Adaptation-Based Programming in Java

Tim Bauer Martin Erwig Alan Fern Jervis Pinto
School of EECS
Oregon State University
{bauertim,erwig,afern,pinto}@eecs.oregonstate.edu

Abstract

Writing deterministic programs is often difficult for problems whose optimal solutions depend on unpredictable properties of the programs' inputs. Difficulty is also encountered for problems where the programmer is uncertain about how to best implement certain aspects of a solution. For such problems a mixed strategy of deterministic programming and machine learning can often be very helpful: Initially, define those parts of the program that are well understood and leave the other parts loosely defined through default actions, but also define how those actions can be improved depending on results from actual program runs. Then run the program repeatedly and let the loosely defined parts adapt.

In this paper we present a library for Java that facilitates this style of programming, called *adaptation-based programming*. We motivate the design of the library, define the semantics of adaptation-based programming, and demonstrate through two evaluations that the approach works well in practice. Adaptation-based programming is a form of program generation in which the creation of programs is controlled by previous runs. It facilitates a whole new spectrum of programs between the two extremes of totally deterministic programs and machine learning.

Categories and Subject Descriptors D [*3*]: 3

General Terms Languages

Keywords Java, Reinforcement Learning, Partial Programming, Program Adaptation

1. Introduction

Programs written in traditional programming languages are typically deterministic, that is, at every step they specify exactly which action to take next. In contrast, there are many problems that do not lend themselves easily to such a deterministic implementation. This is particularly the case for applications that involve decisions for which the best choice depends on aspects of the program input that are unpredictable. Consider, for example, the implementation of an intelligent agent for a real-time strategy game. If the choices are to retreat or attack, to wait or advance, to move toward or away from an object, etc., the best course of action depends on many aspects of the current game state. It is practically impossible to anticipate all the situations such an agent can be faced with and program a specific strategy for each and every case. The programmer is left with considerable uncertainty about how to write deterministic code that makes good choices across all possibilities. Note, however, that even in these situations a programmer is still often able to specify a numeric reward/penalty function that distinguishes good program behavior from poor behavior. For example, in a real-time strategy game this reward might relate to the damage inflicted on an enemy versus the damage taken.

In situations like these, it would be nice if a programmer could leave specific decisions that they are uncertain about as open and instead provide "reward" signals that indicate whether the program is performing well or not. Of course, this would only be useful if the program could automatically "figure out" how to select actions at the open decision points in order to maximize the reward obtained during program executions. One approach to achieve this behavior is to provide functionality for the program to learn, over repeated runs of the program, to optimize the selection of actions at open decision points. This learning functionality should not be visible for the programmer, who needs only to specify their uncertainty about decisions and provide the reward signal.

We call this approach to programming *adaptation-based programming* (ABP) because programs are written in a way that defines their own adaptation based on the situations encountered at runtime. In this paper we describe the realization of ABP as a library for Java. The design of this library [4] has to address many questions. For example:

- How can program decisions be made adaptable? We need constructs to mark parts of a program as adaptable, and we must be able to specify default and adaptive behaviors as well as rewards and the allocation of rewards to behaviors.
- What granularity of adaptation should be supported? How can rewards be shared among different decisions, and how can they be kept separate if needed? How can adaptation be properly scoped with respect to program time and locality?
- What mechanisms do programmers need for controlling the adaptation process? Under which conditions can a decision be considered fully adapted as opposed to still adapting? How can adaptations be made persistent?

Answers to these questions determine the design of a library for adaptation-based programming, and will be addressed in Section 2 by discussing variations of a small ABP example. Specifically, this section provides a programmer's point of view on the library design, which is an important aspect since ease of use and clarity of concepts is an essential precondition for a widespread adoption of a library.

Our library is designed to be used by regular programmers, not machine-learning experts. Hence this work is distinct from Java reinforcement learning libraries (e.g. [20]). In fact, the user is insulated from the details of the RL algorithm used and they need not understand its implementation to use our library. By design the library interface and concepts necessary to use it are minimal.

Before we describe the details we briefly discuss the question regarding the scope of program adaptation supported by our ABP library. One could certainly envision the adaptation of, say type

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PEPM'11, January 24–25, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0485-6/11/01...$10.00

definitions, in response to repeated program executions to optimize data representations. Similarly, the change of control structures could be a response to feedback gathered from runtime information. However, one practical problem with such an approach is that it generally requires recompilation of programs after adaptation, which leads to a more complicated framework compared to a system that only works completely within a single compiled program. Moreover, if the adaptation system does not require recompilation, this will certainly lead to a more efficient runtime behavior of adaptation, which is a very important aspect since problems that are amenable to ABP often require an enormous number of program runs to successfully adapt, see also Section 4. The design of our Java ABP library is therefore based on compilation-invariant adaptations.

The remainder of this paper is structured as follows. After illustrating the use of our Java ABP library in the next section, we will provide a more formal description of the semantics of ABP in Section 3. In particular, we will describe the meaning of an ABP program as an optimization problem and then define the meaning through the learning of good (or optimal) policies. Section 4 provides an evaluation of our library using a dice game and a real-time strategy game. We discuss related work in Section 5, future work in Section 6, and conclude with Section 7.

2. Library Support for ABP in Java

To illustrate our program generation approach suppose we are implementing a simple hunting simulation involving two wolves hunting a rabbit. Our goal is to write a program where the wolves work together to trap the rabbit.

In Section 2.1 we provide some implementation details about the example application to set the stage for the application of ABP. In Section 2.2 we then illustrate how to realize the application using our ABP library. We will also use the example to motivate the design of the library components. In Section 2.3 we discuss some additional aspects regarding the programmer's control over the machine learning process. In particular, we show how a programmer can improve (that is, speed up) the adaptation process by coding insights about the domain. Finally, we discuss in Section 2.4 the flexibility and safety that our library design offers for working with multiple adaptations.

2.1 Example Scenario

Our game world's state is represented by a two-dimensional grid as in the class given below.

```
class Grid {
  Grid moveWolf1(Move m);
  Grid moveWolf2(Move m);
  Grid moveRabbit(Move m);

  boolean rabbitCaught();

  ...

  static Grid INITIAL;
}
```

This class maintains the coordinate positions of each animal on the grid and implements game logic and rules. We omit some of the details that are not important for the following discussion. Changes to the state are made via three move methods `moveWolf1`, `moveWolf2`, and `moveRabbit`. Each of these methods returns a new `Grid` with the move applied. Additionally, a method `rabbitCaught` is given that checks to see if the rabbit has been captured in the current game grid. The static constant `INITIAL` corresponds to the instance of the world with the wolves at the top-left and the rabbit at the bottom right. This is used for the game's initial configuration.

Permissible moves for each animal are described via a `Move` enumeration.

```
enum Move {
  STAY,LEFT,RIGHT,UP,DOWN;

  static final Set<Move> SET =
    unmodifiableSet(EnumSet.allOf(Move.class));
}
```

For our example the `Rabbit` class given below is an interface for any number of fixed strategies that the rabbit may take.

```
abstract class Rabbit {
  abstract Move pickMove(Grid g);
  static Rabbit random();
}
```

The static method `random` returns a rabbit that moves randomly.

