A Problem-solving Agent to Test Rational Agents
A Case Study with Reactive Agents

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

Keywords: Test Agents, Selection of Test Cases, Rational Agents.

Abstract: Software agents are a promising technology for the development of complex systems, although few testing techniques have been proposed to validate these systems. In this paper, we propose an agent-based approach to select test cases and test the performance of rational agent. Interactions between agent and environment are realized in order to evaluate the agent performance for each test case. As a result, we obtain a set of test cases where the agent has not been well evaluated. Based on this result, the approach identifies the goals that are not met by the agent and reported to the designer.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Agent is an entity capable of perceiving their environment by means of sensors and act in this environment through actuators. The concept of rational agents refers to those agents, which act to achieve the best expected result, namely the best measure of performance (Russell and Norvig, 2004).

Since agent-based systems are increasingly taking over the operations and controls in the organization management, vehicles and automated financial systems, guarantees that these complex systems work properly are required. In this sense, a research about software engineering methods, including requirements engineering, architecture and testing techniques in order to provide appropriate mechanisms for software development and support tools it is desirable (Nguyen et al., 2011).

In particular, due to the peculiar properties of rational agents (reactive properties, learning, goal and utility orientation), the application of standard testing techniques is difficult and do not guarantee the reliability of these systems (Nguyen et al., 2009).

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 their own decisions of own agents, which may be different from the user's perspective, since the same test input may result in different executions (Nguyen et al. 2009), (Silveira et al., 2013).

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, 2011).

In this paper we present an approach to testing rational agents programs, leading to the designer the relevant information about the performance of the agent program, useful to improve its design and efficiency.

2 BACKGROUND

2.1 Rational Agents

A software agent is a program able to perceive its environment by means of sensors and acting in this environment by means of actuators (Russell and Norvig, 2004). Depending on the context in which the task can be completed, the agent is able to select the most suitable way (Poutakidis, 2009).

From the point of view of the agent designer, an agent is rational if, based on perception, the agent is able to make correct decisions in order to reach the goals established by the designer. When the agent is not able to accomplish all the goals, decisions in

de V. Silveira F., Augusto L. de Campos G. and I. Cortés M..
A Problem-solving Agent to Test Rational Agents - A Case Study with Reactive Agents.
DOI: 10.5220/0004899105050513
In *Proceedings of the 16th International Conference on Enterprise Information Systems* (ICEIS-2014), pages 505-513
ISBN: 978-989-758-027-7
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

order to achieve a greatest degree of satisfaction defined according to some criterion, are expected. This criterion is known as measuring problem-solving performance (Russell and Norvig, 2004).

Based on these principles, four types of rational agent programs are defined (Russell and Norvig 2004): (i) Simple Reflex agents, where condition-action rules are used to select the actions to be executed based on the current perception, (ii) Model-based Reflex agents, where the agent is able to store its current state in an internal model, (iii) Goal-based agents, where agents are model based agents that set a specific goal and select the actions that lead to that goal. This allows the agent to choose a goal state among multiple possibilities; (iv) Utility-based Agents, where considering the existence of multiple goal states, it is possible to define a measure of how desirable a particular state is.

2.2 Agents Testing

Software testing is an activity that aims to evaluate and improve the product quality by identifying defects and problems. A test consists in checking the dynamic behavior of a program along a set of test cases properly selected (Poutakidis, 2009).

At agent level, the tests are directly related to the test cases created to evaluate the agent's goals (Houhamdi, 2011). The agent can have its own internal goals and knowledge, which can be changed at runtime, and these can affect the result returned, if any. The autonomous agent testing to require more than a single test on the component requires that the same test be applied in different contexts. Ensure the variety of contexts tested to declare that the agent behaves correctly is a difficult but important (Nguyen et al., 2009).

Testing agents involves the testing of the agent goals. The agent should be able to achieve their goals and to act correctly in cases where its expected goal cannot be achieved. This requirement may or may not be sufficient to cover the agent components, planes, beliefs etc. If the adequacy criterion is not reached, more test cases should be defined to complete the test agent (Houhamdi 2011).

3 RELATED WORKS

In agent-oriented software engineering several approaches to testing agent have been proposed. Is a challenging activity and a process for structured agent testing is still expected (Houhamdi 2011).

In this section we consider the following criteria in order to evaluate the attendance 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 agent been tested.

