IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 14, 2020, accepted October 6, 2020, date of publication November 9, 2020,
date of current version November 19, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3036668

A Metadata-Driven Approach for Testing Self-Organizing Multiagent Systems

NATHALIA NASCIMENTO[1], PAULO ALENCAR[1], (Member, IEEE), CARLOS LUCENA[2], (Member, IEEE), AND DONALD COWAN[1], (Life Member, IEEE)

[1]David R. Cheriton School of Computer Science, University of Waterloo (UW), Waterloo, ON N2L 3G1, Canada
[2]Department of Informatics, Pontifícia Universidade Catolica do Rio de Janeiro (PUC-Rio), Rio de Janeiro 22541-041, Brazil

Corresponding author: Nathalia Nascimento (nmoraesdonascimento@uwaterloo.ca)

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Ontario Research Fund (ORF) of the Ontario Ministry of Research, Innovation, and Science, and the Centre for Community Mapping (COMAP).

ABSTRACT Multiagent Systems (MASs) have multiple different characteristics, such as autonomy, and asynchronous and social features, which make these systems difficult to understand. Thus, there is a lack of procedures guaranteeing that multiagent systems once implemented would behave as desired. Determining the reliability of such systems is further complicated by the fact that current agent-based approaches may also involve non-deterministic characteristics, such as learning, self-adaptation and self-organization (SASO). Nonetheless, there is a gap in the literature regarding the testing of systems with these features. This paper presents an approach based on metadata and the publish-subscribe paradigm to develop test applications that address the process of failure diagnosis in a self-organizing MAS. The novelty of the proposed approach involves its ability to test self-organizing MAS systems in the context of local and global behavior. To illustrate the use of this approach, we developed a self-organizing MAS system based on the Internet of Things (IoT), which simulates a set of smart street lights, and we performed functional ad-hoc tests. The street lights need to interact with each other in order to achieve the global goals of reducing energy consumption and maintaining the maximum value of visual comfort in illuminated areas. To achieve these global behaviors, the street lights develop local behaviors automatically through a self-organizing process based on machine-learning algorithms.

INDEX TERMS Metadata-oriented testing, publish-subscribe, failure diagnosis, multiagent system, self-organizing, Internet of Things (IoT), machine learning, neuroevolution.

I. INTRODUCTION

Multiagent Systems (MASs) involve different characteristics, such as autonomy, asynchronous and social features, which makes these systems more difficult to understand. Thus, there is a lack of procedures guaranteeing that multiagent systems would behave as desired [1]. Further complicating the situation is the fact that current agent-based approaches may also involve non-deterministic characteristics, such as *learning*, *self-adaptation* and *self-organization* (SASO) [2], [3]. Nonetheless, there is a gap in the literature regarding the inspection of systems with these features. For example, there are few approaches to test the local interactions between agents in a self-organizing or self-adaptive MAS system and the global behavior that emerges from these interactions [4], [5]. One reason is the difficulty of specifying expected results for non-deterministic applications, especially in actual environments.

We consider here the definition of self-organizing systems that has been used by the editors of the IEEE International Conference on Self-Adaptive and Self-Organizing Systems, as follows [6]:

> Self-organizing systems work bottom-up. They are composed of a large number of components that interact according to simple and local rules. The global behavior of the system emerges from these local interactions, and it is difficult to deduce properties of the global system by studying only the local properties of its parts.

In [7], we present a preliminary version of a publish-subscribe-based architecture that was implemented[1] to make feasible the development of multi-level tests based on logging for multiagent systems. By using this platform, it is possible

The associate editor coordinating the review of this manuscript and approving it for publication was Xiaolong Li.

[1]The source of the test system is available at http://www.inf.puc-rio.br/ nnascimento/MAS-tests.html

to test the behavior of individual agents and the behavior of group of agents. However, we only showed the usability of our platform by testing a very simple MAS application - a marketplace to buy and sell books on-line. Therefore, the goal of this paper is to improve this architecture and present an approach that makes it possible to diagnose failures in a more complex MAS application, a self-organizing one.

The novelty of the proposed approach involves its ability to test self-organizing MAS systems in the context of local and global behavior. As a self-organizing MAS system enables the emergence of social features based on the behavior of individual agents, to test this kind of system our new approach is able to analyze the activities performed by single agents, the interaction among the agents and the behavior that is exhibited by the whole system. Because if an agent fails, its failure may be related to a previous and an unexpected behavior of another agent in the environment, the proposed approach enables testing not only a single agent, but also the group of agents that interact with the agent even when no messages are exchanged between the agent and the group. In addition, to test self-organizing applications, our new approach promotes the development of tests separated into two categories: global and local levels. Section V provides more details about our proposed approach.

To illustrate and evaluate the use of the proposed approach, we developed a self-organizing MAS application by using the ''Framework for the Internet of Things'' (FIoT) [3], which is an agent-based framework for the development of self-adaptive and self-organizing applications based on the Internet of Things (IoT).

This experiment is presented in Section IV. The remainder of this paper is organized as follows. Section III presents the related work. Section II presents the background, briefly describing the publish-subscribe based architecture to generate tests and the Framework for the Internet of Things (FIoT). Section V describes the approach to test self-organizing systems. Section 6 evaluates the test approach, presenting the experimental results and evaluation. The paper ends with some concluding remarks and a discussion about potential future work in Section 7.

II. BACKGROUND

In this section, we first provide a brief introduction to Multilevel-based Testing, which is one of the key concepts of our proposed approach. Next, we provide an overview of the Framework for Internet of Things (FIoT), which we use to instantiate a self-organizing MAS application. We also briefly describe some concepts involved in our proposed solution.

A. MULTILEVEL-BASED TESTING

According to Nguyen *et al.* (2009) [8], a full testing process of a multiagent system consists of five levels: unit, agent, integration (or group), system (or society) and acceptance.

Unit test tests all units of an agent, such as goals, knowledge base, plans, etc. Agent test examines the capability of a specific agent to fulfill its goal and to sense and affect the environment. Integration test verifies the interaction of agents and the interaction of agents with the environment, ensuring that a group of agents and environmental resources work correctly together. System test tests the quality properties that the intended system must reach, such as performance. Acceptance test verifies if the MAS execution meets stakeholder goals.

B. FIoT: A FRAMEWORK FOR INTERNET OF THINGS

The Framework for the Internet of Things (FIoT) [3] is an agent-based software framework to generate different kinds of applications for IoT. It is based on MAS and artificial intelligence paradigms such as neural networks and evolutionary algorithms.

The main role of FIoT is to produce MAS-based applications with decentralized, autonomous, self-organizing features. Basically, it supports the development of three types of agents: (i) Manager Agents; (ii) Adaptive Agents; and (iii) Observer Agents. The primary role of the Manager Agent is to detect new things that are trying to connect to the system and make that connection. Adaptive Agents control things at the scenario and must execute three key activities in sequence namely: (i) collect data from the thing; (ii) make decisions; and (iii) take actions. The Observer Agent examines the environment to determine if the system is meeting its global goals. See more details about these agents in [3].

C. DESIGNING SELF-ORGANIZING MAS THROUGH NEUROEVOLUTION

Evolutionary algorithms, such as genetic algorithm, is a well known approach to develop self-organizing multiagent systems [9]. It allows the emergence of features that were not defined at design-time, such as a communication system [10]. In short, the genetic algorithm is a population-based search algorithm, in which each individual is a solution in a problem space. The individuals are evaluated by using a fitness function, and the fittest individuals are selected to produce offspring of the next generation.

Nolfi *et al.* [11] describe some experiments where the behavior of agents is autonomously configured through a neuroevolution algorithm [12]. Each agent uses an artificial neural network to sense the environment and behave accordingly. To optimize their neural networks, finding the fittest configuration (e.g synaptic weights and neural architecture), Nolfi *et al.* [11] propose a genetic algorithm. Therefore, each individual of the genetic algorithm population represents a configuration of the agent's neural network. In such case, each gene of an individual may represent the strength of a connection between two neurons.

The interested reader may consult more extensive papers [9] and [3].