A game proceeds as follows. First, the rabbit may observe the location of the wolves and then move one square in any direction. Next, the first wolf gets to observe the rabbit's move and move itself. After that the second wolf gets to observe both prior moves and then move itself. If at the end of a round either wolf has landed on the rabbit's square, the rabbit is captured. Otherwise, the hunt continues.

To make the problem more interesting we allow the x-coordinate of the world to wrap. Hence an animal at the left end of the grid can wrap around to the right end and vice versa. In this way, the rabbit could always escape if the wolves close from the same direction. Hence, the wolves must be smart enough to cooperate and close from opposite directions.

A programmer solving this problem with the above definitions might initially sketch out pseudocode such as that given below.

```
Rabbit rabbit = Rabbit.random();
Grid g0 = Grid.INITIAL;
while (true) {
  Move rabbitMove = rabbit.pickMove(g0);
  Grid g1 = g0.moveRabbit(rabbitMove);

  // ... pick move for wolf 1
  Move wolf1Move = ???
  Grid g2 = g1.moveWolf1(wolf1Move);

  // ... pick move for wolf 2
  Move wolf2Move = ???
  Grid g3 = g2.moveWolf2(wolf2Move);

  if (g3.rabbitCaught()) {
    ... raise our score a large amount
    break;
  }

  g0 = g3;
  ... lower our score a small amount
}
```

The grid `g0` represents the world state at the beginning of the loop each iteration. Again this includes the location of each animal. We move each animal as described before in the rules. The `Grids` `g1`, `g2`, and `g3` correspond to the game state after each animal's move.

For now, we are uncertain about how to select the wolf moves, so we indicate that with question marks. Near the end of the loop, we check to see if the rabbit has been captured. If not, we reassign `g0` and perform another round.

This pseudocode motivates some observations.

- There is a natural sense of constrained uncertainty when our wolves select their moves. They have to pick one of a small finite set of moves.
- We can consider the score as a reward or indicator of success or failure. It can be used as a metric telling us how successful our wolves are. Moreover, it suggests that it is easier to classify good and bad solutions than to generate them.

- There is an inherent dependency between each wolf's strategy. They must work together. Moreover, the reward or score applies to both as they share a common goal.

2.2 Adaptation Concepts

The coupling between choice and reward suggests that some sort of abstraction could automatically select sequences of moves and then evaluate those moves by checking the score. Over time, this abstraction could identify better and better sequences of choices so as to optimize their average reward. Implementing these notions is the goal of our ABP library which we now describe.

We define an *adaptive variable* (*adaptive* for short) as one of these points of uncertainty in a program where we must make some decision amongst a small discrete set of choices. A value generated by one of these variables is called an adaptive value. We call the location of this uncertain selection a *choice point*.

An adaptive can suggest a potential action at a choice point. But in order to do so, it requires some unique descriptor that identifies the state of the world. In our above example, there are two points of uncertainty, namely the wolf move selection indicated with `???`. In our library we represent adaptive variables via the `Adaptive` class shown below.

```
public class Adaptive<C,A> {
  public A suggest(C context, Set<A> actions);
}
```

Adaptive values are parameterized by two type variables. The first (`C`) corresponds to the context or world state, and the second (`A`) corresponds to the type of permissible actions that the adaptive can take.

The context parameter `C` gives the library a clean and efficient description of what the world looks like at any given point. In our hunting example, this will initially be the `Grid` class.

The `suggest` method is our way of asking the adaptive for an appropriate value for some context. Moreover, we pass a set of permissible actions for the adaptive to choose from. We are asking the adaptive value, "If the state of the world is `context`, what is a good move from the set of `actions`?"

In our wolf hunt example, each of our wolves could be represented by an adaptive. The context type parameter would be the world state `Grid`, and the action type would simply be a `Move`. We illustrate this shortly.

The dependency between the moves we select for our wolves elicits another important observation: multiple adaptives might share a common goal. Under this view our adaptive wolves must share a common reward stream. The score in the game (the reward) applies to both wolves, not just one. This sharing of rewards asks for a scoping mechanism that allows the grouping of multiple adaptive variables.

We define this common goal as an *adaptive process*. It represents a goal that all its adaptives share, it distributes rewards (or penalties) to its adaptives, and it manages various history information that its adaptive variables learn from.

We show the basic interface in our ABP framework defining an adaptive process below.

```
public class AdaptiveProcess {
  static AdaptiveProcess init(File status);
  public <C,A> Adaptive<C,A>
           initAdaptive(Class<C> contextClass,
                        Class<A> actionClass);

  public void reward(double r);
  public void disableLearning();
}
```

A new adaptive process is created via the class's `init` method. The source file argument permits the `AdaptiveProcess` to automatically be persisted between runs. The first time the program is run, a new file is created to save all information about adaptives. When the program terminates, the process and all its adaptives are automatically saved. Successive program invocations will reload the learning process information from the given file. This persistence permits the adaptive variables contained in the process to evolve more effective strategies over multiple program runs.

Individual adaptives are created with the `initAdaptive` method. The context and action type parameters are passed in as arguments, typically as class literals. This permits us to dynamically type check persisted data as it is loaded.

In our wolf example we would use the following code to initialize our process and the adaptives for each wolf.

```
public class Hunt {
  public static void main(String[] args){
    AdaptiveProcess
      hunt = AdaptiveProcess.init(new File(args[0]));
    Adaptive<Grid,Move>
      w1 = hunt.initAdaptive(Grid.class,Move.class),
      w2 = hunt.initAdaptive(Grid.class,Move.class);
```

We specify the aforementioned rewards of an adaptive process with the `reward` method. Positive values indicate positive feedback and tell the process that good choices were recently made, negative values indicate bad choices were recently made.

Upon receiving a reward, the adaptive process will consider the previous actions it has taken and adjust its view of the world accordingly. This permits later calls to `suggest` to generate better decisions. Details of the mechanics are described more formally in Section 3. We discuss the final method of `AdaptiveProcess`, `disableLearning`, later.

Continuing with the previous block of code, the body of our game we sketched out earlier in pseudo code could be implemented as follows.

```
    Rabbit rabbit = Rabbit.random();
    Grid g0 = Grid.INITIAL;
    while (true) {
      Move rabbitMove = rabbit.pickMove(g0);
      Grid g1 = g0.moveRabbit(rabbitMove);

      // ... pick move for wolf 1
      Move wolf1Move = w1.suggest(g1,Move.SET);
      Grid g2 = g1.moveWolf1(wolf1Move);

      // ... pick move for wolf 2
      Move wolf2Move = w2.suggest(g2,Move.SET);
      Grid g3 = g2.moveWolf2(wolf2Move);

      if (g3.rabbitCaught()) {
        // ... raise our score a large amount
        hunt.reward(CATCH_REWARD);
        break;
      }

      g0 = g3;
      // ... lower our score a small amount
      hunt.reward(MOVE_PENALTY);
    }
  }
}
```

The interesting pieces of this code are those near the comments where we filled in adaptive code and we discuss them here. First, the wolf strategies are as simple as calls to the adaptive variables' `suggest` methods. In each case, we pass in the current game state (the `Grid`) and the set of all moves `Move.SET`. Note that every move is legal; we simply translate moves into walls as `Move.STAY` for simplicity.

