Monitoring and Diagnosis of Faults in Tests of Rational Agents based on Condition-action Rules

Francisca Raquel de V. Silveira, Gustavo Augusto L. de Campos and Mariela I. Cortés
State University of Ceará (UECE), Fortaleza, Ceará, Brazil

Keywords: Test Rational Agents, Diagnosis of Faults, Performance Evaluation, Test Monitoring.

Abstract: In theoretical references available to guide the design of agents, there are few testing techniques to validate them. It is known that this validation depends on the selected test cases, which should generate information that identifies the components of the agent tested that are causing unsatisfactory performance. In this paper, we propose an approach that aims to contribute to the testing of these programs, incorporating the ProMon agent in the testing process of rational agents. This agent monitors the testing and diagnosis of faults present in a tested agent, identifying the subsystem information-processing agent that is causing the faults to the designer. The first experiments are aimed at evaluating the approach by selecting test cases for simple reactive agents with internal states and in partially observable environments.

1 INTRODUCTION

An Agent is an entity capable of perceiving their environment by means of sensors and act in this environment through actuators. The behavior of an agent can be described by the function of the agent, capable of mapping any sequence of specific perceptions for an action. This function of the agent is implemented concretely by an agent program, which is executed in an adequate architecture. Ideally, rational agents should act in order to reach the best-expected outcome, evaluated according to a performance measure (Russell and Norvig, 2013).

Since agent-based systems are increasingly assuming several areas as patient care, battlefield simulation, intrusion detection, games, etc., guarantee the correct operation of these systems need to be given to users. This requires an investigation of structures of software engineering, including requirements engineering, architecture and testing techniques to provide suitable development processes and software verification (Nguyen et al., 2012).

Although there are some efforts to support the development of agent-based systems, little has been done toward proposing methods and techniques to test the performance of these systems (Nguyen et al., 2009). The testing of rational agents involves the adaptation and combination of already existing techniques for software testing, in order to detect different faults and to make the software agents more reliable (Houhamdi, 2011a; Houhamdi, 2011b).

One of the possible reasons for the absence of techniques for a testing agent is the difficulty to applying the techniques that are able to guarantee the reliability of these systems, due to the peculiar properties and the specific nature of software agents, which are designed to be distributed, autonomous and decision-making, which cannot be reproduced, which means it is not possible to guarantee that two executions of the system lead to the same state, even if the same inputs are used. As a consequence, searching for a specific error can be difficult, since you cannot reproduce it with each new execution (Nguyen et al., 2012).

The testing of conventional software with predictable inputs and outputs is a non-trivial activity. Testing autonomous agents is a challenge, since the execution of actions is based on decisions of own agents, which may be different from the user's perspective, since the same test input may result in different executions. Due to the peculiar properties and the specific nature of the software agents, it is difficult to apply software testing techniques capable to guarantee the confidence of these systems (Nguyen et al., 2012; Silveira et al., 2014).

de V. Silveira F., L. de Campos G. and Cortés M..
Monitoring and Diagnosis of Faults in Tests of Rational Agents based on Condition-action Rules.
DOI: 10.5220/0005364005850592
In *Proceedings of the 17th International Conference on Enterprise Information Systems* (ICEIS-2015), pages 585-592
ISBN: 978-989-758-096-3
Copyright © 2015 SCITEPRESS (Science and Technology Publications, Lda.)

This paper presents an approach that aims to contribute to the process of rational agents testing. The assumptions are that any test depends on the selected test cases, which should generate information to identify the components in the structure of the artificial agent tested program that are causing unsatisfactory performance. More specifically, the proposed approach consists of designing an agent that performs the monitoring tests of the rational agent and the designer manages relevant information about the performance and agent faults during testing, while making improvements in the agents of the program.

2 BACKGROUND

2.1 Rational Agents

The rational agents select its actions aiming at the best possible outcome, or in the presence of uncertainty, the best expected outcome as a performance measure established to evaluate their behavior. Designing rational agents in complex task environments is a nontrivial task (Russell and Norvig, 2013; Silveira et al., 2013).