D. FAILURE DIAGNOSIS WITH LOGS CONTAINING METADATA ANNOTATIONS

Araújo and Staa [13] investigated common approaches for testing distributed systems. According to these authors, there are several approaches that perform diagnosis based on log collection. Nonetheless, they have some limitations, such as the need of (i) organizing logs in a centralized architecture and in an adequate time order; (ii) providing visualization tools to assist manual inspection; and (iii) increasing the log details in order to enable the tool to also diagnose the application's logic. Therefore, they presented a diagnosing mechanism based on logs of events annotated with contextual information, allowing a specialized visualization tool to filter them according to the maintainer's needs.

In their approach, each logged event records a set of properties, represented as tags. A tag is a key-value pair where the value is optional. Every event must contain a basic set of tags which are: 1) *timestamp*: used to sort all events into a single timeline; 2) *message*: a description of the event; 3)*request id*: used to identify the type of event; 4) *device*: used to identify the device that originated the event; 5) *module*: the module that triggered the notification; and 6) *line*: the line of code where the notification command was inserted.

E. RabbitMQ: PUBLISH-SUBSCRIBE PLATFORM

RabbitMQ [14] is a message-oriented middleware, which generates asynchronous, decoupling applications by separating sending and receiving data through a client and scalable server architecture. It can be easily integrated into an application to operate as a common platform to send and receive messages, maintaining messages in a safe place to live until received. RabbitMQ is a multi-platform that may be deployed in Java, C, Python, and many other programming languages. It can also be deployed in a cloud infrastructure.

By using RabbitMQ, it is possible to build a logging system based on the publish-subscribe architecture. The publisher is able to distribute log messages to many receivers, while the consumers have the possibility of selectively receiving the logs. Publisher and consumers communicate through queues. Each queue has a particular routing key that is a list of words, delimited by dots. There can be as many words in the routing key as you like, up to the limit of 255 bytes. These words can be anything, but usually they specify some features connected to the message. For example, if a developer specifies that a log message must meet the pattern "(month).(day).(deviceId).(typeLog)", the valid routing keys would be "november.11.device01.error" and "november.15.device01.info" [14].

Therefore, a message sent with a particular routing key will be delivered to all the queues that are bound with a matching binding key. However there are two important special cases for binding keys [14]:

* *(star)* can substitute for exactly one word; and

(hash) can substitute for zero or more words.

III. RELATED WORK

As discussed by Serrano *et al.* [15], there is an abundant research addressing the testing process at the agent level, while there is few approaches for testing MAS at the integration level. For example, Coelho *et al.* [16] and Koeman *et al.* [17] provide tests for a specific agent without considering it into a group.

In addition, most of the approaches for testing the interactions among a group of agents are already only based on the concept of communication sniffer, that is an agent that can intercept messages. For example, Serrano *et al.* [15], which provides an approach for testing MASs at the group level, uses ACLAnalyser [18], a tool for debugging MAS through the analysis of ACL [18] messages. Thus, by using these current test approaches, if an agent exhibits unexpected behavior (failure), a developer has to inspect this failed agent or messages exchanged between agents to find the fault that caused that failure. However, if an agent fails, its failure may be related to a previous and an unexpected behavior of another agent in the environment. This case would be a real problem to some MAS-based approaches, such as that one proposed by Malkomes *et al.* [19], which promotes the development of cooperative agents without using message communication.

In particular, there is a lack of approaches to test self-organizing MAS system [4], [20], [21]. Gardelli *et al.* [20] provide a theoretical system-oriented approach that aims at anticipating design decisions at the early MAS design stages. Bernon *et al.* [21] provide a simulation-driven approach, which allows the developer to simulate different versions of the application while designing the self-organizing agents. Eberhardinger *et al.* [4] present an approach for measuring the performance of self-organizing mechanisms at design time in order to select the best-fitting mechanism, and illustrate the proposed approach in an existing agent-based system. However, these approaches do not support the design and execution of functional tests, and fault diagnosis.

Kaddoum *et al.* [22] describe some evaluation criteria that are required to analyze self-* systems. Accordingly, designers should consider some questions to validate the well-functioning of the system and of the self-*mechanism, such as "is the system able to solve the problem for which it is conceived?" and "is the system able to self-adapt in an efficient way?". In order to investigate these questions, the authors introduce some performance and robustness metrics, such as time (e.g. the number of steps needed by agents to reach the solution), the quality of solution (i.e. functional adequacy of the designed system) and time for adaptation.

IV. APPLICATION SCENARIO: SELF-ORGANIZING STREETLIGHTS

In short, this experiment involves developing self-organizing streetlights. The overall goal of this application is to reduce the energy consumption while maintaining appropriate visibility in illuminated areas [23]. For this purpose, each streetlight was provided with ambient brightness and motion sensors, and an actuator to control light intensity. In addition, they are able to interact with each other though an wireless communicator.

Each street light is controlled by an AdaptiveAgent. We used a neuroevolutionary algorithm [23] to support the

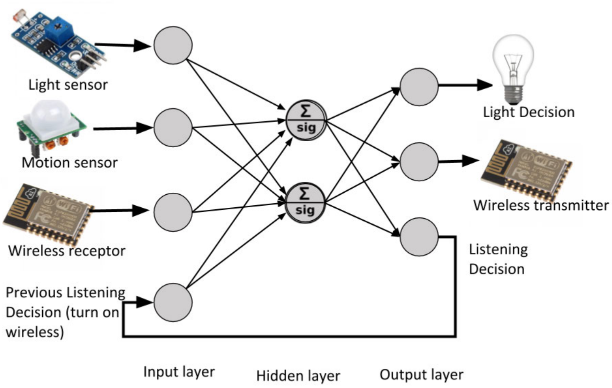


FIGURE 1. The neural network controller for street lights.

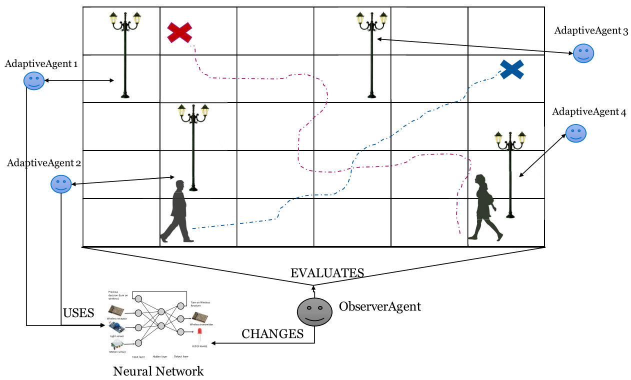


FIGURE 2. Overview of the general application architecture.

design of the street behaviors of the street lights automatically. Figure 1 illustrates the three-layer neural network with sigmoid function that is used by each streetlight to determine the communicating signals, and whether it turns on its lights or not.

As shown in Fig 2, an ObserverAgent evaluates the overall application performance and uses a genetic algorithm to optimize the AdaptiveAgents neural network.

Each solution is evaluated after the simulation ends based on energy consumption, the number of people that finished their routes before the simulation ends, and the total time spent by people moving during their trip:

$$fitness = (1.0 \times pPeople) - (0.6 \times pTrip) - (0.4 \times pEnergy) \quad (1)$$

Equation (1) shows the values to be calculated for the evaluation in which *pPeople* is the percentage of people that finished their routes before the simulation ends; *pEnergy* is the percentage of energy that was consumed by streetlights out of the maximum energy value that could be consumed during the simulation. We also considered the use of the wireless transmitter to calculate energy consumption; *pTrip* is the percentage of the total duration of people's trips out of the maximum time value that their trip could last; and *fitness* is the fitness of each candidate that encodes the proposed neural network solution.

The interested reader may find more details about the application scenario in [23].