Second, where we referred to scores earlier, we place calls to the `reward` method of the adaptive process representing our hunting goal. After each unsuccessful round we assess a small penalty `MOVE_PENALTY`. Once the rabbit is captured we reward a large amount `CATCH_REWARD`. In our example we use -1 and 1000,

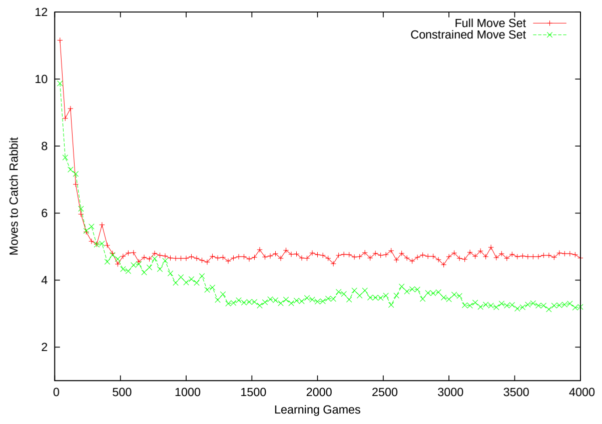

Figure 1. Average number of moves to catch the rabbit.

respectively.

If we run the game over multiple iterations, the average number of moves for the wolves to catch the rabbit drops fairly quickly. After a few thousand runs the average is between 4 and 6 moves. With grid dimensions of 3×4 from the initial state, the worst-case optimal solution is 4 moves. That is, if the rabbit stays as far away from the wolves as possible each round, it will take up to 4 moves to capture it. With a random rabbit, we should expect smaller averages.

The reason for this initial discrepancy is two-fold. First, a few thousand iterations is not a long time for the adaptive process to learn an optimal strategy; indeed after a few thousand more games, the average drops lower. Second, and more importantly, even once an optimal strategy is found, the adaptive process will continue to search for better ones and may try suboptimal strategies. An initially bad looking move might lead to a better overall solution. Hence, it is necessary for the adaptive process to operate suboptimally as its adaptive variables try various move sequences.

Recall the `disableLearning` method of `AdaptiveProcess` whose description we deferred earlier. This method tells the adaptive process to suspend its search through suboptimal values and use only the best moves for every given world state. This is a means for the programmer to indicate to the adaptive process that it should stop searching and work deterministically with the knowledge it currently has. During practice we explore new strategies, but during the big game we use what we know works. If we transition to this optimal mode and run the wolf hunt program shown before, the average number of moves to catch the rabbit drops to around five after just a few thousand rounds.

In Figure 1 the plot titled “Full Move Set” shows the average number of rounds necessary to catch the rabbit as a function of the number of games learned when playing in this optimal mode. We discuss the second plot presently.

2.3 Optimizing the Adaptation Behavior

Recall in one extreme we consider algorithms that are fully specified, where the programmer completely describes how to solve the problem and manually handles all cases. Such algorithms can be complicated even for easy tasks. In an opposite extreme we offload the entire strategy to some function that we learn via machine learning algorithms. But in doing so we forfeit control over various aspects of it. Furthermore, debugging such algorithms is difficult.

One of the goals of our ABP library is to offer a middle ground; the programmer can specify some of the analytical solution, but leave small pieces to be learned. An example of this is shown by the `suggest` method of `Adaptive`. The second argument takes a set of permissible moves.

Let us suppose that our programmer implementing the wolf hunt game observed that any successful strategy would require the two wolves moving in opposite directions. In that case, we could easily specify this partial knowledge by passing in a subset of the permissible moves. We illustrate this by showing a slice of the earlier code inside the game loop.

```
// ... pick move for wolf 1
Move wolf1Mv = w1.suggest(g1,setOf(LEFT,DOWN,STAY));
Grid g2 = g1.moveWolf1(wolf1Mv);

// ... pick move for wolf 2
Move wolf2Mv = w2.suggest(g2,setOf(RIGHT,DOWN,STAY));
Grid g3 = g2.moveWolf2(wolf2Mv);
```

Including the above constraint gives the more efficient “Constrained Move Set” plot shown in Figure 1. (The `setOf` function just wraps a list into a `Set`.) This constrained learning permits a better solution, and one that can typically be learned faster since the adaptives must explore fewer alternatives.

2.4 Varying Context and Action Types

In general, different adaptives within a process may contain different context and action types. Recall the method from `AdaptiveProcess` to create a new adaptive.

```
public <C,A> Adaptive<C,A>
          initAdaptive(Class<C> contextClass,
                       Class<A> actionClass);
```

Hence, the type variables `C` and `A` for the context and action type are applied to the adaptives, not the process. For example, suppose our hunting game contained a lion and a wolf working together. These different animals might have completely different capabilities and different move types. Imagine that a lion can `POUNCE` as well as do the other basic moves. Then the adaptive representing the lion might have a `LionMove` as its action type and the wolf adaptive could have a `WolfMove`.

The parametric polymorphism here gives us some extra static type checking and saves us having to explicitly cast objects. We are less likely to ask the wrong adaptive for an incorrect move type.

3. Formalizing ABP: Semantics and Learning

In the previous section, we illustrated how an `AdaptiveProcess` allows a programmer to explicitly encode their uncertainty about a program via `Adaptive` objects, as well as the desired program behavior via `reward` statements. The intention is for the resulting adaptive program to learn, over repeated executions, to make choices for adaptive values that maximize reward. However, the notions of “learning” and “maximize reward” have not yet been made precise. In this section, we first formalize these notions by providing a semantics that defines how a program with an `AdaptiveProcess` defines a precise optimization problem that serves as the learning objective. Next, we describe how this semantics allows us to draw on work in the field of reinforcement learning (RL) to solve the optimization problem over repeated runs of the program. Finally, we describe some useful extensions beyond the basic semantics, which are supported by the library and provide the programmer with additional flexibility.

3.1 Induced Optimization Problem

For simplicity, and without loss of generality, we describe the semantics for programs that include a single `AdaptiveProcess` and where the set of adaptive objects appearing in the program is fixed and statically determined. We also assume that the context and action types of each adaptive are discrete finite sets. The relaxation of these assumptions is discussed at the end of this section. Recall that the Wolf-Rabbit program from Section 2 contains a single `AdaptiveProcess` and two adaptives, each responsible for selecting actions for a wolf via calls to the `suggest` method.

The programmer did not need to specify an implementation for `suggest`, but rather expects that the ABP machinery will optimize this method for each adaptive in order to produce good program behavior, that is, capture the rabbit quickly.

To specify the optimization problem induced by an adaptive program, we first introduce some definitions. A *choice function* for an adaptive with context-type C and action-type A is a function from C to A. A *policy* for an adaptive program is an assignment of a choice function to each adaptive that appears in the program. In the Wolf-Rabbit example, a policy is a pair of functions, one for each wolf, that returns a movement direction given the current game context.