The work of Artificial Intelligence is to design the agent program, which implements the function of the agent and will run on any architecture, ie, a computing device with actuators and sensors. Depending on the environment, the design of the agent can be performed considering four basic types of agent programs: (1) simple reactive agents (select actions based on current perception, ignoring the historical perceptions), (2) model-based reactive agents (the agent keeps an internal state that depends on the historical perceptions), (3) goal-based agents (beyond the internal state, the agent keeps information about the goals which describe desirable situations); (4) utility-based agents (have a utility function that maps a state in an associated degree of happiness). In the environment where the agent does not know the possible states and the effects of their actions, the conception of a rational agent can request an agent program with learning capabilities (Russell and Norvig, 2013).

The four types of programs agents can be subdivided into three main subsystems that process information. The first, the perception subsystem, maps a perception data (P) in an abstract representation (State) useful to the agent, see: P → State. The second, the update internal state subsystem, maps representing the current perception and information about the internal state (IS) held by the agent on a new internal state, next: State x IS → IS. Finally, the decision-making subsystem, maps information about the internal state on a possible action (A), action: IS → A (Wooldridge, 2002).

For the simple reactive agent program, the action function selects actions based on the current perception, mapped by the see function, and a set of rules in the condition-action format. The next function in model-based reactive agents keep a description of the environmental state of the agent in memory. The action function of the goal-based agents programs selects its actions using the information processed by the next function and information on the goals that describe the desirable situations in the environment. The action function of utility-based agents uses a utility function to map descriptions of the environmental state as an associated happiness degree.

2.2 Testing Agents

Software testing is an activity that aims to evaluate and improve the product quality by identifying defects and problems. A successful test for detecting defects is the one that makes the system operate incorrectly and as a consequence, exposes the defects (Sommerville, 2011; Pressman and Maxim, 2014).

Due to the peculiar properties of rational agents (reactive properties, of memory, goal-based and utility, and the learning) and its task environments, there is a demand for new test techniques related to the particular nature of agents. For the testing of intelligent agents, it is necessary that the existing software testing techniques are adapted and combined at aiming to detect different faults, making software agents more reliable. Most works of literature consist of adaptations of the techniques from conventional software testing. In the case of the rational agents, we know that these adaptations should seek to evaluate the rationality of actions and plans executed by the agent in its task environment (Houhamdi, 2011a; Houhamdi, 2011b).

Test input selection for intelligent agents presents a problem due to the very fact that the agents are intended to operate robustly under conditions which developers did not consider and would therefore be unlikely to test (Padgham et al., 2013).

Some approaches focus on the production of the test artifacts to support the development methodologies for agent systems (Nguyen, 2008). The assumption in most studies is that a good evaluation of agent depends on the test case selected.

Good test cases should provide the generation of information about the unsatisfactory performance of the components in the structure of the agent and the operation of these components in an integrated manner (Houhamdi, 2011b).

3 RELATED WORKS

In agent-oriented software engineering several approaches to testing agent programs have been proposed. However, it is a challenging activity, and the process for structured agent testing is still expected (Houhamdi, 2011a; Houhamdi, 2011b).

In this section we consider the following criteria in order to evaluate the perfomance of the existing approaches to testing agent programs: (i) the notion of rational agents, (ii) utilization of test cases generated according to the agent goals, (iii) adoption of a measure to evaluate the agent performance, (iv) evaluation of the plans used by the agent to reach the goals, and (v) monitoring the performance measure of the tested agent.

A goal-oriented approach for the testing of agents is presented in (Houhamdi, 2011b) that complements the Tropos methodology (Mylopoulos and Castro, 2000) and reinforces the mutual relationship between the analysis and testing objectives. It also defines a structured process for the generation of tests for agents by providing a method to derive test cases from the agent goals. This strategy does not present: (i) the notion of rational agents, (ii) a measure to performance evaluation of the agent and (iii) any simulation to support the monitoring of the agent behavior to perform the actions involving the agent's goals.

An evolutionary approach for testing autonomous agents is adopted by (Nguyen et al., 2012). It proposes to apply a recruitment of the best test cases for evolving agents. Each agent is given a trial period in which the number of tests with different difficulty levels are executed. This approach is focused in the BDI architecture. Thus, considering the evaluation criteria that is not treated: (i) the notion of rational agents and (ii) a simulation to monitoring the agent behavior performing an action that involves goals.