A goal-oriented approach for the testing of agents is presented in (Houhamdi, 2011) that complements the Tropos methodology (Castro and Mylopoulos, 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.

Nguyen (2008) proposes a systematic and comprehensive method of goal-oriented testing for agents, which encompasses the development process of the agent according to the methodology Tropos (Castro and Mylopoulos 2000). This work presents a methodology for the production of test artifacts from the specifications and design of agents, which are used to detect problems. The test cases are automatically generated and evolve guided by mutation and quality function.

An evolutionary approach to testing autonomous agents is adopted by (Nguyen et al., 2009). It is proposed to apply a recruitment of the best test cases for evolving agents. For each agent is given a trial period in which the number of tests with different difficulty levels are executed.

Both approaches (Nguyen, 2008) and (Nguyen et al., 2009) are focused in the BDI architecture. Thus, considering the evaluation criteria is not treated: (i) the notion of rational agents and (ii) a simulation to monitoring the agent behavior.

4 PROPOSED APPROACH

This section presents the proposed approach and the aspects involved in the testing of the rational agents. The approach is centered in a problem-solving agent for test cases selection, *Thestes*.

4.1 Selection Test Case of Rational Agents

In the context of rational agents, the testing consists

in identify situations where the agent was not well valued considering an evaluation measure informed by the designer according environment aspects.

Table 1 specifies a measure of performance evaluation to the cleaner agent (a version adapted from Russell and Novirg (2004)) that should clean up the environment and maximize cleaning and energy. The first and second columns describe part of the agent perception and the possible actions related to each perception. The third and the fourth columns are associated to the cleaning and energy goals, respectively, and involve two scalar functions (av_E and av_C) to measure the agent performance.

Table 1: Measure of performance evaluation.

$Ep^k = (P^k$,	$Action^k)$	$av_E(Ep^k)$	$av_C(Ep^k)$
..., Clean, ...	Aspire	-1.0	0.0
..., Clean, ...	Righ, Left, Above, Below	-2.0	1.0
..., Clean, ...	No operate	0.0	0.0
..., Dirty, ...	Aspire	-1.0	2.0
..., Dirty, ...	Righ, Left, Above, Below	-2.0	-1.0
..., Dirty, ...	No operate	0.0	-1.0

The achievement of goals which are implicit in the performance measure established by the designer is a hint of the rationality of the agent program. However, it will depend of the adequacy of the internal environment properties, and the mechanism used in implementation and execution of the agent program (architecture), to the external environment properties, where the task will be performed. Thus, the test process of those systems should evaluate the program performance in its task environment, identifying the goals that are not met (desired states) in the external environment and the components of the program internal environment that are restricting the satisfaction of the goals.

In this context, the efficacy of the test process depends of the test cases selected. Not always the best cases are available initially and, depending of agent task environment, there may be a lot of cases to be observed. Thus, the selection of a set of test cases is a search problem in a space of status composed by a big family of sets of possible cases. An optimal test case is one that the agent obtains the minimum performance value as possible. Be:

- **Agent**: the rational agent program to be tested;
- **Env**: the environment program able to interact with *Agent*;
- **InteractionProtocol**: a description of the interaction protocol between *Agent* and *Env*;
- **Ω**: a set of feasible environments to instantiate *Env* and to test *Agent*;
- **P(Ω)**: subsets of possible environments to be described in Ω;
- **TestCASE ∊ P(Ω)**: a subset of test cases in the set P(Ω), where:
  - **$Case_i$ ∊ TestCASE**: a specific environment description in *TestCASE*;
- **H(TestCASE) ∊ $P((PxA)^{NInt})$**: set of histories of length N_{int} of *Agent* in *Env*, considering *InteractionProtocol* and all cases in *TestCASE* such that ∀i ϵ {1,...,NCases}, ∀t ϵ {1,...,NInt}:
  - **$h(Case_i)$ ∊ $(PxA)^{NInt}$**: history of length N_{Int} of *Agent* in *Env* correspondent the $Case_i$ ∊ *TesteCASE*;
  - **$Ep^k(h(Case_i))$ ∊ PxA**: episode in interaction k, $k \le N_{Int}$, of the history of *Agent* in *Env* correspondent the case $Case_i$ ∊ *TesteCASE*;
- **fad(H(TestCASE)) = (f_1(H(TesteCASE)), ..., f_m(H(TesteCASE))) ∊ Rm**: an array of m objective functions (implicit) in the measure of performance evaluation made by the designer ($m \ge 1$), which measures the adequacy of *Agent* in *Env* considering a set of histories *H(TestCASE)*, where, ∀m ϵ {1, ..., m}:

