Rafael H. Bordini, Louise A. Dennis and Yves Lespérance (eds)
Pre-proceedings of the 7th International Workshop on Engineering Multi-Agent Systems (EMAS 2019)
13th-14th May 2019, Montreal, Canada
cgi.csc.liv.ac.uk/~lad/emas2019/accepted/EMAS2019_paper_33.pdf

From Programming Agents to *Educating* Agents – A Jason-based Framework for Integrating Learning in the Development of Cognitive Agents

Michael Bosello and Alessandro Ricci

Alma Mater Studiorum – Università di Bologna
Department of Computer Science and Engineering, Cesena Campus
michael.bosello@studio.unibo.it, a.ricci@unibo.it

Abstract. Recent advances and successes of machine learning techniques are paving the way to what is referred as *Software 2.0 era* and *cognitive computing*, in which traditional programming and software development is meant to be replaced by such techniques for many applications. If we consider agent-oriented programming, we believe that such developments trigger new interesting scenarios blending cognitive architecture such as the BDI one and techniques like Reinforcement Learning (RL) even more deeply compared to what has been proposed so far in the literature. In that perspective, we aim at exploring the integration of cognitive agent-oriented programming based on BDI with learning techniques so as to systematically exploit them in the agent development stage. The approach should support the design of BDI agents in which some plans can be explicitly programmed and others instead can be learned by the agent during the development/engineering stage. In that view, the development of an agent is metaphorically similar to an *education process*, in which first an agent is created with a set of basic programmed plans and then grow up in order to learn plans to achieve the goals for which the agent is meant to be designed. This paper present and discuss this medium-term view, introducing a first model for a BDI agent programming framework integrating RL, a first implementation based on Jason programming language/platform and sketching a roadmap for this research line.

1 Introduction

Machine learning and cognitive computing techniques have been getting a momentum in recent years, thanks to several factors, including theoretical developments in contexts such as (deep) neural networks, reinforcement learning, Bayesian networks, the availability of big data and the availability of more and more powerful parallel computing machinery (GPU, cloud) [2,12,13,16]. Their deeper and deeper impact in real-world applications is celebrating a new "AI Spring" era, which is generating a strong debate in the literature as well [19]. Actually, the impact is not only about applications but also about how applications are *programmed* and *engineered*. In particular, a vision of *Software 2.0 era* is emerging [17], in which traditional programming and software development is meant to be more and more replaced by e.g. machine learning and cognitive computing techniques, towards the "the end of programming" era [16,10].

Besides the hype and the marketing-oriented claims, if we consider *agent-oriented programming* [23], and – more generally, multi-agent systems (MAS) engineering – we believe that such recent developments would trigger new interesting scenarios blending cognitive architectures such as the BDI one [21] and techniques like Reinforcement Learning (RL) [27] even more deeply than what has been already proposed so far in literature. As far as authors' knowledge, existing research integrating BDI-based agents and MAS with learning techniques mainly focused on improving agent adaptation, exploiting learning to improve e.g. plan or action selection at runtime. As a further approach, our objective is to explore the integration of cognitive agent-oriented programming based on BDI with learning techniques so as to systematically exploit them in the agent development stage. The basic idea is that an agent developer could integrate the explicit programming of plans – when developing a BDI agent – with the possibility that, for some goals, it would be the agent itself to learn the plans to achieve them, by exploiting reinforcement learning based techniques. This is not only for a specific ad hoc problem, but as a general feature of the agent platform.

In that view, the development of an agent is metaphorically similar to an *education process*, in which first an agent is created with a set of basic programmed plans and then *grow up* in order to learn plans to achieve the goals for which the agent is meant to be designed. We believe that this vision would trigger interesting research directions about the evolution of agent-oriented programming in the *software 2.0 era*. In the remainder of the paper, we present and discuss this view, with a first proof-of-concept framework based on the Jason agent platform [6,7].

After giving a background and an account about related works (section 2), we first describe our approach integrating learning in the loop of BDI agent programming, using AgentSpeak(L) [20] as reference model (section 3). Then, we describe an implementation of the framework on top of the Jason agent language/platform (section 4), including some testing using a simple example and a discussion (section 5). We close the paper sketching a roadmap for this research line (section 6).

2 Background and Related Works

In this section, we first provide an overview about basic concepts of Reinforcement Learning (RL) – taking [27] as main reference – and then an account of existing research works in the context of agent-oriented programming – especially focusing on BDI-based model [21] – proposing an integration with RL. It is worth remarking that in this setting we do not intend to consider the latest advances in RL, but just the core foundational layer useful to present and discuss our approach.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning method with the key idea that a goal-oriented *agent* learns how to fulfill a task by interacting with its *environment*.

Any problem of learning *goal-directed* behavior can be reduced to three signals passing back and forth between an agent and its environment: one signal to represent the choices made by the agent (the *actions*), one signal to describe the basis on which the choices are made (the *states*), and one signal to define the agent's goal (the *rewards*).

A state is defined as whatever information is available to the agent about its environment, some of what makes up a state could be based on the memory of past perceptions or even be entirely mental or subjective, i.e. the states can be anything we can know that might be useful in the decision process. The agent must decide what action to take as a function of whatever state signal is available. The actions too might be either internal – changing the agent's mental state – or external – affecting the environment. For example, some actions might control what an agent chooses to think about, or where it focuses its attention. The reward is a single real number obtained at each interaction step that the agent seeks to maximize *over time*. The reward signal thus defines what the good and bad events for the agent are, i.e. it is your way of communicating to the agent what it must achieve.

