Improving the Performance of Complex Agent Plans Through Reinforcement Learning

Matteo Leonetti and Luca Iocchi

Sapienza University of Rome
Department of Computer and System Sciences
via Ariosto 25, 00185
Rome, Italy

Abstract. Agent programming in complex, partially observable, and stochastic domains usually requires a great deal of understanding of both the domain and the task in order to provide the agent with the knowledge necessary to act effectively. While symbolic methods allow the designer to specify declarative knowledge about the domain, the resulting plan can be brittle since it is difficult to supply a symbolic model that is accurate enough to foresee all possible events in complex environments, especially in the case of partial observability. Reinforcement Learning (RL) techniques, on the other hand, can learn a policy and make use of a learned model, but it is difficult to reduce and shape the scope of the learning algorithm by exploiting a priori information. We propose a methodology for writing complex agent programs that can be effectively improved through experience. We show how to derive a stochastic process from a partial specification of the plan, so that the latter's perfomance can be improved solving a RL problem much smaller than classical RL formulations. Finally, we demonstrate our approach in the context of Keepaway Soccer, a common RL benchmark based on a RoboCup Soccer 2D simulator.

1 Introduction

Despite the great deal of research on planning over many years and in many different domains, planning in dynamic and uncertain domains is still a challenging task. In many applications, agents operate in highly dynamic and uncertain environments where most of the changes are not a consequence of the agent behavior. They usually have limited knowledge of the environment and noisy sensors. Many approaches rely on a transitional model of the domain; in these cases the knowledge about the environment is encoded and exploited for planning either offline or online.

As stated by Bonet and Geffner [2], creating a controller that maps observations into actions has been mainly achieved in three ways:

- the *programming* approach, where the controller is programmed by hand in a suitable high-level procedural language;
- the *planning* approach, where the controller is derived automatically from a suitable description the actions, sensors and goals;

Dagstuhl Seminar Proceedings 10081
Cognitive Robotics
http://drops.dagstuhl.de/opus/volltexte/2010/2634

– the *learning* approach, where the controller is derived from experience.

The programming approach allows to encode procedural information about how the task must be performed, but it makes improving the agent's behavior quite difficult, leaving little or no room for automation. The planning approach, on the other hand, allows to provide the agent with declarative knowledge about the environment, but is sensitive to inaccuracy of the model: in the class of environments we are considering, a declarative model cannot in general be able to foresee all possible events that can cause the plan to fail. This issue, especially in robotics, leads to the need for *execution monitoring* [14], that constitutes a whole research field. Finally, the learning approach can learn both a model of the environment and/or a *policy*, but it is particularly difficult for the designer to shape the search space, even when his/her knowledge could reduce the learning burden significantly.

In spite of many efforts to planning and learning in complex domains, hand-crafted plans still have a major role in many applications, even though they require a lot of effort from the designer and the results are of limited use in highly dynamic and uncertain domains.

Some relevant works in the direction of integrating a priori knowledge into a learning framework are present (cf Section 5). However, these works have limited applicability and do not scale to the class of problems we consider.

In this paper we introduce a novel use of Reinforcement Learning (RL) to improve planning from experience, while still allowing the designer to write a knowledge base or a set of plans. The proposed approach allows to convey prior knowledge to the agent in a straightforward way, more specifically in the form of a partially specified plan (or a set of plans). This is in contrast with standard approaches to learning to perform a specific task, which usually require a non negligible effort in the definition of the features of the environment to feed the learning algorithm, a careful choice of a function approximator and also the definition of proper actions.

The novelty of the approach is in the application of well established RL theory and methods in a novel learning state space, which is obtained directly from the plan, and it is considerably smaller with respect to standard formulations, thus not requiring function approximation. The proposed approach is targeted at all the "real world" applications in which the knowledge about the domain, even from the designer perspective, does not allow him/her to establish which plan (in a set of admissible ones) is going to perform better. In this context the agent behavior can automatically adapt during plan execution gaining benefit from experience.

To verify the effectiveness of our solution we implemented and tested it in the *Keepaway* domain [16, 8], a benchmark for learning algorithms at the edge of what RL can currently face. Our learning method could learn a behavior that significantly outperforms former results in the same setting and converges to the optimal solution several times faster.

2 Plan Representation

Our approach addresses the planning problem in complex, dynamic environments and is suited for reactive systems. In the rest of the paper we consider reactive plans represented as generic state machines, like state charts [6], in which every state corresponds to a set of *actions* and each transition corresponds to an *event*.

A *plan state* is a configuration of the machine that encodes the plan, as opposed to an environment state that is a configuration of the variables that represent the agent knowledge. Each plan state is associated the set of *actions* that is being executed in that point of the plan. Notice that the same set of actions may occur several times in different plan states. The state of the whole system is the Cartesian product of the plan and the environment state spaces.

An *event* is a general happening in the environment that can be whatever the agent is capable of detecting, for instance: a condition that becomes true, a message received from another agent, a timeout expired or a joint that reached its target position.

The representation of plans considered in this paper is based on a transition system defined over plan states and events. Such a transition system determines a set of plans, or *plan schemas*, as formally stated in the following definition.