$$f_m(H(TestCASE)) = \frac{1}{NCases}\sum_{i=1}^{NCases} Ev_m(h(Case_i)) \quad (1)$$

that measures the adequacy of *Agent* in *Env*, as the achievement of the goal *m* in the evaluation measure, considering the histories in *H(TestCASE)* and ∀i ϵ {1, ..., NCases}:

$$Ev_m(h(Case_i)) = \sum_{k=1}^{NInt} ev_m(Ep^k(h(Case_i))) \quad (2)$$

where $ev_m(Ep^k(h(Case_i)))$ is the value of reward / penalization in goal *m* assigned by the evaluation of the episode *k* of the history associated to the $Case_i$ ∊ *TesteCASE*;

- **finad(H(TestCASE)) = (– f_1(H(TestCASE)), ..., – f_m(H(TestCASE))) ∊ Rm**: an array of m objectives associated the array fad(H(TestCASE), measures the inadequacy of *Agent* in *Env* considering the histories in *H(TestCASE).*

Problem:

'maxime' finad(H(TestCASE))

subject to: TestCASE ϵ P(Ω) and

H(TestCASE) ϵ $P((PxA)N_{Int})$

The formulation to the test case selection problem considers that, if the agent program is

inadequate, the objective functions of inadequacy, i.e., the objective functions in the evaluation measure modified by minus signs (-), will be maximized. Depending on the objectives, there may not be an optimal set. In this case, the task is to find a satisfactory set of cases, i.e., when the performance of the agent program in the environment is unsatisfactory and, consequently, allowing to detect their limited properties.

4.2 Agent of Problem Solving of Test Case Selection

This section outlines an agent of problem solving of test cases selection for rational agents programs (*Thestes*). The program uses a local search strategy, based on population and oriented by an utility function, for finding sets of satisfactory test cases, i.e., specific environments where the histories associated to *Agent* in *Env* have low performance.

4.2.1 Structure of the Agent Program *Thestes*

Thestes agent incorporates and processes the information in the selection problem formulation and other information sent by designer in $Perception^k$, in order to select a satisfactory solution to be sent in $Action^k$ for the designer in order to improve the performance of Agent, if necessary. This interaction scheme between the designer and *Thestes* should be continued until the rational performance of *Agent* is considered satisfactory.

Figure 1 illustrates the structure of the problem solving agent, *Thestes*. This structure consists in an adaptation of the utility-oriented agent program, specified by Russell and Norvig (2004), and considering the abstract architecture of the agent with internal state, specified by Wooldridge (2002).

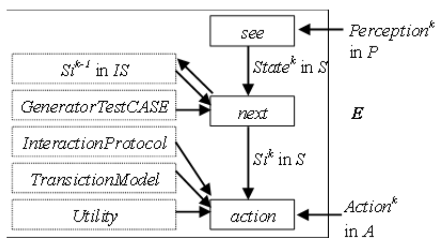


Figure 1: Structure of agent program *Thestes*.

More specifically, the perception subsystem, *see*, is responsible to map the information necessary to test *Agent* in a computational representation, $State^k$, useful to the processing of the others two subsystems (*next* and *action*): (*Agent*, *Env*, *ParametersSearch*, *ParameteresSimulation*). The subsystem *next* updates the internal state in $State^k$, and generates an initial set *TestCASE*. The set is randomly generated or specific to test certain aspects of the internal structure of *Agent* and, in order to facilitate the next stage of decision make (*action*): (*TestCASE, Agent, Env, ParametersSearch, ParametersSimulation*).

Finally, considering the updated internal state, the *action* function starts a process of local search in order to find a satisfactory action $Action^k$. This function uses information about a state transition model, *ModelTransition*, to generate new test cases from *TestCASE*. In addition, the interaction protocol, *InteractionProtocol*, and an utility function, *Utility*, are used to, respectively, to obtain the histories corresponding to test cases in the set and to evaluate the performance of *Agent* in this histories.