We refer to each successive stages of interaction between the agent and the environment as time steps. In the case of a BDI agent, a reasoning cycle can be pretty assumed as a step. In some applications, there is a natural notion of final time step, that is, when the agent-environment interaction breaks naturally into subsequences that are referred as *episodes*. Related to BDI, this is the case of agents pursuing *achievement goals*. In many cases, the agent-environment interaction does not break naturally into identifiable episodes but goes on continually without limit — these are called *continuing tasks*—i.e., *maintenance goals* in the BDI case.

The agent learns a *policy* as a result of the learning process. A policy is a function that maps states to actions. we seek to learn and exploit an optimal policy, but we need to behave non-optimally to explore all the possibilities. A classic method to balance the *exploitation* and *exploration* phases is to use an ε-greedy policy with which the agent behave greedily but there is a (small) ε probability to select a random action.

In many cases of interest, the agent has only *partial* information about the state of the world, so, the states signal is replaced by an *observations signal* that depend on the environment state but provide only partial information about it. In the BDI case, this is directly modeled by percepts and, therefore, by beliefs about the environment. The fundamental property of a state, known as the Markov property, is that it can be used to predict the future. A stochastic process has the Markov property if the conditional probability distribution of future states depends only upon the current state, not on the sequence of events that preceded it. From the observations, the agent recover an approximate state i.e., a state that may not be Markov. Actually, we can partially drop the Markov property; however, this implies that long-term prediction performance can degrade dramatically. An approximate state will play the same role in RL algorithms as before, so we simply continue to call it a state.

2.2 Integrating RL into BDI Agents and AOP

Generally speaking, the integration of learning capabilities has been a main research topic in agents and MAS literature since their roots [29]. A first work providing preliminary results about integrating learning in BDI multi-agent systems is [14], proposing an extension of the BDI architecture to endow agents with learning skills, based on induction of logical decision trees. Learning, in that case, is about plan failures, that an agent should reconsider after its experience.

Other works in literature exploits RL to improve plan/action selection capability in BDI agent, making them more adaptive [25,1,24]. In [18] an extension of BDI is proposed so as to get a model of decision making by exploiting the ability to learn to recognize situations and select the appropriate plan based upon this.

In [3,4] Jason is used to realize Temporal Difference (TD) and SARSA, two reinforcement learning methods, in order to face the RL problem with a more appropriate paradigm which has been remarkably effective. [28] proposes a hybrid framework that integrates BDI and TD-FALCON, a neural network based reinforcement learner. BDI-FALCON architecture extends the low-level RL agent with a desire and intention modules. In this way, the agent has explicit goals in the desire module instead of relying on an external signal, enhancing the agent with a higher level of self-awareness. The intention module and its plan library permit to reason about a course of actions instead of individual reactive responses. If there isn't a plan for a situation, the agent performs the steps suggested by the FALCON module and, if the sequence succeeds, a new plan is created with indications about the starting state, the goal state, and the actions sequence. When the agent uses a plan, it updates the confidence of the plan according to the outcome.

Also in [15] a hybrid approach BDI-FALCON is proposed. Here, the focus is on the abstraction level: BDI provides a high-level knowledge representation but lacks learning capabilities, meanwhile low-level RL agents are more adept at learning. The layered proposal wants to retain both advantages. An alternative vision is provided in [11], where a policy is learned and then is used to generate a BDI agent.

3 The Basic Idea

The simple idea of this paper is to extend the BDI agent development process with a learning stage in which we can specify plans in the plan library whose body is not meant to be explicitly programmed but learned, using a learning by experience technique. In so doing, the development of an agent accounts for: *(i)* explicitly programming plans as in the traditional way—we will refer to them as *hard plans*; and *(ii)* implicitly defining plans by allowing the agent itself to learn them by experience (*soft* plans). Soft plans are meant to become part of the plan library like hard plans and can be selected and used at runtime – in terms of instantiated intentions – without any difference (but allowing for *continuous learning*, if wanted). Actually, at runtime, soft and hard plans are treated in a uniform way: intentions are created to carry on plan execution, hard plans could trigger the execution of soft plans and vice versa.

To support the learning stage, we would need to run the agent in a proper environment supporting this learning by experience, like the simulated environments typically used in RL scenarios. Besides, before deploying the agent, there could be some *assessment* of the soft plans, eventually using an assessment environment which could be different from the one used for training. The assessment is actually very similar to a traditional validation stage, including tests that consider the soft plans and their integration with hard plans. If the agent overcomes the assessment, it can be deployed—otherwise, the process goes back to the learning stage, possibly changing also plans in the hard part.

Given this general idea, in the remainder of the section we first introduce the model integrating key concepts of RL into a BDI framework, and then we describe an extension of the BDI reasoning cycle supporting the learning process. We will use AgentSpeak(L) [21] and Jason [6] as concrete abstract and practical BDI-based languages—nevertheless, we believe that the core idea is largely independent of the specific BDI agent programming language or framework adopted.

To exemplify the approach, we will use the simple gridworld example (p. 80, [27]), in which an agent is located into a bi-dimensional grid environment, where it has to reach some destination cells without knowing in advance the best path for doing it.

