Experiments with Multiple BDI Agents with Dynamic Learning Capabilities

Chapter · June 2016

DOI: 10.1007/978-3-319-39387-2_23

5 authors, including:

Amelia Bădică
University of Craiova
18 PUBLICATIONS **41** CITATIONS
SEE PROFILE

Costin Badica
University of Craiova
216 PUBLICATIONS **829** CITATIONS
SEE PROFILE

Maria Ganzha
Polish Academy of Sciences
170 PUBLICATIONS **728** CITATIONS
SEE PROFILE

Marcin Paprzycki
Instytut Badań Systemowych Polskiej Akademi…
342 PUBLICATIONS **2,246** CITATIONS
SEE PROFILE

Experiments with Multiple BDI Agents with Dynamic Learning Capabilities

Amelia Bădică[1], Costin Bădică[1], Maria Ganzha[2], Mirjana Ivanović[3], and Marcin Paprzycki[2]

[1] University of Craiova, A.I.Cuza 200530, Romania,
ameliabd@yahoo.com, cbadica@software.ucv.ro
[2] Polish Academy of Sciences, Systems Research Institute, Warszawa, Poland
Maria.Ganzha@ibspan.waw.pl, paprzyck@ibspan.waw.pl
[3] Faculty of Sciences, Novi Sad, Serbia
mira@dmi.uns.ac.rs

Abstract. In this paper we show how multiple BDI agents, enhanced with temporal difference learning capabilities, learn their utility function, while they are concurrently exploring an uncertain environment. We focus on the programming aspects of the agents using the Jason agent-oriented programming language. We also provide experimental results showing the behavior of multiple agents acting in a Markovian grid environment. We consider agents with the perception function affected by the intermittent faults and Gaussian noise, as well as agents for which their action function is not always successful.

Keywords: BDI agent, reinforcement learning, agent-oriented programming

1 Introduction

Our research is focused on narrowing the gap between agent-oriented programming and learning. Agents, acting in an uncertain and dynamic environment, can use reinforcement learning (RL, hereafter) to either learn their utility function in passive learning, or an optimal policy that maximizes their utility in active learning. RL assumes that an agent is using observed rewards (also known as reinforcements) that are perceived from the environment to measure its welfare following its actions in an uncertain and dynamic environment [11].

Agent-oriented programming (AOP, hereafter) is concerned with the development of better programming models for the engineering of multi-agent systems (MAS, in what follows). Currently AOP is a hot research topic that resulted in quite a large number of proposed AOP languages [1]. The AgentSpeak(L) programming language, represented by its Jason incarnation, based on the Java platform, can be considered as the *de facto* standard of AOP [4].

Temporal Difference Learning (TDL, hereafter) is a passive RL method that can be used by an agent to learn its utility function, while it is acting according to a given policy in an uncertain and dynamic environment. In [2] we reported our initial approach and results for modeling and implementation of the TDL using Jason. Here we expand the research in the following directions:

– We consider multiple agents acting concurrently and asynchronously, Each agent gets individualized percepts in the environment, while agent actions are processed concurrently by the environment.
– We consider agents with the perception function affected by intermittent faults and Gaussian noise.

2 Background

2.1 Agent-Oriented Programming

Brief Overview of AgentSpeak(L) The software agent paradigm was proposed about two decades ago to capture the new model of a "computer system situated in some environment that is capable of flexible autonomous action in order to meet its design objectives" [7]. Historically, agent-oriented programming, here understood as computer programming based on the agent paradigm, was firstly proposed more than 20 years ago as "a new programming paradigm, one based on cognitive and societal view of computation" [10].

AgentSpeak(L) is an abstract AOP language firstly introduced in [8]. Jason is a Java-based implementation, as well as an extension of the AgentSpeak(L) [4, 6]. AgentSpeak(L) follows the paradigm of practical reasoning, i.e. reasoning directed towards actions, and it provides an implementation of the belief-desire-intention (BDI, hereafter) architecture of software agents [8]. According to this view, an agent is a software module that (i) provides a software interface with the external world, and (ii) contains three components: belief base, plan library and reasoning engine.

