To appear in *AAAI Spring 2009 Symposium on Agents that Learn from Human Teachers* (AAAI), Palo Alto, United States, March 2009.

# A Task Specification Language for Bootstrap Learning

**Ian Fasel, Michael Quinlan, Peter Stone**
Department of Computer Sciences
The University of Texas at Austin
1 University Station C0500
Austin, Texas 78712-1188
{ianfasel,mquinlan,pstone}@cs.utexas.edu

**Abstract**

Reinforcement learning (RL) is an effective framework for online learning by autonomous agents. Most RL research focuses on domain-independent learning *algorithms*, requiring an expert human to define the *environment* (state and action representation) and *task* to be performed (e.g. start state and reward function) on a case-by-case basis. In this paper, we describe a general language for a teacher to specify sequential decision making tasks to RL agents. The teacher may communicate properties such as start states, reward functions, termination conditions, successful execution traces, task decompositions, and other advice. The learner may then practice and learn the task on its own using any RL algorithm. We demonstrate our language in a simple BlocksWorld example and on the RoboCup soccer keepaway benchmark problem. The language forms the basis of a larger "Bootstrap Learning" model for machine learning, a paradigm for incremental development of complete systems through integration of multiple machine learning techniques.

## Introduction

Traditionally, research in the reinforcement learning (RL) community has been devoted to developing domain-independent algorithms such as SARSA (Sutton & Barto, 1998), Q-learning (Watkins, 1989), prioritized sweeping (Moore & Atkeson, 1993), or LSPI (Lagoudakis & Parr, 2003), that are designed to work for any given state space and action space. However, the *modus operandi* in RL research has been for a human expert to re-code each learning environment, including defining the actions and state features, as well as specifying the algorithm to be used. Typically each new RL experiment is run by explicitly calling a new program (even when learning can be biased by previous learning experiences, as in transfer learning (Soni & Singh, 2006; Torrey *et al.*, 2005; Taylor & Stone, 2005). Thus, while standards have developed for describing and testing individual RL algorithms (e.g., RL-Glue, White *et al.* 2007), no such standards have developed for the problem of describing complete tasks to a preexisting agent.

In this paper we present a new language for specifying complete tasks, and a framework for agents to learn a new policy for solving these tasks. The language allows a user (or "Teacher") to provide information such as start states, reward functions, termination conditions, task decompositions, successful execution traces, or relevant previously learned tasks. An agent can then practice the task to learn a policy on its own using any RL algorithm.

The language and framework are targeted at enabling users to develop complete systems that may need to learn multiple different tasks using different learning techniques or sources of training data. Moreover, a teacher can use policies for tasks that were previously learned to "Bootstrap" learning for more complex tasks, either by suggesting earlier policies to be used as abstract actions (i.e., "options" Sutton, Precup, & Singh 1999), or by specifying that an earlier policy can serve as a source of prior experience to be transferred to learning the new task.

## Why an RL Task Language?

We are motivated by the recently proposed "Bootstrap Learning" (BL) paradigm for machine learning [1], whose ambition is to integrate all forms of machine learning into a single agent with a natural human-instruction interface. In the BL setting, a human teacher provides structured, step-by-step lessons involving multiple instruction methods to a learning agent, in order to gradually build its ability to perform a variety of complex tasks. For example, rather than starting "*tabula rasa*" as in traditional machine learning, a BL agent might first be given lessons teaching it to perform various types of statistical pattern recognition or logical inference, the results of which can be used in later lessons as primitive state variables in more complex sequential decision making tasks. The final aim of the BL project is to create autonomous agents which can be taught by end users in the field to solve multiple, complex problems by combining many different teaching and learning methods.

The overall goal of BL represents a multi-year, multi-institution project. In this paper, we present a key initial component to accomplish this goal: a formal language that allows a human teacher to specify tasks to a learning agent, so that it may set up and initiate learning of new policies autonomously. We refer to our proposed language as the "BL Task Learning" language, or simply the BLTL language, and likewise refer to the framework for using the BLTL language as the BLTL framework.

[1] Bootstrapped learning proposer information pamphlet, http://www.darpa.mil/IPTO/solicit/closed/BAA-07-04_PIP.pdf

A key goal of the BLTL language is to enable multiple methods for instruction of processes, and multiple RL algorithms, to be used in the same agent. Previous work on human-to-agent instruction has included task demonstrations (Schaal, 1997), task decompositions (Dayan & Hinton, 1993; Dietterich, 1998; Sutton, Precup, & Singh, 1999), general advice about actions and states which the learner can incorporate into the value function (Maclin & Shavlik, 1996; Kuhlmann *et al.*, 2004), and identification of previously acquired policies that can be used for initializing the policy to be learned for the new task (Soni & Singh, 2006; Torrey *et al.*, 2005; Taylor & Stone, 2005). Typically, research on these types of instruction has required that a human set up each individual learning task from scratch and manually invoke each one. In this paper, we allow the teacher to provide *advice*, indicate relevant previous experience (enabling *transfer learning*), use previously taught tasks as primitive actions in new tasks, or specify portions of a more complex task which are to be refined by learning (enabling *task decomposition*). Because parts of larger tasks can each be learned using a different technique, we enable multiple learning methods to be synergistically integrated in a single RL agent that is far more capable than an agent using any one learning algorithm.