Given an adaptive program P and a policy π, we define the execution of P on input x with respect to π to be an execution of P where each call to an adaptive's `suggest` method is serviced by evaluating the appropriate choice function in π. Each such execution is deterministic and results in a deterministic sequence of calls to the `reward` method, each one specifying a numeric reward value. The sum of these rewards is of particular interest and we will denote this sum by $R(P,\pi,x)$, noting that R is a deterministic function of its arguments. In our Wolf-Rabbit example, the program input x might correspond to an initial position of the wolves and rabbit or a random seed that determines those positions and $R(P,\pi,x)$ depends on whether the wolves catch the rabbit and if so how long it took when following π.

Intuitively, we would like to find a π that achieves a large value of total reward $R(P,\pi,x)$ for inputs x that we expect to encounter. More formally, let D be a distribution over possible inputs x in our intended application setting. For instance, in the Wolf-Rabbit example, D might be the uniform distribution over possible initial positions of the wolves and rabbit, or might assign probability one to a single initial position. Given such a distribution D and an adaptive program P, we can now define the induced optimization problem to be that of finding a policy π that maximizes the expected value of $R(P,\pi,x)$ where x is distributed according to D. That is, our goal is to find π^* as defined below.

$$\pi^* = \arg\max_{\pi\in\Pi} \mathbb{E}\left[R(P,\pi,x)\right],\ x \sim D \qquad (1)$$

That is we wish to find the arguments that maximizes the expression where Π is the set of all policies, $\mathbb{E}[\cdot]$ is the expectation operator, and $x \sim D$ denotes that x is a random variable distributed according to distribution D. In order to make this a well defined objective we assume that all program executions for any π and x terminate in a finite number of steps so that $R(P,\pi,x)$ is always finite.[1]

In all but trivial cases finding an analytical solution for π^* will not be possible. Thus, the ABP framework attempts to "learn" a good, or optimal, π based on experience gathered through repeated executions of the program on inputs drawn from D. Intuitively, during these executions the learning process explores different possibilities for the choices of the various adaptives in order to settle on the best overall policy for the program. Once this policy is found, the learning process can be turned off and the policy can be used from that point onward to make choices for the adaptives.

3.2 Learning Policies

Now consider how we might learn a good, or optimal, policy via repeated program executions. First, note that given a particular policy π we can estimate the expected reward $\mathbb{E}\left[R(P,\pi,x)\right]$ by first sampling a set of inputs $\{x_1,\ldots,x_n\}$ from D, then executing the program on each x_i with respect to π to compute $R(P,\pi,x_i)$, and then averaging these values. A naive learning approach then is to estimate the expected reward for each possible policy and then return the policy with the best estimate. This, however, is not a practical alternative since in general there are exponentially many policies. For example, in our Wolf-Rabbit example, the number of choice functions for each of the wolf adaptives is equal to the number of mappings from contexts to the five actions, which is exponential in the number of contexts. Thus, this naive enumeration approach is only practical for problems where the number of adaptives and contexts for those adaptives is trivially small.

To deal with the combinatorial complexity we leverage work in the area of RL [18] for policy learning. RL studies the problem of learning controllers that maximize expected reward in controllable stochastic transition systems. Informally, such a system transitions among a set of control points with rewards possibly being observed on each transition. Each control point is associated with a set of actions, the choice of which influences (possibly stochastically) the next control point and reward that is encountered. An optimal controller for such a system is one that selects actions at the control points to maximize total reward. It is straightforward to view an adaptive program as such a transition system where the control points correspond to the adaptives in a program. In particular, each program execution can be viewed as a sequence of transitions between control points, or adaptives, with interspersed rewards, where the specific transitions and rewards depend on the actions selected by the adaptives.

More formally, it has been shown [2, 13] that state-machine or program-like structures that are annotated with control points are isomorphic to Semi-Markov Decision Processes (SMDPs), which are widely used models of controllable stochastic transition systems. The details of SMDP theory and this result are not critical to this paper. However, the key point is that there are well-known RL algorithms for learning policies for SMDPs based on repeated interaction with those systems. This means that we can use those algorithms as a starting point for the learning mechanisms of our ABP library. In particular, our first version of the library is based on an algorithm known as SMDP Q-learning [5, 12], which extends the Q-learning algorithm [22] from Markov Decision Processes to SMDPs. Below we describe SMDP Q-learning in terms of its implementation in our ABP library. Naturally, future work will investigate other learning approaches, both existing and new, to understand which approaches are best suited for optimizing the adaptive programs produced by end programmers.

In the context of ABP, SMDP Q-learning aims to learn a *Q-function* for each adaptive in a program. A Q-function for an adaptive with context-type C and action-type A is a function from $C \times A$ to real numbers. In our current library, the Q-function for each adaptive is simply represented as a table, where rows correspond to the possible contexts and columns correspond to possible actions. The Q-function entry for adaptive object o, context c, and action a will be denoted by $Q_o(c,a)$. Intuitively, SMDP Q-learning aims at learning values so that $Q_o(c,a)$ indicates the goodness of adaptive o selecting action a in context c. Thus, after learning a Q-function, the choice function for each adaptive o simply returns the action that maximizes $Q_o(c,a)$ for any given context. Accordingly, the policy for the program after learning is taken to be the collection of the choice functions induced by the learned Q-function. It remains to describe the formal semantics of the Q-function and the algorithm used to learn it from program executions.

The intended formal meaning of the Q-function entry $Q_o(c,a)$ is the expected future sum of rewards until program termination after selecting action a in context c for adaptive o and then assuming that all other adaptives act optimally. If the table entries satisfy this definition, then selecting actions that maximize the Q-functions results in an optimal policy. SMDP Q-learning initializes the Q-function tables arbitrarily (often to all zeros) and then incrementally updates the tables during program executions in a way that moves the tables toward satisfying the formal definition of the Q-function. Under certain technical assumptions the algorithm is guaranteed to

[1] If this does not hold, for example, for a continually running program, other standard objectives are available including temporally-averaged reward and total discounted reward. There are straightforward adaptations to the learning algorithm described below for both of these objectives [11].

converge to the true Q-function and hence the optimal policy. We now describe the simple updates performed by the algorithm.

SMDP Q-learning initializes the Q-table to all zeros and then iteratively executes the adaptive program P on inputs drawn from D while exploring different action choices for the adaptives that it encounters. In particular, when the adaptive process is in learning mode, each call to the `suggest` method of an adaptive is serviced by the Q-learning algorithm, which returns one of the possible actions for that adaptive and also updates the Q-table based on the observed rewards. Specifically, for the i'th call to `suggest` for the current program execution, let o_i denote the associated adaptive, c_i denote the associated context, and a_i denote the action selected by Q-learning for that call to suggest. Also let r_i denote the sum of rewards observed via calls to the `reward` method between o_i and o_{i+1}. Note that in some cases there will be no calls to `reward` between o_i and o_{i+1} in which case $r_i = 0$. After encountering the $(i+1)$'th call to `suggest`, SMDP Q-learning performs the following two steps:

(1) **Update Q-function.** Update $Q_{o_i}(c_i, a_i)$ based on r_i and the Q-table information for context c_{i+1} of adaptive o_{i+1} (see Equation 2 as detailed later).

(2) **Select Action.** Select an action to be returned by `suggest` for the adaptive o_{i+1}.

Notice that this learning algorithm does not require the storage of full trajectories resulting from the program executions, rather it only requires that we store information about the most recent and current adaptives encountered. It remains to provide details for each of these two steps.

