected, the agent stands by to help other testing agents or to return back to phase 1 if a new organization message is eavesdropped, otherwise, the agent sends a request to its neighbors asking to find a path between its connected components and starts a timer.
- Upon receiving a request, the agent joins the requester's graph and its own, and verifies whether the new graph is connected (**line 9**). If it is, the agent sends an answer containing its graph, otherwise, the agent forwards the request after adding its own graph to it.
- Upon receiving an answer, the agent checks if it is the final recipient (**line 19**). If it is, the agent considers its graph connected and waits for a reorganization of the SuT, otherwise, the agent forwards the answer to the agent it forwarded the request for.
- After sending a request, if no answer is received before the timeout, the agent will issue a request to an operator.
- **At line 21,** the agent raises an error as no adequate answer was received in the given time lapse.
- **At line 23,** if a testing agent eavesdrops an organization message, it gets back to phase 1 in order to rebuild its graph and resets its timer since the detected error might be corrected by the reorganization.

The message used for the cooperation between testing

Phase 2: Checks the SuT graph
An error can be detected, otherwise the organization is considered correct until the next reorganization

$TErr$ is the maximum duration an agent waits for a response before considering that there is no path between its connected components and raising an error

1: **while** not eavesdropping SuT message **do**
2: **if not** ISCNTD($myGraph$) **then**
3: $T \leftarrow$ STARTTIMER( )
4: SNDREQ($neighborhood$, myG)
5: **else**
6: STANDBY( )
7: **when** receiving request from neighbor
8: $mergedG \leftarrow$ JOIN($neighbG$, myG)
9: **if** ISCNTD($mergedG$) **then**
10: SNDRESP($neighbor$, myG)
11: **else if not** request already sent **then**
12: SNDREQ($neighborhood$, $mergedG$)
13: **when** receiving response from neighbor
14: $mergedG \leftarrow$ JOIN($neighbG$, myG)
15: **if** respDest = me **and** ISCONCTD($mergedG$) **then**
16: ADDTOSUTGRAPH($responseG$)
17: $T \leftarrow$ RESETTIMER( )
18: STANDBY( )
19: **else**
20: SNDRESP($respDest$, $responseG$)
21: **when** $T > TErr$
22: ERROR( )
23: $T \leftarrow$ RESETTIMER( )

Gets back to phase 1

Fig. 2. Description of the phase 2 in testing agent behavior

agents is formed of five fields:

1) **Message type** indicates the type of the message:
   - **Request** is used when a testing agent's graph is disconnected. The agent will send a request to its neighborhood asking for a path of representatives linking its connected components. 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.
2) **Sender id** indicates the id of the first agent which sent the message.
3) **Destination id** is most of the time a wildcard id meaning that all the neighbors should process it. It is used in the response type message.
4) **Message id** is used to differentiate messages from the same sender.
5) **SuT graph** is the known SuT graph of the sender. It is represented as lists of representatives where one list gathers all the representatives that are known to be connected. As a request is shared throughout the system, this field will include impacted representatives of the SuT.

The main difficulty in designing phase 2 behavior is to avoid false positives. When an agent computes two components as being disconnected, it has to rely on the other agents to check if these components are really disconnected or if there is such a path that connects them. We had to define an algorithm to find if a path between two components exists when this information is distributed among testing agents that may not be directly connected. As far as we know, no such algorithm exists and thus we proposed here our own: an algorithm where the requester asks its neighborhood and its neighborhood propagates the request adding their knowledge until no new knowledge can be added or a path between the components is found. It assures to clearly distinguish between true and false positives since once a potential error is found, all the agents of the system will share their knowledge to all the other agents until either an error or a solution is found. As the testing system is completely independent from the SuT, generated testing messages do not interfere nor disturb the SuT. Optimising the testing messages load will be tackled in a future work.