Finally, the generated information by *action* function, *Thestes* send to the designer important information in $Action^k$: (1) the current set *TestCASE* as solution to the selection problem and the best cases found by the solution, (2) the histories corresponding to *Agent* in *Env*, (3) the performance values, considering each objective in measure of performance evaluation.

4.2.2 Transition Model

The transition model indicates the function that modifies the set of test cases considering the test cases in the current solution, Pop^t = *TestCASE*. Moreover, the model regards a transition model pre-defined and the corresponding performance values, measured by a Utility function to generate new test cases, Pop^{t+1}. The generic transition model considers the *NCases* in a current set Pop^t and chooses: (a) the test case that will be modified, and (b) the changes that have to be made in these cases.

Several transition models may be implemented to achieve the above choices. In this paper the implementation of the model is based on population-based metaheuristcs using Genetic-Algorithm (GA). Likewise, several transition models may be implemented considering other population-based metaheuristcs.

4.2.3 Simulation of *Agent-Env* Interaction

Thestes knows *Agent* and *Env*, as well as the protocol *InterationProtocol*. In addition, the *action* function considers ParametersSimulation in the decision making process. Thus, it was conceived an interaction mechanism that simulates the interactions between *Agent* and *Env*, according

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

InteractionProtocol. The process starts when *Env* is initialized with information of a test case ($Case_i \in TestCASE$) and annotates the episodes ($Ep^k(h(Case_i)) \in PxA$) of the history ($h(Case_i) \in (PxA)^{NInt}$). Figure 2 shows the simulation.

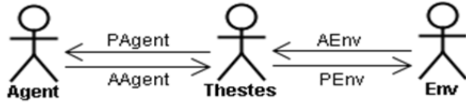


Figure 2: Simulation of the interactions *Agent-Env*.

The mechanism set *Env* with information in *PEnv* with a specific test case belongs to the current solution set *TestCASE*. Then, *Env* stores this information and send the information of the current state in *AEnv*. The mechanism annotates this information and forwards it to *Agent*, which perceives in *PAgent*. *Agent* processes the information and selects an action that is sent in *AAgent* to the simulation mechanism. After that, the simulation closes the annotation of a history episode, and sends in *PEnv* the information about *action* to *Env*. Other interaction can be started, which should be repeated until the annotation of a full history.

4.2.4 Utility Function

During the generation of new test cases along the search process, the Utility function enables the agent to obtain performance measures considering the test cases modified by *TransitionModel*. These measures allow *Thestes* to evaluate the test cases and select a subset of *NCases* that is better than the latter.

Thus, this first approach, considering that the utility function is preferably independent, i.e., the utility degree of a goal is independent of the values assumed by the others, were incorporated in *Thestes* two way of Utility function, i.e., a function additive:

$$\mathrm{Utility}(f_{inad}(H(TestCASE))) = \sum_{m=1}^{M} u_m\left(-f_m\left(H(TestCASE)\right)\right), \quad (3)$$

and other in linear format, where $w_m \geq 0$, $m = 1,\ldots,M$.

$$\mathrm{Utility}(f_{inad}(H(TestCASE)) = -\sum_{m=1}^{M} w_m * f_m(H(TestCASE)) \quad (4)$$

The selection problem of test case was reformulated as:

'maximize' $\mathrm{Utility}(f_{inad}(H(TestCASE)))$

subject to: $TestCASE \in P(\Omega)$ e $H(TestCASE) \in P((PxA)^{NInt})$

That is, considering that $У = f_{inad}(P(\Omega))$ is the representation of the mapping of $P(\Omega)$ in the space of objectives:: $У = \{y \in R^M | \ y = f_{inad}(H(TestCASE))$), $TestCASE \in P(\Omega)$ and $H(TestCASE) \in P((PxA)^+)\}$; the problem can be established as:

'maximize' Utility(y)

subject to: $y \in У$

5 EXPERIMENTAL EVALUATION OF *Thestes* AGENT

In this section, we illustrate the operation of *Thestes* agent, solving a problem of test case selection. In our experiments, two versions of the cleaner agent were implemented: (i) simple reactive agent and (ii) reactive agent with internal state. Both are evaluated in an environment with multiples places rooms considering the power and cleaning attributes, according to Table 1.