## The BL Task Learning Framework

Traditional RL research focuses on algorithms for a learning agent, behaving in a single environment, to update a policy by which it chooses actions given fixed state variables. In the BL paradigm, the automated student is responsible not just for learning a specific policy, but also for the larger problem of knowing *how* to engage in learning, provided a task specification. Thus the BL student must also be able to identify the task, and initialize, terminate, and evaluate episodes (as well as single steps) during practice. By analogy, a human student who has been given a set of practice problems, say to learn long-division, must not only learn the procedure for division, but also must know which problems to work on, how to evaluate her performance (for instance by checking the answers and noticing how long it takes to solve each problem), and that she should continue to work through the entire set of homework problems—or at least as many as it takes to master the concept and get a good grade. To enable a teacher to provide instructions to a BL student, the BLTL language must be able to address all of these elements of learning.

The standard RL definitions of agents and environments, as described in (Sutton & Barto, 1998), are as follows:

**Environment:** stores all the relevant details of the world, such as the current state representation, the transition probabilities, and transition rewards.

**Learning Agent:** both the learning algorithm and the policy maker for acting in the environment. The agent needs to decide which action to take at every step, and may update its policy as it receives experience in the world.

**Experiment Program:** this is the control loop, which steps the agent through the environment in multiple episodes, and collects information about performance.

A popular, freely available implementation of these components in code is the RL-Glue framework (White *et al.*, 2007), which has formed the basis of several "bakeoffs" and contests for comparing RL algorithms.

The BLTL framework encompasses the Environment and Learning Agent components identically to the traditional RL framework (indeed we implement and connect these components using a subset of the RL-Glue package). However in the BLTL framework, the Experiment Program is incorporated into an internal component called the *Trainer*. The Trainer component monitors the environment and decides when to initiate and terminate learning of a particular policy for a particular task. It makes these choices autonomously and is not under direct external control. During an episode of learning, the Trainer monitors the environment and checks for the termination conditions (as specified in the task specification), combines rewards from the environment with any additional rewards specified in the task specification, and directs the Learning Agent and the Environment to the initial conditions as specified by the teacher (to the extent possible given the implementation – in a simulation environment it may be able to "teleport" to an initial condition, however in a physical implementation it may have to e.g., walk a robot to a starting position).

The BLTL framework therefore encompasses all of the concepts in traditional RL, but additionally makes the control and monitoring of the Learning Agent and Environment an integral part of the complete autonomous system, not a separate component that must be supplied by an external human user. In order to allow an external teacher to specify a task, the BLTL framework defines a common name space in which the state variables and functions (such as numerical functions or sorting routines) required to define the state and action spaces are accessible to the Teacher, Trainer, and Learning Agent.

In order to use the BLTL framework, an end user (i.e., the programmer wishing to write lessons for RL tasks) must first implement the necessary functions for the Trainer and Learning Agent to monitor and take actions in the Environment, and supply any additional learning algorithms he wishes to test if not already available. The BLTL language can then be used to set up and teach any number of lessons in the environment while the BL agent runs continually, sensing and acting in the world. Although the initial setup requires about the same effort as the standard approach to RL research for a *single* task, the BLTL framework makes it simple for a teacher to supply lessons for *multiple* tasks, and to reuse previously learned policies as abstract actions in new tasks.

## Language Primitives

We can now specify the primitive functions needed for a teacher to describe to a BL agent how to practice a task. The specification of these functions is a main contribution of this paper. In aggregate, these functions enable a teacher to specify a completely new RL task to a BL agent for the purpose of learning. These functions involve initialization, describing the rules for termination conditions, the reward, and the state and action spaces. For clarity, we refer to initialization, taking a series of steps, then terminating as an *episode*.

When deciding how to act in the world, the Learning Agent takes as input a state vector at each step. This state vector may be the result of complex operations on the raw world state (which may have been learned in previous lessons), therefore the language includes commands for constructing a list of functions whose outputs are concatenated into the state vector presented to the agent. Similarly, in order to allow learned actions, as well as "primitive" actions, to be used in a policy, the language also includes commands for constructing a list of behavior functions that serve as the action space for the current RL task.

The following list of functions with informal descriptions represents our proposed language for teacher-student interaction regarding sequential decision making problems. A complete and formal specification of the functions is beyond the scope of this paper.