Q-Function Update When the $(i+1)$'th call to `suggest` is encountered, the Q-learning algorithm performs an update to the Q-table entry $Q_{o_i}(c_i, a_i)$ according to the following equation,

$$Q_{o_i}(c_i, a_i) \leftarrow (1-\alpha)Q_{o_i}(c_i, a_i) + \alpha(r_i + \max_a Q_{o_{i+1}}(c_{i+1}, a)) \qquad (2)$$

where α is a real-valued *learning rate* between 0 and 1. Intuitively, this update is simply moving the current estimate of $Q_{o_i}(c_i, a_i)$ toward a refined estimate given by $r_i + \max_a Q_{o_{i+1}}(c_{i+1}, a)$. Since the update for entry $Q_{o_i}(c_i, a_i)$ is done at the $(i+1)$'th call to suggest and references the Q-table for o_{i+1} via the max operation, the above update is not well defined for that last adaptive in the program execution. Thus, we adjust the update for the last adaptive as follows. Let t be the number of calls to `suggest` throughout the program execution until termination. When the program terminates we update $Q_{o_t}(c_t, a_t)$ according to,

$$Q_{o_T}(c_T), a_T) \leftarrow (1-\alpha)Q_{o_T}(c_T, a_T) + \alpha \cdot r_T \qquad (3)$$

where r_T is the total reward observed after o_T until the program terminates.

Theoretically, the above update rule is guaranteed to converge to the true Q-value, provided that α is decayed according to an appropriate schedule. However, the theoretically correct decay sequences typically lead to impractically slow learning. Thus, in practice, it is common to simply select a small constant value for α. The default in our library is 0.01.

Action Selection After performing a Q-table update for a call to `suggest`, Q-learning selects an action to be returned by `suggest` for the adaptive. In theory, there are many options for this choice that all guarantee convergence of the Q-learning algorithm to the true Q-function. In particular, any action selection strategy that is *greedy in the limit of infinite exploration (GLIE)* suffices for convergence. A selection strategy is GLIE if it satisfies two conditions: (1) For every adaptive and context it tries each action infinitely often in the limit, and (2) In the infinite limit it selects actions that maximize the Q-function, that is, for adaptive o_i in context c_i select $a_i = \arg\max_a Q_{o_i}(c_i, a)$. One simple and common GLIE strategy is ε-greedy exploration [18] and is what is currently implemented in our library. This strategy selects the greedy action with probability $1-\varepsilon$ and selects a random action with probability ε for $0 < \varepsilon < 1$. If ε is decreased at an appropriate rate, this strategy will be GLIE. In practice, however, it is common to simply select a small constant value for ε, which is used throughout the learning period. The default value in our library is 0.3.

To summarize, the SMDP Q-learning algorithm implemented in our library has two parameters: the learning rate α and the exploration constant ε, which have small default values (0.01 and 0.3 respectively), but can also be set by the programmer if desired. The Q-function table is initialized to all zero values and updated each time a call to `suggest` is encountered. It is worth noting that, in addition to the conditions mentioned above, the convergence of Q-learning also requires that the adaptive contexts satisfy certain theoretical assumptions [12]. In practice these assumptions rarely hold, but nevertheless, Q-learning has proven to be a practically useful algorithm in many applications even when such conditions are not satisfied [7].

3.3 Extensions

Multi-Process Adaptive Programs Recall that the above semantics were defined for programs that include a single adaptive process. In some cases it is useful to include multiple adaptive processes in a program, and our library supports this. Each such process has its own set of adaptives and its own reward statements throughout the program. For example, in the Wolf-Rabbit example, a programmer might want to include one adaptive process for the wolves and a different adaptive process for the rabbit. The rabbit process could be used to learn avoidance behavior for the rabbit, with the reward structure for this process the opposite of that of the wolves. In particular, yielding positive rewards for each step the rabbit stays alive and a large negative reward for being caught.

As another example, it is sometimes the case that a program can be broken into independent components that can each be optimized independently yet still yield overall good program behavior. For example, suppose we are writing a controller for a video game character where we must control both high-level choices about which map location to move to next and lower-level choices about exactly how to reach those locations. The problems of selecting a good location and how to get to those locations often decouple and we could treat these as separate adaptive processes. The motivation for doing this is that it is possible that optimizing the individual processes is easier than attempting to optimize a single process that mixes all of the decisions together.

Our framework handles multi-process programs by simply running independent learning algorithms on each of them. Theoretically, this extension puts us in the framework of Markov Games [9], which are the game-theoretic extension of Markov Decision Processes. Such games can be either adversarial, where the objectives of the different processes are conflicting (such as the first example) or cooperative, where the objectives of the processes agree with one another as in the second example. Understanding the convergence properties of RL algorithms in such game settings is an active area of research [16] and is much less understood than the single process case. As such we do not pursue a characterization of the solution that will be produced by our library at this time, but note that in practice the type of independent training pursued by our library has often been observed to produce useful results [14, 19].

Multi-Episode Adaptive Programs In the above framework, learning occurred over many repeated executions of the adaptive program, each execution using a program input drawn from some distribution. In the Wolf-Rabbit example, each program execution corresponds to a single game. While it is possible to use scripts to “train” the program through repeated executions, this is often not convenient for a programmer. Rather, it may be preferable to be able to effectively run a large number of games via one execution of the program and learn from all of those games. In our Wolf-Rabbit example, one way this might be done is to simply have a high-level loop in the main program that repeatedly plays games, allowing the `AdaptiveProcess` to learn from the entire game sequence.

In concept, learning may be successful on such a multi-game program execution. However, this basic approach has a subtle flaw, which can result in some amount of confusion for the learning algorithm. In particular, the learning algorithm will observe a long sequence of calls to `suggest` and to `reward`. In the case where a single program execution corresponds to multiple games, the calls to `suggest` and `reward` do not all correspond to the same game, but rather are partitioned across multiple games. It is clear to the programmer that actions selected by adaptives in one game have no influence on the outcome or reward observed in future games. However, this is not made explicit to the learning algorithm and can result in a more difficult learning problem. In particular, the learning algorithm will spend time trying to learn how the actions choices in a previous game influence the rewards observed in the next and future games. After enough learning, the algorithm will ideally "figure out" that indeed the different games are independent, however, this may take a considerable amount of time depending on the particular problem.

In order to allow programmers to perform such multi-game training, while avoiding the potential pitfall above, we introduce the concept of an *episode* into the library. In particular, we give the programmer the ability to explicitly partition the sequence of `suggest` and `reward` calls into independent sub-sequences. Each such independent sub-sequence will be called an episode, which in the Wolf-Rabbit example corresponds to a single game. We instantiate this concept in our library by adding the following method to the `AdaptiveProcess` class.

```
public void endEpisode()
```

The programmer can then place calls to this method at the end of each episode, which make the episode boundaries explicit for the learning algorithm. Thus, the learning algorithm will now see a sequence of calls to `suggest` and `reward` between calls to `endEpisode`.