3.1 A First Model

In devising the model, we aim at abstracting from the specific RL algorithm that can be used. To that purpose, we consider key common RL concepts – observations, actions, rewards, episodes – and how they are represented into a BDI setting (see Fig. 1, top). These concepts are used by a component – referred as *RL reasoner* – extending the BDI reasoner (interpreter) (see Fig. 1, bottom). The classic BDI reasoner handles the hard plans – i.e., with a body – and the RL reasoner handles the soft plans—whose behavior is learned.

Observations are modeled as a subset of the agent beliefs that will be used by the particular algorithm to construct the state, including those that are necessary to understand when a goal is achieved. Recall that the observations are a generalization of states and if we want to represent a Markov state we just need to include all its aspects as observations. In the gridworld case, for instance, the agent must reach a specific position moving towards four directions. In this case, the observations include the current position of the agent (`pos(X, Y)`) and a belief about having reached the target (`finish_line`)—if this belief is missing, it means that the agent has not achieved the target in the current state.

The *action set* can contain both primitive actions (a BDI agent action) and compound actions (a BDI agent plan) so that representing different levels of planning granularity. To have a common representation for both cases, we represent actions as plans, i.e. the set of actions selectable by the RL reasoner is a subset of the plan library defined by the plans which are relevant for the goal and applicable in the current context. In Fig. 1, these plans are still referred as *Actions* in the plan library, and as *Behavior* (i.e., the learned policy) when instantiated at runtime, wrapped into an intention. In the gridworld example, the action set has one parametric plan to move the agent in one direction: `!move(D)`, where `D` could be `up`, `down`, `left`, `right`.

Rewards are represented by rules reflecting agent desires—we call them *Motivation Rules*. These rules make it possible to weight the current situation of an agent according to some goal to be achieved. We can see these rules as the generators of internal stimuli in the agent like a reward signal in neuroscience, which is a signal internal to the brain that influences decision making and learning. The signal might be triggered by the external environment, but it can also be triggered by things that do not exist outside the agent and which can be represented as beliefs as well. We move the reward, that in RL comes

RL	Proposed BDI+ construct	Representation in BDI
Observations	Belief about Learning	Belief subset
Actions	Actions	Relevant Plans
Rewards	Motivational Rule	Belief Rule
Episode	Terminal Rule	Belief Rule
Policy	Behavior	Intention

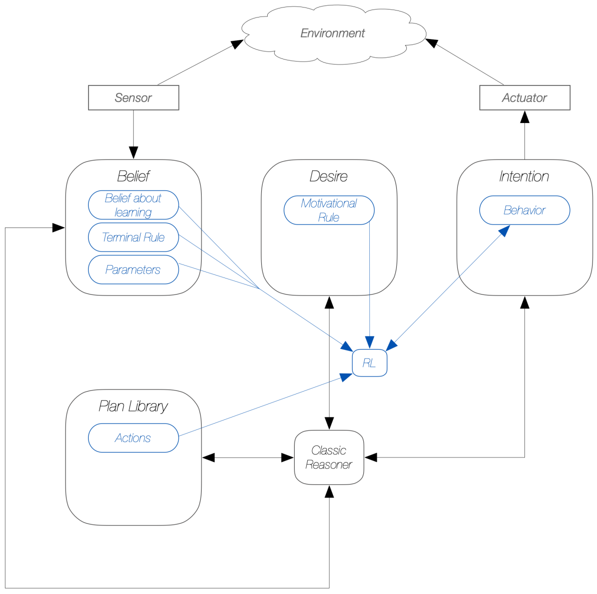

Fig. 1: (Top) The mapping between RL concepts and their counterpart in the BDI model. (Bottom) A graphic representation of the BDI model with the addition of our constructs.

from the environment, into the agent. This is crucial to separate the agent desires from the environment so as to allow an agent to define its own goals and rewards about them. In the AgentSpeak(L)/Jason model, we can represent Motivational Rules as Prolog-like rules in the belief base:

```
reward(Goal, Reward) :- < some condition over the belief base >
```

In the gridworld example, the motivational rule will give a positive reward when the finish line is reached (while pursuing a `reach_end` goal) and a negative reward in other steps.

```
reward(reach_end, 10) :- finish_line.
reward(reach_end, -1) :- not finish_line.
```

Finally, we must include a notion of *episode*. An episode is an event or a group of events occurring as part of a sequence. Like in the case of rewards, we can assume that the agent (designer) may have her vision about how to define episodes starting from a relevant

```
B ← B'; /* B' are initial beliefs */
I ← I';   /* I' are initial intentions */
RL Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize Q(s, a), for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that Q(terminal, ·) = 0

while true do
    get next percept $\rho$ via sensors;
        B ← brf(B, $\rho$);
        D ← options(B, I);
        I ← filter(B, D, I);
        $\pi$ ← plan(B, I, Ac); /* Ac is the set of actions */
        while ( ($\pi$ is learning plan and S is not terminal) or
                    not ($\pi$ is learning plan and empty($\pi$)) ) and
                    not (succeeded(I, B) or impossible(I, B)) do
            if $\pi$ is learning plan then
                O ← related(B, I)
                S' ← state(O)
                R ← motivationalRule(B)
                Choose A' from $\mathcal{A}$(S') using policy derived from Q //(e.g. $\varepsilon$-greedy)
                If S not null and A not null then
                    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma^* Q(S', A') - Q(S, A)]$
                end-if
                S ← S'
                A ← A'
                execute(A)
            else
                $\alpha$ ← first element of $\pi$;
                execute($\alpha$);
                $\pi$ ← tail of $\pi$;
            end-if
            observe environment to get next percept $\rho$;
            B ← brf (B, $\rho$);
            if reconsider(I, B) then
                D ← options(B, I);
                I ← filter(B, D, I);
            end-if
            if not sound($\pi$, I, B) then
                $\pi$ ← plan(B, I, Ac)
            end-if
        end-while
end-while
```