The agent's external world consists of the physical environment, as well as possibly other agents. Consequently, the agent interface provides three elements: sensing interface, actuation interface and communication interface. The agent uses its sensing interface to get percepts from its physical environment. The agent uses its actuation interface to perform actions on its physical environment. Finally, the agent uses its communication interface to interact by exchanging messages with other agents.

The *belief base* defines what an agent "knows" or "believes" about its environment at a certain time point. The BDI architecture does not impose a specific structuring of the belief base other than as a generic container of beliefs.

The *plan library* defines the agent's "know-how" and it is structured as a set of behavioral elements called plans. A plan follows the general pattern of event-condition-action rules and it is composed of three elements: triggering event, context and body. The *plan body* specifies a sequence of agent activities. AgentSpeak(L) provides three types of activities: actions, goals, and belief updates. Actions define primitive tasks performed by the agent either on the environment (external actions) or internally (internal actions). Goals represent complex tasks. AgentSpeak(L) distinguishes between test goals and achievement goals. Belief updates represent the assertion $+b$ or the retraction $-b$ of a belief b from the belief base.

The *plan context* is represented by a conjunction of conditions that define the context, in which a plan can be applied. The *triggering event* specifies the event that can trigger the selection of the plan for execution. The plan is actually selected for execution if and only if its context logically follows from the belief base.

Each Jason agent contains a component called a "reasoning engine" that controls the agent execution by "interpreting" the Jason code. The reasoning engine performs a reasoning cycle that consists of a fixed sequence of steps. Basically, each agent performs the following sequence of steps during the reasoning cycle: the agent perceives the environment, updates its belief base, receives communication from other agents, selects an event, selects an applicable plan and adds it to its agenda, selects an item (called intention) for execution (from the agenda) and, finally, executes the next step of the partially instantiated plan that represents the top of the currently selected intention. We can think of each intention as a stack of partially instantiated plans (somehow similar to a call stack) that represents an agent execution thread. The agent agenda is organized as a list of stacks representing the agent intentions. Each stack represents one focus of attention of the agent. Using this approach an agent can execute concurrent activities to manage multiple focuses of attention [4].

The behavior of the reasoning engine is parameterized according to several selection functions that represent nondeterministic choice points of the agent interpreter: S_M (message selection), S_E (event selection), S_O (option selection), and S_I (intention selection).

Engineering Jason agents Jason programming language and system [6] is an implementation, as well as an extension of the AgentSpeak(L) that allows programmers to build experimental MAS. Jason is based on Java. The agent program is written in Jason, while the environment, including the management of the environment state, the agent percepts and the effect of agent actions must be programmed in Java. Additionally, the programmer can customize the agent class, as well as the agent architecture to alter the default behavior of selection and perception functions of the Jason interpreter. This approach has the following advantages: i) the clean separation of the agent logic from the environment logic; ii) the extensibility of the agent sets of percepts and actions to match a specific environment that is the most suitable for the problem in hand; iii) the customization of the agent interpreter to match more specific application requirements.

Agent Code The agents are programmed in Jason following the BDI metaphor. The basic constructs of Jason are beliefs and plans, as has been described above.

Environment Code Environment implementation is realized in Java, by extending the *Environment* class. Usually the programmer has to provide an implementation for the *init* method, to initialize the environment, as well as the *executeAction* method, to update the environment state after the execution of each agent action. Percepts are represented using a *Literal* class and they are added to the environment state using method *addPercept* of class *Environment*. An agent action is a structured term represented using the *Structure* class that provides methods for checking its functor and its arguments.

Agent Class and Architecture Code The agent class can be customized to overwrite the default behavior of selection functions. This can be achieved by sub-classing the *Agent* class of the Jason package to overwrite the definition of the selection Java methods.