An approach to the selection of test cases for testing rational agents is proposed by Silveira et al., (2014). It proposes a utility-based agent that uses aspects present in population-based metaheuristics to find satisfactory sets of test cases, i.e., descriptions of specific environments in which the histories associated with the tested agent program in its environment had lower performance. Analyzing the presented criteria, this approach does not treat: (i) a simulation to monitor the agent behavior performing an action that involves goals.

4 PROPOSED APPROACH

This section presents the proposed approach and the aspects involved in monitoring and diagnosis of faults in the test of rational agents. The approach is centered on the monitoring agent, *ProMon*.

4.1 Overall Approach

The proposed approach to the test of rational agents is based on Silveira et al., (2014) that consider the notion of rational agents in the use of test cases. The test cases are generated according with (i) the goals contained in the performance evaluation measure of the tested agent, (ii) the simulation of the interactions of the tested agent with its environment (histories) and (iii) strategies of the utility-based multi-objective local searches for finding test cases and corresponding histories in which the agent is not well evaluated. The proposed approach considers an agent to monitor the testing and diagnosis of faults in the tested agent, *ProMon*.

More specifically, the approach considers in addition to *Designer* four program agents involved, i.e., the agent program being tested *Agent* designed by designer, the task environment program, *Env*, an agent program for selecting test cases, *Thestes* and a monitor agent program *ProMon*. Figure 1 illustrates the interactions between these agents.

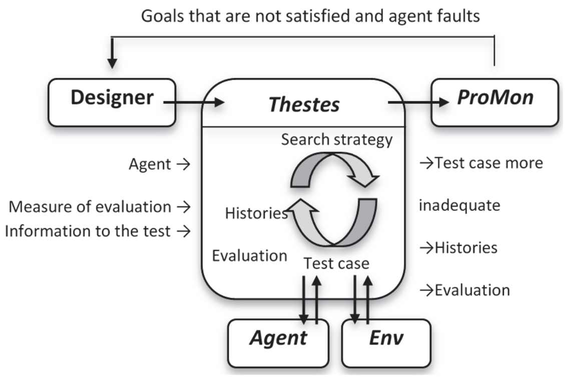

Figure 1: Overview of the approach.

The *Designer* is responsible for designing the rational agent program *Agent*, the measure of performance evaluation and setting other information necessary for the agent *Thestes* to start the process of testing the agent for *Env*. The *Thesthes* agent consists of a solution for the problems when selecting the test case that performs local searches in the state of test utility-based cases. This agent sends a set of efficient solutions determined by multiobjective search strategy, i.e., describing test cases in which *Agent* has the most inappropriate behavior, a set of corresponding histories and their utility values, to the *ProMon*. The *ProMon* agent receives this information and identifies it for the Designer: the goals in the evaluation measure that are not being adequately met by the *Agent*, the episodes in the histories of *Agent* in *Env* which are faults, i.e., that are "distant" from the ideal, and what are the corresponding ideals episodes.

4.2 *ProMon* Agent

This section outlines the main concepts in monitoring the agent program *ProMon*, highlighted in Figure 1. The section is divided into two subsections. The first subsection briefly describes aspects of rational agents that were considered by *ProMon* agent in the composition of both events (measures) as the diagnosis of faults. The second subsection highlights the main function that the agent uses to perform the diagnosis.

4.2.1 Notions of Rational Agents in *ProMon*

The conception of the *ProMon* agent believes that, depending on the environment, the design of rational agent can consider four basic types of agent programs, i.e.: simple reactive, model-based reactive, goal-based and utility-based. More specifically, the conception considers a synthesis of the views of Wooldridge (2002) about agent abstract architectures, and of Russell and Norvig (2004) about the four structures of the mentioned agents programs.

Figures 2 and 3 illustrate some of the information modules (subsystems) proposed by Wooldridge (2002) to the simple reactive internal state agents, respectively, considered by the *ProMon* agent for the diagnosis of faults in the tested agent.

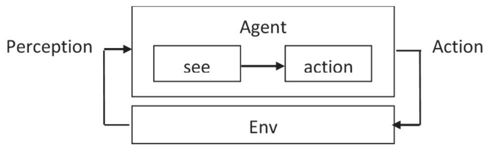

Figure 2: Subsystems of the simple reactive agent.