C. Testing agents specifications

A testing agent has three main modules: an eavesdropping module capturing the organization messages and formatting them to be passed to the second module, the module generating the graph and analyzing it and a communication module, used to send and receive testing messages. The first module does not require much memory or computing power as its only role is to receive, parse and format SuT messages. The second module requires more computing power as it has to analyze the connectivity of a graph. The graph is represented as an adjacency matrix with each representative as a line/column, giving a use of R2 with R the number of known representative. The last module does not require a lot of computing power as it only formats, sends, and receives messages. However, it has the largest memory usage as it has to keep a table of all the forwarded requests: an entry includes the sender id, message id and the data, *i.e.*, $2 + R$ bytes with R the number of representatives in the data (up to the total number of representatives). Overall, the computing power of an agent should be enough to compute a depth-first-search using an adjacency matrix, and memory should be enough to store the adjacency matrix and a communication table containing up to $Q \times M$ entries, with Q the number of requests and M the number of testing agents. The number testing agents is directly related to the number of SuT agents and more importantly to their physical configuration and density. Thus, when all the SuT agents are near each other, only a few testing agents can be deployed, as long as they can process all the SuT messages. In a spread SuT, more testing agents may have to be deployed since all the SuT agents should be tested.

V. Experimentation

A. Experimental protocol

As a first step in the validation of the proposed testing approach and testing system, we ran simulations in which we deployed a system executing the MWAC model. We used the MASH simulator [16]. Our goal is to validate our approach with software simulations and then deploy the validation and MWAC agents as hardware and/or embedded software agents in order to be able to have a fully hardware/embedded software simulation in a close to deployment environment. As we had to detect an error in the self-organization algorithm, we couldn't inject an error within the SuT since this would restrict and bias the autonomy of the agents of the SuT. To build our test cases, we used the known error in the self-organization process of MWAC described in Section III-B.

This configuration contains one hundred and nine MWAC agents and thirty-five testing agents. We ran twenty simulations with this configuration. Then, we additionally generated thirteen other configurations from the SuT. Those configurations contained from thirty-one to one hundred and twenty-one MWAC agents and nine to twenty-six testing agents. We ran five simulations for each generated configuration. Random generation was performed by adding MWAC agents one by one and ensuring that the newly placed agent was in the neighborhood of one already placed agent (except for the first one which was randomly placed). This method tends to generate one cluster of agents and so not many errors were expected to be found (no chain of agents or small packs). For each of the eighty-five simulations, error detection was corroborated by manual comparison of the results and the state of the SuT (which errors were detected and which weren't or shouldn't have been). Each simulation was set to last seven minutes: two minutes given for the SuT to organize itself

TABLE I
SIMULATION RESULTS

Total number of simulations	85
Failed simulations	1
Successful simulations	84
Number of faulty configurations	20
Number of false positives	None (over 84 configurations)
Number of false negatives	None (over 64 configurations)

and five minutes for the testing agents to either raise an error, or enter the standby mode. During our experiments, we encountered a scalability issue in our simulation tool and had to adjust the testing system: we added a static organization where for ten testing agents, one was responsible of sending, answering and forwarding requests and the other nine sent to it their knowledge to be aggregated. This greatly reduced the number of exchanged messages at a cost of low complexity.

B. Results

One of the eighty-five simulations did not yield any result because the SuT couldn't stabilize its configuration (even after waiting for ten more minutes). On the remaining eighty-four simulations, twenty contained errors. All the detected errors could be found so no false positive were detected. On the sixty-four simulations where no errors were detected, we also couldn't find any organization errors. So, no false negatives were detected. These results are summarized in table I.

Errors were caused by the inadequate choice of roles by some agents as their algorithm only considers their direct neighbors and not the whole system. The chosen roles satisfy the rules defining the correct formation of groups (see Definition. III.4) but lead to disconnecting the groups they belong to. The example in Fig. 3, a correct organization, from another simulation, is also given as a comparison. The left organization is disconnected as the agents 51, 52, 56, 57 and 58 are simple members and so do not forward messages. Therefore, agents of groups 55 and 52 can only send messages to each other.