It is straightforward to exploit this episode information in the SMDP Q-learning algorithm. The only adjustment is that upon encountering a call to `endEpisode` an update based on Equation 3 is performed instead of Equation 2. Further, no update is performed when encountering the first call to `suggest` after any `endEpisode`. The effect of these changes is to avoid Q-table updates that cross episode boundaries. The formal semantics for the multi-episode setting is almost identical to the one described above. The only difference is that we define $R(P,\pi,x)$ to be the average total reward across episodes during the execution of P on input x with respect to policy π. With this change the overall optimization problem is as specified in Equation 1.

Finally, we note that the concept of multi-episode program executions can be particularly useful when combined with multi-process programs. For example, there can be cases where the natural episode boundaries of two different processes in a program do not coincide with one another. Since calls to `endEpisode` are associated with individual adaptive processes, our library can easily handle such situations. The above multi-process example involving navigation in a video game environment provides a good example of when this situation might arise. The adaptive process corresponding to the high level choice about which location to move to next may have episode boundaries corresponding to complete games. However, these boundaries are not natural for the process dedicated to the details of navigating from one location to another. Rather, the natural episode boundaries for that process correspond to the navigation sequences between the goal locations specified by the high-level process.

```
void yahtzeePlayer(AdaptiveProcess player,
                   Adaptive<GameCtx,Category> c1,
                   Adaptive<GameCtx,Category> c2,
                   GameState s1) {
 for (int i = 1; i <= 13; i++) {
     Category cat1 = c1.suggest(getCtx(s1),
                                s1.getEmpty());
     State s2 = rollFor(cat1);
     Category cat2 = c2.suggest(getCtx(s2),
                                s2.getEmpty());
     State s3 = rollFor(cat2);

     //out of rolls here
     State s4 = assignBest(s3.getEmpty());
     player.reward(s4.score - s3.score);
 }
}
```

Figure 2. An Adaptive Yahtzee Algorithm

4. Evaluation

In Section 2 we have already provided positive experimental results on our example Wolf-Rabbit program. It was shown that through the use of our library the wolves could effectively learn to capture the rabbit. In this section we describe two more substantial applications of our library to problems where encoding a full solution by hand is not trivial. The first domain is the dice game of Yahtzee, and the second domain is multi-unit tactical battles in a real-time strategy game. In each case, we write adaptive programs using our library and show that the learning mechanism is able to effectively optimize the program behavior.

4.1 Yahtzee

Yahtzee is a well-known dice game. Players roll some dice and then apply the combination of numbers given to some category such as "three of a kind". Each category can only be applied once. Additionally during a round, a subset of the dice may be re-rolled. The goal is to maximize one's score over 13 rounds.

Here we consider the problem of using our library to create a program that can achieve good performance in Yahtzee. The structure of our adaptive program is based on observing the fundamental decisions that are made in each round of play. Specifically, each round involves up to two rolls of the dice. The decision about which of the dice to re-roll can be decomposed into a decision about which category to apply and then selecting the dice to re-roll with the aim of achieving the target category. Given a particular target category, it is relatively easy for humans to select a good set of dice to re-roll and accordingly straightforward to write standard code to select that set of dice. However, the selection of the target category before each roll is a key source of uncertainty that is not straightforward to program. This motivates an adaptive program where the adaptives correspond to choices about target categories and where the selection of dice to re-roll is hand-coded by the programmer without the use of adaptive elements. Our program shown in figure 2 has two `Adaptive` objects `c1` and `c2` with identical contexts and actions. The actions selected from are simply the set of empty categories at a particular moment in the game. The context represents the die faces and the *number* of empty categories remaining. The first re-roll is executed by first obtaining the set of empty categories in the current game and then making a call to the `suggest` method of `c1` to obtain a target category. This category is then passed as an argument to our hard-coded function `rollFor` which selects dice and re-rolls them based on the category. Next, this same process is repeated, but using the c2 adaptive instead of `c1`. After the second re-roll, which means that a category must be selected for scoring, we use a simple scoring function `assignBest` to pick the empty category that will result in the highest score. Finally, after each round the adaptive process receives a reward equal to the increase in score achieved in that round. Note that with this reward function the total reward over a game is equal to the game score. Thus a program that optimizes the expected reward also optimizes the expected Yahtzee score.

We trained the program by having it play several million games over a period of approximately 30 minutes. The results in Table 1 show that the average score for our program (labeled ABP) before learning was 119, while after learning the average score was 195. This shows that the learning process results in a significant improvement in the program performance.

Table 1 gives the performance of our adaptive Yahtzee player compared to a state-of-the-art Monte-Carlo planning algorithm [8] called UCT. We see that the slow version achieves an average score of 208 compared to the 195 achieved by our adaptive program. However, UCT requires 6 orders of magnitude more time per game to achieve this result. On the other hand, UCT-fast achieves an average score of 161, which is significantly worse than our adaptive program, while still requiring several orders of magnitude more computation time. Unlike UCT our approach does require a training period to gradually improve its performance, however, this is a one-time cost that UCT will incur for each move. More important to us than the timing comparison though is the score comparison. With very little programming effort we were able to use our adaptive library to achieve competitive results compared to a state-of-the-art planning algorithm.

Program	Avg. Score	Avg. Game Time (sec)
ABP (before learning)	119	0.001
ABP (after learning)	195	0.001
UCT-fast	161	0.8
UCT-slow	208	152.0

Table 1. Performance on Yahtzee. All results are averaged over 1000 games.

We also verified that our program is learning useful information by observing the score at regular intervals during learning. Figure 3 shows the improvement in performance for our Yahtzee program as the number of learning games increases. Each point in the graph represents the average score over 1000 games played by the adaptive program in non-learning mode. Note that after only a small number of learning games, performance jumps from 119 to approximately 145. After this the performance steadily improves.

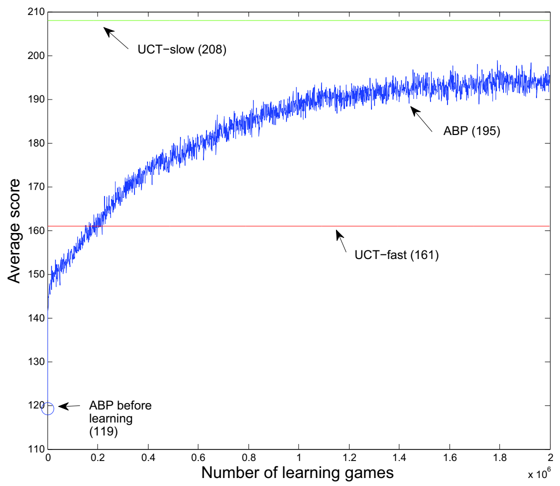

Figure 3. Performance of the Yahtzee program as the number of learning games is increased.

It is important to note that the context provided to the adaptives is not ideal. To make truly optimal decisions the contexts would need to encode the precise information about which categories are still empty. Rather our current contexts only include information about the number of empty categories (in addition to the values showing on the dice). Unfortunately, expanding the context to include exact information about which categories are open exceeds reasonable memory limitations. This is due to the fact that our current library requires the storage of Q-tables, which grow linearly in the number of contexts. There are 2^{13} possible combinations of open categories and expanding our context space by this factor was not practical. As mentioned in Section 6, this limitation can be removed using more advanced RL techniques which do not require storing Q-functions as explicit tables, but rather use more compact representations. We expect that including such techniques in future versions of the library will offer even better learning performance as more detailed contexts can then be used.