Definition 1 (Plan Schema). *A Plan Schema is a tuple* $\langle S, s_0, F, E, \Phi, A, L, T \rangle$ *where:*

- S *is a finite set of plan states*
- s_0 *is the initial plan state*
- $F \subset S$ *is a set of final plan states*
- E *is a finite set of events*
- Φ *is a set of conditions*
- A *is a set of actions*
- $L : S \rightarrow \wp(A)$ *is a total labeling relation that maps plan states on actions*
- $T : S \times E \times \Phi \rightarrow S$ *is a transition relation augmented with a triggering event and a condition. For each* $s \in S$, $e \in E$ *and* $\phi \in \Phi$, ϕ *must entail the pre-conditions of all the actions in* $L(T(s, e, \phi))$

The underlying assumption is that all actions are indeed procedures that involve some actuators and then take time to execute. During the execution of an action the environment state changes continuously while the plan state does not. Indeed this representation does not model explicitly the agent's knowledge, but only the execution state of the plans.

The outcome of actions may be uncertain and we assume that a knowledge base (KB) exists such that at any moment it is possible to check whether or not it entails a certain condition. We also assume that appropriate modules keep such a KB updated with respect to the agent's perceptions.

In addition to events, edges are labeled with guard conditions that must hold for the edge to be enabled. The behavior of the machine is the following: the current state contains the currently executed set of actions which is performed until one of the events associated to the outgoing edges happens. Whenever such an event is sensed by the agent we say that the event *triggers* the transition which

makes it available for execution. For the edge to be actually enabled at that time another condition must be met, namely the guard of the transition must hold. When an edge is *triggered* (the associated event happens) and *enabled* (its guard condition holds) it is allowed to be followed and the next state represents the set of actions the agent is to execute next. If an action was present in the previous plan state and it is not in the next one that action must be terminated. On the other hand, if an action appears in the next state it must be started. All actions that are both in the previous and next plan state keep being executed. To make the operational semantic clearer we assume that all events are external (i.e., they cannot be generated by the machine itself) and transitions are instantaneous, so that no event can be lost during a transition execution. Final states are absorbing states that cannot be left once entered and determine the execution termination.

If the state machine is deterministic (it can never happen that two transitions are triggered and enabled at the same time), then the plan schema is actually a single plan since no choices are left to the executor and the entire behavior is specified. On the other hand, if the machine is non-deterministic the plan schema represents multiple plans and each non-deterministic choice is a fork among them. Nothing prevents different plans from sharing common paths and depart only where their behavior differs.

Such a state machine can easily represent any reactive, conditional plan with also while-loops. Transformation from plans in classical state-based representation to plan schemas as defined above is straightforward, since events may model post-conditions that become true and the guards can easily represent the pre-condition of the following action. But events can do much more, they can represent communication among agents (recall that an event can be associated to the receipt of a message), allowing the specification of multi-agent plan schemas. Events can also represent unexpected conditions (not necessarily the awaited post conditions), so that the plan may also account for interrupts. Finally, it is possible to easily extend the representation for hierarchical plans in which actions can be low level behaviors or state machines themselves, even if for this paper we limit the analysis to non-hierarchical plans. Thus, the proposed plan representation is quite general and we do not pose any restriction on the origin of plans, they can come from anything between automatic generation and handcrafting. The only assumption we require is that plans must be *correct*, in the sense that they should reach goal situations without violating action preconditions or domain constraints. Checking correctness of input plan schemas is outside the scope of the proposed approach.

2.1 Keepaway Soccer example

In order to make the plan representation and execution clear, we show a simple example borrowed by the *Keepaway Soccer* domain proposed by Stone and Sutton [16, 8]. Keepaway Soccer is a subtask of RoboCup Soccer in which one team, the *keepers*, must keep possession of the ball in a limited region as long as possible while another team, the *takers*, tries to gain possession. The task is *episodic* and one episode ends whenever the takers manage to catch the ball or the ball leaves the region.

Keepaway soccer retains some of the complexity of real world for the sensors are noisy, the environment is highly dynamic, also due to adversarial agents, and the communication among agents is limited.

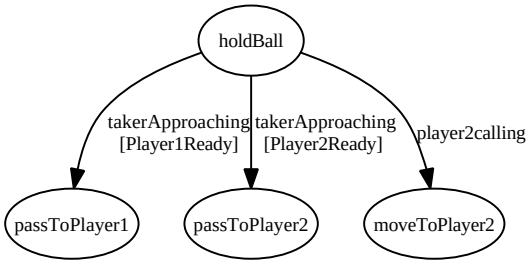

Fig. 1. An example of a simple plan. Actions label states, events and guards label transitions.

As an example, consider the plan schema in Figure 1. In the initial state the agent simply holds the ball until an event occurs. If *takerApproaching* happens two transitions are triggered. When at least one transition is triggered the post-conditions are checked, and if a transition is also enabled (its condition at that moment holds), it is followed setting the plan in a new state. Of course not necessarily there must be at least one enabled transition when an event happens and some events may be uncaught. In that case, the system remains in its current state waiting for the next event to happen. Notice that, if the guards *Player1Ready* and *Player2Ready* are not mutually exclusive, two transitions can be triggered and enabled at the same time. Thus, the transition system is non-deterministic and, in the same situation, two plans are available: the first one is $\langle holdBall, passToPlayer1\rangle$ while the second one is $\langle holdBall, passToPlayer2\rangle$.

In other words, in general, plan schemas are a compact way of representing multiple plans providing for different alternatives to achieve a goal.