Fig. 2: BDI practical reasoning extended with RL based on SARSA, in pseudo-code.

ensemble of situation. This condition is well established by a rule that asserts in which belief state a coherent group of events ends up in an episode, after which a new episode begins. We refer to this rule as a *Terminal Rule*.

```
episode_end(Goal) :- < some condition over the belief base >
```

In the gridworld example, the episode for achieving a `reach_end` goal ends when the finish line is reached:

```
episode_end(reach_end) :-  finish_line.
```

It is worth noting that this approach could be applied only in the case of achievement goals. In the case of maintenance goals (*continuing tasks* in [27]) we would need to reconsider how an episode is modeled.

3.2 Extending the Reasoning Cycle

In our framework, a BDI agent is then equipped with general-purpose learning capabilities that are triggered as soon as a soft plan must be learned, for some goal. Fig. 2 shows the pseudo code of a classic BDI agent reasoning cycle (as defined in [30,7]) extended with such learning capabilities The RL algorithm used in the example is SARSA, adapted for the context; our additions are in red.

In the standard cycle, the function *plan* in line (11) generates a plan to achieve the intention *I*, we consider that this function is extended to include the case in which the agent, for any reason, is not able to produce a plan to pursue the intention. In this case, the agent can choose to instantiate a soft plan that relies on RL to achieve the intention. The execution loop condition remains the same for the hard plans; instead, a soft plan continues until a terminal state is met or until the intention succeeds or becomes impossible to reach.

The execution of soft plans is between (16) and (25). First, the agent extracts the observations from the belief base according to the goal to achieve (16), then, a state is built from these observations (17). The reward is obtained from the motivational rule that quantifies the fulfillment of the goal in accordance with the current beliefs (18). The action is selected following the RL policy and the current value function *Q* (19). In (21) the value-action function is updated. Finally, the selected action is carried out (25).

4 Proof-of-Concept Implementation in Jason

We developed a first proof-of-concept implementation on top of Jason, exploiting its extensibility. Knowledge required by the RL part is uniformly represented by specific beliefs, referred as *beliefs about learning*. The framework abstracts from the specific RL algorithm but, depending on the characteristics of the problem, there will be different constraints on it. Critical factors are the knowledge about the environment and the state/action space dimensionality: if the state/action space dimensionality increases or more environment features are hidden (Markov property), then the constraints on the algorithm will be more stringent. To deal with this problem, we consider the possibility for the programmer to specify some domain knowledge so as to reduce state/action space and thus obtaining a more efficient/effective learning.

All the RL items are represented as beliefs, including rules, and plans. In this way, the agent can control everything related to the reinforcement learning process. For the BDI agent, RL is a black box and vice versa. We can see the black box as a block that we can change without affecting the agent and that can implement any RL algorithm.

4.1 RL Concepts Representation in Jason

All beliefs about learning include as first parameter a ground term representing the goal for which we want a soft plan, i.e. whose plan is learned. This is useful to support multiple goals with soft plans at the same time. In the gridworld, for instance, we identify the goal with `reach_end`.

In order to reduce the state space, we need to declare which beliefs shall be considered relevant for a goal, so that they will be used as observations. We do this with beliefs `rl_observe(G, O)`, where *G* is a ground term that refers to the goal and *O* is the list of the beliefs that will be considered for the goal. In gridworld example:

```
rl_observe(reach_end, [ pos(_,_) ]).
```

As introduced in the previous section, Motivation Rules defines the rewards for some goal given the current context:

```
rl_reward(G, R) :- ...
```

where *G* is the goal and R is a real number. The body of the rule represents the state for which this reward must be generated, i.e. it represents the goal state. In the gridworld example:

```
rl_reward(reach_end, 10) :- finish_line.
rl_reward(reach_end, -1) :- not finish_line.
```

At each execution, the RL reasoner gets the sum of all the rewards of the Motivational Rules for which the body is true on the basis of the agent beliefs.

Similarly, Terminal Rules are in the form of `rl_terminal(G) :- ...`, asserting when the end of an episode is reached. In the gridworld example:

```
rl_terminal(reach_end) :- finish_line.
```

The action set is represented as a set of (hard) plans, identified by an `@action` annotation: `@action[rl_goal(`g_1`, ..., `g_n`)]` where g_1`, ..., `g_n is the list of goals for which the plan/action can be used. In the gridworld example:

```
@action[rl_goal(reach_end),
     rl_param(direction(set(right, left, up, down)))]
+!move(Direction) <- move(Direction).
```

This is used to inform the RL reasoner that the `move` action, wrapped into the corresponding plan, is relevant for learning how to achieve the `reach_end` goal. The annotation allows for specifying also parameters that are used in the action/plan, specifying the range of the values: `@action[rl_goal(`g_1`, ..., `g_n`), rl_param(`p_1`, ..., `p_m`)]` where p_1`, ..., `p_m is the list of literals whose names match the names of the variables—these literals must contain a predicate that defines the type of the parameter and its range. To define an action space in which the action set is not the same in all states we can use the context of the plan—if the context is not satisfied for the current state, the plan will not be considered by the RL algorithm.