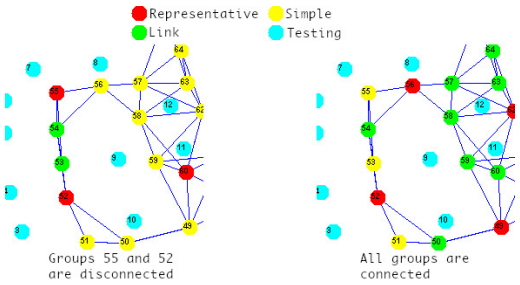

Fig. 3. Example of error in MWAC self-organization

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented multi-embedded-agent systems and the issues arising when testing them. We then focused on the test of self-organizations and showed that there was few work done in this area, even though there is a real need to test and trust self-organization processes in MEAS as it is one of their major benefits. Based on this work, we proposed a non-intrusive approach to test self-organization process in MEAS by validating at run-time their results and applied it on a case study: the Multi-Wireless-Agent communication process. The choice of this particular case study was motivated by the fact that we were aware of an error in its self-organization process. Future work includes extending the simulations and enhancing our simulator by automating the verification process and increasing data precision. Another perspective of our work consists of improving the cooperation between testing agents to reduce the number of exchanged messages and increase its scalability without having to trade-off with its precision. We also plan to work on the stability issue we detected during one of our simulations by adding a stability error to the error model of self-organization for our case study.

REFERENCES

[1] J. P. Müller and K. Fischer, *Application Impact of Multi-agent Systems and Technologies: A Survey*. Springer Berlin Heidelberg, 2014, pp. 27–53.

[2] J.-P. Jamont and M. Occello, "Meeting the challenges of decentralised embedded applications using multi-agent systems," *International Journal of Agent-Oriented Software Engineering*, vol. 5, no. 1, pp. 22–68, 2015.

[3] M. Wooldridge and N. R. Jennings, "Intelligent agents: theory and practice," *The Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, 1995.

[4] C. Barnier, O. Aktouf, A. Mercier, and J. Jamont, "Toward an embedded multi-agent system methodology and positioning on testing," in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, vol. 1, 2017, pp. 239–244.

[5] C. Rouff, "A test agent for testing agents and their communities," in *Proceedings, IEEE Aerospace Conference*, vol. 5, 2002, pp. 5–2638.

[6] Y. Demazeau, "From interactions to collective behaviour in agent-based systems," in *Proceedings of the 1st. European Conference on Cognitive Science*, 1995, pp. 117–132.

[7] J. Sudeikat and W. Renz, "A systemic approach to the validation of self–organizing dynamics within mas," in *Agent-Oriented Software Engineering IX*, vol. 5386, 2009, pp. 31–45.

[8] C. Rouff, R. Buskens, L. Pullum, X. Cui, and M. Hinchey, "The adaptiv approach to verification of adaptive systems," in *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, 2012, pp. 118–122.

[9] N. Bulling, M. Dastani, and M. Knobbout, "Monitoring norm violations in multi-agent systems," in *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, 2013, pp. 491–498.

[10] H. A. Abbass, J. Harvey, and K. Yaxley, "Lifelong testing of smart autonomous systems by shepherding a swarm of watchdog artificial intelligence agents," 2018, preprint on webpage at http://arxiv.org/abs/1812.08960. [Online]. Available: http://arxiv.org/abs/1812.08960

[11] N. Bakar and A. Selamat, "Agent systems verification : systematic literature review and mapping," *Applied Intelligence*, vol. 48, no. 5, pp. 1251–1274, 2018.

[12] J. Jamont and M. Occello, "A self-organization process for communication management in embedded multiagent systems," in *2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'07)*, vol. 1, 2007, pp. 55–58.

[13] J. Luo, L. Xu, J.-P. Jamont, L. Zeng, and Z. Shi, "Flood decision support system on agent grid: method and implementation," *Enterprise Information Systems*, vol. 1, no. 1, pp. 49–68, 2007.

[14] J.-P. Jamont, M. Occello, and A. Lagrèze, "A multiagent approach to manage communication in wireless instrumentation systems," *Measurement*, vol. 43, no. 4, pp. 489–503, 2010.

[15] A. Baudet, O.-E.-K. Aktouf, A. Mercier, and J.-P. Jamont, "Toward testing self-organizations in multi-embedded-agent systems," in *Software Engineering for Resilient Systems*, vol. 11732, 2019, pp. 97–108.

[16] J.-P. Jamont, M. Occello, and E. Mendes, "Decentralized intelligent real world embedded systems: a tool to tune design and deployment," in *Proceedings of the 11th International Conference on Practical Applications of Agents and Multiagent Systems*, vol. 7879, 2013, pp. 133–144.