**BeginTaskDescription("name")** Prepare to start learning a task. Where `name` defines the new policy created from learning this task.

**Environment("simulator" or "function")** Tells the agent what world to interact with. The agent can be given a `simulator`, or it may directly invoke a known transition function.

**BeginEpisode("world state")** Specifies that an episode is to begin by initializing to the given `world state`, which must be available in the simulator.

**OnEpisodeEnd("option")** What to do when an episode ends. The choice of options depends on the abilities of the simulator, and might include e.g. `Restart`, `RestartFromPointX` etc. By default, the trainer will call the agent's `StepReward` function at the end of an episode.

**StepReward("function")**
Let `function` be the reward function. For example `TimeElapsed(now,lastStepTime)`. In continuous-time environments, this is called just before the next action is taken (or on termination) so the reward is associated with the correct state-action. A common form of `function` is a conditional expression `(f ? X : Y)`, meaning "if `f` is true then value `X`, otherwise value `Y`", in order to give rewards only at the end of an episode.

**AddToStateSpace("function")** Add a function to the list of state space variables, which will be concatenated into the state vector. These functions generally take as input the raw sensory variables, and may have either real valued or binary output.

**AddToActionSpace("function")** Add a function to the list of possible actions. For example in GridWorld this could be `MoveUp()` etc. These could also be complex options like `MoveUpUntilReachedWall()`, which might have been taught in earlier lessons.

**WrapperPolicy("policy", "condition")** This is used when the task has already been decomposed and the current agent is only to learn a policy for a subtask. The command tells the agent that it should run a pre-existing policy `policy`, but to switch to learning the current policy when `condition` is satisfied. `policy` could be a previously learned policy from a prior lesson, or a built-in.

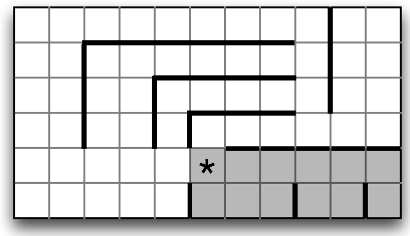

**Figure 1**: There are many possible tasks in this gridworld. For instance, one task may be for an agent to reach the starred location from any starting point in the shaded area.

**AddTerminationCondition("function")** Add a boolean function to be evaluated at the end of each timestep, which if it evaluates to `true`, will end the episode. It is possible to specify multiple termination conditions.

**StateSpaceDefine("function", "name")** A function that takes the raw state space as input, and places the output into a variable with `name`, to be used by state space functions. This function will be executed before calls to the state space functions.

**SourcePolicy("sourcepolicy","iscopy")** Indicates the policy currently being learned should be based on `sourcepolicy`. If the boolean value `iscopy` is `true` this is done by simply copying the old policy to the current one. Otherwise it should be used as the source for a more complex transfer method.

**AddAdvicePolicy("policy")** Adds an entire policy as a form of advice. For example this could be a hand generated policy that the agent could use while learning. It would be up to the underlying learning algorithm how this is handled.

**AdviseAction("state", "vals", "function")** When `state` equals `vals`, recommend the action represented by the function `function`. `vals` could be a boolean function, such as `Between(2,4)`. It is the responsibility of the learning algorithm to handle the advice (e.g., as in (Maclin & Shavlik, 1996) or (Kuhlmann *et al.*, 2004)).

**StartLearning("until")**
Starts learning. Any combination of conditionals can be passed into `until`, such as `number_of_timesteps = k`, `wait_for_stop_message`, `learning_converged`.

## Teaching a lesson in GridWorld

We first illustrate using the BLTL language to teach a lesson in GridWorld. The following sections will then show how the BLTL framework can be used to specifying tasks in widely varying domains, including BlocksWorld and RoboCup Soccer Keepaway.

In this illustrative lesson, the goal is to learn to get to the location marked with a star in Figure 1 from any starting point in the shaded region. Once the Environment (call it `GridWorld`) and a Learning Agent (e.g., tabular action-value function + SARSA) have been implemented, the first step in teaching is to initialize learning and the environment:

```
BeginTaskDescription("GridWorldSubtask");
Environment("GridWorld");
```

The Teacher now specifies how to start an episode, using a function provided by the simulator for starting a player at a random location within a region:

```
BeginEpisode("RandomWithinRegion(6,5,6,2)");
```

In this particular world, the possible actions are to move one space in the cardinal directions. The state is the current location. Therefore the teacher says:

```
AddToActionSpace("Up1()")
AddToActionSpace("Right1()")
AddToActionSpace("Down1()")
AddToActionSpace("Left1()")
AddToStateSpace("PositionX()")
AddToStateSpace("PositionY()")
```

An episode concludes when the agent reaches the starred location, and the reward is simply the time elapsed. The function `NumSteps()` and identifier `EpisodeStart` must be provided to indicate the number of steps since the beginning of an episode. It also must provide a function for checking the location of the agent.

```
AddTerminationCondition("AtLocation(6,5)");
StepReward("NumSteps(EpisodeStart)");
OnEpisodeEnd("Restart");
```

Finally the agent is told it may start practicing by calling `StartLearning`. The training agent will then run the learning agent through several episodes, resetting the world as needed, as the agent repeatedly takes actions and learns from the reinforcement it receives. Once the agent has learned and mastered this task, we can use the learned policy as an abstract action in a future lesson, for instance to reach the upper left corner.