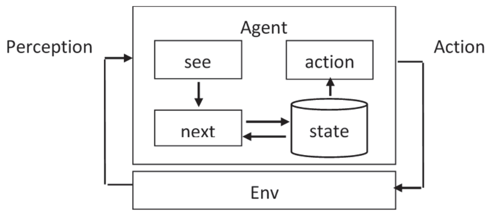

Figure 3: Subsystems of the agent with internal state.

4.2.2 Functionality of the *ProMon* Agent

The *ProMon* agent receives: (1) a test case set, TestCASES containing these test cases where the *Agent* had an inadequate behavior, (2) the histories of the *Agent* in *Env* and the values associated with performance evaluation, episode by episode. Considering this information, the *ProMon* agent must send it to the designer: (1) episodes in all histories that the *Agent* failed and corresponding ideal episodes, and (2) an identification of the fault type, indicating the information processing subsystem of the *Agent*, i.e.: (a) perception subsystem, see, (b) update internal state subsystem, next, and (c) decision-making subsystem, action.

The *ProMon* agent was designed as a model-based reactive agent. More specifically, the monitoring process and diagnosis of fault performed by the *ProMon* agent was proposed to be conducted in two stages, corresponding to the processing of a subsystem of next and action. In the first stage, the next function of the agent receives the histories associated from the cases in TestCASES H(TestCASES) and identifies all the episodes containing faults in these histories.

Thus, as in the case of the *Thestes* agent, this function considers the ProtocolInteraction protocol, and environment program *Env* and a fully observable version of the tested agent (omnipresent), denoted by *Agent** to identity whether an episode in an interation K, in a history associated case I in TestCASES, and generates two episode sets: the ideal episode set in the interaction, Ep_{ideals}^{K}, and the set of episodes with faults of the histories associated cases in TestCASES, Ep_{faults}.

According to the test case problem formulated by the *Thestes* agent, the notation used is:

- **Agent:** a rational agent program to be tested;
- **Agent*:** a omnipresent version of program to be tested;
- **Env:** an environment program able to interact with *Agent* through of ProtocolInteraction;
- **h(Case$_i$) $\in$ (PxA)^{NInt}:** a history of size $N_{Int}$of *Agent* in *Env* corresponding to the Case$_i$

$\in$TestCASES;

- **$h^*(Case_i) \in (PxA)^{NInt}$:** a history of size N_{Int} of *Agent** in *Env* corresponding to the $Case_i \in$ TestCASES;
- **$Ep^K(h(Case_i)) \in PxA$**or, more specifically, $Ep((P^K, A^K))) \in PxA$ an episode in the interaction K of the history of *Agent* in *Env* corresponding to the case $Case_i \in$TestCASES.
- **$Ep^K(h^*(Case_i)) \in PxA$** or, more specifically, $Ep((P^K, A^{K*}))) \in PxA$ an episode in the interaction K of the history of *Agent** in *Env* corresponding to the $Case_i \in$ TestCASES.

The ideal episodes set in an interaction K, Ep_{ideals}^K, consists of all possible episodes that would be produced by *Agent** in interaction, $Ep((P^K, A^{K*}))$,which satisfies at least one of the following two conditions:

(1) For every attribute m in measure of performance evaluation:

$$ev_m(Ep((P^K, A^{K*}))) \geq ev_m(Ep((P^K, A^K)));$$

(2) A^{K*} is better than or equivalent to A^K while considering the viewpoint of the designer.

And, consequently episodes $Ep((P^K, A^K))$, produced by tested agent *Agent* which do not belonging to Ep_{ideals}^Kset, must compose the episodes with faults set, Ep_{faults}.

The condition (2) depends on the viewpoint of the designer. It was introduced to identity faults that are not perceived directly for the specification of measuring performance evaluation. Thus, unlike the condition (1), which considers the measure of performance evaluation, the condition (2) should be specified according to the scope of the tested agent, *Agent.*

Ending the stage of identifying the episodes that are faults in all interactions K in histories associated with the cases in TestCASES, held by next function of *ProMon* agent, the second stage of monitoring and diagnosis of agent can be initiated. In this stage, the action function of the agent uses the episodes Ep_{faults} set and an action function based in condition-action rules to identify the fault type associated to the episode with faults in an specific interaction $Ep((P^K, A^K))$, considering itself as the episode, the values of evaluation are associated to the episode in all the attributes m in performance measure $ev_m(Ep((P^K, A^K)))$, and the ideal episodes $Ep((P^K, A^{K*}))$ in Ep_{ideals}^K, produced by *Agent** in same interaction.