4.2 Tactical Battles in Wargus

Wargus [21] is a real-time strategy (RTS) game, which runs on the open-source RTS game engine Stratagus. RTS games involve controlling large numbers of game units in order to build an economy, a military, and to eventually overtake an opponent via military force. A major challenge for RTS game developers is developing the game AI. Unfortunately, current game AIs are relatively static and non-adaptive, which degrades their entertainment value for experienced human players. One of the reasons for this is that it is extremely difficult and time-consuming to hand-code flexible strategies for RTS games that can handle the breadth of situations that will eventually occur. Thus most approaches to game AI development centers around scripted behaviors. ABP offers an alternative paradigm for developing such strategies, allowing programmers to leave certain choices open to be optimized by the program automatically via repeated play. Here we consider the application of our library to the Wargus sub-problem of tactical battles.

Tactical battles are an important aspect of RTS games, where two groups of military units come into close contact and battle until one side is destroyed. Here we consider a tactical battle scenario in Wargus where we control 4 ground military units against 4 other ground military units controlled by the native Stratagus AI. The goal of the experiment is to write an adaptive program using our library that can learn to defeat the Stratagus AI by as large a margin as possible, where the margin of a win is measured with respect to the remaining hit points of our forces at the end of the game. For this purpose we have created a Java API to the Stratagus engine, that allows for Java programs to easily control units in the game.

Writing a good strategy even for this relatively small scenario can be challenging: Is it better to gang up on an enemy unit at any cost? Or is there some point after which we should attack the nearest enemy? How do we get our units to stay in formation? We are interested in writing an adaptive program using minimal effort that can learn to win this battle by automatically finding a good way to handle the above issues. More generally we are interested in using our library to learn a strategy that works well for any tactical battle, though that is beyond the scope of this work.

Our adaptive program must control the activities of each of our 4 military units. In this experiment we limit the actions that each unit considers to actions that attack one of the enemy units. After issuing an attack command to the Stratagus engine, the attacking unit pursues the target until coming into attack range and then continually inflicts damage on the target. Thus, the fundamental decision that must be made at each point in time is which enemy unit each of our units should attack. This is also the key source of uncertainty faced by the programmer when designing a control program for the units. With this motivation the structure of our adaptive program is as follows.

Our adaptive program contains a single adaptive process with a single adaptive that has a binary action set over the values `target` and `non-target`. The context of this adaptive will be specified later. After every 10 cycles of the Stratagus engine our program enters a decision phase where a target enemy unit is selected for each friendly unit. This decision phase contains nested loops over each of our friendly units and each enemy unit, which considers each pair of friendly and enemy units. For each pair (f,e), where f is a friendly unit and e is an enemy unit, the `suggest` method of our adaptive is called with a context that depends on (f,e). If the method returns `target` then e is set as the current target of f, otherwise the target of f remains unchanged. Thus, at the end of this phase, each friendly unit is assigned to attack the last enemy unit for which the adaptive returned `target`. A call to the `reward` method is made at the beginning of each decision phase with an argument equal to $\mathrm{HP}_e - \mathrm{HP}_f$, where HP_e and HP_f are the total hit points deducted from the enemy and friendly units respectively during the previous 10 game cycles. Thus the total reward over a single game is equal to the difference in total damage inflicted on the enemies minus the total damage inflicted on the friendly units, as desired.

It remains to specify the context of our adaptive. The context should capture useful information for making a decision about whether an enemy unit e is a good target for friendly unit f. Some of the key pieces of information relevant to this decision are: (1) the nearness of f to e, which we discretize to the values `CLOSE`, `NEAR`, `FAR`, (2) The number of other friendly units already targeting e, which can be one of four values, and (3) the health of e, which we discretize as `HEALTHY`, `MEDIUM`, `WEAK`. The context space for our adaptive is the cross-product of these three features, yielding a total of 36 possible contexts. These contexts are trivial to compute. The challenge is in selecting the best decision based on the context, which is a job we have left to the learning process.

We allowed our adaptive program to learn for 1000 repeated games of this 4 vs. 4 battle. Before training, our adaptive program achieved a health difference of -23, which indicates that the program was losing to the native Stratagus AI. By the end of training, the adaptive program learned to defeat the Stratagus AI by a hit point difference of 44, winning by a significant margin. For comparison purposes we wrote a deterministic program that implemented a simple strategy of attacking the nearest enemy unit. Initially, we expected this strategy to do reasonably well and at least win the game by a small margin based on our inspection of the map. Surprisingly, this deterministic program performs quite poorly and loses to the Stratagus AI by a margin of -22 hitpoints. This shows that the adaptive program is apparently learning a strategy that somehow trades-off proximity, the amount of "ganging up", and the enemy health. Overall these results are promising and suggest the investigation of ABP for more sophisticated scenarios and other aspects of RTS games.

5. Related Work

Our work is inspired by a variety of previous efforts in the field of reinforcement learning (RL). RL [18] is a subfield of artificial intelligence that studies algorithms for learning to control a system by interacting with the system and observing positive and negative feedback. RL is intended for situations where it is difficult to write a program that implements a high-quality controller, but where it is relatively easy to specify a feedback signal that indicates how well a controller is performing. Thus, pure RL can be viewed as an extreme form of ABP where the non-adaptive part of the program is trivial, requiring the RL mechanisms to solve the full problem from scratch. As such, successful applications of RL typically require significant expertise and experience. It is somewhat of an art to formulate a complex problem at the appropriate abstraction level so that RL will be successful.

The inherent complexity of pure RL led researchers to develop different mechanism for humans to provide natural forms of "advice" to RL systems, for example, in the form of a set of rules that specify hints about good behavior in various situations [10], or example demonstrations of good behavior by a domain expert [1]. However, these forms of advice still require an RL expert who is very familiar with the underlying algorithms for their successful application. In addition, the expressiveness of the types of advice that can be provided are quite limited, particularly in comparison with programming languages.

The desire to increase the expressiveness of advice provided to RL systems has resulted in research on hierarchical reinforcement learning [6, 13]. Here a human specifies behavioral constraints on the desired controller, or program, to be learned in the form of subtask, or sub-procedure, hierarchies. The hierarchies specify potential ways that the high-level problem can be solved by solving some number of sub-problems, and in turn how those sub-problems can potentially be broken down and so on. Not all of the possibilities specified by the hierarchies will be successful or optimal, but the space of possible controllers can be dramatically smaller than the original unconstrained problem. Given these constraints, RL algorithms are often able to solve substantially more complex problems.

Provided with enough constraints the hierarchies described above can be viewed as defining programs. This idea was made explicit under the name partial programming, where a simple language based on hierarchical state machines was developed to provide guidance to an RL agent [2]. This language was soon replaced by the development of ALISP [3], which was a direct integration of RL with LISP. The key programming construct that ALISP adds to LISP is the choice point, which is qualitatively similar to `Adaptive` objects in our library. The primary focus of work on ALISP has been to develop adaptation rules for choice points and to understand the conditions under which learning in the infinite limit will result in controllers that achieve certain notions of optimality when executed in the world/environment. A more recent proposal for an adaptive programming language is A^2BL [17], which integrates RL with the agent behavior language (ABL). The proposal for A^2BL can be viewed as an instance of ABP for a language that is specialized to behavioral-based programming of software agents. Few details concerning a concrete syntax, implementation, semantics and learning rules are currently available for A^2BL.

