Towards Adaptive Programming

Integrating Reinforcement Learning into a Programming Language

Christopher Simpkins Sooraj Bhat
Charles Isbell, Jr.
College of Computing
Georgia Institute of Technology
{simpkins,sooraj,isbell}@cc.gatech.edu

Michael Mateas
Computer Science Department
University of California, Santa Cruz
michaelm@cs.ucsc.edu

Abstract

Current programming languages and software engineering paradigms are proving insufficient for building intelligent multi-agent systems–such as interactive games and narratives—where developers are called upon to write increasingly complex behavior for agents in dynamic environments. A promising solution is to build *adaptive systems*; that is, to develop software written specifically to adapt to its environment by changing its behavior in response to what it observes in the world. In this paper we describe a new programming language, An Adaptive Behavior Language (A^2BL), that implements adaptive programming primitives to support *partial programming*, a paradigm in which a programmer need only specify the details of behavior known at code-writing time, leaving the run-time system to learn the rest. Partial programming enables programmers to more easily encode software agents that are difficult to write in existing languages that do not offer language-level support for adaptivity. We motivate the use of partial programming with an example agent coded in a cutting-edge, but non-adaptive agent programming language (ABL), and show how A^2BL can encode the same agent much more naturally.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Algorithms, Languages, Design

Keywords Adaptive Programming, Reinforcement Learning, Partial Programming, Object-Oriented Programming

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

1. Introduction

In this paper we present a language, A^2BL, that is specifically designed for writing adaptive software agents. By adaptive software we refer to the notion used in the machine learning community: software that learns to adapt to its environment during run-time, not software that is written to be easily changed by modifying the source code and recompiling. In particular, we use Peter Norvig's definition of adaptive software:

> Adaptive software uses available information about changes in its environment to improve its behavior (Norvig and Cohn 1998).

1.1 The Need for Adaptivity in Agent Software

We are particularly interested in programming intelligent agents that operate in real environments, and in virtual environments that are designed to simulate real environments. Examples of these kinds of agents include robots, and non-player characters in interactive games and dramas. Unlike traditional programs, agents operate in environments that are often incompletely perceived and constantly changing. This incompleteness of perception and dynamism in the environment creates a strong need for adaptivity. Programming this adaptivity by hand in a language that does not provide built-in support for adaptivity is very cumbersome. In this paper we will demonstrate and analyze the construction of an agent for a simple world, `Predator-Food`, in which the agent must simultaneously pursue food and avoid a predator. We will show the difficulties of programming an adaptive agent for even this simple environment using ABL, an advanced agent programming language. We will then show how A^2BL, with its built-in adaptivity and support for partial programming, makes the construction of the same agent much easier.

1.2 How to Achieve Adaptive Software

Norvig identifies several requirements of adaptive software—adaptive programming concerns, agent-oriented concerns, and software engineering concerns—and five key

technologies—dynamic programming languages, agent technology, decision theory, reinforcement learning, and probabilistic networks—needed to realize adaptive software. These requirements and technologies are embodied in his model of adaptive programming given in Table 1.

Traditional Programming	Adaptive Programming
Function/Class	Agent/Module
Input/Output	Perception/Action
Logic-based	Probability-based
Goal-based	Utility-based
Sequential, single-	Parallel, multi-
Hand-programmed	Trained (Learning)
Fidelity to designer	Perform well in environment
Pass test suite	Scientific method

Table 1. Peter Norvig's model of adaptive programming (Norvig 1998).

A^2BL integrates two of Norvig's key technologies: agent technology and reinforcement learning. We will explain how A^2BL implements Norvig's adaptive programming model and argue that A^2BL satisfies many of Norvig's requirements, with the rest slated for future development. Before we proceed, we expand on Norvig's view of the role of machine learning in general, and reinforcement learning (RL) in particular in the realization of adaptive programming, and discuss related work in integrating reinforcement learning into programming languages.

1.3 The Path to Adaptive Software: Integrating Machine Learning into a Programming Language

One of the promises of machine learning is that it allows designers to specify problems in broad strokes while allowing a machine to do further parameter fine-tuning. Typically, one thinks of building a system or agent for some specific task and then providing it some kind of feedback, allowing it to learn. In this case, the agent is the point of the exercise. A^2BL embeds this notion within a programming language itself by extending it with adaptive behaviors. The power of such a merger of machine learning and a programming language is that it allows for what has become known as *partial programming*; that is, it allows a designer to specify what he knows how to express exactly and leave the system to learn how to do the rest. In the following sections we explain how this marriage of machine learning and programming languages supports the partial programming paradigm.