3 Learning Framework

The learning framework is focused on exploiting the non determinism of a plan schema to make an informed choice.

Reinforcement Learning allows us to make use of experience to improve an agent's performance over time and seems a reasonable choice to achieve our goal. RL has been thoroughly studied within the MDP framework, since this framework provides a formal and neat mathematical notation for studying an important class of sequential decision problems. In traditional RL applications it is assumed that all relevant knowledge about an agent's environment can be encoded in a structure, usually a Markov Decision Process (MDP). Moreover, both in "model-free" and "model-based" RL techniques, it is assumed that even

though the agent might not know exactly what the structure of the MDP is (e.g. the transition matrix, etc), all sample observations are drawn from some underlying MDP. In the class of problems we are considering, however, assuming the existence of a fully observable MDP, or even trying to come up with a reasonable possible encoding for the states, which could somehow guarantee that the Markovian assumption is respected, might be infeasible. One reason for that is that it can be quite hard, or even impossible, to represent all the required information about other agents, their policies, unpredictable events, parallel action execution, unexpected changes in the task or in the environment, etc. In other words, it might be unreasonable or infeasible to assume that the task being solved can be well represented by an MDP. This is still true despite the sophisticated work on function approximation.

For this reason we rely on a generic knowledge base that reflects the beliefs of the agent about the environment, without building a dynamic model of it. In the following, we will define a stochastic decision process by deriving it from the plan and we will use this model to gather the experience to use in subsequent trials.

The state of the system is composed by both the state of the plan and the state of the environment but the latter is in general not completely known. The reward depends on how the state of the environment is perceived by the agent. In order to make a decision in non-deterministic choice points, we want to look forward in the plan having a value function associated with plan states, but not looking forward in the environment state space trying to predict the outcome of actions (i.e. the next environment state).

The plan executor must adhere to the state machine operational semantic as long as the choices are deterministic. Whenever a non-deterministic choice must be taken, the executor can refer to the value function to evaluate the alternatives and then exploit or explore as usual in RL.

3.1 Problem Definition

In order to properly characterize the stochastic process associated to the previously described state machine, and to set the proposed method in the RL framework, we define it in terms of a Semi Non-Markov Decision Process (SNMDP), since the actions do not have the same duration and the process is in general non Markovian.

We first show the construction of the SNMDP with an example and then we provide its formal definition. Suppose that at some point of the plan schema you have a choice point like the one previously described (Figure 1). The nodes that allow for non-determinism (i.e., that have more than a transition associated with the same event, and whose guard conditions are not mutually exclusive) are split into a number of nodes equal to the constituent events of the condition. In the example, the event *takerApproaching* (abbreviated as *ta*) is associated to the conditions Player1Ready (*p1r*) and Player2Ready (*p2r)*. This gives raise to four possible constituents, namely: only *p1r* is true, only *p2r* is true, both are true or none is. To the first three we associate a state and an arc from the

original *holdBall* state. The last situation, in which none of the conditions holds, translates into a loop on the *holdBall* state.

The resulting graph is represented in Figure 2. All the created edges correspond to the same non-deterministic action of the SNMDP reported as *ta*. Since it is caused by the perception of the event *ta*, the result of this action depends on the environment and cannot be chosen by the agent. In this section we make use of the term "action" as it is in the literature of stochastic processes when we refer to the SNMDP. Therefore, while an action in the plan schema is the actual intervention of the agent in the environment, an *action* in the SNMDP is an instantaneous transition available to the controller. An *action* in the SNMDP causes a change in the state of the process but cannot modify the state of the environment while this is the primary intention of an action in the plan schema.

Each node associated to a constituent of the conditions is connected to the action node containing the actions enabled by that constituent. In our example, *p1r* is connected to the node representing the action *passToPlayer1* (*pp1*), while *p2r* is connected to *passToPlayer2* (*pp2*) and *p1r&p2r* is connected to both. At this level the edges reaching different action nodes are associated to different *actions* of the SNMDP. The resulting graph has a choice point in the state *p1r&p2r* since in that case two actions are simultaneously available.

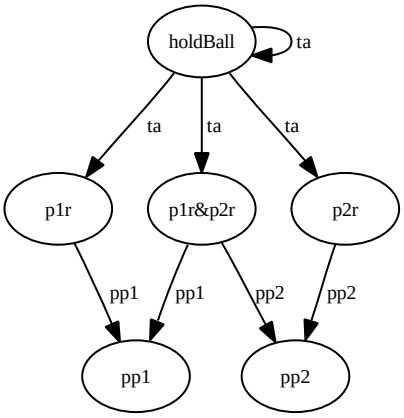

Fig. 2. Creation of the SNMDP. The original node with the action *holdBall* is split creating nodes to represent the conditions

The number of nodes in which a choice point in the original plan is split is exponential in the number of conditions. This is not surprising, and in the case of full observability and discrete state space this number would be equal to the number of states storing an entire Q-function. Nonetheless, the underlying assumption is that the domain is continuous and partially observable so that there is no notion of single state and considering single states or many small regions is not possible nor desirable. Hence, even if it is possible to consider

function approximation, it is not necessary for the number of nodes generated in practice.