The architecture of an agent is responsible with the agent interface with the middleware layer. Basically this is concerned with the agent $\leftrightarrow$ middleware software interfaces

for perceiving and acting, as well as for sending and receiving messages. These interfaces can be customized for example to simulate faults in the effector-sensorial and / or communication subsystem of an agent. The update of the agent architecture can be achieved sub-classing the *AgArch* class to overwrite perception, action, and communication Java methods.

2.2 Temporal Difference Learning in Markovian Environments

In RL, the agent is using the observed rewards (known also as reinforcements) which are part of its percepts, to learn an optimal policy for acting in an uncertain and dynamic environment [11]. RL assumes that the agent environment is uncertain and dynamic, thus leading to the nondeterminism of agent actions. Therefore, the RL adopts a Markovian model of the environment.

Specifically, for a Markovian environment E, we denote with $p(e'|e,a)$ the probability of the environment to transit into state e' given its current state is e and the agent executes action a. Obviously, $\sum_{e' \in E} p(e'|e,a) = 1$ for all $e \in E$ and $a \in Ac$. In practice many of the values $p(e'|e,a)$ will be 0, as taking action a in the current state e possibly reaches only few neighboring states of e from E.

In each state e of the environment the agent receives a reward $R(e)$ represented by a positive or negative real number. Thus, an agent percept is a pair $(e, R(e))$. The agent must decide what to do for each perceived state e of the environment, using its private strategy. This is called a policy and it is defined by a function $\pi : E \rightarrow Ac$, with $\pi(e)$ denoting the action recommended by policy π to the agent in state e.

The agent utility depends on the sequence of rewards received on each state of the environment history. Usually the agent horizon for decision making is considered infinite, while the utility function is additive with discounted rewards: $U_h([e_0, e_1, e_2, \ldots]) = \sum_{i \geq 0} \gamma^i R(e_i)$, where $\gamma \in (0, 1]$ is the discount factor.

While the environment is Markovian, many different environment histories are possible, starting from a given initial state $e_0 = e$, for the same agent policy. Therefore, it is natural to define the true utility of a state e as the expected utility of all environment histories $H(e) = [e_0 = e, e_1, e_2, \ldots]$ starting with e. Basically, each such environment history $H(e)$ can occur with a given probability that depends on the stochastic model of the environment, so the utility $U^\pi(e)$ for the given agent policy π is the weighted average of the utilities of each possible environment history, i.e. $U^\pi(e) = \mathbb{E}[U_h(H(e))]$, where $\mathbb{E}[\cdot]$ denotes the expected utility.

It can be easily shown that U^π satisfies a Bellman system of equations for a given policy π, i.e. $U^\pi(e) = R(e) + \gamma \sum_{e' \in E} p(e'|e, \pi(e)) U^\pi(e')$ for all $e \in E$. So, at least in principle, $U^\pi(e)$ can be determined by solving the Bellman equations. However, in a realistic agent system this is not possible, as the agent does not know the model of the environment. In an extreme scenario, the agent does not even know the set E of states. In fact, the agent discovers the elements of E while it explores the environment.

The agent can use a passive RL method to learn U^π. Methods of passive RL include direct utility estimation, adaptive dynamic programming and temporal difference learning – TDL [9]. In this paper we are considering TDL, due to its simplicity. Nevertheless, the approach can also be extended to other RL methods, either passive or active.

The idea of TDL is to use each observed transition $e \rightarrow e'$ to adjust the value of $U^{\pi}(e)$, so that it better approximates the Bellman equations. The updated value of $U^{\pi}(e)$ is $U^{\pi}(e) + \alpha(R(e) + \gamma U^{\pi}(e') - U^{\pi}(e))$. As can be noticed, TDL uses a very simple mathematical equation and it does not need to estimate the stochastic model of the environment (i.e. the probability distribution $p(e'|e, a)$).

3 Experiments

The starting point of our experiments was the initial implementation of the TDL, using the Jason platform that was reported in [2]. Here, we consider only the updates that were necessary to adapt the setup for running concurrently multiple agents with altered perceptual functions.