The *ProMon* agent was designed with generic rules to the case in which the *Agent* is a simple reactive agent or a model-based agent. The antecedents in these rules consist of two conditions associated to the conditions (1) and (2) previously described. These conditions associated indicate the reason why an episode with fault $Ep((P^K, A^K)$ was inserted into an Ep_{fault}set, i.e.:

(1)’ there is at least an attribute x which $ev_x(Ep((P^K, A^{K*}))) > ev_x(Ep((P^K, A^K)))$, while the rest of the attributes $ev_m(Ep((P^K, A^{K*}))) \geq ev_m(Ep((P^K, A^K)))$;

(2)’ A^{K*} is better than A^K.

The results of the rules are messages sent to the designer of the *Agent*. The first message indicates the condition is satisfied, the reason for the episode belonging to the Ep_{fault}set. The second message consists of a disjunction involving rules of the form “consequent **if** antecedent”, i.e., “Fault in Module X **if** Condition Y is satisfied”. These are messages that are sent to the designer to evaluate the conditions described in its antecedents and to perceive what processing information modules in *Agent* are possibly causing the faults, according to the conditions (1)’ and (2)’.

Thus, as in the antecedent of each rule suggested is proposals involving the outputs of the processing information modules of the tested agent *Agent* and the agent with fully observable *Agent**, the approach with the *ProMon* agent assumes that the designer has control of the tested agent and is able to compare the processing performed by the modules of *Agent* with the processing performed by the modules of *Agent**.

When the perception subsystem of *Agent** produces an information $State^K$ different from the produced subsystem of the *Agent* and the designer assumes that the decision-making subsystem of *Agent* would be able to select an ideal action A^{K*}, i.e., one of the actions selected by *Agent** and presents in Ep_{ideals}^K, in case it had the information $State^K$, the rules indicate that the fault is in the perception subsystem of the *Agent*. If there is no fault in see function, the rules indicate the fault is in decision-making subsystem of *Agent*, i.e., despite perceiving how *Agent**, the designer assumes that the tested agent *Agent* could not make equivalent decisions. Besides the two possibilities, the rules also indicate that fault may be present in the two subsystems.

For the case when the internal state of the *Agent* is different from the information produced by a perception subsystem of the *Agent** and the designer assume that the decision-making subsystem of the *Agent* would be able to select an ideal action A^{K*} to

State^K, the rules indicate that the fault is in the next function. Whereas there is no fault in the next function, these rules indicate that the fault is in decision-making subsystem of the model-based reactive agent *Agent*. Finally, the rules indicate that the fault may be present both in next and action functions.

It is noteworthy that the rules applied to the model-based reactive agent considers no possibility of the fault in the see function of this agent, meaning that, specifically in this case, the designer knows the limitations of the agent in terms of observing the environment and, therefore, designed a next function, i.e., to minimize the low reliability of the see function. Thus, for the designer, it is more important to perceive if there are faults in the next function. However, if desired, may include rules for fault in the see function of the model-based reactive agent. Although not specified, the rules adopted for this agent can be adapted to the goal-based and utility-based agents, since these agents can also be described in terms of components: see, next and action.

5 EXPERIMENTAL EVALUATION

In this section, we illustrate the operation of *ProMon* agent. In our experiments, two versions of cleaner agent (described by Russell and Norvig, 2013) are implemented: (i) simple reactive agents and (ii) reactive agent with internal state. Both are evaluated in environment with several places considering the energy and cleaning attributes.

5.1 Measure of Performance Evaluation

Table 1 shows the measure of performance evaluation used in the experiments. Ideally a cleaner agent program should clean the environment and maximize levels of cleaning the environment and energy in your battery at the end of the task. The first column describes some of the information on the perceptions of the agent in each possible episode. The second column describes the possible action on these episodes. In the third and fourth columns, respectively, associated to the energy and cleaning goals, two scalars functions (ev_E e ev_C) to measure the performance of the agent in each episode of your history in the environment. The fifth column highlights only the episodes that represent inappropriate behavior, probably due to a fault in see function and/or in set of condition-action rules in action function of cleaner programs.