To give a formal definition of the SNMDP we have informally previously introduced, we define the set $C_{cnd}(s, e)$ of the constituent events generated by overlapping conditions in a specific choice point (denoted as $\langle s, e \rangle$) of a plan schema $PS = \langle S, s_0, F, E, \Phi, T, A, L \rangle$ as follows: if there exist k conditions $\phi_1 \dots \phi_k$ and a state s_j s.t. $\langle s, e, \phi_i, s_j \rangle \in T$ for each $i \in \{1, \dots, k\}$ then $C_{cnd}(s, e) = \wp(\{\phi_k\}) \setminus \emptyset$ while $C_{cnd}(s, e) = \emptyset$ otherwise. In our example $C_{cnd}($ *holdball*, *takerApproaching*$) = \{\{p1r\}, \{p2r\}, \{p1r, p2r\}\}$.

Next, we define the set S_c of the states generated by condition overlapping in all choice points:

$$S_c = \{\langle s, e, cond \rangle | s \in S, e \in E, cond \in C_{cnd}(s, e)\}$$

In our example

$$\begin{aligned} S_c = \{ & \langle holdball, takerApproaching, \{p1r\} \rangle, \\ & \langle holdball, takerApproaching, \{p2r\} \rangle, \\ & \langle holdball, takerApproaching, \{p1r, p2r\} \rangle, \\ & \langle holdball, player2calling, \{true\} \rangle \} \end{aligned}$$

Those states constitute the second layer of Figure 2 except for the last one since the event *player2calling* has been omitted for simplicity. Finally, we also define a utility function S_c^e to select in S_c the states that are generated by a specific choice point as follows:

$$S_c^e(s, e) = \{\langle s, e, cond \rangle \in S_c | cond \in C_{cnd}(s, e)\}$$

Time has not been addressed yet so far. We consider time in discrete timesteps and actions can take multiple timesteps to complete. We use the following notation:

- t_k: the time of occurrence of the k^{th} transition. By convention we denote $t_0 = 0$
- $s_k = s(t_k)$ where $s(t) = s_k$ for $t_k \leq t \leq t_{k+1}$
- $a_k = a(t_k)$ where $a(t) = a_k$ for $t_k \leq t \leq t_{k+1}$

We define a Semi Non-Markov Decision Process $SNMDP = \langle S_{sp}, A_{sp}, P_{sp}, r_{sp} \rangle$ such that:

- $S_{sp} = S_c \cup S$, is the state set. The set S_c is the set generated by overlapping conditions, whereas S is borrowed directly from the plan schema and accounts for *action states*, that is states that are not the result of a choice point split but are associated to actions in execution. The first and last layer of Figure 2 are an example of the states in S while the intermediate layer is an example of the states in S_c.
- $A_{sp} = \{a \in \wp(A) | \exists s \in S \; s.t. \; L(s) = a\} \cup E$, is the action set. The first part is the co-domain of the labeling function in the plan schema. We create an action for each possible set that labels the states of the plan schema. Notice

that those actions are deterministic and we give them the same name of their target state. In our example of Figure 1 the co-domain of labeling function is $\{\{holdBall\}, \{pp1\}, \{pp2\}, \{mp2\}\}$. In this example there is no parallelism, so all sets are singletons. You can spot the corresponding actions in Figure 2. The set E (events in the plan schema) is used to define the actions on which the agent has no control. These actions are non-deterministic and their outcome depends on the environment. Again, in Figure 2, ta is an example of such an action.

- $P_{sp}(s', a, \tau, s) = Pr(t_{k+1} - t_k \leq \tau, s_{k+1} = s' | s_k = s, a_k = a)$ is the probability for action a to take τ time steps to complete, and to reach state s' from state s
  - if $a \notin E$: the action is deterministic. An action that *is not in* E connects a state in S_c to the state in S (second to third layer in the example) labeled with the actions enabled by the condition in that state. Moreover, those actions do not reflect any change in the environment so they always complete in zero time. That is,

$$P_{sp}(s', a, \tau, s) \begin{cases} = 1 \text{ if } \exists\, s_i, e, \phi. \\ \qquad \langle s_i, e, \phi, s' \rangle \in T \\ \qquad \wedge\, s \in C_s^e(s_i, e) \\ \qquad \wedge\, L(s') = a \wedge\ \tau = 0 \\ = 0 \quad \text{otherwise} \end{cases}$$

    A state s is connected to the state s' by a iff s is a state generated by a condition constituent, it is linked to s' by the plan schema, and a is the label of that link.
  - if $a \in E$: the action is non-deterministic. These actions take the time spent in the previous state waiting for the event. An action that *is in* E connects a state in S to itself and to all the condition states that its split generate (first to second layer in the example). Therefore, events cannot connect all pairs of states, which translates into:

$$P_{sp}(s', a, \tau, s) \begin{cases} = 0 \text{ if } s \notin S \vee \\ \qquad s' \notin C_s^e(s, a) \cup \{s\} \\ = \int_H p(t_{k+1} - t_k \leq \tau, s_{k+1} = \\ \qquad s' | s_k = s, a_k = a, \boldsymbol{h}) \\ \qquad p(\boldsymbol{h})\, d\boldsymbol{h} \\ \text{otherwise} \end{cases}$$