3.1 Experimental Setup

We consider a MAS comprising a team of BDI agents that explore the 3×4 rectangular grid firstly introduced in [9], as shown in Figure 1, using a statically defined policy. The goal of each agent is to compute the utility value of each state. The actions available to agents are: up, $down$, $left$ and $right$. When a trial is finalized, i.e. an agent reached a goal state, a new trial must be prepared by generating a new initial state. For this purpose we introduce a special agent action called *null*.

The effect of a normal agent action is uncertain. If the agent attempts to move in a certain direction it will succeed with probability 0.8 or it will fail by changing direction to the left or to the right of the intended direction with probabilities equal to 0.1. Grid squares are represented as pairs of integers $(row, column)$ with $row \in \{1, 2, 3\}$ and $column \in \{1, 2, 3, 4\}$. The grey square from position $(2, 2)$ defines an obstacle. Also the grid walls are considered obstacles. An attempt of the agent to move in the direction of an obstacle will fail, leaving the agent in the initial position. States $(2, 4)$ and $(3, 4)$ are goal states such that $(3, 4)$ is a successful goal state, where the agent receives a positive reward of $+1$, while $(2, 4)$ is a failure goal state, where the agent receives a negative reward of -1. For each of the other states the agent receives a small negative reward of -0.04 with the meaning of the small energy consumed to take an action. We consider the agent policy specified in Figure 1. For example, in state $(1, 1)$ the agent takes the action up, while in state $(3, 2)$ the agent takes the action $right$.

We are using the same application model as in [2]. The model was updated to accommodate the execution of multiple concurrent and possibly different agents that are sharing the same grid environment. In what follows we are only focusing on these updates. For the other details concerning our experimental setup, the reader should consult [2]. Basically the updates were:

i) Definition of multiple agents;
ii) Expanding the environment implementation to support the execution of multiple concurrent agents with individualized perception;
iii) Altering the agent perception function to simulate agents sensing faulty percepts, by updating the agent architecture.

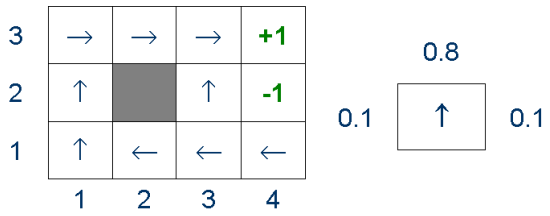

Fig. 1. Sample environment and agent policy.

iv) Altering the agent action function to simulate agents performing unsuccessful actions, by updating the agent architecture.

The Jason platform provides facilities for developing software agents that act concurrently in a shared environment programmed in Java. The Java code that defines the environment, is based on introducing the class *MDPEnvN*, while the Java code that defines the agent architecture is based on introducing the class *TDLAgentArch*. Using these definitions, the script that defines a MAS composed of 4 agents is shown in Figure 2. Node that, according to this definition, the agents are using the JADE [3] infrastructure for exchanging information.

```
MAS team {
    infrastructure: Jade
    environment: MDPEnvN
    agents: tdlAgent agentArchClass TDLAgentArch #4;
}
```

Fig. 2. Script that defines a MAS composed of 4 agents of type *tdlAgent*.

We have updated the environment Java code introduced in [2] that defines the environment *MDPEnvN* class, as follows:

i) We added two member variables that define the number of agents, as well as the root of the name of our agents. By default, an agent member of our *team* MAS has the name composed of the given name `"tdlAgent"` (see Figure 2) and a counter (taking values from 1 to 4 for the MAS defined in Figure 2).
ii) We added a member variable representing the array of agent names.
iii) We added a member variable representing the map that associates each agent to its current location on the grid. The association is using the agent name as key.
iv) We updated the *updatePercepts* and *executeAction* methods that perform individualized perception and action, by adding a parameter to represent the agent name.
v) We added a constructor of the *MDPEnvN* class for initializing the array of agent names and the mapping of agent names to their initial locations.
v) We updated the code of the *init* method to initialize the agents' percepts.