5.1 The Environment and the Agent in Testing

More specifically, the task of *Thestes* is to select a set of environments formed by *n x n* places (*Env*) which are satisfactory to test the cleaner agent program based on condition-action rules (*Agent*). An environment differs of others as regards the localization and the amount of dirty places. Every environment is partially observable, i.e., the cleaner agent perceives the environment, but the *see* function can only map the state of the place where the agent is.

5.2 Tested Agents

The simple reactive cleaner agent program (SR_Partial) focuses on the selection of actions based on the current perception. Thus, the agent ignores the historical perceptions obtained in a partially observable environment, i.e., the *see* function (SR_Partial) allows perceive only the state, dirty or clean, of the current place. Figure 3 shows the condition-action rules of SR_Partial.

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

Figure 3: 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 4 shows the condition-action rules of RIS_Partial.

if state is *Dirty* **then do** *Aspire*
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 4: Condiction-action rules RIS_Partial.

5.3 Problem-solving *Thestes* Agent

The *Thestes* agent presupposes the existence of a current set *TestCASE* containing NCases of test, which is an initial solution generated by *next* function. The *action* function of agent considers four main components: (1) *TransitionModel* of states that operates on *TestCASE* to generate new sets, (2) a utility function to evaluate the generated sets, (3) a strategy to select from the evaluated sets a new current set more useful, and (4) a test for the new current set *TestCASE*, established in terms of inadequacy associated with performance of *Agent* in *Env*. This set is instantiated with each NCases in current set, and the number of changes made until the current interaction. This information is used to decide if the *action* function can stop the process.

For the components (1), (3) and (4), notions of genetic algorithm (Holland, 1975) are used. In (2) a Utility function in linear format is used, as described in Section 4.2.4.

In the context of GA, *TestCASE*, containing descriptions of environment with *n x n* places, was represented by a population, each individual encodes an environment and each gene encode the place state, in terms of dirt. The *TransitionModel* based in GA considers *TestCASE* as a population that is able to go through the steps of selection pairs (roulette method), crossover and mutation. This process promotes the evolution and allows the *action* function to simulate, evaluates the utility of individuals and composes a new population. To prevent the loss of the best test case in the previous generation, the elitism strategy is used.

5.4 Experiment with *Thestes* Agent

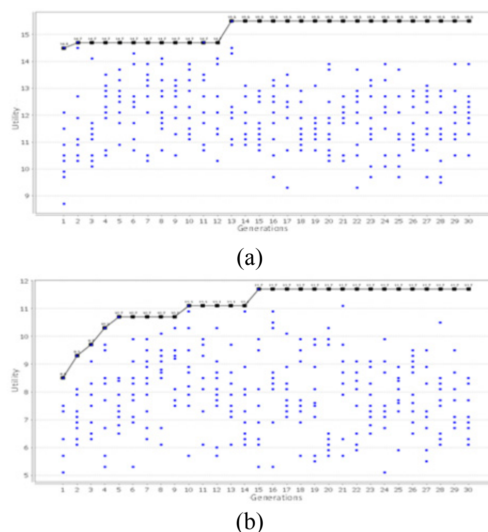
This section presents an experiment conducted with *Thestes*. *SearchParameters* describes the size (n^2) and the amount (NCases) of environments in TestCASE. The probability mutation (Mut), the maximum number of executions (Kmax), and the maximum utility value that can be achieved by a history (Umax) are given. Information about Kmax and Umax define the stop condition in the test. Information in *SimulationParameters* describes the maximum number of interaction between *Agent* and *Env* in any simulation (Nint), i.e., the maximum number of episodes in each history, and the number of simulation realized in the same environment (Ns). Table 2 presents this information.

Table 2: SearchParameters and SimulationParameters.

SearchParameters					SimulationPar	
NCases	n^2	Mut	Kmax	Umax	Nint	Ns
10	25	0,6	30	100000	25	5

The maximum number of interactions (Nint) of *Agent* with *Env*, to be simulated in each of 10 cases (*NCases*) in the population is 25. The maximum utility value (Umax) is high, considering that the stop condition in testing process of the search strategy is defined considering the realization of 30 cycles of executions of *action* function (Kmax). For each test case were realized five simulations (Ns).

Figure 5 presents the results produced by *Thestes* using the Utility function in linear format, described in Section 4.2.4. Values of equals weights to the two goals (cleaning and energy) in evaluation measure are given, i.e., $w_L = w_E = 0{,}5$, over 30 generations. In (a) the results of SR_Partial and in (b) the results of RIS_Partial are showed, respectively.