    If a connection between s and s' through e exists according to the plan schema, the value of the transition function is the probability for the event a to happen in the state $\langle s, \boldsymbol{h} \rangle \in S \times H$ where H is the domain of (continuous) hidden variables. Since those variables are not observable, the sample distribution is the (hidden) underlying one marginalized over the hidden variables. This makes the stochastic process non Markovian due to partial observability.
- $r_{sp}(s', a, k, s_t)$ is the reward function. It is Markovian (but the total reward in general is not) and we define its value to be 0 if $a \notin E$. Therefore the

immediate reward is non-zero only for events. Since events can take time (the time spent waiting in the previous state for the event to happen) the reward is defined in terms of immediate rewards as:

$$r_{sp}(s', a, k, s_t) = \sum_{i=1}^{k} \gamma^{i-1} r_{t+i}$$

where the r_{t+i} are the instantaneous rewards collected during the action execution, and γ such that $0 \leq \gamma \leq 1$ is the discount factor. Instantaneous rewards are defined over perceptions, that is they are a function of the state of the knowledge base.

In order to define a decision problem, we establish a performance criterion that the controller of the stochastic decision process tries to maximize. As such, we consider the expected discounted cumulative reward, defined for a stochastic policy $\pi(s, a)$ and for all $s \in S_{sp}$ and $a \in A_{sp}$ as:

$$\begin{aligned} Q_\pi(s, a) &= E\{\sum_{i=1}^{\infty} \gamma^{i-1} r_i\} \\ &= \sum_{s' \in S_{sp}} \sum_{\tau=0}^{\infty} \pi(s, a) P_{sp}(s', a, \tau, s) \cdot \\ &\cdot \Big(r_{sp}(s', a, \tau, s) \; + \\ &+ \gamma^{\tau} \sum_{a' \in A_{sp}(s')} \pi(s', a') Q_\pi(s', a') \} \Big) \end{aligned} \tag{1}$$

where $A_{sp}(s)$ is the set of actions for which $P_{sp}(s', a, \tau, s) > 0$ for some value of τ. The optimal discounted reward function is defined as:

$$Q^*(s, a) = max_\pi \; Q_\pi(s, a), \; s \in S_{sp}, \; a \in A_{sp} \tag{2}$$

3.2 Learning Algorithm

Since the stochastic process is in general non Markovian, extra attention must be paid at the algorithm used to evaluate the expected reward of a given policy. Usual algorithms based on a value function for MDPs make use of temporal difference (TD) methods to compute the expected reward from a state onward. The actual proof of convergence for TD relies on the Markov property and, even if Sarsa(λ) can be quite robust to partial observability [9], it is in general not guaranteed to converge. It has also been shown that adding memory to the observations can solve some POMDPs [11] and the plan schema allows to add arbitrary memory: if the plan schema is a tree the whole history is considered, but loops can create any situation in between memory-less and full memory. Practically, Sarsa(λ) should converge to a policy that, even if suboptimal, can allow for behavior improvement. A sound algorithm for the general case is MCESP by

Perkins [13], and good other candidates can be found in policy search methods, whose evaluation on our framework we leave for future work. For a brief review of results we can leverage, please refer to Section 5

The value function, that is the cumulative discounted reward from each state executing each action onward, will converge to the *expected value* of the reward influenced by the experience. It might happen that a choice point in the stochastic process corresponds to a region of the actual state space in which no action is in most of the cases better than any other. In such a case the value of all the available actions in that choice point would average out each other giving no reliable estimation of the expected reward. For this reason, a method (such as the aforementioned MCESP and Sarsa(λ)) that performs some form of Monte Carlo update must be preferred, so that is does not spoil the estimation of the former states. If the available knowledge does not allow to separate the conflicting regions in the actual state space, the agent cannot do any better.

4 Experimental Evaluation

The learning approach described in this paper has been tested in *Soccer Keepaway* on the 3 vs 2 task, i.e. with three keepers and two takers. Although the agents learn separately and there is no communication involved in the task, Keepaway is still a multi-agent task since the agents share the reward signal and each agent's action has an impact on all the others. Thus, credit assignment is particularly difficult since the reward for the whole *team behavior* is received by each agent as if it was its own.

In our work, we focus on the keepers and leave the takers' behavior to their predefined policy that consists in both of them following the ball. We refer to Stone at al. [16] and especially to the more recent work by Kalyanakrishnan and Stone [8] as representatives of the "RL way" to face Keepaway Soccer and we show our methodology applied to this task. As in that last reference, we consider the problem of learning both a behavior for the agent in possession of the ball and a behavior for the agents that are far from the ball. This is an additional challenge since the two behaviors interact making credit assignment even more problematic.

The first step consists in devising a proper set of actions. We consider three actions for the agent closer to the ball and three for the other two agents. The actions available to an agent close enough to kick are *holdBall* that just keeps possession of the ball, *passToCloser* that passes the ball to the agent that is closer to the kicker and *passToFree* that passes the ball to the agent whose line of pass is further from the takers. The first action is clearly wrong since a player cannot hold the ball indefinitely without being reached by the takers but we added it as a control, to make sure that our algorithm assigns the correct value to it and never chooses that action after convergence. The actions available to the agents far from the ball are *searchForBall* that just turns in place, *getOpen* that is the default handcoded behavior provided by the framework described as "move to a position that is free from opponents and open for a pass from

the ball's current position", and *goToTheNearestCorner* that goes to the corner closer to the agent.