V. TEST APPROACH: MULTILEVEL-BASED DESIGN

The main goal of a self-organizing system is to achieve global properties through local interactions. Therefore, we propose to execute several functional ad-hoc tests at local and global levels. The idea of the tests at the global level is to verify if the self-organized system solves the overall problem. If these global tests pass, we can conclude that the most basic tests (the intern ones), which were modeled at the local level, are also satisfying the functional requirements. If a global test fails, we need to understand which part of the system generated the failure, verifying the internal tests results. However, if we were executing tests at system level (performance) or evaluating how the system self-organize, we should verify the local tests independently of the global tests results. For example, according to the performance tests proposed by Kaddoum *et al.* [22] to self-* systems, we could verify whether the agents can reach the global solution by executing a desired number of steps.

We need to customize these tests according to the application. In general, at the global level, we should verify if the self-organized system is able to solve the problem for which it is conceived [22]. For example, our streetlight application has the goal of achieving an specific energy consumption target and maintaining the maximum visual comfort in illuminated areas in order to enable people to finish their routes. If the multiagent system does not solve this problem, we should investigate local tasks to understand why the self-organizing process failed, as depicted in Fig 3.

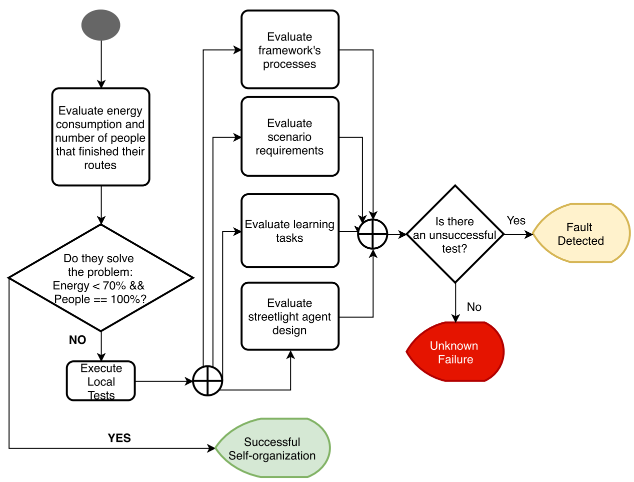


FIGURE 3. Testing steps.

In our illustrative example, we can investigate the failures generated by the tasks associated with the framework (i.e. the ManagerAgent cannot identify new streetlights at the scenario), to the agent design (i.e. streetlight agents must detect people, but they do not have motion sensors), tasks related to the application scenario (i.e. streetlights should communicate, but the distance between them is higher than the wireless range), or the tasks related to the learning algorithm execution (i.e. the ObserverAgent is executing the genetic algorithm wrongly, selecting the worst solutions to compose a new generation instead of the best solutions).

A. DESIGN AND IMPLEMENTATION: AN ARCHITECTURE BASED ON METADATA AND THE PUBLISH-SUBSCRIBE PARADIGM

We developed a publish-subscribe-based architecture as a foundation for generating different kinds of test applications for MASs at different levels. Our goal is to provide mechanisms to capture and process logs generated by agents automatically. As depicted in Fig 4, their architecture consists of three layers: MAS Application (L1), Publish-Subscribe Communication (L2), and Test Applications (L3). The Publish-Subscribe Communication layer uses the RabbitMQ platform [14] for delivering logs from agents (publishers) to be consumed by test applications (subscribers).

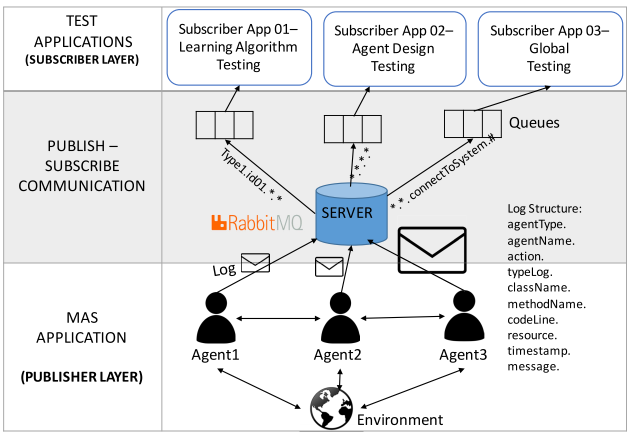


FIGURE 4. A publish-subscribe-based architecture to test MASs.

Each agent publishes logs with annotations that are composed of the following tags:

- *agentType*: the type of the agent (e.g OBSERVER, STREETLIGHT). In JADE, it refers to the name of the container where this agent lives;
- *agentName*: the name provided for the agent by the system developer/user (e.g streetlight01, streetlight02, observer01);
- *action*: the event that caused the log generation (e.g readMotionSensor, selectBestIndividuals, switchStreetLight);
- *typeLog*: types of logs (e.g error, info, warning);
- *className*, *methodName*, *codeLine*: necessary information to identify which parts of the code generated the event;
- *resource*: the main resource that has been manipulated or requested by an agent during an event execution (e.g neuralController, streetlight01Info, memory). It may be used to investigate all events that are related to a specific resource;
- *timestamp*: time that the log was created. It is used to sort all events into a single timeline [13];
- *message*: a description of the event.

Thus, a log message must meet the pattern "(agentType).(agentName).(action).(typeLog). (className). (methodName).(codeLine).(resource).(timestamp). (message)." Each application will have a set of values that each tag may assume, except the message tag is an open field.

All agents in the MAS application layer are also a TestableAgent type. As shown in Fig 5, a Testable agent extends the JADE agent. Thus, it complies with FIPA specifications. A Testable agent uses the RabbitMQ properties to send logs with annotations as messages. These logs can be published from any part of the agent's code. Via the TestableAgent class and JADE properties, some tags have their values attributed autonomously, such as agentType, agentName and timestamp.

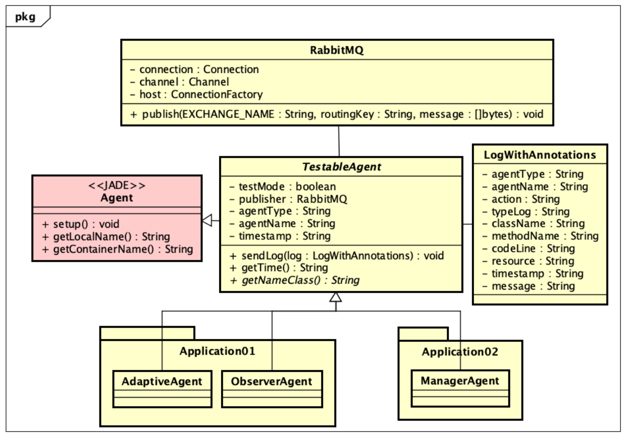


FIGURE 5. Simplified class diagram for creating testable MASs.

The RabbitMQ autonomously delivers log messages to queues according to their tags' values. As shown in Fig 4, each test application defines a binding key in order to subscribe itself to consume messages from a specific queue. For example, a test application that monitors only error logs from the Observer agent must have the binding key "Observer.*.*.error.#." Therefore, this application will consume any log with the tuples (agentType,Observer) and (typeLog,error). It is also possible to create applications that use multiple bindings. For example, if a performance test application needs to calculate the number of Adaptive agents that are connected to the system, this application will have to consume logs with different action values. Thus, it needs to consume logs with the tuples (action,connectToSystem) and (action,beDestroyed).

Test applications do not interfere on the execution of each other. Each test class extends the class RabbitMQConsumer that starts an independent process to consume messages from a specific queue. We used the Template Method Pattern [24] to model the consumeMessage method. Thus, to consume and process particular log messages, a test class must overwrite

```
public abstract class FIoTAgent extends TestableAgent {
```

```
public class ManagerAgent extends FIoTAgent {
     @Override
    public String getNameClass() {
        return "ManagerAgent";
    }

    @Override
    public boolean getTestMode() {
        return true;
    }
```

```
public class ObserverAgent extends FIoTAgent{
     @Override
    public String getNameClass() {
        return "ObserverAgent";
    }
    @Override
    public boolean getTestMode() {
        return true;
    }
```

```
public class AdaptiveAgent extends FIoTAgent {

    @Override
    public String getNameClass() {
        return "AdaptiveAgent";
    }
    @Override
    public boolean getTestMode() {
      return true;
    }
```

FIGURE 6. Making FIoT agents as Testable Agents.

```
public class LogAdaptiveAgent implements LogValues{

   public enum Action implements LogValues.Action {
        connect, waitMsgFromSmartThing,receiveMsgFromSmartThing,
        receiveInputDataFromSmartThing,getController,useControllerToGetOutput,
        sendOutputToSmartThing;
    }
    public enum TypeLog implements LogValues.TypeLog{
        INFO, WARNING, ERROR;
    }
    public enum Resource implements LogValues.Resource{
          controller, smartThing;
    }
    public enum MethodName implements LogValues.MethodName{
        setup,create,setControlLoop,setInput,getOutput,controlLoop,read,
        readInputDevice;
    }
}
```

FIGURE 7. Setting log values for each Testable FIoT agent: AdaptiveAgent.

and customize the methods getListBindingKey() and processData().