## Learning in Multiple Domains

So far we have implemented the BLTL famework and tested it in two domains. The first is a simple, deterministic domain with discrete states and actions, and the second is a complex, stochastic, multi-agent domain with continuous state and discrete actions. We provided two learning agents, which the BLTL framework automatically selects based on the task description as follows:

First, if the task's state and action spaces are discrete, the BLTL framework selects a leaning agent that implements the R-Max (Brafman, Tennenholtz, & Schuurmans, 2002) algorithm, which aggressively explores unknown state-action pairs and rapidly learns an effective policy starting from any given state.

Alternatively, if the state space is continuous, the BLTL framework selects an agent which implements Q-learning with a neural network action-value function approximator, using Experience Replay (Lin, 1992) in repeated "batches" to update the value function. The agent implements neural networks with many hidden layers initialized in a stage-wise manner using a probabilistic algorithm as follows: the input layer and first hidden layer are trained as a restricted Boltzmann machine (RBM) (Ackley, Hinton, & Sejnowski, 1985) using contrastive divergence (Hinton, 2002), using the observed states as data. Subsequent hidden layers are then trained in series, where the activities of the hidden units of the previous layer are treated as data to train a new RBM at the next layer, forming a "stack" of RBMs. A many-hidden-layer network initialized this way is referred to as a "deep belief network" and described in detail in (Hinton, Osindero, & Teh, 2006; Hinton & Salakhutdinov, 2006). We have found that the combination of experience replay and deep belief nets makes this learning agent very efficient without requiring additional parameter tuning (Fasel, Kalyanakrishnan, & Stone, 2008), making it an appropriate learning agent for the BLTL framework.

A primary goal of future work (as discussed in the Conclusions section) is to add more possible learning agents and enhance the ability of the BLTL framework to automatically select appropriate learning agents based on the task description. However these two methods are already enough to learn in two very different domains.

### Deterministic Domain: Blocks World

A major aim of the BLTL framework is to make it easy to teach agents complex tasks by building from simpler previous tasks. In this domain our task was to learn to make a stack containing three blocks in a particular order. The agent had previously learned the notion of `On(block,flatsurface)` and also the action `MoveOnto(block,flatsurface)`. The instructions for describing this task to the agent are given in Figure 2.

```
BeginTaskDescription("MakeStack");
Environment("BlocksWorld");
BeginEpisode( "RandomStackABC()" );
AddToActionSpace( "MoveOnto(A,table)" );
AddToActionSpace( "MoveOnto(A,B)" );
AddToActionSpace( "MoveOnto(A,C)" );
AddToActionSpace( "MoveOnto(B,table)" );
AddToActionSpace( "MoveOnto(B,A)" );
AddToActionSpace( "MoveOnto(B,C)" );
AddToActionSpace( "MoveOnto(C,table)" );
AddToActionSpace( "MoveOnto(C,A)" );
AddToActionSpace( "MoveOnto(C,B)" );
AddToStateSpace( "On(A,table)" );
AddToStateSpace( "On(A,B)" );
AddToStateSpace( "On(A,C)" );
AddToStateSpace( "On(B,table)" );
AddToStateSpace( "On(B,A)" );
AddToStateSpace( "On(B,C)" );
AddToStateSpace( "On(C,table)" );
AddToStateSpace( "On(C,A)" );
AddToStateSpace( "On(C,B)" );
AddTerminationCondition( "IsAStack(C,A,B)" );
StepReward( "(IsAStack(C,A,B) ? 100 : 0)" );
OnEpisodeEnd( "RestartFromBeginning" );
StartLearning();
```

**Figure 2**: BLTL instructions for the Blocks World MakeStack task.

Because the state-space was discrete, the BLTL framework selected the R-Max agent. After several episodes, the BL agent was able to learn an effective policy for building a particular stack, in this case the stack (C,A,B), with results that were comparable to a standard (non-BLTL) implementation.