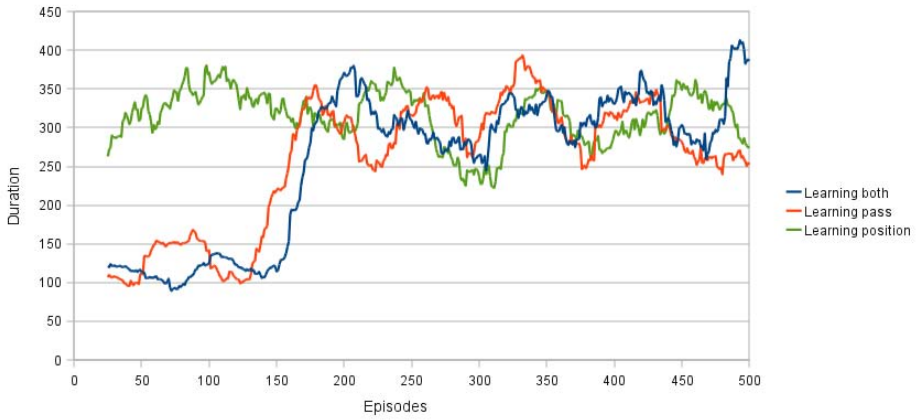

Fig. 3. A representative run of experiments. The x axis is the number of episodes in the run while y axis is the hold time in tenths of seconds

After the definition of the available actions we create a plan schema to accommodate our choice points. The entire plan schema used in these experiments is shown in Figure 4. States labeled with *noaction* have the empty action set associated, while *noevent* is a special event that takes zero time. This event has no impact on learning but it allows to add states in the plan schema convenient for representation and readability. Similarly, when no condition is indicated the guard of that edge is assumed to be *true*, i.e. the condition that is always fulfilled. Again, this is just syntactic sugar and does not affect the method. The leftmost node is the initial state, the control flows from left to right and it reaches the rightmost node within a simulation server cycle. In each cycle the agent must send a command to the server, thus performing an action, therefore every path from the leftmost to the rightmost nodes contains exactly one action. All of the conditions except those that guard the edges with event *takerApproaching* are mutually exclusive and leave no choice to the executor. As already mentioned, in the case of the passing actions since all three of them are triggered by the same event (*takerApproaching*) and their conditions (*true*) always hold at the same time, there is a non-determinism that can be exploited to make an informed choice. *TakerApproaching* is triggered when the agent perceives that a taker is closer than a certain threshold, *actionPerfomed* happens when the previous actions has queued its command for the server, and the conditions are similarly defined over state variables. In a similar way, the three choices for the positioning behavior *getOpen*, *searchForBall* and *goToTheNearestCorner* are taken into account when the player is not the one closest to the ball. In this setting even the simple Sarsa algorithm converged to the optimal solution.

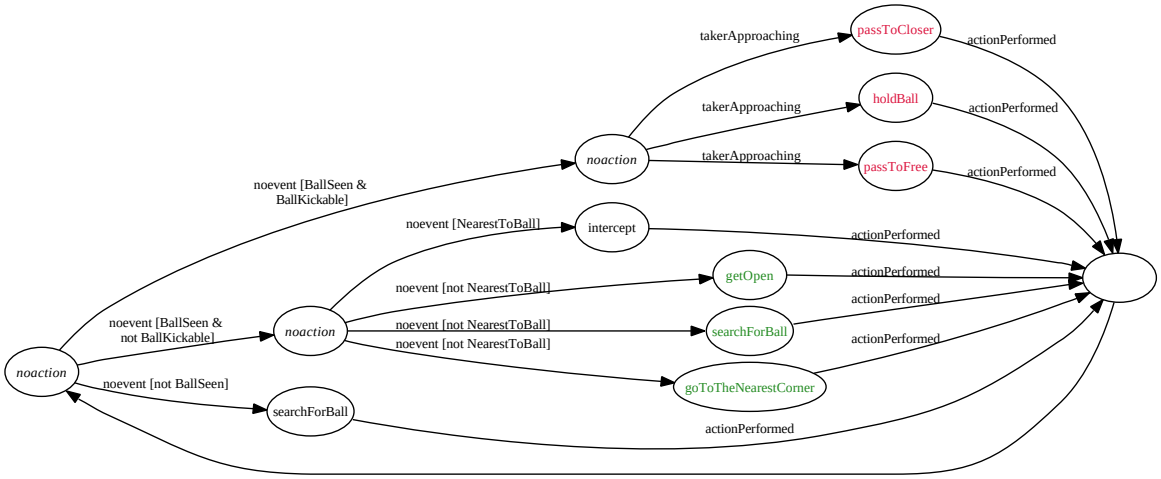

Fig. 4. The plan schema for a keeper with choice points on the passing and positioning behaviors