By using queues, the publisher generates a set of information elements without the need of knowing which applications will consume them. In addition, more than one application can consume the same data, but giving them different treatments. To understand more about the characteristics of RabbitMQ that we used in our approach, see https://www.rabbitmq.com/tutorials/tutorial-five-java.html (Accessed in 03/2019).

B. ADAPTING FIoT AGENTS TO BE TESTABLE AGENTS

Our first step was to allow FIoT agents to publish logs during the application execution, extending the TestableAgent class, as shown in Figure 6. Then, we set the log values that can be published by each Testable FIoT agent (see Figures 7 and 8). For example, the AdaptiveAgent can use the word 'receiveIputDataFromSmartThing' to replace the tag action in the annotated log, while the ObserverAgent can use 'startGeneticAlgorithm'.

VI. TESTS AND RESULTS

Our test approach takes two perspectives into account: the local and the global. The local perspective considers the tasks that an individual agent in the collection of streetlight agents must execute, such as collecting data, switching the light and communicating with the other agents. The global perspective takes the global tasks into account, such as verifying whether the self-organized system guarantees that people finish their routes before the simulation ends and whether the system achieves a pre-specified energy consumption target.

```
    public enum Action implements LogValues.Action {
        connect, chooseAdaptationMethod,changeControllerConfiguration,
        startExecutionWithControllerConfiguration,
        startExecutionWithoutAdaptation,
        readSimulationResults,
        calculateEnergy,calculatePeople,calculateTripDuration,calculateFitness,
        achieveEnergyTarget,achievePeopleTarget,
        reseiveMsgFromSmartThing,sendMsgToSmartThing,
        startGeneticAlgorithm, generateFirstPopulation, startNewGeneration,
        selectBestFitIndividuals, finishGeneticAlgorithm;
    }
    public enum TypeLog implements LogValues.TypeLog{
        INFO, WARNING, ERROR;
    }
    public enum Resource implements LogValues.Resource{
         solution,individual,generation,dataset,agent,smartthing;
    }
    public enum MethodName implements LogValues.MethodName{
        setup,create,observerLoop,executingNeatControl, trainingNeatControl,
        readResultSimulation,answerMessage,makeCopyGeneration;
    }
}
```

FIGURE 8. Setting log values for each Testable FIoT agent: ObserverAgent.

In this experiment, we have one test application consuming logs related to the global perspective, while we have two test apps related to the local perspective: one to monitor the ObserverAgent and its learning algorithm execution and another one to monitor the streetlight agents.

By using our proposed architecture, we created some test applications to execute functional tests at local and global levels. Within these two test levels, we also explore other perspectives, such as: (i) a framework perspective (i.e. evaluating the agent interactions generated because of the framework that we used to create the application); (ii) a learning perspective (i.e. a test application to inspect the interactions generated because of the learning algorithms); (iii) a MAS designing perspective (i.e. a test application to evaluate the sensors, actuators and analysis architecture that were selected to compose the agent), and (iv) a scenario perspective (i.e. a test application to consume the logs generated by the application scenario).

This section presents part of the test plan that we created and performed for testing the application presented in the section IV.

A. ACTIVITY DIAGRAMS FOR FIoT AGENT BEHAVIOR

In order to support the identification of functional tests, we first created activity diagrams for the street light agents and for the ObserverAgent, as depicted in Figures 9 and 10.

B. LOCAL AND GLOBAL TESTS

We executed various test cases, taking seven parameters into account: (i) level (e.g. local or global); (ii) sub-level (e.g. related to the learning, framework, agent design or scenario requirements); (iii) function (e.g. composed of a set of actions; for example, the function evaluateSolution may be composed of the actions calculateEnergy and calculateNumberPeople); (iv) procedure (e.g. a general description of the test); (v) input (e.g. a resource, a component); (vi) expected value (e.g. the result that will be produced when executing the test if the program satisfies its intended behavior); and (vii) validation method (e.g. the strategies that a tester performs to evaluate the system, comparing the program execution against expected results). Each test case execution produced several logs with meta-information annotations, which were

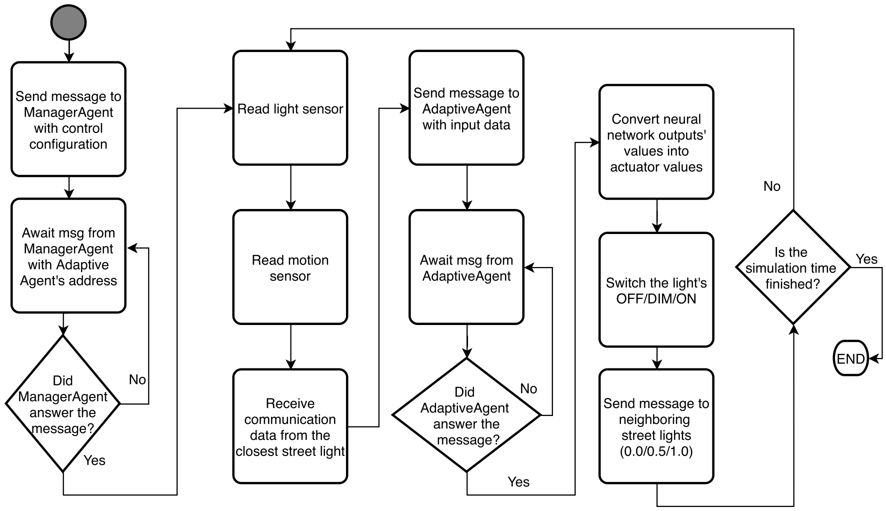


FIGURE 9. Activity diagram of the streetlights.

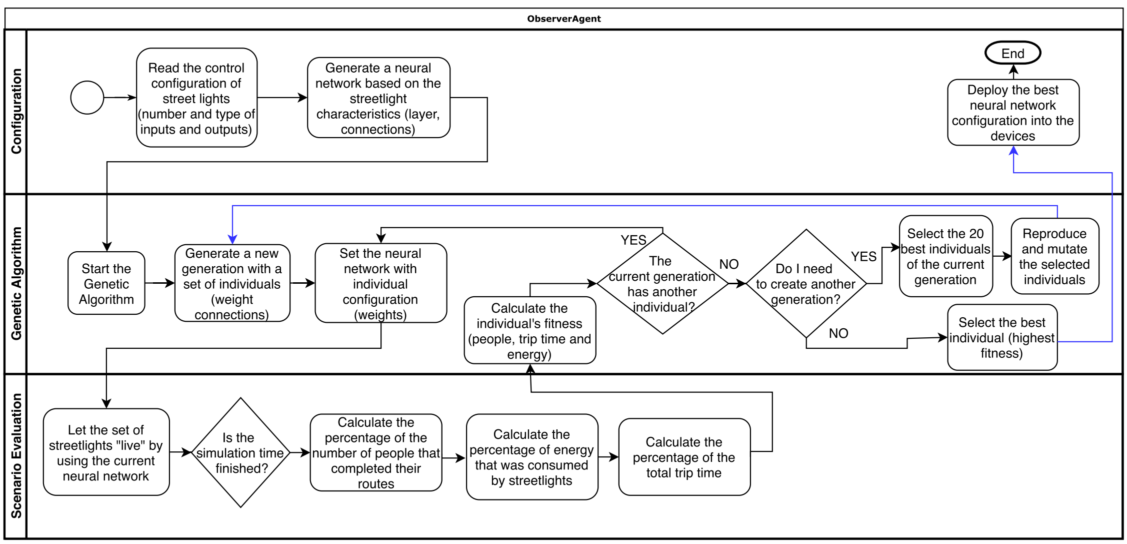


FIGURE 10. Activity diagram of the ObserverAgent.

consumed by test applications. Then, we used these logs as a validation method, as shown in Table 1.