The code that describes these updates is presented in Figure 3. Note that only a part of the Java code of the *executeAction* method is presented. We have shown the fragment that determines the current agent position and that sets the next agent position, after the execution of the agent action.

3.2 Experiments with Agents with Faulty Perception

We considered two kind of faults in the agents' perception function: (i) perception affected by intermittent faults (IF faulty perception) and (ii) perception affected by Gaussian noise (GN faulty perception). We ran an experiment with 4 agents such that the first agent had IF faulty perception, the second agent had a GN faulty perception, while the third and fourth agent were not faulty.

We followed the hints from [4] for the simulation of faulty perception, by overwriting the *perceive* method of the agent architecture class. In order to simplify the coding, we developed a single agent architecture class that was named *TDLAgentArch*. This class provides different implementations of the *perceive* method, depending on the agent type (here represented by the agent id). The code of the *perceive* method is shown in Figure 4.

For the agent with IF faulty perception, we altered the reward perceived between the 2000-th and the 2010-th perception by adding a value of 100. For the agent with GN faulty perception we always added to the perceived reward a Gaussian signal with mean 0 and standard deviation 1.0 (see Figure 4).

We have reused the experimental context from [2]. The Jason program was not altered for the experiments with faulty perception. So basically we have reused the static policy that defines agents' actions in each state and the learning factor (the same for all agents).

We ran 100000 iterations for each agent. Taking into account that the environment is Markovian, this resulted in different numbers of trials for each agent: 18937 for agent 1, 18921 for agent 2, 18647 for agent 3, and 19071 for agent 4. Below, we only present selected results obtained for agent 1 (with IF faulty perception) and agent 2 (with GN faulty perception).

Figure 5 presents values of utilities $u(1,1)$ and $u(2,3)$ for the agent with IF faulty perception. We can observe that sometime after the 2000-th iteration the utilities are severely increased (comparing to their actual value). However this effect being intermittent, it does not last too long. So, sometime before the 3000-th iteration the values of the utilities appear to converge to their correct values. Note that at this time the effect of the intermittent faulty perception was canceled. Actually (not shown on Figure 5) $u(1,1)$ converges to a value close to 0.7, while $u(2,3)$ converges to a value close to 0.65. Note that Figure 5 displays the values of utilities only up to the 9000-th iteration, although we ran in our experiment 100000 iterations, in order to better capture, in the figure, the effect of the intermittent fault.

Figure 6 presents the values of utilities $u(1,1)$ and $u(2,3)$ for the agent with GN faulty perception. The effect of the Gaussian noise is pretty obvious, being more accentuated at the start of the exploration process. As agent 2 is progressing, the effect of the noise is decreasing, and we can observe a convergence of the utilities to values that we found similar to those obtained by the other agents, either faulty or non-faulty.

```
public class MDPEnvN extends Environment {
    // ...
    final int nAgents = 4;
    final String agName = new String("tdlAgent");
    String agentsNames[] = new String[nAgents];
    Map<String,AgentPosition> agentsPositions =
        Collections.synchronizedMap(new HashMap<String,AgentPosition>(1));

    public MDPEnvN() {
        // Initialize the array of agents names
        for (int i=0;i<nAgents;i++) { agentsNames[i] = agName + (i+1); }
        // Set initial position of each agent to (agentStartRow,agentStartColumn)
        for (int i=0 ; i<nAgents ; i++) {
            agentsPositions.put(agentsNames[i],
                new AgentPosition(agentStartRow,agentStartColumn));
       }
    }

    public void init(String[] args) {
        for (int i=0;i<nAgents;i++) { updatePercepts(agentsNames[i]); }
    }

    private void updatePercepts(String agent) {
        int agentRow,agentColumn;
        // Remove previous percepts of the agent
        clearPercepts(agent);
        // Determine the agent's current position
        AgentPosition ap = agentsPositions.get(agent);
        agentRow = ap.getRow();
        agentColumn = ap.getColumn();
        // Determine the literal percept and add it to the list of agent's percepts
        double r = rewards[agentRow][agentColumn];
        String agentPos = new String("pos(");
        agentPos += agentRow; agentPos += ",";
        agentPos += agentColumn; agentPos += ",";
        agentPos += r; agentPos += ",";
        agentPos += (isExitState(agentRow,agentColumn) ? "t" : "n");
        agentPos += ")";
        addPercept(agent,Literal.parseLiteral(agentPos));
    }

    public boolean executeAction(String ag, Structure action) {
        // ...
        int agentNewRow,agentNewColumn;
        int agentRow,agentColumn;
        AgentPosition ap = agentsPositions.get(ag);
        agentRow = ap.getRow(); agentColumn = ap.getColumn();
        // ...
        if (! walls[agentNewRow][agentNewColumn]) {
            agentsPositions.replace(ag,new AgentPosition(agentNewRow,agentNewColumn));
        }
        updatePercepts(ag); // update the agent's percepts for the new
                            // state of the world (after this action)
        return true;        // all actions succeed
    }
}
```