Table 1: Measure of performance evaluation.

$Ep^K = (P^K,$	$A^K)$	$ev_E(Ep^K)$	$ev_C(Ep^K)$	Fault
...,C, ...	Suck	-1.0	0.0	x
...,C, ...	Right, Left, Below, Above	-2.0	1.0	
...,C, ...	No-op	0.0	0.0	
...,D, ...	Suck	-1.0	2.0	
...,D, ...	Right, Left, Below, Above	-2.0	-1.0	x
...,D, ...	No-op	0.0	-1.0	x

It is noteworthy that the measure of performance evaluation in Table 1 does not implicitly point out the negative aspects of the see function, since the value assigned to each episode is independent of the state of the other places in the environment, different to the place where the cleaner is.

5.2 Tested Agents

The simple reactive cleaner agent program (SR_Partial) focuses on the selection of actions based on current perception, ignoring the historical perceptions obtained in a partially observable environment, i.e., the see function of SR_Partial allows perception only for the state, dirty or clean, of the place in which the agent is. Figure 4 shows the condition-action rules of SR_Partial.

if state is *Dirty* **then do** *Suck*
if state is *Clean* **then do** random motion (*Right, Left, Above, Below*)

Figure 4: Condition-action rules of SR Partial.

The second agent program tested was designed according to the structure of the reactive with internal state with partially observable environment (RIS_Partial). This agent has an internal state with store the historic of perceptions that are considered to the action selection process. Figure 5 shows the condition-action rules of RIS_Partial.

If state is *Dirty* **then do** *Suck*
If state is *Clean* and NotVisit(north) **then do** *Above*
If state is *Clean* and NotVisit(south) **then do** *Below*
If state is *Clean* and NotVisit(east) **then do** *Right*
If state is *Clean* and NotVisit(west) **then do** *Left*
If state is *Clean* and visited all **then do** random action

Figure 5: Condition-action rules of RIS Partial.

5.3 *ProMon* Agent

The *ProMon* agent considers that the ideal episodes set in a interaction K, Ep_{ideals}^{K}, consists of all possible episodes that would be produced by *Agent** in interaction, $Ep((P^K, A^{K*}))$ compared to the corresponding episode produced by *Agent*, $Ep((P^K, A^K))$, in the same interaction.

In the case of cleaner agent, three types of episodes with faults $Ep((P^K, A^K))$ may occur satisfying the Condition (1)', corresponding to lines 1, 5 and 6 of Table 1.

For episodes with faults that compete for a satisfaction of Condition (2)', two types may occur, considering two possible situations identified by designer, but that are not explicitly defined in Table 1:

i. Agent moved to a neighboring place different from another neighboring place that contained dirt;
ii. Agent moved unnecessarily to a neighboring place previously visited.

Thus, *ProMon* identifies the faults associated with the episodes considering five possible of flawed episodes and the condition-action generic rules. It is noteworthy to identity the subsystem that is causing the fault is provided by the designer interpreting the messages sent by *ProMon*.

5.4 Experiment with *ProMon* Agent

This section presents the experiments considering the monitoring and diagnosis of faults of cleaner agent made by *ProMon* agent.

Table 2 shows five episodes of the simulation of the interaction between *Agent* in *Env* generated by the *Thestes* agent, in the test case which achieved the best average value of utility for simple reactive agent.

Table 2: Partial history of *Agent*in*Env*.

K	P^K	A^K	$-ev_E(P^K, A^{K)}$	$-ev_C(P^K, A^K)$
1	...,Clean,...	Below	2.0	-1.0
2	...,Clean,...	Right	2.0	-1.0
3	...,Clean,...	Below	2.0	-1.0
4	...,Dirty,...	Suck	1.0	-2.0
5	...,Clean,...	Left	2.0	-1.0

The selected environment is composed of places with the following configuration: [[C,C,C,C,D], [C,C,D,D,C], [C,D,D,D,D], [C,C,C,D,D], [C,C,D,D,D]]. The utility value is U = 15.5 and the values of inadequacy: $-f_E$ = 49.0 e $-f_C$ = -26.0. The other episodes related to the history of *Agent* in *Env* follow the same pattern.