RL algorithm parameters can be specified as beliefs, enabling the complete control of the learning process by the programmer and the agent. In the gridworld example, some parameters are:

```
rl_parameter(alpha, 0.26).
rl_parameter(gamma, 0.9).
rl_parameter(policy, egreedy).
```

Finally a couple of internal actions – `rl.execute(G)` and `rl.expectedreturn(G, R)` – are provided to drive and inspect the learning process.

The internal action `rl.execute(G)` makes it possible to perform one run (episode) of the learning process and/or execute the soft plan. The action, implemented in Java, wraps the core part of the RL algorithm. To that purpose, the Java bridge makes it possible to reuse existing RL libraries, when useful, including libraries written in other languages such as Python ones. The action carries on and improves the soft plan under learning and its execution completes when the episode is completed (or until an action failure). The soft plan's intention is carried out like any other intention—so the RL execution competes with the other intentions for the agent attention and further execution. Typically, a full learning process involves the repeated execution of learning episodes, in this case by executing multiple times the `rl.execute` action.

The internal action `rl.expectedreturn(G, R)` gets the estimate of future rewards `R` for the goal `G` on the basis of the current state and learned policy, i.e. the *expected return*. This could be used to understand the performance of the leaned soft plan for some goal given the current situation of the agent. For instance, if the expectation of the learned behavior in the current state is poor, we can fall back on another plan. As a result, we obtain a notion of *context* for soft plans.

4.2 Jason-RL Reasoning Cycle

The Jason reasoning cycle defines how the Jason interpreter runs an agent program, it can be seen as a refinement of the BDI decision loop [7]. There are ten main steps: in our framework, some of these steps are extended to include learning aspects. Figure 3 shows our extended architecture based on the original one; the red components are the extensions. The detailed description of the cycle can be found in [7]. In the following, we focus on our extensions. In a learning agent, after the update of the belief base (2a), the maps that track the *Belief about Learning* are updated to reflect the new belief; we call this process *Observation Update Function* OUF (2b). In this way, when the observations are required, the agent doesn't need to iterate multiple time the belief base. In step (7a), when the plan's context is bound to the expected reward, the value is asked to the RL black box and then verified against the threshold (7b). Finally, a new step (11) shall be added to the sequence when the next action of the intention selected in (9) is the RL execution. At this stage, the information that the RL process needs in order to continue shall be provided to it. The observations and the parameters are taken from the belief base, plus, the Motivational Rules and the Terminal Rules are checked against the belief base to retrieve the reward and the terminal status. The RL reasoner needs also the set of relevant actions; this is formed by the action set defined in the plan library after the non-applicable actions are eliminated through the same check context function of (7). So, the agent provides these data to the RL black box and then obtain the next suggested action (12). This action is pushed on top of the intention queue and, if the state is not terminal, under this action is put a new call to the RL execution action (13). The next time this intention is further execute, the selected action will be performed, at the subsequent intention execution, another action will be requested and so on. Recall that an environment action or an achievement goal suspend the intention until the execution is done, same way internal actions are entirely executed before return. Since that and given that the RL selected action is above the next RL step in the same intention, the subsequent RL execution will never begin before the end of the previous action.