To validate a test case, the test application must verify whether the logs are appearing in the order described in the Validation Method column. Therefore, after the developer informs the logs from the validation column, the test application will automatically create a state machine, where each state represents an action. For example, Figures 11 and 12 illustrate the state machine that were created to validate the execution of the global test "evaluate solution" and the local test "switch the light ON", respectively. As shown, the verification program defines the transition between states as a log. A transition will only occur when the expected log appears. Each state has a maximum wait time for the expected log(s). Thus, if the maximum wait time exceeds a threshold, an error linked to the current state will be generated. This situation indicates that an agent performed an unexpected behavior and the action was not successful executed. For example, if the multiagent system does not self-organize to a satisfactory solution, it will not produce the log "OBSERVER.observer.achieveEnergyTarget.#". Thus, an error linked to the state "calculateEnergy" will be generated, as depicted in Fig 11.

In order to force test failure and verify if these test applications were able to identify faults, we forced certain classes to act incorrectly during the execution of the program over some local tests. For example, to test the function "switch the light ON", we inserted a defect that makes some streetlights to go dark during the simulation. Therefore, a streetlight agent that switched its light ON on the

TABLE 1. Functional tests at local and global levels (Simplified Table).

Level	Sub-level	Func.	Procedure	Input	Expected value	Validation Method (logs sorted into a timeline)
Local	Framework	create Adaptive Agent to the streetlight	Manager Agent creates a new Adaptive Agent to the streetlight	Control configuration (number of inputs and outputs)	Adaptive Agent with the selected control	1)MANAGER. receiveMsgFromSmartThing. *.*.*.smartThing.# 2)MANAGER. createAdaptiveAgent.INFO.# 3)AdaptiveAgent.lightsAgent. connect.# 4)MANAGER. sendMsgToSmartThing.INFO.# 5)AdaptiveAgent.lightsAgent. receiveInputDataFromSmartThing.#
	Scenario	collect data	streetlight 10 (node10) reads its sensors data	streetlight's motion and light sensors, and communication input	Adaptive Agent receives data from the streetlight's sensors	1)lightContainer.node10. receiveWirelessData.# 2)lightContainer.node10. readLightSensor.# 3)lightContainer.node10. readMotionSensor.# 4)lightContainer.node10. sendMsg.*.*. msgAdaptiveAgent 5)AdaptiveAgent.lightsAgent. receiveInputDataFromSmartThing.#
	Learning	process output	AdaptiveAgent uses a neural network to process sensors data and generate output	streetlight's sensors data	Adaptive agent calculates two outputs (led and wireless data)	1)AdaptiveAgent.lightsAgent. useControllerToGetOutput.# 2)AdaptiveAgent.lightsAgent. sendOutputToSmartThing.#
	Learning	change the neural network	ObserverAgent uses an individual's genes to set the ANN weights (see subsection II-C)	an individual from the current generation	the ANN weights sequence is equal to the current individual	1)OBSERVER. chooseAdaptationMethod.# 2)OBSERVER. selectNeuralConfiguration.# 3)OBSERVER. useIndividualGenesToANN.# 4)OBSERVER. startExecutionWithController Configuration.#
	Scenario	switch the light ON	Streetlight Agent (node 10) switches the light ON	neural network's light output value is positive	node10's light sensor detects a value equal or higher than its light brightness	1)lightContainer.node10. receiveNeuralNetworkCommand.# 2)lightContainer.node10. switchLightON.# 3)lightContainer.node10. detectLight.# 4)lightContainer.lights. finishSimulation.#
Global	MAS	evaluate the selected solution	Observer Agent analyzes the energy consumption and whether everyone finished their routes during the selected solution	the best individual of the last generation	energy consumption is less than 70% and everybody finished their routes	1)OBSERVER. startExecutionWith ControllerConfiguration.# 2)OBSERVER. readSimulationResults.# 3)OBSERVER. calculateEnergy.# 4)OBSERVER. achieveEnergyTarget.# 5)OBSERVER. achievePeopleTarget.# 6)OBSERVER. calculateFitness.#

previous execution, did not detect brightness on the current execution and failed. As the test application did not receive the log "LIGHT.light1.detectLight.info.#", its state machine indicated a failure in the state "switchLightON," as depicted in Fig 13. Considering that a person can only move if his current and next positions are not completely dark, it interferes on the overall solution evaluation. Consequently, if a person does not finish his or her route, the test at the global level will also fail. Fig 14 depicts the logs that were generated by agents while this situation was being executed. Fig 15 depicts the global test that was executed without this defect.

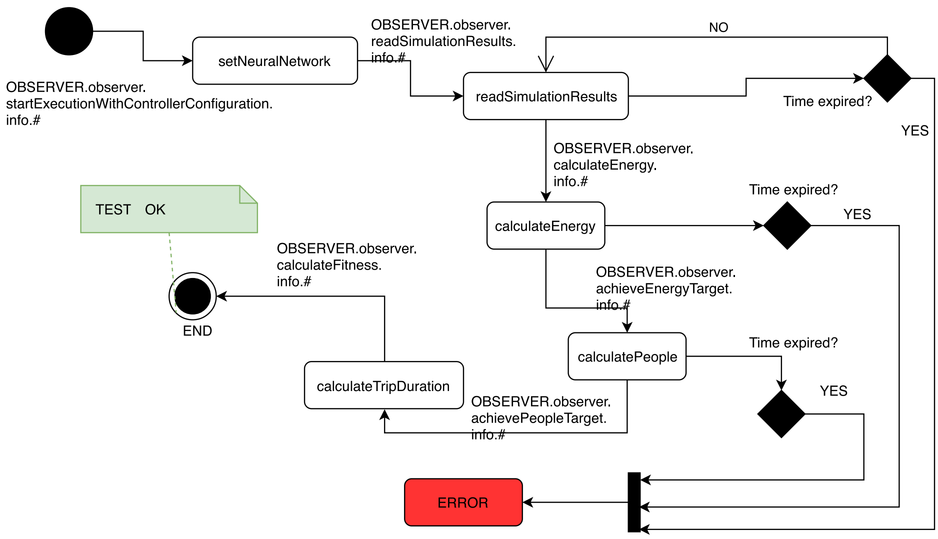


FIGURE 11. Simplified state machine for verifying test cases generated for the function "evaluate selected solution".

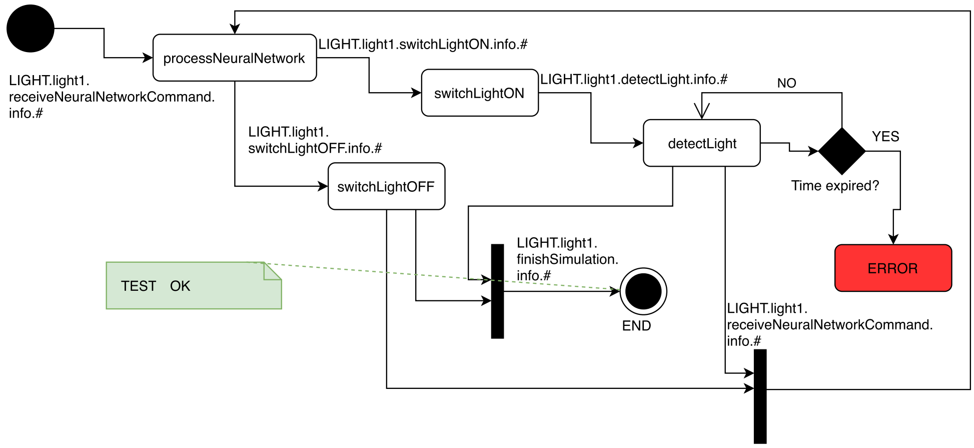


FIGURE 12. Simplified state machine for verifying test cases generated for the functions "switch the light ON" and "switch the light OFF".