We then changed the task so that the agent could use an additional, previously learned action `MakeStack-Naive(block,block,block)`. This action is a complete policy that succeeds in making a stack *only if* all the blocks begin on the table. By adding the additional command:

```
AddToActionSpace( "MakeStack-Naive(C,A,B)" )
```

our agent could easily take advantage of this new action and could learn an effective policy more quickly, using `MakeStack-Naive` as a single action when it encountered three blocks on the table, instead of having to relearn an entire sequence of actions from that stage. The downside to adding this action was that the total state-action space was now larger, and sometimes the agent would learn to *unstack* a partial stack so that it could execute MakeStack-Naive. Nevertheless, adding this new action was as simple as adding a single extra command.

## Non-Deterministic Domain: RoboCup

Here we consider an even more complex domain, RoboCup soccer, and show how to describe a subtask within RoboCup to learning agents.

Robocup is a fully distributed, multiagent domain with both teammates and adversaries. There is hidden state, meaning that each agent has only a partial world view at any given moment. The agents have noisy sensors and actuators, meaning that they do not perceive the world exactly as it is, nor can they affect the world exactly as intended. Perception and action are asynchronous, prohibiting the traditional AI paradigm of using perceptual input to trigger actions.

RoboCup is a good example for the BLTL framework because mapping from the low-level state description to the low-level action language requires several levels of intermediate concepts. The primitive percepts indicate perceived distance and angle to objects in the environment, such as: (see ((goal r) 15.3 27) ((ball) 8.2 0) which indicates that the right goal is 15.3 m away and 27 degrees to the right and the ball is 8.2 m straight ahead. Meanwhile, the actions are parametric, enabling agents to dash forward with a power ranging from [0,100], turn a specified angle from [-180,180], or, when the ball is nearby, kick in a specified direction with a power ranging from [0,100].

**Keepaway Soccer** *Keepaway* is a subtask of RoboCup soccer, in which one team, the *keepers*, tries to maintain possession of the ball within a limited region, while the opposing team, the *takers*, attempts to gain possession. Whenever the takers take possession or the ball leaves the region, the episode ends and the players are reset for another episode (with the keepers being given possession of the ball again). A sample starting configuration is shown in Figure 3.

Parameters of the task include the size of the region, the number of keepers, and the number of takers. Figure 3 shows screen shots of games with 3 keepers and 2 takers (called 3 vs. 2, or 3v2 for short) playing in a 20m x 20m

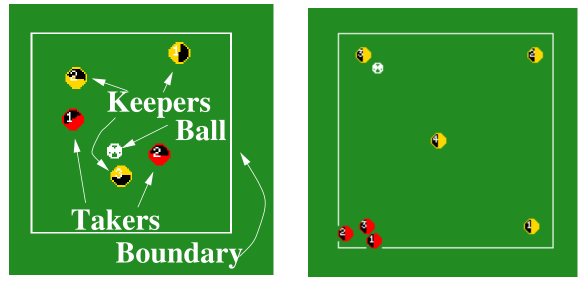


**Figure 3**: **Left:** A screen shot from the middle of a 3v2 keepaway game in a 20m× 20m region. **Right:** A starting configuration for 4v3 keepaway in a 30m×30m region.

region and 4 vs. 3 in a 30m x 30m region.[2]

Keepaway has received considerable attention as a testbed for RL algorithms (Pietro, While, & Barone, 2002; Torrey *et al.*, 2005; Stone, Sutton, & Kuhlmann, 2005).However, to the best of our knowledge, in all cases the task has been fully specified manually. In this paper, we focus on how the keepaway task can be taught to an agent using the BLTL framework, using some of the more advanced features such as task decomposition and "Advice".

In this example, our goal is for each keeper to learn a policy for what to do when it possesses the ball – i.e., the ball handling policy. When a keeper does not have the ball, it should follow policies that have already been specified (i.e `Receive(GetOpen)` or `Recieve(GoToBall)`), which are described in (Stone, Sutton, & Kuhlmann, 2005). Note that these policies could be taught to the agent in prior lessons. We therefore will specify to the agent that it will normally follow the "Keeper" policy, but when the state "BallIsKickable" is true, it will use the policy currently being learned. Note that the states and actions for when the keeper does not have the ball are different from when it does have the ball. An illustration of the entire agent policy is given in Figure 4.

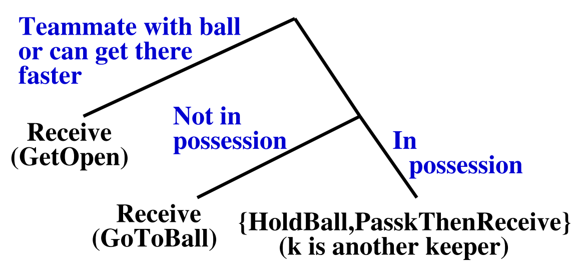


**Figure 4**: If the keeper does not have the ball, then it follows a predetermined policy to either GetOpen or Receive. When it has the ball, it has a choice to either HoldBall or PasskThenReceive. How to make this choice is the focus of this lesson.