Fig. 3. Updates of the environment Java code.

```
public class TDLAgentArch extends AgArch {
    final String rootAgName = new String("tdlAgent");
    final int nAgents = 4;
    int[] perceptCount = new int[nAgents];
    final double stdDev = 0.1;
    final double mean = 0.0;
    Random r = new Random();

    // Agents' names are tdlAgent1, tdlAgen2, ...
    private int getAgId() {
        String agName = getAgName();
        String id = agName.replace(rootAgName,"");
        return (new Integer(id)).intValue();
    }

    public TDLAgentArch () {
        for (int i=0; i<nAgents; i++) {
            perceptCount[i] = 0;
        }
    }

    public List<Literal> perceive() {
        // Get the default perception
        List<Literal> per = super.perceive();
        // Alter percept
        int agId = getAgId();
        double v3 = 0.0;
        if (per != null) {
            Iterator<Literal> ip = per.iterator();
            if (ip.hasNext()) {
                perceptCount[agId-1]++;
                Literal l = ip.next();
                // Third argument of the percept per(Row,Col,Reward,TorN) is the reward
                NumberTerm t3 = (NumberTerm)(l.getTerm(2));
                if ((agId == 1) && // Agent with IF faulty perception.
                   (perceptCount[agId-1] >= 2000) && (perceptCount[agId-1] <= 2010)) {
                    // Alter percepts received between 2000th and 2010th perception.
                    try { v3 = t3.solve(); }
                    catch (NoValueException e) {}
                    v3 = v3+100.0;
                }
                else if (agId == 2) { // Agent with GN faulty perception
                    double noise = r.nextGaussian()*stdDev + mean;
                    try { v3 = t3.solve(); }
                    catch (NoValueException e) {}
                    v3 = v3+noise;
                }
                t3 = new NumberTermImpl(v3);
                l.setTerm(2,t3);
            }
        }
    }
    return per;
}
```

Fig. 4. Java code for simulating faulty perception.

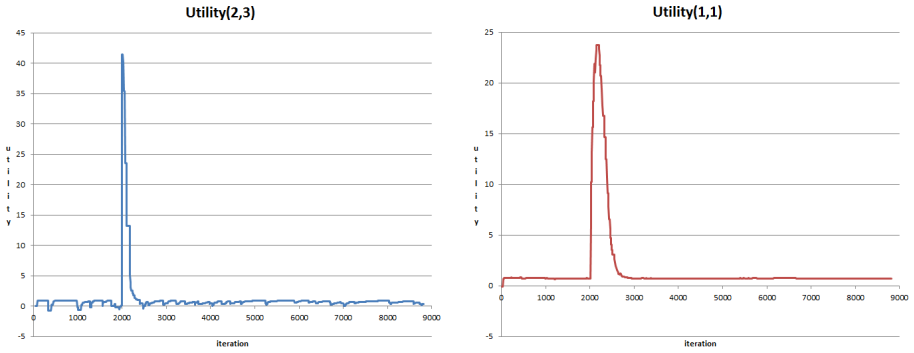

Fig. 5. Results for agents with IF faulty perception.