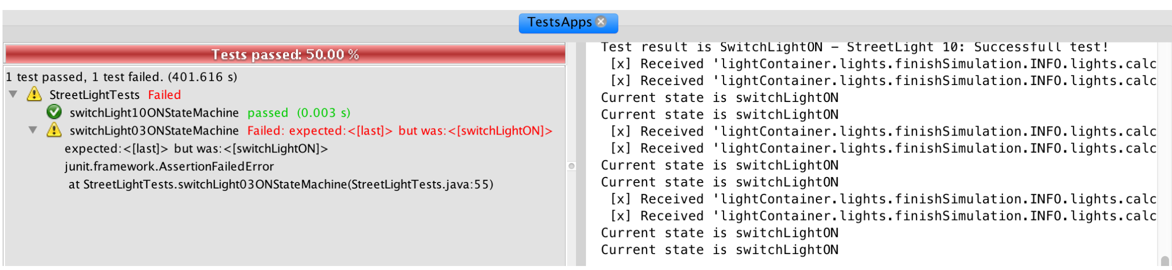


FIGURE 13. Executing the state machine to test the function "switch the light ON": failure generated between states "switchLightON" and "detectLight" - specific log was not consumed.

Using our proposed solution, a test application can automatically select those logs from different agents that are essential for a specific test case and present them sorted in a single timeline. As a result, the interface depicted in Fig 16 shows just the logs that were consumed by the evaluation test application according to this binding key list. In addition, all logs are organized in a single timeline. As shown, not all logs depicted in Fig 17 were presented in this interface, but only the logs relevant to the execution of this test case. Thus, we were able to verify these logs in order to find the fault that generated the failure indicated by the state machine.

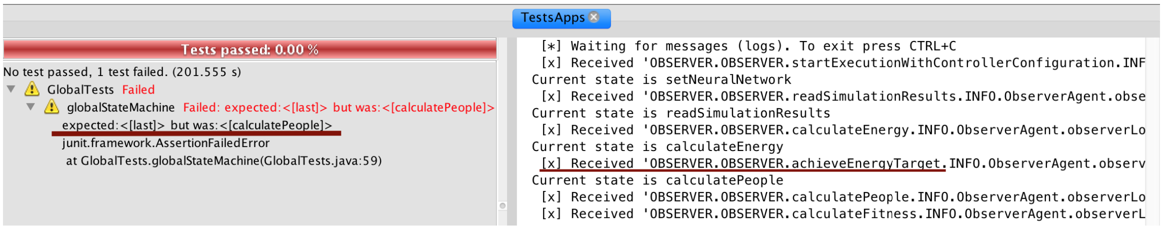


FIGURE 14. Executing the state machine to test the evaluation solution: failure generated between states "calculatePeople" and "calculateTripDuration" - because the machine did not receive the log that indicates that everyone finished their routes during the selected solution.

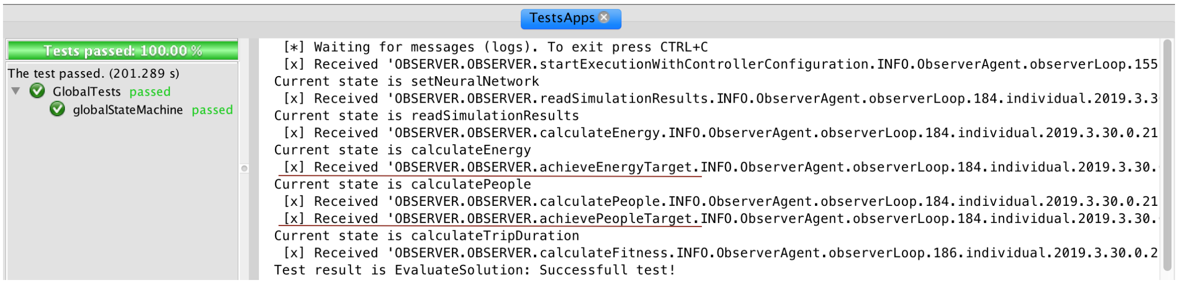
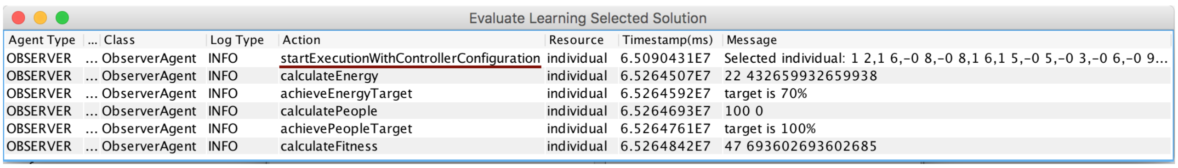


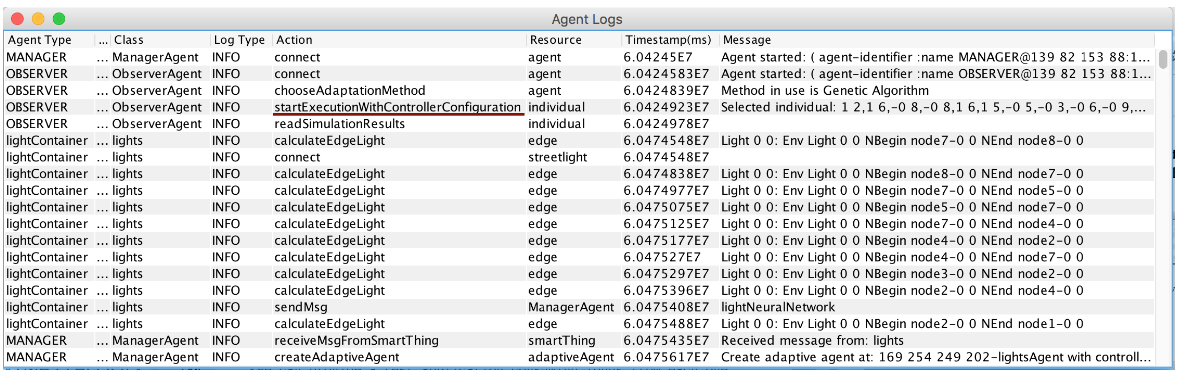
FIGURE 15. Executing the state machine to test the evaluation solution.



Evaluate Learning Selected Solution

Agent Type	... Class	Log Type	Action	Resource	Timestamp(ms)	Message
OBSERVER	... ObserverAgent	INFO	startExecutionWithControllerConfiguration	individual	6.5090431E7	Selected individual: 1 2,1 6,-0 8,-0 8,1 6,1 5,-0 5,-0 3,-0 6,-0 9...
OBSERVER	... ObserverAgent	INFO	calculateEnergy	individual	6.5264507E7	22 432659932659938
OBSERVER	... ObserverAgent	INFO	achieveEnergyTarget	individual	6.5264592E7	target is 70%
OBSERVER	... ObserverAgent	INFO	calculatePeople	individual	6.5264693E7	100 0
OBSERVER	... ObserverAgent	INFO	achievePeopleTarget	individual	6.5264761E7	target is 100%
OBSERVER	... ObserverAgent	INFO	calculateFitness	individual	6.5264842E7	47 693602693602685

FIGURE 16. Subscribing to receive only logs related to the evaluation solution testing.