**Required Actions, Predicates** Before beginning the lesson, the student must be capable of several skills, many of which are composed of multiple primitive actions and perceptions. These skills must either be previously learned via other lessons, or be "innate" (i.e., programmed in).

Table 1 summarizes the required actions and predicates for the keepaway task. The actions are implemented as

[2] Flash files illustrating the task and are available at http://www.cs.utexas.edu/~AustinVilla/sim/keepaway

| Actions | Predicates |
|---|---|
| HoldBall() | CanGetToBallFaster() |
| PassBall(k) | BallIsKickable() |
| GetOpen() | TakerHasBall() |
| GoToBall() | BallOutOfBounds() |
| Receive() | dist(a, b) |
| PassToKThenReceive(k) | ang(a, b, c) |
| InitializeKeepaway(nK, nT, h, w) | min(a, b) |
| | SortKeeperDistances() |
| | SortTakerDistances() |

**Table 1**: Actions and Predicates required for Keepaway

```
1  BeginTaskDescription("BallHandlingPolicy");
2  Environment("RoboCupSoccerSimulator");
3  BeginEpisode( "InitializeKeepaway(3, 2, 25, 25)" );
4  WrapperPolicy("KeeperPolicy", "BallIsKickable" )
5  AddToActionSpace( "HoldBall()" );
6  AddToActionSpace( "PassKThenReceive()" );
7  AddAdviceActions( "HandCodedBallHandling" );
8  StateSpaceDefs( "SortKeeperDistances", "K");
9  StateSpaceDefs( "SortTakerDistances", "T" );
10 AddToStateSpace( "dist(K[1], C)" );
11 AddToStateSpace( "dist(K[2], C)" );
12 AddToStateSpace( "dist(K[3], C)" );
13 AddToStateSpace( "dist(T[1], C)" );
14 AddToStateSpace( "dist(T[2], C)" );
15 AddToStateSpace( "dist(K[1], K[2])" );
16 AddToStateSpace( "dist(K[1], K[3])" );
17 AddToStateSpace( "dist(K[1], T[1])" );
18 AddToStateSpace( "dist(K[1], T[2])" );
19 AddToStateSpace( "Min(dist(K[2], T[1]), dist(K[2], T[2]))" );
20 AddToStateSpace( "Min(dist(K[3], T[1]), dist(K[3], T[2]))" );
21 AddToStateSpace( "Min(ang(K[2], K[1], T[1]),
22                   ang(K[2], K[1], T[2]))" );
23 AddToStateSpace( "Min(ang(K[3], K[1], T[1]),
24                   ang(K[3], K[1], T[2]))" );
25 AddTerminationCondition( "TakerHoldsBall" );
26 AddTerminationCondition( "BallOutOfBounds" );
27 StepReward( "TimeElapsed(now, lastStepStart)" );
28 OnEpisodeEnd( "RestartFromBeginning" );
29 StartLearning();
```

**Figure 5**: BLTL instructions for the Keepaway ball-handling task.

their names suggest. `InitializeKeepaway(nkeepers, ntakers, h, w)` initializes the playing field as in Figure 2 with the specified height and width and number of keepers and takers.

Each of the actions requires several predicates, some of which define abstract mathematical relationships while other are domain related, such as `BallOutOfBounds()` which evaluates to `true` if the ball is out of bounds.

Following the bootstrap learning philosophy, each of the above skills and primitives should have been taught to the student by the teacher previously. Those that are sequential decision making tasks, such as GetOpen() may be taught using the language we propose in this paper. Others, such as BallOutOfBounds() may be more suited to other learning approaches such as supervised learning. As such, they must be specified to the player using different communication primitives. Because this paper focuses on task specification for RL tasks, those primitives are beyond the scope of this paper. Assuming that the above skills and primitives *have* already been taught to the student, the teacher can use them to specify the keepaway task using the BLTL language.

**The Keepaway lesson** For the agent to learn an effective policy for ball handling in keepaway, the teacher must first specify how to start and end an episode, then tell the student what the state and action spaces are, then allow the student to practice the game over multiple episodes.

Figure 5 shows the series of instructions needed to specify the keepaway ball-handling task. Most lines are self-explainatory and are similar to those used in the GridWorld and BlocksWorld examples. The 13 state variables when the keeper has the ball, defined by the series of AddToStateSpace() functions, are the same as the ones commonly used for learning this task (Stone, Sutton, & Kuhlmann, 2005), and consist of several distances and angles among the keepers and takers. The `StateSpaceDefs` are needed to create intermediate variables needed to define those 13 state variables.