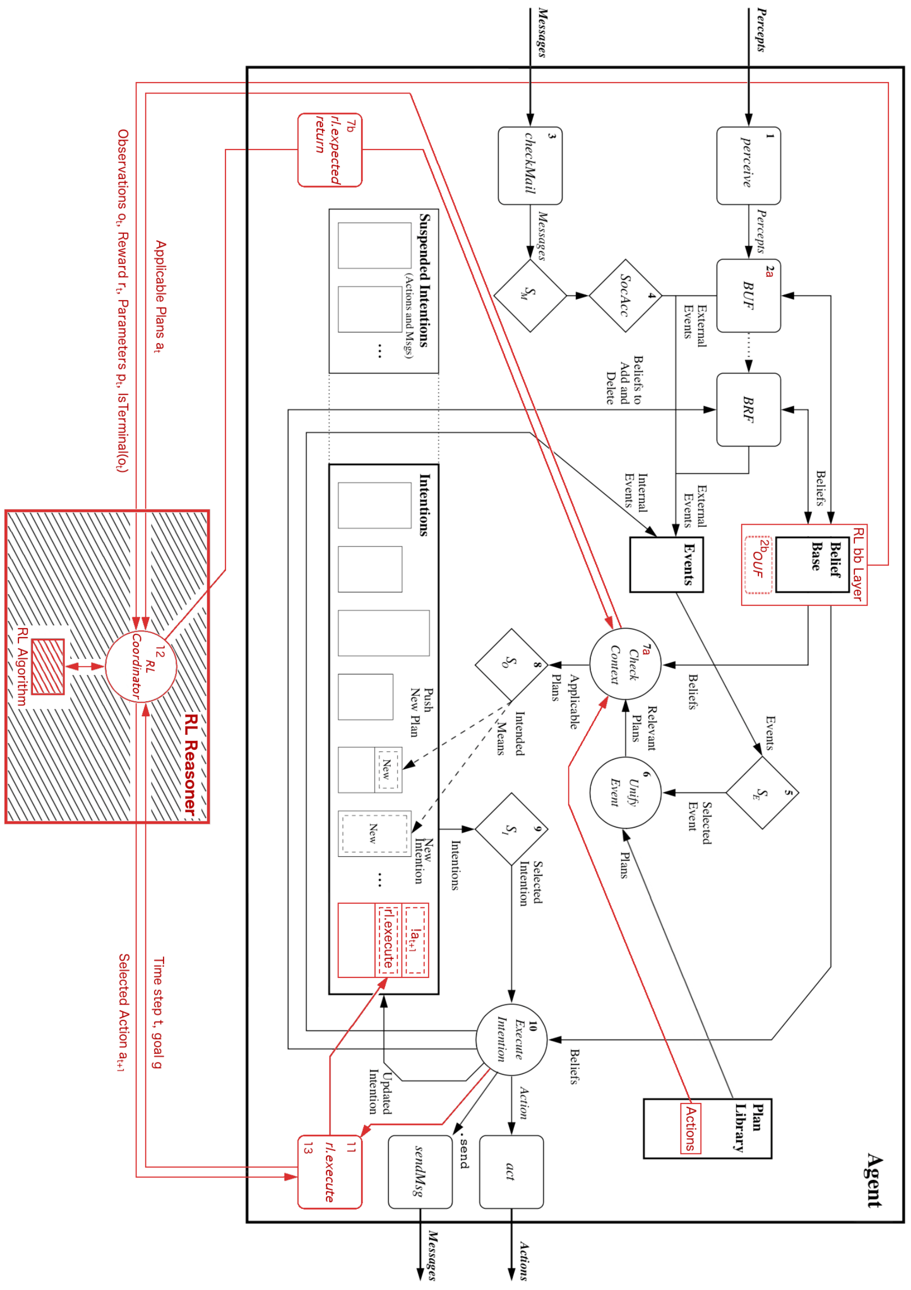

Fig. 3: The Jason Reasoning Cycle extended with learning aspects.

```
rl_parameter(policy, egreedy).
rl_parameter(alpha, 0.26).
rl_parameter(gamma, 0.9).
rl_parameter(epsilon, 0.22).
rl_parameter(epsilon_decay, 0.99992).

rl_observe(reach_finish, [ pos(_,_) ]).

rl_reward(reach_finish, 10) :- finish_line.
rl_reward(reach_finish, -1) :- not finish_line.

rl_terminal(reach_finish) :- finish_line.

@action[rl_goal(reach_finish), rl_param(direction(set(right, left, up, down)))]
+!move(Direction) <- move(Direction).

/* in this case we run an infinite learning process - actually it could be
   stopped when the performance (expected return) is considered good enough */

!start.
+!start : true <- rl.execute(reach_finish); !start.
```

Fig. 4: Full source code of the grid-world agent.

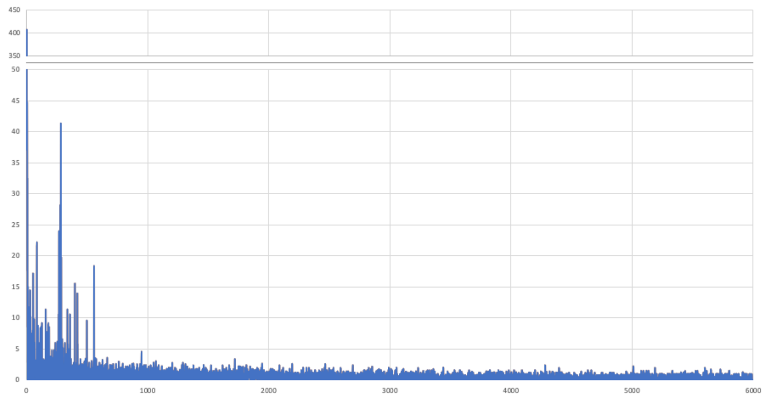

Fig. 5: Chart of simulation results: x is the episode number and y is the average error.

In Appendix A the interested reader can find further details about the Jason implementation. The full implementation is available here[1].

[1] `https://github.com/MichaelBosello/jacamo-rl`

5 Discussion

In order to test our proof-of-concept implementation, we used the gridworld problem introduced in section 3, performing first simple tests over small (5x5) grids. It is worth remarking here that both the kind of the problem used – the gridworld –and the size of the environment are clearly not useful for evaluating the approach from the point of view of the performance, the scalability and the generality as well. It has been used essentially for testing the framework developed so far.

The agent source code is shown in Fig. 4[2]. At every episode, the agent appears in a random place and seek to reach a fixed target position. SARSA algorithm performs properly in this task with a ε-greedy policy with epsilon decay (i.e., the exploration probability decreases with increasing steps). Parameters have been: alpha = 0.26, gamma = 0.9, epsilon = 0.22, epsilon decay = 0.99992. The agent learns the policy in about 1000 episodes, and when epsilon decreases under 0.05 (with this decay, approximately after 5000 episodes), the behavior becomes near optimal. The chart in figure 5 shows the average error (how many extra steps were made compared to the minimum path) on five trials with 6000 episodes.