Agent Logs

Agent Type	... Class	Log Type	Action	Resource	Timestamp(ms)	Message
MANAGER	... ManagerAgent	INFO	connect	agent	6.04245E7	Agent started: (agent-identifier :name MANAGER@139 82 153 88:1...
OBSERVER	... ObserverAgent	INFO	connect	agent	6.0424583E7	Agent started: (agent-identifier :name OBSERVER@139 82 153 88:1...
OBSERVER	... ObserverAgent	INFO	chooseAdaptationMethod	agent	6.0424839E7	Method in use is Genetic Algorithm
OBSERVER	... ObserverAgent	INFO	startExecutionWithControllerConfiguration	individual	6.0424923E7	Selected individual: 1 2,1 6,-0 8,-0 8,1 6,1 5,-0 5,-0 3,-0 6,-0 9,...
OBSERVER	... ObserverAgent	INFO	readSimulationResults	individual	6.0424978E7	
lightContainer	... lights	INFO	calculateEdgeLight	edge	6.0474548E7	Light 0 0: Env Light 0 0 NBegin node7-0 0 NEnd node8-0 0
lightContainer	... lights	INFO	connect	streetlight	6.0474548E7	
lightContainer	... lights	INFO	calculateEdgeLight	edge	6.0474838E7	Light 0 0: Env Light 0 0 NBegin node8-0 0 NEnd node7-0 0
lightContainer	... lights	INFO	calculateEdgeLight	edge	6.0474977E7	Light 0 0: Env Light 0 0 NBegin node7-0 0 NEnd node5-0 0
lightContainer	... lights	INFO	calculateEdgeLight	edge	6.0475075E7	Light 0 0: Env Light 0 0 NBegin node5-0 0 NEnd node7-0 0
lightContainer	... lights	INFO	calculateEdgeLight	edge	6.0475125E7	Light 0 0: Env Light 0 0 NBegin node7-0 0 NEnd node4-0 0
lightContainer	... lights	INFO	calculateEdgeLight	edge	6.0475177E7	Light 0 0: Env Light 0 0 NBegin node4-0 0 NEnd node2-0 0
lightContainer	... lights	INFO	calculateEdgeLight	edge	6.047527E7	Light 0 0: Env Light 0 0 NBegin node4-0 0 NEnd node7-0 0
lightContainer	... lights	INFO	calculateEdgeLight	edge	6.0475297E7	Light 0 0: Env Light 0 0 NBegin node3-0 0 NEnd node2-0 0
lightContainer	... lights	INFO	calculateEdgeLight	edge	6.0475396E7	Light 0 0: Env Light 0 0 NBegin node2-0 0 NEnd node4-0 0
lightContainer	... lights	INFO	sendMsg	ManagerAgent	6.0475408E7	lightNeuralNetwork
lightContainer	... lights	INFO	calculateEdgeLight	edge	6.0475488E7	Light 0 0: Env Light 0 0 NBegin node2-0 0 NEnd node1-0 0
MANAGER	... ManagerAgent	INFO	receiveMsgFromSmartThing	smartThing	6.0475435E7	Received message from: lights
MANAGER	... ManagerAgent	INFO	createAdaptiveAgent	adaptiveAgent	6.0475617E7	Create adaptive agent at: 169 254 249 202-lightsAgent with controll...

FIGURE 17. Subscribing to receive logs from all agents.

C. TEST RESULTS

As shown in Table 1, we executed some functional tests at local and global levels. By using state machines, the test applications were able to validate these test cases by comparing the logs consumed from the MAS publisher against the logs listed in the "Validation Method" column. In addition, we also conducted some tests by inserting software failures and verifying if our test software could be useful for detecting these faults. As a result, after the state machine had indicated a failure, the developer could use the interface to identify the fault and reduce the diagnosis time.

VII. CONCLUSION AND PROSPECTS

We presented a promising decoupled architecture that allows a developer to execute tests simultaneously and independently while running a MAS. In addition, we provided evidence of the applicability of our proposal, using it to test a self-organizing MAS application. We showed that it is possible to develop different tests for a self-organizing multi-agent system at local and global levels by using logs containing meta-information annotations and a publish-subscribe technology.

In the following we are proposing future directions that we intend to investigate.

A. OTHER APPLICATION DOMAINS

In this paper, we described a self-organizing application in the IoT domain. But, our approach can also be applied to other application domains. For example, we may consider autonomous vehicles applications [25] or self-organizing swarm robotics [10], where the robot behavioral mechanisms are automatically generated by using a learning algorithm. Floreano *et al.* [10] describes a set of robotic agents that self-organizes to forage in an environment containing a food and a poison sources. Their overall goal is to increase the robot density around the food. Thus, these robotic agents may learn to distinguish the poison source from the food source and to signal to the other robots the food position. Therefore, we could develop a test application at the global level to evaluate if all robots are at the food source after the simulation ends. At the local level, we can evaluate the learning algorithm and the physical characteristics of the robots, such as their sensors and actuators.

B. PHYSICAL ENVIRONMENT

The self-organizing process can occur in a simulated or in a physical environment. However, many devices could be damaged if we were to use real equipment, since several configurations must be tested during the training process. Therefore, to execute the training algorithm, we decided to simulate how smart street lights behave in a fictitious neighborhood. After the training process, we can transfer the evolved neural network to physical devices and observe how they behave in a real scenario. As our approach is based on a publish-subscribe platform, it works independent on the programming language. But we need to adapt our physical streetlights to publish logs at runtime.

C. TESTING PREDICTION AND SELF-ADAPTIVE APPLICATIONS

There are other non-deterministic characteristics that have been usually associated to current MAS systems, such as learning and self-adaptation. It is possible to extend our approach to test these kinds of applications. For example, we describe in [26] a multiagent architecture to monitor fruit storage and offer predictions about shelf life. Analogously, this application has a global goal of achieving an specific target accuracy. If this system does not present a desired result to the new dataset entries, we can implement local tests to evaluate the sensors measuring the storage conditions, to test the back-propagation algorithm, and the communication among the agents.

D. TESTING SELF-ORGANIZING NEURAL NETWORKS

According to Amari [27], non supervised learning scheme is sometimes called self-organization. It occurs when a neuron modifies its weights depending only on its state and input signal, without a teacher or error signal. In such case, tests at the global level may evaluate the general purpose of the self-organizing neural network, while tests at the local level may evaluate each neuron, verifying the algorithms for encoding inputs and decoding outputs, whether the input signals received by each neuron is part of the information source, whether the output of a neuron is received as an input by another neuron, etc. In addition, we can also develop a test to consume logs from the application scenario, allowing us to create a map between context changes [28] and neural changes.