Note that the functions map relatively easily from natural language to the formal specification. For example, “learn a new soccer-related task” is represented in line 2; “learn when to hold the ball and when to pass if you have the ball, and use your existing policy when you don’t have the ball” is represented in line 4; and “base your decision in part on the distances of the players to the center of the field” is represented in lines 10–24. Such NLP in a constrained domain is currently possible (Kuhlmann *et al.*, 2004) and would enable a domain expert with no special RL knowledge to express the learning task to the BL student.

**Advice and Transfer learning** At any stage during learning, the teacher may wish to give advice to influence policy learning, as in (Kuhlmann *et al.*, 2004). For example, the teacher may advise that if taker 1 is more than 50m away, the student should consider holding the ball. The command would be:

```
AdviseAction("dist(K[1],T[1])", "GreaterThan(50)",
             "HoldBall()");
```

If we have already trained the 3v2 example shown above, we could use it to initialize a new policy that learns 3v2 on a larger field, as in (Taylor & Stone, 2005). If the previous learned policy was called “BallHandlingSmallField”, then we would change line 3 and add a new line after it as follows:

```
BeginEpisode("InitializeKeepaway(3,2,50,50)");
SourcePolicy("BallHandlingSmallField",true);
```

In our example, instead of using the above methods for advice and transfer, we told the agent to use the existing hand-coded policy as advice (this can also be seen as a form of transfer). This was done on line 7 with the use of the `AddAdviceActions` command. The learning agent we implemented can make use of advice actions from many sources – for instance, if the environment provided an interface for a human teacher to “remote control” the agent, thus providing instruction by demonstration, the syntax could have been:

```
AddAdviceActions( "RemoteControl" );
```

## Keepaway Results

Since the focus of this paper is on the language for a teacher to communicate an RL task to a student, the main mark of success is whether a BL student can, with no prior knowledge of the properties of the desired task, begin learning the desired task as specified by a teacher.

Initially, using this protocol, we were able to initiate learning for keepaway identically to the description in Stone, Sutton, & Kuhlmann (2005), which does not use advice, and verified that we can obtain similar results. We then used the BLTL language to add the hand-coded policy as a form of advice, where the learning agent implemnts an advice-taking algorithm similar to that of (Kuhlmann *et al.*, 2004). The experience-replay agent uses advice by simply adding 10 to the reward at each timestep if the action it takes agrees with the action that was suggested by the advice, and subtracting 10 from the reward if the action it selects disagrees with the advice. Additionally, the agent executes the action selected by the hand coded policy with probability $\alpha$, and takes the action recommended by its own learned policy according $(1 - \alpha)$. Initially $\alpha = 1$, but the value of $\alpha$ decays exponentially so that it reaches zero after the first 30 episodes (each episode is one game of keepaway). Figure 6 shows average game times as learning progresses, where each point on the curve represents the average time of the previous 50 games (or all previous games if fewer than 50 have been played). The agents quickly achieve good average game times of about 15 seconds (equal to the hand-coded policy), and eventually achieve average game times of up to 25 seconds, which is among the best average game times ever reported on standard keepaway. The saw-tooth appearance of the learning curves is due to the fact that each player is learning a policy independently, in batch-mode, so each spike represents the time that a new policy update has occurred in one of the agents. Since the agents are learning independently, from the perspective of an individual agent the world is not a true semi-MDP, because the environment (the other agents) are changing. This makes the optimal policy a moving target and makes experience replay slightly unstable. Nevertheless, the agents reach extremely good performance in a very small number of games.

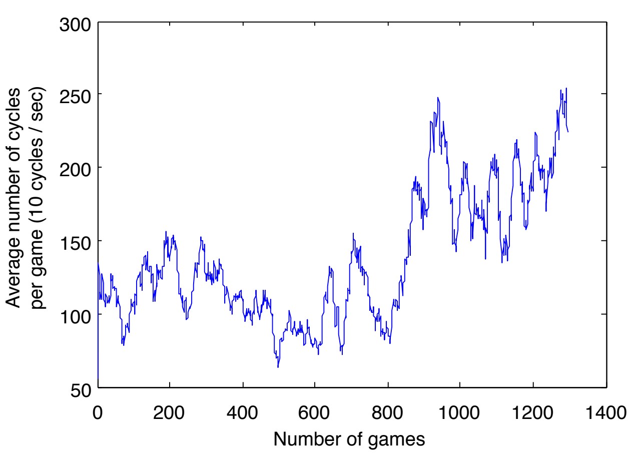


**Figure 6**: Performance results with Keepaway using advice. Peak performance of 25 second average game times is among the best reported in the literature.

## Conclusions and Future Work

The main contribution of this paper is the introduction of an architecture and interface language for teaching sequential decision making tasks to reinforcement learning agents. The BLTL language allows tasks to be specified concretely in terms of starting states, reward functions, and termination conditions. In addition, the teacher may provide advice and suggest sources for transfer learning, and may decompose complex tasks into multiple smaller lessons, allowing different policies learned with different methods to be combined synergistically. The BLTL language forms the cornerstone for the larger Bootstrap Learning project, which integrates even more machine learning methods for teaching agents to solve many different types of problems, not just sequential decision making tasks.