An important semantic distinction between the work in this paper and existing languages for partial programming in RL is that the semantics of languages such as ALISP are tightly tied to an interface to a world/environment that is external to the program. That is, an ALISP program by itself does not have a well defined semantics from a learning perspective until it is coupled with a world/environment, or more formally a Markov decision process (MDP). This requirement is most clearly reflected by the fact that the ALISP language does not include native reward statements. Rather rewards are assumed to be provided by an external MDP. A likely reason for the tight coupling to MDPs is that ALISP grew out of the area of RL where MDPs are already an assumed entity. However, this strict coupling to MDPs makes it difficult for a programmer who is not knowledgeable about RL to understand the semantics and exploit the potential power of the language.

On the other hand, the semantics provided for our ABP Java library are not tied to the notion of an external world, environment, or MDP in any way. Rather the semantics are defined completely in terms of just a program and a distribution over its inputs, which could be a constant input. In this sense, our library can be immediately applied in any context that Java programs might be written. Importantly, it is straightforward to write adaptive programs using our library that do interface to an external world, but this is not a native requirement of the library. Our work is arguably the first to develop an ABP framework for a language as widely used as Java. With the exception of ALISP, which is based on LISP, all other work on ABP that we are aware of has been in the context of non-mainstream and highly specialized languages, for example, A^2BL and PHAM [2], which greatly diminishes their potential impact. Our primary motivation was to develop a library for ABP that is highly flexible and has the potential for wide use by non-RL experts. We believe that our current library represents significant progress in identifying some of the key constructs needed to make this goal a reality in the object-oriented Java language.

There are numerous RL libraries and frameworks for Java and other object-oriented languages (e.g. Rl-Glue [20]). However, such libraries are designed for people with extensive experience in RL and their goal tends to be providing a test harness for experimenting with new algorithms. In contrast, ABP's goal is to provide a library that a non-expert can use to naturally describe adaptive constructs in their programs with.

6. Future Work

One of the immediate directions for future work is to support adaptives for which the number of possible contexts is enormous. For example, in Yahtzee we would have liked to have used a richer context for the adaptives that encoded the precise set of open categories. However, this was not practical due to the use of a table-based Q-function representation. To support such large context spaces we will implement support for compact representations of Q-functions and other related structures, which grow sub-linearly with respect to the number of contexts. This will also require developing learning algorithms that operate directly on these compact representations. The RL literature has studied a variety of such representations and learning algorithms [7], and we will initially draw on that work.

More fundamentally we believe that there is much work to be done with respect to understanding how different RL algorithms interact with different types of programming patterns involving adaptives. In particular, is it possible to analyze the structure of a program or its execution patterns in order to derive learning algorithms that learn more quickly? In addition, developing notions of equivalency-preserving transformations of adaptive programs might also be useful for automatically transforming programs to ones that are easier for learning. Some encouraging preliminary results are available in [15] where we use program analysis to automatically feed the learning algorithm better information. The resulting method achieves impressive results on a set of complex adaptive programs.

Finally, we are interested in understanding how programmers will think about and use ABP. In this direction, we plan to conduct studies where programmers use our library to write adaptive programs for problems where ABP appears beneficial. These studies will hopefully inform future library design decisions as well as direct research on the learning algorithms to better handle situations that are likely to arise with programmers.

7. Conclusion

For many problems, programmers have uncertainty about various choices to be made when solving a problem with a deterministic program. However, despite this uncertainty the programmer is still often able to specify a reward signal that indicates whether a program's run-time behavior is good or bad. The ABP paradigm is aimed at matching the programmer's knowledge in such situations, by allowing them to directly encode their uncertainty along with a reward signal. The goal then is for the resulting adaptive program to learn to optimize its decisions over repeated runs of the program.

The work in this paper has developed the first Java library for ABP and arguably the first ABP implementation for any language as widely used as Java. We provided examples of this library, specified a semantics for adaptive programs written using this library, and demonstrated its utility on non-trivial application problems. This work has set the stage for a more widespread use of the ABP paradigm by programmers that are not experts in machine learning.

Acknowledgments

This work is supported by the National Science Foundation under the grant CCF-0820286 "Adaptation-Based Programming".

References

[1] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *ICML*, pages 1–, 2004.

[2] David Andre and Stuart J. Russell. Programmable reinforcement learning agents. In *NIPS*, pages 1019–1025, 2000.

[3] David Andre and Stuart J. Russell. State abstraction for programmable reinforcement learning agents. In *AAAI/IAAI*, pages 119–125, 2002.

[4] Tim Bauer, Martin Erwig, Alan Fern, and Jervis Pinto. ABP. http://groups.engr.oregonstate.edu/abp/.

[5] Steven J. Bradtke and Michael O. Duff. Reinforcement learning methods for continuous-time markov decision problems. In *NIPS*, pages 393–400, 1994.

[6] Thomas G. Dietterich. The maxq method for hierarchical reinforcement learning. In *ICML*, pages 118–126, 1998.

[7] L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4(237-285):102–138, 1996.

[8] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, pages 282–293, 2006.

[9] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *ICML*, pages 157–163, 1994.

[10] Richard Maclin, Jude W. Shavlik, Lisa Torrey, Trevor Walker, and Edward W. Wild. Giving advice about preferred actions to reinforcement learners via knowledge-based kernel regression. In *AAAI*, pages 819–824, 2005.

[11] S. Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22(1):159–195, 1996.

[12] R.E. Parr. *Hierarchical control and learning for Markov decision processes*. PhD thesis, University of California, Berkeley, 1998.

[13] Ronald Parr and Stuart J. Russell. Reinforcement learning with hierarchies of machines. In *NIPS*, pages 1043–1049, 1997.

[14] Leonid Peshkin, Kee-Eung Kim, Nicolas Meuleau, and Leslie Pack Kaelbling. Learning to cooperate via policy search. In *UAI*, pages 489–496, 2000.

[15] Jervis Pinto, Alan Fern, Tim Bauer, and Martin Erwig. Robust learning for adaptive programs by leveraging program structure. In *ICMLA '10: Proceedings of the 2010 International Conference on Machine Learning and Applications*, Washington, DC, USA, to appear. IEEE Computer Society.

[16] Y. Shoham, R. Powers, and T. Grenager. Multi-agent reinforcement learning: a critical survey. In *AAAI Fall Symposium on Artificial Multi-Agent Learning*, 2004.

[17] Christopher Simpkins, Sooraj Bhat, Charles Lee Isbell Jr., and Michael Mateas. Towards adaptive programming: integrating reinforcement learning into a programming language. In *OOPSLA*, pages 603–614, 2008.

[18] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2000.

[19] Ming Tan. Multi-agent reinforcement learning: Independent versus cooperative agents. In *ICML*, pages 330–337, 1993.

[20] Brian Tanner and Adam White. RL-Glue : Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research*, 10:2133–2136, September 2009.

[21] The Wargus Team. Wargus. http://wargus.sourceforge.net/.

[22] C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.