With this information, *Thestes* is sent by *ProMon* to perform the monitoring and diagnosis of faults. Table 3 illustrates the episodes 1 to 5 in the history of *Agent* in *Env* shown in the Table 2.

Table 3: Partial history of SR_Partial.

History – Ep^1, Ep^2, Ep^3, Ep^4, Ep^5									
$-ev_E$	$-ev_C$	$-ev_E$	$-ev_C$	$-ev_E$	$-ev_C$	$-ev_E$	$-ev_C$	$-ev_E$	$-ev_C$
2.0	-1.0	2.0	-1.0	2.0	-1.0	2.0	-1.0	2.0	-1.0
Fault: No		Faul: No		Fault: No		Fault: No		Fault: Yes	
0,0,0,0,1 **0**,0,1,1,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,0,0,0,1 ***0***,**0**,1,1,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,0,0,0,1 **0**,***0***,1,1,0 0,**1**,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,0,0,0,1 **0**,**0**,1,1,0 0,***0***,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,**0**,0,0,1 **0**,**0**,1,1,0 **0**,***0***,1,1,1 0,0,0,1,1 0,0,1,1,1	

Symbols used: 0 (clean place), 1 (dirty place), **0** (clean place visited), **1** (dirty place visited), **0** (agent is in a clean place), **1** (agent is in dirty place), ***0***(agent was in a clean place), ***1*** (agent was in dirty place), **0** (agent is in a visited clean place), **1** (agent is in visited dirty place).

Table 4 shows the ideals episodes set produced by *Agent** to the episodes shown in Table 3.

Table 4: Ideals episodes produced by *Agent**.

Ideal Histories – Ep^1, Ep^2, Ep^3, Ep^4, Ep^5									
$-ev_E$	$-ev_C$	$-ev_E$	$-ev_C$	$-ev_E$	$-ev_C$	$-ev_E$	$-ev_C$	$-ev_E$	$-ev_C$
2.0	-1.0	2.0	-1.0	2.0	-1.0	2.0	-1.0	2.0	-1.0
Fault: No		Faul: No		Fault: No		Fault: No		Fault: Yes	
0,**0**,0,0,1 0,0,1,1,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,***0***,**0**,0,1 0,0,1,1,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,**0**,***0***,0,1 0,0,**1**,1,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,**0**,**0**,0,1 0,0,**0**,1,0 0,0,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,**0**,**0**,0,1 0,0,***0***,**1**,0 0,0,1,1,1 0,0,0,1,1 0,0,1,1,1	
0,**0**,0,0,1 0,0,1,1,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,***0***,0,0,1 0,**0**,1,1,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,**0**,0,0,1 0,***0***,**1**,1,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,**0**,0,0,1 0,**0**,***0***,1,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,**0**,0,0,1 0,**0**,**0**,**1**,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1	
0,**0**,0,0,1 0,0,1,1,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,***0***,0,0,1 0,**0**,1,1,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,**0**,0,0,1 0,***0***,1,1,0 0,**1**,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,**0**,0,0,1 0,**0**,1,1,0 0,***0***,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,**0**,0,0,1 0,**0**,1,1,0 0,***0***,**1**,1,1 0,0,0,1,1 0,0,1,1,1	
0,0,0,0,1 **0**,0,1,1,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,0,0,0,1 ***0***,**0**,1,1,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,0,0,0,1 **0**,***0***,1,1,0 0,**1**,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,0,0,0,1 **0**,**0**,1,1,0 0,***0***,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,**0**,0,0,1 **0**,**0**,1,1,0 **0**,***0***,1,1,1 0,0,0,1,1 0,0,1,1,1	
0,0,0,0,1 **0**,0,1,1,0 0,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,0,0,0,1 ***0***,0,1,1,0 **0**,1,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,0,0,0,1 **0**,0,1,1,0 ***0***,**1**,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,0,0,0,1 **0**,0,1,1,0 **0**,***0***,1,1,1 0,0,0,1,1 0,0,1,1,1		**0**,0,0,0,1 **0**,0,1,1,0 **0**,***0***,**1**,1,1 0,0,0,1,1 0,0,1,1,1	

The SR_Partial agent program is not made faults in the first four interactions maintained with *Env*, because the four episodes produced by SR_Partial belong to the ideal episodes set Ep_{ideals}^{K}, generate in the four interactions (K = 1, …, 4) of *Agent** in *Env* (each column in Table 3 of the episodes 1 to 4 corresponds to a episode ideals set in a interaction), as outlined in Table 4.