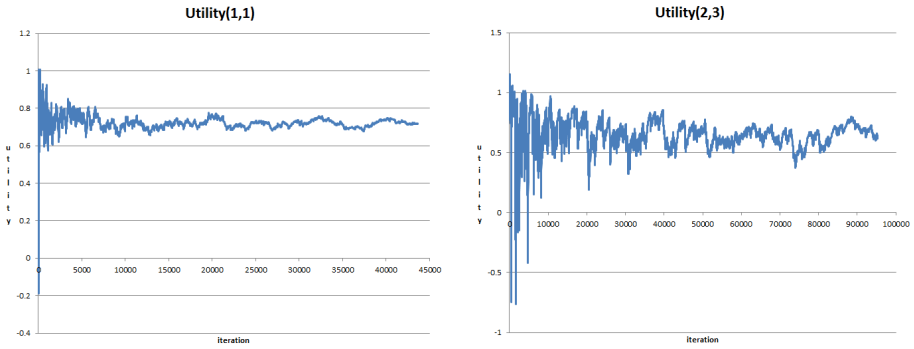

Fig. 6. Results for agents with GF faulty perception.

3.3 Experiments with Agents with Faulty Action

We also performed experiments with agents experiencing unsuccessful actions. In order to keep things simple, we have considered an agent for which the actions $left$, *right*, *up*, and *down* (but not *null* !) can fail. Failure is understood here as: i) the action does not produce any effect in the environment; ii) the result of the action reports failure to the agent, causing the corresponding agent plan to fail. Simulation of unsuccessful actions was achieved by overwriting method *act* of the *AgArch* class, as shown in Figure 7.

There are already proposed solutions, in the literature, for dealing with plan failure [5]. In our experiment we adopted a simple solution, by using contingency plans.

According to the model introduced in [2], TDL agents were defined as proactive agents with the goal *!keep_move* of continuously exploring the environment. Achievement of *!keep_move* assumes to update the utility function based on the current percept and to continue to explore the environment via goal *!continue_move*. Then, for the achievement of *!continue_move*, if the exploration was not terminated by reaching the

```
public class TDLAgentArch extends AgArch { // ...
    public void act(ActionExec action, List<ActionExec> feedback) {
        int i = r.nextInt(2); // randomly set the flag to indicate faulty action
        String afunctor = action.getActionTerm().getFunctor();
        if (! afunctor.equals("null") && (i==0)) {
            action.setResult(false);
            feedback.add(action);
        }
        else { // calls the default implementation
            super.act(action,feedback);
        }
    }
}
```

Fig. 7. Script that defines the architecture of an agent that can perform faulty actions.

upper bound of the number of iterations (i.e. when *below_limit(M)* context condition in Figure 8 is true), there are two cases:

i) a new trial was terminated by reaching a finals state (i.e. *non_terminal_state(St)* context condition from Figure 8 is true); in this case the agent executes a *null* action that, according to our assumptions, cannot fail;
ii) the last explored state is not terminal (i.e. *terminal_state(St)* context condition from Figure 8 is true); in this case the agent selects a move according to its static policy. In this case the action can fail, producing a failure of the *!continue_move* goal.

Therefore, we have updated the Jason plan library, introduced in [2] for the implementation of TDL agents, by adding a contingency plan using the blind commitment strategy for the achievement of the *!continue_move* goal. This simply restates the *!continue_move* goal causing the failed action to be re-executed until it succeeds. The Jason code is shown in Figure 8.

```
+!continue_move(M,St) : non_terminal_state(St) & below_limit(M) <-
   !do_one_move(St);
   !!keep_move.
+!continue_move(M,St) : below_limit(M) & terminal_state(St)  <-
   ?last_trial(N); N1 = N+1; -+last_trial(N1);
   ?check_trial(N1);
   null;
   -+last_action(null);
   !!keep_move.
+!continue_move(M,St) : not below_limit(M) <-
   ?last_trial(N);
   .print("END OF RUN. TRIALS: ",N," ITERATIONS: ",M);
   ?print_results.
-!continue_move(M,St) : below_limit(M) <-
   !continue_move(M,St).
```

Fig. 8. Plans associated to the *!continue_move* goal.