5.1 About the RL algorithm used

The framework aims at modeling the three fundamental RL signals without any assumption on the RL algorithm behind them. Nevertheless, depending on the RL algorithm, different kinds of environments may be considered, with a different impact on the performances. In literature, three main characteristics of the environments are typically considered to properly select the RL algorithms: the Markov property, the type of task (episodic or continuing), and the state and action spaces dimension. A detailed discussion of this aspect is out of the scope of this paper.

Here it is interesting to consider if and how our model/framework would be expressive and flexible enough to include more advanced RL approaches used to tackle complex environments. For instance, in literature function approximation is exploited to tackle partial observability, possibly using nonlinear methods such as neural networks, in particular *deep* networks—such as in *deep* reinforcement learning. In our framework, this accounts for changing/plugging a proper RL reasoner component, without changing the whole interpreter architecture.

If the RL reasoner component cannot be centralised – being based e.g. on cloud services – then an interesting perspective is to consider the possibility to partially *externalise* the functionalities of the RL reasoner into an *artifact* in the A&A perspective [22], exploiting e.g. the full JaCaMo platform [5]. In that case, the RL reasoner is modeled as an external tool extending the cognitive capability of the agent and possibly wrapping the use of cloud-based services handling the management of e.g. deep neural networks, in a cognition-as-a-service style [26].

[2] The example is available in the source code of the framework.

6 The Road Ahead

The objective of this paper was to introduce a novel perspective on the integration of learning in BDI Agents programming and agent-oriented programming. In that perspective, the development stage of an agent accounts for setting up a first version of the agent, eventually including some programmed plans (hard plans), and then *grow up* the agent by making itself learning some other plans (soft plans), according to the need. Soft plans become part of the plan library and at deployment time the agent can exploit them like the hard ones, in a uniform way.

In the paper, we described a first proof-of-concept model and implementation based on Jason. In the current state, the framework is not meant to be ready to tackle real-world problems but to be a first tool in order to further explore and develop this idea. In that perspective, many interesting aspects – from our point of view – are worth to be investigated in future work. A list of main ones follows:

- Extending the investigation by considering different kinds of RL algorithms, with different complexities, and a different set of examples/problems as well, eventually doing a rigorous analysis of the computational complexity and properties of the computations performed by the extended reasoning cycle. Among the large spectrum of RL-based approaches, two are particularly interesting with respect to the objective of our research line. The first is about Hierarchical Reinforcement Learning, extending the reinforcement learning paradigm by allowing the learning agent to aggregate actions into reusable subroutines or skills [8]. In BDI case, reusable subroutines or skills are modelled as plans triggered by subgoals. The second one is about shaping in reinforcement learning[3] [9]. There, “education” is realised through the creation of a proper learning environment and, in particular, through demonstration.
- Exploring further the development/education process *lifecycle*, analysing how the different stages – development / training, validation / assessment, deployment/monitoring – are related.
- Designing and developing proper tools to be embedded in existing IDEs – or extending them – to support this process. Including simulators, which become an essential part of the picture.
- Exploring how software engineering aspects such as modularity, extensibility, reusability, composability can be framed when dealing with soft plans, aside to hard plans. Can we introduce kind of *incremental* learning to extend existing soft plans?
- Exploring how environment first-class abstractions such as artifacts [22] could be useful to better structure, modularise and make the way in which actions – and observations as well – are currently considered more dynamic.
- Beyond single agent perspective: what does it mean an education process for a multi-agent system? what does it mean an education process for an agent *organisation*?
- Methodologies – what’s the impact on AOSE methodology. Or, can we exploit existing AOSE methodology to effectively support this process or do we need to extend them?

[3] We thank the reviewers for this suggestion

– Planning – in the paper we did not consider at all planning, being our framework focused on learning. Nevertheless, it would be interesting and important to extend the conceptual framework to consider also the role that planning can do in such agent education process.