The SR_Partial agent committed a fault in episode 5, i.e., there is a better action with 'Left' in this episode, i.e., 'Right' that leads the agent to the neighboring dirty place. In this case, the fault indicates that whoever is causing this fault is the see function of the agent, since the action function of SR_Partial could be able to choose the 'Right' action if it knew that the place to your right was dirty. In this case, despite the evaluation values of SR_Partial and *Agent** are equals, the agent avoids to gain a point by not moving to a dirty place.

The same procedure is performance for all the histories of *Agent* in *Env*. Thus, as expected, the cleaner agent with simple reactive architecture and partial observability presents the worst performance in the evaluation, to realize a brief analysis in the condition-action rules the agent doesn't consider the perceptions and the actions previously related to energy and cleaning objectives. Since the designer receives the information of the episodes that are flawed, it is possible to make changes in its internal structure to improve its performance, allowing it to run actions better.

6 CONCLUSIONS

Considering which rational agent should be able to accomplish your goals, appropriate tests should be developed to evaluate the actions and plans executed by the agent when achieving these goals in its task environment. The motivation of this research is due the gap in terms of testing techniques specifically applied to autonomous agents, so that they can evaluate the behavior and the confidence of agent-based systems.

The information generated by the approach indicated a measure of average utility associated with the performance of the tested agent and objectives as evaluation that is not being satisfied. Considering the best set of histories of the agent in its environment, associated to the selected test case set by approach the end of the search process, the designer and/or other auxiliary automatic systems can identity those problematic episodes, and what subsystems processing information and information associated modules are causing the unsatisfactory performance on the agent.

For future work we suggest the development of conditional-action rules for goals-based and utility-based agent programs as well as investigation into other aspects that may be included in the diagnosis of the *ProMon* agent that contribute to the identification of problematic episode agents.

REFERENCES

Houhamdi, Z. 2011. Multi-Agent System Testing: A Survey. In *International Journal of Advanced Computer Science and Applications (IJACSA)*. v. 2, n. 6.

Houhamdi, Z., 2011. Test Suite Generation Process for Agent Testing. In *Indian Journal of Computer Science and Engineering (IJCSE)*. v. 2, n. 2.

Mylopoulos, J., Castro, J., 2000. Tropos: A Framework for Requirements-Driven Software Development. In *Information Systems Engineering: State of the Art and Research Themes, Lecture Notes in Computer Science*, Springer.

Nguyen, C. D., Perini, A., Tonella, P., Miles, S., Harman, M., Luck, M., 2012. Evoluctionary Testing of Autonomous Software Agents. In *Autonomous Agents and Multi-Agent Systems*. v. 25, n. 2, p. 260-283.

Padgham, L., Zhang, Z., Thangarajah, J., Miller, T. 2013. Model-Based Test Oracle Generation for Automated Unit Testing of Agent Systems. In *IEEE Transactions on Software Engineering*. v. 39, n. 9.

Pressman, R. S.; Maxim, B. 2014. *Software Engineering*: A Practitioner's Approach. 8 ed. McGraw-Hill.

Russell, S., Norvig, P., 2013. *Inteligência Artificial*: uma abordagem moderna, Campus. São Paulo, 3rd edition.

Silveira, F. R. V.; Campus, G. A. L.; Cortés, M. I. 2013. Rational Agents for the Test of Rational Agents. IEEE Latin America Transaction, v. 11, n. 1, feb.

Silveira, F. R. V., Campos, G. A. L., Cortés, M. I., 2014. A Problem-solving Agent to Test Rational Agents.In *16th International Conference on Enterprise Information Systems (ICEIS 2014)*.

Sommerville, I., 2011. *Engenharia de Software*, Pearson Addison Wesley. São Paulo, 9th edition.

Wooldridge, M., 2002. *An Introduction to MultiAgent Systems*.John Wiley & Sons.