REFERENCES

[1] N. A. Bakar and A. Selamat, "Agent systems verification : Systematic literature review and mapping," *Int. J. Speech Technol.*, vol. 48, no. 5, pp. 1251–1274, May 2018.
[2] G. Di Marzo Serugendo, J. Fitzgerald, A. Romanovsky, and N. Guelfi, "A generic framework for the engineering of self-adaptive and self-organising systems," in *Proc. Dagstuhl Seminar Proc.*, 2008, pp. 1–5.
[3] N. M. do Nascimento and C. J. P. de Lucena, "FIoT: An agent-based framework for self-adaptive and self-organizing applications based on the Internet of Things," *Inf. Sci.*, vol. 378, pp. 161–176, Feb. 2017.
[4] B. Eberhardinger, H. Ponsar, D. Klumpp, and W. Reif, "Measuring and evaluating the performance of self-organization mechanisms within collective adaptive systems," in *Proc. Int. Symp. Leveraging Appl. Formal Methods*, 2018, pp. 202–220.
[5] J. Brings, M. Daun, K. Keller, P. Aluko Obe, and T. Weyer, "A systematic map on verification and validation of emergent behavior in software engineering research," *Future Gener. Comput. Syst.*, vol. 112, pp. 1010–1037, Nov. 2020.
[6] O. Babaoglu and H. Shrobe, Eds., *Ninth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2015)*. Cambridge, MA, USA: IEEE Press, Sep. 2015. [Online]. Available: http://www.saso-conference.org/
[7] N. Nascimento, C. J. Viana, A. Staa, and C. Lucena, "A publish-subscribe based architecture for testing multiagent systems," in *Proc. 29th Int. Conf. Softw. Eng. Knowl. Eng.*, Jul. 2017, pp. 1–5.
[8] C. D. Nguyen, A. Perini, C. Bernon, J. Pavón, and J. Thangarajah, "Testing in multi-agent systems," in *Proc. Int. Workshop Agent-Oriented Softw. Eng.*, 2009, pp. 180–190.
[9] V. Trianni and S. Nolfi, "Engineering the evolution of self-organizing behaviors in swarm robotics: A case study," *Artif. Life*, vol. 17, no. 3, pp. 183–202, 2011.
[10] D. Floreano, S. Mitri, S. Magnenat, and L. Keller, "Evolutionary conditions for the emergence of communication in robots," *Current Biol.*, vol. 17, no. 6, pp. 514–519, Mar. 2007.
[11] S. Nolfi, J. Bongard, P. Husbands, and D. Floreano, *Evolutionary Robotics*. Cham, Switzerland: Springer, 2016, pp. 2035–2068.
[12] K. O. Stanley, J. Clune, J. Lehman, and R. Miikkulainen, "Designing neural networks through neuroevolution," *Nature Mach. Intell.*, vol. 1, no. 1, pp. 24–35, Jan. 2019.
[13] T. P. de Araújo, R. Cerqueira, and A. von Staa, *Supporting Failure Diagnosis With Logs Containing Meta-Information Annotations* (Monografias em Ciencia da Computacao), vol. 14. Rio de Janeiro, Brazil: PUC-Rio Departamento de Informatica, 2014, p. 21.
[14] (Nov. 2020). *Rabbitmq*. [Online]. Available: https://www.rabbitmq.com/
[15] E. Serrano, A. Muñoz, and J. Botia, "An approach to debug interactions in multi-agent system software tests," *Inf. Sci.*, vol. 205, pp. 38–57, Nov. 2012.
[16] R. Coelho, E. Cirilo, U. Kulesza, A. von Staa, A. Rashid, and C. Lucena, "JAT: A test automation framework for multi-agent systems," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Oct. 2007, pp. 425–434.
[17] V. J. Koeman, K. V. Hindriks, and C. M. Jonker, "Automating failure detection in cognitive agent programs," in *Proc. Int. Conf. Auto. Agents Multiagent Syst.*, 2016, pp. 1237–1246.
[18] J. A. Botía, A. López-Acosta, and A. G. Skarmeta, "ACLAnalyser: A tool for debugging multi-agent system," in *Proc. 16th Eur. Conf. Artif. Intell. (ECAI)*. Valencia, Spain: IOS Press, 2004, pp. 967–968.
[19] G. Malkomes, K. Lu, B. Hoffman, R. Garnett, B. Moseley, and R. Mann, "Cooperative set function optimization without communication or coordination," in *Proc. 16th Int. Conf. Auto. Agents Multiagent Syst.*, 2017, pp. 1–5.

[20] L. Gardelli, M. Viroli, and A. Omicini, "On the role of simulations in engineering self-organising MAS: The case of an intrusion detection system in TuCSoN," in *Proc. Int. Workshop Eng. Self-Organising Appl.* Springer, 2005, pp. 153–166.

[21] C. Bernon, M.-P. Gleizes, and G. Picard, "Enhancing self-organising emergent systems design with simulation," in *Proc. Int. Workshop Eng. Societies Agents World*, 2006, pp. 284–299.

[22] E. Kaddoum, C. Raibulet, J.-P. Georgé, G. Picard, and M.-P. Gleizes, "Criteria for the evaluation of self- systems," in *Proc. ICSE*, 2010, pp. 29–38.

[23] N. Moraes do Nascimento and C. J. P. de Lucena, "Engineering cooperative smart Things based on embodied cognition," in *Proc. NASA/ESA Conf. Adapt. Hardw. Syst. (AHS)*, Jul. 2017, p. 15.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *Proc. Eur. Conf. Object-Oriented Program.*, 1993, pp. 406–431.

[25] L. Caetano, M. Viana, E. Cirilo, P. Alencar, and C. Lucena, "Using governance and adaptive normative multiagent systems for dynamic vehicle platoon formation," in *Proc. IEEE Int. Conf. Big Data*, Dec. 2019, pp. 3770–3774.

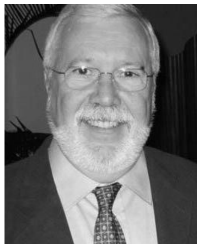
[26] N. M. D. Nascimento, C. J. P. D. Lucena, and H. Fuks, "Modeling quantified Things using a multi-agent system," in *Proc. IEEE/WIC/ACM Int. Conf. Web Intell. Intell. Agent Technol. (WI-IAT)*, Dec. 2015, pp. 26–32.

[27] S.-I. Amari, "Mathematical foundations of neurocomputing," *Proc. IEEE*, vol. 78, no. 9, pp. 1443–1463, 1990.

[28] N. Nascimento, P. Alencar, C. Lucena, and D. Cowan, "A context-aware machine learning-based approach," in *Proc. 28th Annu. Int. Conf. Comput. Sci. Softw. Eng.*, 2018, pp. 40–47.

NATHALIA NASCIMENTO received the bachelor's degree in computer engineering from the State University of Feira de Santana (UEFS), in 2013, and the master's degree in informatics and the Ph.D. degree in software engineering from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), in 2015 and 2019, respectively. She is currently a Postdoctoral Researcher with the University of Waterloo. Her research interests include multiagent systems, software reuse, machine learning, and the Internet of Things.

PAULO ALENCAR (Member, IEEE) is currently a Research Professor with the David R. Cheriton School of Computer Science, University of Waterloo, and an Associate Director of the Computer Systems Group (CSG). He received international research awards from organizations, such as Compaq and IBM. He has published over 200 refereed publications and has been a member of program committees of numerous highly-regarded conferences and workshops. His recent research in information technology and software engineering has focused on high-level software architectures, design, components, and their interfaces; software frameworks and application families; software processes, automated workflows and work graphs, and evolution; Web-based approaches and applications; open and big data applications; context-aware and event-based systems; software agents; machine learning; cognitive chatbots; artificial intelligence; and formal methods. He has been a Principal or a Co-Principal Investigator in projects supported by NSERC, ORF-RE, IBM, SAP, CITO, CSER, Bell, and funding agencies in Canada U.S., Brazil, Germany, and Argentina. He is also a member of the Association of Computing Machinery (ACM), the Institute of Electrical and Electronic Engineers (IEEE), an Association for the Advancement of Artificial Intelligence (AAAI), the Waterloo Water Institute (part of the Global Water Futures initiative), and the Waterloo Artificial Intelligence Institute.



CARLOS LUCENA (Member, IEEE) received the B.Sc. degree from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil, in 1965, the M.Math. degree in computer science from the University of Waterloo, Canada, in 1969, and the Ph.D. degree in computer science from the University of California at Los Angeles, in 1974. He has been a Full Professor with the Departamento de Informatica, PUC-Rio, since 1982. He is a member of the IEEE Computer Society and various other scientific organizations. He received many international awards, such as the ACM Fellow (in 2013) for contributions in software engineering and multiagent systems, and for computer science leadership in Brazil.

DONALD COWAN (Life Member, IEEE) is currently a Distinguished Professor Emeritus of computer science with the University of Waterloo and the Director of the Computer Systems Group. He has made contributions to computer science in areas, such as computer science, software engineering, and complex applications. He is the author and coauthor or an editor of over 300 refereed articles and 17 books in computer/communications, software engineering, education, environmental information systems, and mathematics. He has supervised over 120 graduate students and postdoctoral fellows. His group has developed over 80 Web-based and mobile software systems for many application areas. The University of Waterloo recognized his contributions to development of graduate students by presenting him with the Award of Excellence in Graduate Supervision. He was recognized for his research and support of the development of computer science in Brazil by being awarded the National Order of Scientific Merit (Grand Cross), the country's highest civilian scientific honour, by the President of Brazil. In 2009, he received the Waterloo Award, the City of Waterloo's highest civic honour, for his contributions to the City of Waterloo. In 2010, he was named a Distinguished Scientist by the Association for Computing Machinery. In 2011, he received a Doctor of Science (Honoris Causa) from the University of Guelph for his contributions to Computer Science and Software Engineering.