(a)

(b)

Figure 5: Utility value of *TestCASE* in generations.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

The nonlinear points represent the test cases in *TestCASE* in each generation. The linear points identify the best test case in the generation. In the case of simple reactive agent with partial observability (Figure 5 in (a)), the best test case was obtained in the fourteenth generation, with utility value equal to 15.5. This case was used as the reference for the cases in *TestCASE* in the later generation with utility values less than the best. In case of the reactive agent with internal state (Figure 5 in (b)), the best test case was obtained in fifteenth generation, with utility value equal to 11.7.

As expected, considering the utility values of the best test cases, the *Thestes* agent appoints that the more inadequate behavior to the simple reactive agent, when compared with the reactive agent with internal state. In this case, the annotation of the places that were visited is useful to enable the conception of a subsystem of decision making (*action*) more refined. The set of rules in this case is able to avoid the unnecessary visit to the places that have been visited in previous interactions. This behavior is not possible for the SR_Partial.

In the experiments, *Thestes* aims to select a set satisfactory *TestCASE*. In this sense, in each generation, the test case where the agent had the most inadequate behavior is preserved, i.e., the utility value is equal to the maximum value between the 10 cases. Cases whose utility values are lower than the minimum value are not important.

Figure 6 presents the values of the inadequacy function related to cleaning and energy associated with 10 test cases in *TestCASE* in 30 generations to SR_Partial in (a) and RIS_Partial in (b).

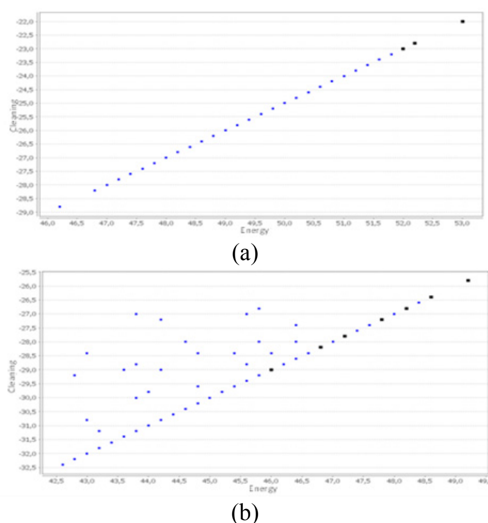


Figure 6: Values of inadequacy of Cleaning and Energy.

Considering the test case in *TestCASE* with higher utility value as shown in Figure 5, is possible observe that for both the attributes, cleaning and energy, shows in Figure 6, the simple reactive agent with partial observability has more inappropriate behavior, com $-f_E = 53.0$ and $-f_C = -22.0$, while the reactive agent with internal state has a less inadequate behavior with $-f_E = 49.2$ e $-f_C = -25.8$.

The simple reactive agent with partial observability has limited vision of the environment, the actions with it selects are not always rational. For example, as the actions of motion are randomly chosen, these actions do not always lead to the dirty place closer, causing the agent spends more energy than necessary and score at least in cleaning. As the reactive agent with internal state stores the historic of perceptions, the agent avoid the places previously visited. Consequently, this agent selects best actions.

Considering that the two goals in the evaluation measure have the same importance value ($w_C = w_E = 0.5$) in the utility function, the approach privileges the cases in which *Agent* has a more inadequate behavior in terms of consumption energy that in terms of cleaning. This is justified because the measure of performance evaluation scores negatively in terms of energy all episodes for *Agent* because the agent always expends energy to perform your actions, and positively in terms of cleaning.

However, the increase of the number of dirty places at the end of the interaction between *Agent* and *Env* in the utility function minimizes the effect mentioned previously and favors those cases where the environment remained with greatest amount dirty places. Figure 7 shows the percentage of dirty places after 25 interactions between *Agent* and *Env*. In (a) the results of SR_Partial and in (b) the results of RIS_Partial are showed, respectively.