References

1. Airiau, S., Padgham, L., Sardina, S., Sen, S.: Enhancing the adaptation of BDI agents using learning techniques. Int. J. Agent Technol. Syst. **1**(2), 1–18 (Apr 2009)
2. Andrew McAfee, E.B.: The Second Machine Age: Work, Progress, and Prosperity in a Time of Brilliant Technologies. W. W. Norton & Company (2014)
3. Badica, A., Badica, C., Ivanovic, M., Mitrovic, D.: An approach of temporal difference learning using agent-oriented programming. In: 2015 20th International Conference on Control Systems and Computer Science. pp. 735–742 (May 2015)
4. Badica, C., Becheru, A., Felton, S.: Integration of jason reinforcement learning agents into an interactive application. In: 2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). pp. 361–368 (Sep 2017)
5. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Science of Computer Programming **78**(6), 747–761 (2013)
6. Bordini, R.H., Hübner, J.F., Vieira, R.: Jason and the Golden Fleece of Agent-Oriented Programming, pp. 3–37. Springer US, Boston, MA (2005)
7. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology). John Wiley & Sons, Inc., USA (2007)
8. Botvinick, M., Niv, Y., C Barto, A.: Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective. Cognition **113**, 262–80 (11 2008). https://doi.org/10.1016/j.cognition.2008.08.011
9. Brys, T., Harutyunyan, A., Suay, H.B., Chernova, S., Taylor, M.E., Nowé, A.: Reinforcement learning from demonstration through shaping. In: Proceedings of the 24th International Conference on Artificial Intelligence. pp. 3352–3358. IJCAI'15, AAAI Press (2015), `http://dl.acm.org/citation.cfm?id=2832581.2832716`
10. end of code, T.: Tanz, jason. Wired (2016)
11. Feliu, J.L.: Use of reinforcement learning (rl) for plan generation in belief-desire-intention (bdi) agent systems (2013), `https://digitalcommons.uri.edu/theses/160`
12. Ford, M.: Architects of Intelligence: The truth about AI from the people building it. Packt Publishing (2018)
13. Gerrish, S.: How Smart Machines Think. MIT Press (2018)
14. Guerra-Hernández, A., El Fallah-Seghrouchni, A., Soldano, H.: Learning in BDI multi-agent systems. In: Proceedings of the 4th International Conference on Computational Logic in Multi-Agent Systems. pp. 218–233. CLIMA IV'04, Springer-Verlag, Berlin, Heidelberg (2004)
15. Karim, S., Sonenberg, L., Tan, A.H.: A hybrid architecture combining reactive plan execution and reactive learning. In: Yang, Q., Webb, G. (eds.) PRICAI 2006: Trends in Artificial Intelligence. pp. 200–211. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
16. Kelly, J.E.: Computing, cognition and the future of knowing (2015), IBM Research and Solutions, white paper
17. Meijer, E.: Behind every great deep learning framework is an even greater programming languages concept (2018), Invited Talk at the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)
18. Norling, E.: Folk psychology for human modelling: Extending the BDI paradigm (2004)

19. Parnas, D.L.: The real risks of artificial intelligence. Commun. ACM **60**(10), 27–31 (Sep 2017). https://doi.org/10.1145/3132724, `http://doi.acm.org/10.1145/3132724`
20. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Van de Velde, W., Perram, J.W. (eds.) Agents Breaking Away. pp. 42–55. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
21. Rao, A.S., Georgeff, M.P.: Bdi agents: From theory to practice pp. 312–319 (1995)
22. Ricci, A., Piunti, M., Viroli, M.: Environment programming in multi-agent systems: an artifact-based perspective. Autonomous Agents and Multi-Agent Systems **23**(2), 158–192 (Sep 2011)
23. Shoham, Y.: Agent-oriented programming. Artif. Intell. **60**(1), 51–92 (Mar 1993), `http://dx.doi.org/10.1016/0004-3702(93)90034-9`
24. Singh, D., Hindriks, K.V.: Learning to improve agent behaviours in goal. In: Dastani, M., Hübner, J.F., Logan, B. (eds.) Programming Multi-Agent Systems. pp. 158–173. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
25. Singh, D., Sardina, S., Padgham, L., James, G.: Integrating learning into a BDI agent for environments with changing dynamics. In: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three. pp. 2525–2530. IJCAI'11, AAAI Press (2011). https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-420, `http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-420`
26. Spohrer, J., Banavar, G.: Cognition as a service: An industry perspective. AI Magazine **36**(4), 71–86 (Winter 2015)
27. Sutton, R.S., Barto, A.G.: Reinforcement learning : an introduction. The MIT Press (2018)
28. Tan, A.H., Ong, Y.S., Tapanuj, A.: A hybrid agent architecture integrating desire, intention and reinforcement learning. Expert Syst. Appl. **38**(7), 8477–8487 (Jul 2011)
29. Weiß, G.: Adaptation and learning in multi-agent systems: Some remarks and a bibliography. In: Weiß, G., Sen, S. (eds.) Adaption and Learning in Multi-Agent Systems. pp. 1–21. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
30. Wooldridge, M.: Introduction to Multi-Agent Systems. Wiley (2009)

A Details about the Implementation

In this appendix, we include further details about how the Jason interpreter has been extended to include the RL reasoner, following the model depicted in Fig. 1 and the architecture shown in Fig. 3

The belief base is amended to track belief about learning and observations instructions to prevent the necessity to loop multiple time over the belief base when the agent needs to retrieve the observations. Every time a belief is added or deleted, the observations maps are updated. To retrieve the reward and verify if the current state is terminal, the related rules are checked to find out if they are a logical consequence of the current belief base.

We develop also two internal actions: `rl.execute` and `rl.expected_return`. When `rl.execute` is carried out, the agent retrieves from the belief base the observations, the algorithm parameters, the reward, the boolean value that assert if the state is terminal. The agent retrieves the plans that are properly labeled, as described in section 1, from the plan library to form the action set. The action set is reduced through the *check context* function: step (7) in 2. All the information are passed to the RL algorithm which returns the next action (an achievement goal in our representation) to execute.

Then, `rl.execute` puts the selected action on top of the current intention's stack. If the current step isn't a final step, another `rl.execute` is pushed in the stack under the added action. As a result, `rl.execute` appears like a mere static plan; `rl.execute` and its intention acts and competes for execution like any other intention. Moreover, the next `rl.execute` will never be executed before the selected action end.

In the proof of concept, we implemented the SARSA algorithm with an ε-greedy policy. An RL algorithm must just implement the interface as follows:

```
public interface AlgorithmRL {
    Action nextAction(Map<Term, Term> parameter,
                      Set<Action> action,
                      Set<Literal> observation,
                      double reward, boolean isTerminal);
    double expectedReturn(Set<Action> action,
                          Set<Literal> observation);
}
```