We have illustrated our language on a GridWorld task, and have implemented and tested the first BL agent on a BlocksWorld task and a RoboCup soccer task. An important next step is to test the BLTL language for other tasks, both in RoboCup and in completely different domains such as flying an unmanned aerial vehicle (UAV). Future work for expanding the BLTL framework to the full Bootstrap Learning framework will include natural language mapping to the BLTL language, and adding more machine learning methods.

The BLTL language also lays the groundwork for future development of agents which can decide for themselves what tasks to learn and how to learn them, rather than waiting for task specifications and advice from a teacher. Current and future work on agents that discover new learning tasks (for example automatic subgoal discovery (McGovern & Barto, 2001)) will benefit from the ability to formulate new tasks using the BLTL language. Similarly, the introduction of BLTL exposes the important future goal of enabling an agent to automatically select, based on task characteristics, from among the large (and still growing) number of domain-independent RL algorithms and possible parameterizations thereof.

## References


Ackley, D. H.; Hinton, G. E.; and Sejnowski, T. J. 1985. A learning algorithm for boltzmann machines. *Cognitive Science* 9:147–169.

Brafman, R. I.; Tennenholtz, M.; and Schuurmans, D. 2002. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. In *Journal of Machine Learning Research*, 213–231.

Dayan, P., and Hinton, G. E. 1993. Feudal reinforcement learning. In Hanson, S. J.; Cowan, J. D.; and Giles, C. L., eds., *NIPS 2005*. San Mateo, CA: Morgan Kaufmann.

Dietterich, T. G. 1998. The MAXQ method for hierarchical reinforcement learning. In *ICML*. Madison, WI.

Fasel, I.; Kalyanakrishnan, S.; and Stone, P. 2008. Reinforcement learning with deep belief networks. (unpublished manuscript).

Hinton, G. E., and Salakhutdinov, R. R. 2006. Reducing the dimensionality of data with neural networks. *Science*.

Hinton, G. E.; Osindero, S.; and Teh, Y. W. 2006. A fast learning algorithm for deep belief nets. *Neural Computation*.

Hinton, G. E. 2002. Training products of experts by minimizing contrastive divergence. *Neural Computation*.

Kuhlmann, G.; Stone, P.; Mooney, R.; and Shavlik, J. 2004. Guiding a reinforcement learner with natural language advice: Initial results in RoboCup soccer. In *The AAAI-2004 Workshop on Supervisory Control of Learning and Adaptive Systems*.

Lagoudakis, M. G., and Parr, R. 2003. Least-squares policy iteration. *Journal of Machine Learning Research* 4:1107–1149.

Lin, L. 1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. In *Machine Learning*, 293–321.

Maclin, R., and Shavlik, J. W. 1996. Creating advice-taking reinforcement learners. *Machine Learning* 22:251–282.

McGovern, A., and Barto, A. G. 2001. Automatic discovery of subgoals in reinforcement learning using diverse density. In *ICML*, 361–368. Williamstown, MA.

Moore, A. W., and Atkeson, C. G. 1993. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning* 13:103–130.

Pietro, A. D.; While, L.; and Barone, L. 2002. Learning in RoboCup keepaway using evolutionary algorithms. In et al., W. B. L., ed., *GECCO 2002*, 1065–1072. New York.

Schaal, S. 1997. Learning from demonstration. In Mozer, M.; Jordan, M.; and Petsche, T., eds., *Advances in Neural Information Processing Systems 9*. Cambridge, MA: MIT Press.

Soni, V., and Singh, S. 2006. Using homomorphisms to transfer options across continuous reinforcement learning domains. In *Proceedings of the Twenty First National Conference on Artificial Intelligence*.

Stone, P.; Sutton, R. S.; and Kuhlmann, G. 2005. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior* 13(3):165–188.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press.

Sutton, R.; Precup, D.; and Singh, S. 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence* 112:181–211.

Taylor, M. E., and Stone, P. 2005. Behavior transfer for value-function-based reinforcement learning. In *AAMAS 2005*, 53–59.

Torrey, L.; Walker, T.; Shavlik, J.; and Maclin, R. 2005. Using advice to transfer knowledge acquired in one reinforcement learning task to another. In *ECML 2005*. Porto, Portugal.

Watkins, C. J. C. H. 1989. *Learning from Delayed Rewards*. Ph.D. Dissertation, King's College, Cambridge, UK.

White, A.; Lee, M.; Butcher, A.; Tanner, B.; Hackman, L.; and Sutton, R. 2007. Rl-glue distribution, *http://rlai.cs.ualberta.ca/rlbb/top.html*.