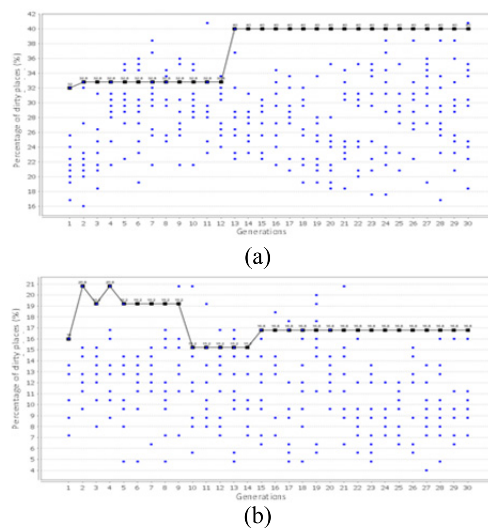


Figure 7: Percentage of dirty places in the TestCASE.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

In general, *Thestes* selected test cases with higher amount of dirty places at the end of the tests, which are considered test cases with worse performance.

Table 3 illustrates five episodes about the simulation of the interaction between *Agent* in *Env*, in environment which obtain average utility value for the SR_Partial.

Table 3: Partial history of Agent in Env.

k	P^k	A^k	$-av_E(P^k, A^k)$	$-av_L(P^k, A^k)$
1	...,Clean,…	Below	2.0	-1.0
2	...,Clean,...	Right	2.0	-1.0
3	...,Clean,...	Below	2.0	-1.0
4	...,Dirty,...	Aspire	1.0	-2.0
5	...,Clean,...	Left	2.0	-1.0

The environment selected 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. As expected, the cleaner agent is more adequate to the environment considering the cleaning objective than the energy objective. A brief analysis of the condition-action rules of the cleaner agent confirms this proposition. The history obtained by RIS_Partial follows the same pattern.

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 not consider the perceptions and the actions previous related to energy and cleaning objectives. As the cleaner agent was designed as a simple reactive agent, little can be done to improve their performance. In this sense, an extension in its structure is required in order to widen the observability of the environment, allowing it to choose actions better. Consequently, the agent will be able to economize energy avoiding places has been visited.

6 CONCLUSIONS

Considering which the 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 this context, techniques that consider the peculiar nature of the agent are required.

The proposed approach considers that in the case of rational agents, where the measure of performance evaluation is established by the designer, multiple objectives, possibly conflicting, must be considered. In the proposed approach, the test results should indicate the average performance of the agent and, especially, the goals that are not being meeting, as well as information about the stories of the agent, which are useful to identify the agent behaviors that need to be improved.

The information generated by the approach indicates a measure of utility associated with the performance of the tested agent and objectives in the evaluation measure that are not being satisfied. Considering the best set of stories of the agent in the environment, associated with the set of test cases selected by the approach to end of the search process, the designer and / or other auxiliary automated systems can identify those problematic episodes with are causing the unsatisfying performance at the agent.

As future work, we suggest a case study with objective-based and utility-based agents. Additionally, adapt the approach to provide a testing strategy capable of test the agent interaction in multiagent systems.

REFERENCES

Holland, J. (1975) Adaptation in natural and artificial systems. University of Michigan Press.

Houhamdi, H. (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. Information Systems Engineering: State of the Art and Research Themes, Lecture Notes in Computer Science, Springer.

Nguyen, C. D. (2008) Testing Techniques for Software Agents. PhD Dissertation. University of Trento.

Nguyen, C. D.; Perini, A.; Tonella, P.; Miles, S.; Harman, M.; Luck, M. (2009) Evoluctionary Testing of Autonomous Software Agents. In: 8th Int. Conf. on Autonomous Agents and Multiagent Systems. Budapest, Hungary.

Nguyen, C.; Perini, A.; Bernon, C.; Pavón, J.; Thangarajah, J. (2011) Testing in multi-agent systems. Springer. v. 6038, p. 180-190.

Poutakidis, D.; Winikoff, M.; Padgham, L.; Zhang, Z. (2009) Debugging and Testing of Multi-Agent Systems using Design Artefacts. Springer Science Business Media, LLC.

Russell, S.; Norvig, P. (2004) Inteligência Artificial: uma abordagem moderna. 2 ed. São Paulo: Prentice-Hall.

Silveira, F. R. V.; Campus, G. A. L.; Cortés, M. I. Rational Agents for the Test of Rational Agents. IEEE Latin America Transaction, vol. 11, n. , feb 2103.
Wooldridge, M. (2002) An Introduction to MultiAgent Systems. John Wiley & Sons Ltd.