We performed different trials learning the two behaviors simultaneously and also the passing behavior and the positioning behavior separately. Our implementation used a greedy policy with optimistic initialization, a value of $\alpha = 0.3$ and $\gamma = 1.0$ which is sound since the task is episodic and the cumulative reward is limited. The reward signal is given by the duration of the episode: at every server cycle each agent receives a reward of $1/10$ of second. Even if the immediate reward is the same after every action, the cumulative reward depends on the previous choices and on the behavior of all the agents resulting in being highly non Markovian. Indeed what each agent aims maximize is actually the *team* performance. A representative trial is plotted in Figure 3 where each point is the average over a window of 50 episodes. With our approach we obtain the optimal behavior (that can be manually verified to be when *passToFree* and *goToTheNearestCorner* are chosen) after about 200 episodes in the case of learning both passing and positioning, with an average episode duration after learning of about 31 seconds. In previous works [16, 8] the best results are about 16 seconds of hold time and they take tens of thousands of episodes to be learned. We also show the learning curves of the single behaviors separately when coupled with the optimal choice for the other one. It appears that the passing behavior is the harder to learn, while positioning is learned in the first few episodes. Moving to the nearest corner without the ball then proves to be the crucial action that outperforms its alternatives quite quickly. In Figure 5 we show the box-plot of the distributions of the episodes' duration before (random behavior) and after learning. Both plots are drawn from 250 runs. We first used the Shapiro-Wilk normality test to check whether the two samples come from a normally distributed population, which turned out to be false for the second sample. Then, we used the non-parametric Mann-Whitney U test to confirm that the two samples do not (are extremely

unlikely to: $p = 5.7271 * 10^{-26}$) come from the same distribution. This means that the learning algorithm has indeed had a statistically significant impact on the duration of the episodes. The domain proved to be extremely noisy and the variance of both the samples is quite noteworthy.

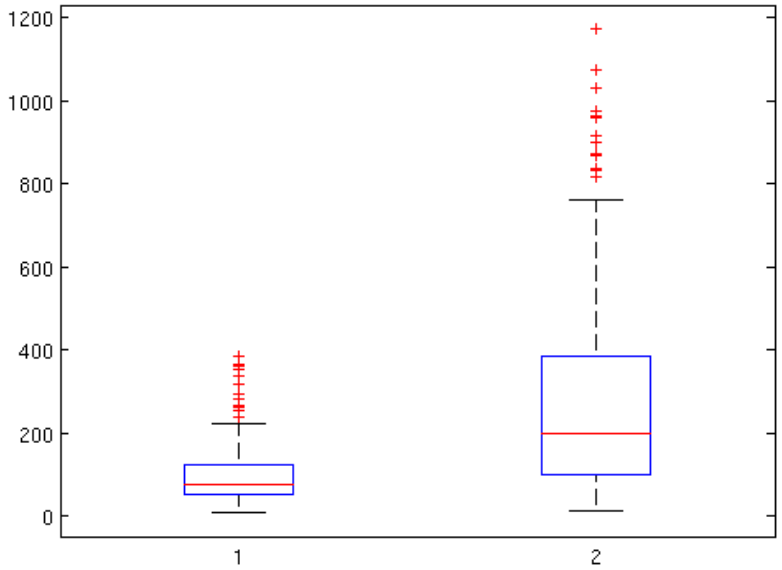

Fig. 5. Box-plot of the distributions of the episodes' duration (1) before and (2) after learning.

At the cost of devising a few (quite simple indeed) actions, and creating a partially specified plan exploiting the designer's knowledge about the task, we obtained a performance twice as high as the previous works in a number of learning episodes thousands of times smaller even with an algorithm as simple as Sarsa. The burden of creating the state representation and tailoring the function approximation for traditional RL is quite remarkable compared to the effort required of a designer to define such a plan schema and implement those actions. Also notice we made little use of perceptions, since conditions and events consider only a few aspects of the environment.

5 Related Work

Our work can be considered as part of the field of Hierarchical Reinforcement Learning (HRL). The overall idea of HRL is the ability of expressing constraints

on the behavior of the agent so that knowledge about the task and the environment can be exploited to shrink the search space. The optimal policy in this setting is the best one among those compatible with the constraints. The approaches closest to ours are Parr and Russell's HAM [1] and Andre and Russell's ALisp [10]. A similar approach can also be found in the field of symbolic agent programming, as this is the case of Decision Theoretic Golog (DTGolog) [3, 5]. All of the mentioned works allow to partially define the agent behavior in a high-level language (hierarchical state machines, Lisp and Golog respectively) and learn (or compute, in the case of DTGolog) the best behavior when this is not specified. While we share their motivation, our work departs from the previous ones in at least two aspects: (1) the formalism we adopt allows for dealing with reactive plans, non atomic actions, and continuous state spaces: these aspects are strictly related, leading to the representation of actions as states (instead of transitions) and to the need for events to both determine the end of an action and to mark those perceptions among the continuous infinity of possible ones that the agent should pay attention to; (2) even more importantly, we do not assume the existence of a Markov process as the underlying environment (an assumption common to all of the previous methods), but we derive a controllable process directly from the plan. Notice that the implementation of Kalyanakrishnan and Stone [8] fixes the behavior of the agent everywhere except for the two aspects they want to learn actually implementing a HAM. Therefore, the performance evaluation we carried can also be considered with respect to HAMs.

As a result of giving up the Markov assumption, and since partial observability is an aspect of common applications we consider in our approach, the control of the stochastic process resulting from the plan schema belongs to the class of problems usually referred to as with *hidden states*. The most general formulation of learning with hidden state are Partially Observable Markov Decision Processes (POMDP) [4]. Most of the methods for POMDPs attempt some state estimation, while we do not.