We ran an experiment by allowing all the agents (either perception faulty or perception non-faulty) to execute actions that can fail according to this model. All the agents were able to finalize the experiment with success producing appropriate utility values.

The only inconvenience that was added by introducing faulty actions (according to this model) was the increase of the learning time. This was caused by the necessity to re-execute failing actions until they were successful.

4 Conclusion

In this paper we show how multiple BDI agents enhanced with TDL capabilities learn their utility function, while they are concurrently exploring an uncertain environment. Our main results are related to the programming aspects of the agents using the Jason agent-oriented programming language and Java. We provide experimental results showing the behavior of different agents acting in a Markovian grid environment: agents with the perception function affected by intermittent faults and Gaussian noise, as well as agents for which their action function can fail. As future work we plan to expand these results in at least two directions: i) by considering active learning strategies, for example Q-learning; ii) by allowing teams of agents to cooperate by exchanging messages while they are exploring the environment, with the goal to either improve the learning process or to exclude faulty and/or malicious agents from the team.

References

1. Bădică, C., Budimac, Z., Burkhard, H.-D., Ivanović, M.: Software agents: Languages, tools, platforms. Comput. Sci. Inf. Syst. 8 (2): 255–298 (2011) doi: 10.2298/CSIS110214013B
2. Bădică, A., Bădică, C., Ivanović, M., Mitrović, D.: An Approach of Temporal Difference Learning Using Agent-Oriented Programming. In: Proc. 20th International Conference on Control Systems and Computer Science (CSCS'2015), 735–742, IEEE, (2015) doi: 10.1109/CSCS.2015.71
3. Bellifemine, F. L., Caire, G., Greenwood, D.:Developing Multi-Agent Systems with JADE, ser. Wiley Series in Agent Technology. John Wiley & Sons Ltd, (2007)
4. Bordini, R. H., Hübner, J. F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason, ser. Wiley Series in Agent Technology. John Wiley & Sons Ltd, (2007)
5. Hübner, J. F., Bordini, R. H., Wooldridge M.: Programming Declarative Goals Using Plan Patterns. In: Baldoni, M., Endriss, U.: Declarative Agent Languages and Technologies IV. Volume 4327 of the series Lecture Notes in Computer Science, 123–140, Springer, (2006) doi: 10.1007/11961536_9
6. Jason: a Java-based interpreter for an extended version of AgentSpeak. http://jason.sourceforge.net//. Accessed in February, 2016.
7. Jennings, N. R. Wooldridge, M.: Applications of Intelligent Agents. In: Jennings, N. R., Wooldridge, M. J. (eds.): Agent Technology. 3–28, Springer Berlin Heidelberg, 1998 doi: http://dl.acm.org/citation.cfm?id=277789.277799
8. Rao, A. S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Van de Velde, Walter and Perram, J. W. (eds.): Agents Breaking Away. Lecture Notes in Computer Science 1038, 42–55, Springer Berlin Heidelberg, (1996) doi: 10.1007/BFb0031845
9. Russell S., Norvig, P.: Artificial Intelligence: A Modern Approach (3rd ed.). ser. Prentice Hall Series in Artificial Intelligence. Prentice Hall, (2010)
10. Shoham, Y.: Agent-Oriented Programming. Artificial Intelligence 60 (11): 51–92, 1993 doi: 10.1016/0004-3702(93)90034-9
11. Sutton, R. S.: Learning to predict by the methods of temporal differences. Machine Learning 3 (1): 9–44 (1998) doi: 10.1007/BF00115009