The stochastic process resulting from the observations in a POMDP (ignoring the underlying state space) is non-Markovian, and it is in some sense similar to the process we generate. The literature about N-MDPs is not as extensive as the one about MDPs, nonetheless some interesting results have been proved. A review of the available results is beyond the scope of this section, but we refer to the analysis by Pendrith and McGarity [11] and Singh et al. [15] about the characteristics of optimal policies in N-MDP and the effects of applying *direct* RL to them. An algorithm sound in the general case (although potentially suboptimal) is provided by Perkins [13] and the role of the exploration policy in the convergence of Sarsa and Q-learning is pointed out by Perkins and Pendrith [12]. Moreover, while examples can be constructed to prove that some (extremely simple indeed) implementation of direct RL on N-MDPs can diverge, there are promising empirical results about eligibility traces and partial observability [9]. Thus, although a comprehensive study about classes of N-MDPs and the traditional RL algorithms able to cope with them is still an issue, the lack of general results about convergence in non Markovian environments does not imply that those methods are doomed, it simply entails that further work is still needed.

We have shown through experiments that standard RL on the SNMDP built from a plan schema as shown before converges in a well-known (quite difficult) benchmark domain.

6 Discussion and Future Work

In this paper we have presented a methodology to write agent programs and to improve the agent's behavior through learning for a quite general category of plans. We have defined a proper controllable stochastic process deriving it from a partial specification of plans, in order to use it as a model for RL algorithms to improve the performance of the agent through experience. Finally, we have discussed the applicability of the available RL algorithms to the particular class of stochastic processes that our method generates and we have proved the effectiveness of our approach on a widely adopted test bed.

In our work we used *actions* as procedures that are usually referred in Hierarchical RL as *skills*. A popular model for skills is provided by the *option* framework [17] in which options and basic actions can be simultaneously considered. The role of options and their utility has been regarded as arguable [7] when the focus is on optimality. Nonetheless, in the class of problems we are considering optimality is quite difficult to achieve anyway, and our approach semi-automatically combining a set of handcoded skills proved to be more effective than *flat* RL which, even though is supposed to eventually reach the optimal behavior, was outperformed making use of a number of training episodes several orders of magnitude lower. In this context, having a good set of reusable skills to combine is of the utmost importance, and the work on temporal abstraction to create them automatically can profitably be integrated with our method providing different levels of intervention. Thus, where flat RL suffers in scaling up the search for a global optimum, the role of skills can be less arguable.

We have demonstrated in this paper a simple application of our approach to Keepaway Soccer limiting for simplicity the number of actions, and by no means obtaining the best behavior achievable. Our method is conceived to scale up to domains in which RL has not yet been successfully applied. In future work we plan to face more complicated settings possibly defining a new benchmark for hierarchical methods. We also plan to extend our formalism to multi-agent plans, exploiting events to represent message delivery or, more in general, coordination signals, thus learning team behaviors and coordination. Finally, we will further investigate the properties of the RL algorithms when applied to the stochastic process generated from a plan schema, and how to make use of the structure of plan schemas to obtain processes that favor convergence and/or optimality.

References

1. D. Andre and S. Russell. Programmable reinforcement learning agents. *Advances in Neural Information Processing Systems*, pages 1019–1025, 2001.
2. B. Bonet and H. Geffner. Planning and control in artificial intelligence: A unifying perspective. *Applied Intelligence*, 14(3):237–252, 2001.

3. C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the National Conference on Artificial Intelligence*, pages 355–362. AAAI Press / The MIT Press, 2000.
4. A. R. Cassandra. *Exact and approximate algorithms for partially observable markov decision processes*. PhD thesis, Providence, RI, USA, 1998. Adviser-Kaelbling, Leslie Pack.
5. C. Fritz and S. McIlraith. Decision-theoretic golog with qualitative preferences. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning, Lake District, UK*, 2006.
6. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
7. N. K. Jong, T. Hester, and P. Stone. The utility of temporal abstraction in reinforcement learning. In *The Seventh International Joint Conference on Autonomous Agents and Multiagent Systems*, 2008.
8. S. Kalyanakrishnan and P. Stone. Learning Complementary Multiagent Behaviors: A Case Study. In *Proceedings of the 13th RoboCup International Symposium*, 2009.
9. J. Loch and S. Singh. Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 141–150. Morgan Kaufmann, 1998.
10. B. Marthi, S. J. Russell, D. Latham, and C. Guestrin. Concurrent hierarchical reinforcement learning. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI*, pages 779–785. Professional Book Center, 2005.
11. M. D. Pendrith and M. McGarity. An analysis of direct reinforcement learning in non-markovian domains. In J. W. Shavlik, editor, *ICML*, pages 421–429. Morgan Kaufmann, 1998.
12. T. Perkins and M. Pendrith. On the existence of fixed points for Q-learning and Sarsa in partially observable domains. In *Proceedings of the Nineteenth International Conference on Machine Learning*, page 497. Morgan Kaufmann Publishers Inc., 2002.
13. T. J. Perkins. Reinforcement learning for pomdps based on action values and stochastic optimization. In *Eighteenth national conference on Artificial intelligence*, pages 199–204, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.
14. O. Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.
15. S. Singh, T. Jaakkola, and M. Jordan. Learning without state-estimation in partially observable Markovian decision processes. In *Proc. of 11th International Conference on Machine Learning*, 1994.
16. P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.
17. R. Sutton, D. Precup, and S. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999.