1.4 The Partial Programming Paradigm: Why Current Programming Models are Ill-Suited to Building Adaptive Software

The model of computation, or "control regime," supported by a language is the fundamental semantics of language constructs that molds the way programmers think about programs. PROLOG provides a declarative semantics in which programmers express objects and constraints, and pose queries for which PROLOG can find proofs. In C, programmers manipulate a complex state machine. Functional languages such as ML and Haskell are based on Lambda Calculus. A^2BL will be multi-paradigmatic, supporting declarative semantics based on reactive planning, procedural semantics through its direct use of Java, and partial programming semantics based on reinforcement learning, in which the programmer defines the agent's actions and allows the learning system to select them based on states and rewards that come from the environment. This point is important: partial programming represents a new paradigm which results in a new way of writing programs that is much better suited to certain classes of problems, namely adaptive agents, than other programming paradigms. A^2BL facilitates adaptive agent programming in the same way that PROLOG facilitates logic programming. While it is possible to write logic programs in a procedural language, it is much more natural and efficient to write logic programs in PROLOG. The issue here is not Turing-completeness, the issue is cognitive load on the programmer. In a Turing-complete language, writing a program for any decidable problem is theoretically possible, but is often practically impossible for certain classes of problems. If this were not true then the whole enterprise of language design would have reached its end years ago.

The essential characteristic of partial programming that makes it the right paradigm for adaptive software is that it enables the separation of the "what" of agent behavior from the "how" in those cases where the "how" is either unknown or simply too cumbersome or difficult to write explicitly. Returning to our PROLOG analogy, PROLOG programmers define elements of logical arguments. The PROLOG system handles unification and backtracking search automatically, relieving the programmer from the need to think of such details. Similarly, in A^2BL the programmer defines elements of behaviors – states, actions, and rewards – and leaves the language's runtime system to handle the details of how particular combinations of these elements determine the agent's behavior in a given state. A^2BL allows an agent programmer to think at a higher level of abstraction, ignoring details that are not relevant to defining an agent's behavior. When writing an agent in A^2BL the primary task of the programmer is to define the actions that an agent can take, define whatever conditions are known to invoke certain behaviors, and define other behaviors as "adaptive," that is, to be learned by the A^2BL runtime system. As we will see in Sections 3 and 4, even compared to an advanced agent programming language, this ability to program partial behaviors relieves a great deal of burden from the programmer and greatly simplifies the task of writing adaptive agents.

1.5 Integrating Reinforcement Learning Into a Programming Language

Among the many different kinds of machine learning algorithms, reinforcement learning is particularly well-suited to the task of learning agent behavior. The goal of a reinforcement learning algorithm is to learn a *policy* – a mapping from states to actions. In other words, for a given agent, a policy concretely answers the question "given the state the agent is in, what should it do?" In Section 2 we will provide a broad overview of AI and machine learning and explain in more detail why reinforcement learning is well-suited to the task of constructing intelligent autonomous agents.

There is already a body of work in integrating reinforcement learning into programming languages, mostly from Stuart Russell and his group at UC Berkeley (Andre and Russell 2001, 2002). Their work is based on *hierarchical reinforcement learning* (Parr and Russell 1998; Dietterich 1998), which enables the use of prior knowledge by constraining the learning process with hierarchies of partially specified machines. This formulation of reinforcement learning allows a programmer to specify parts of an agent's behavior that are known and understood already while allowing the learning system to learn the remaining parts in a way that is consistent with what the programmer specified explicitly.

The notion of *programmable hierarchical abstract machines* (PHAM) (Andre and Russell 2001) was integrated into a programming language in the form of a set of Lisp macros (ALisp) (Andre and Russell 2002). Andre and Russell provided provably convergent learning algorithms for partially specified learning problems and demonstrated the expressiveness of their languages, paving the way for the development of RL-based adaptive programming. Our work builds on theirs but with a focus on practical applications.

1.6 The Path to Adaptive Software Engineering: Practical Languages for Large Agent-Based Applications

We have chosen another language, ABL (which we shall describe in some detail later), as the starting point for our adaptive programming language because ABL is designed for developing intelligent autonomous agents for significant end-user applications, namely games and interactive narratives. A^2BL serves two purposes. First, with a modular implementation of adaptive behaviors that enables the swapping of RL algorithms, A^2BL provides a platform for RL research. Second, A^2BL is the first step towards a language that supports the needs of game designers and social science modelers writing practical, large scale agent systems. It is the second purpose, the practical purpose, that distinguishes our work from previous work in RL-based adaptive programming.

2. Background

In this section, we provide the reader with some basic background knowledge in a few key concepts from Artificial Intelligence (AI). While the presentation here should suffice to understand the remainder of this paper, we provide pointers to more detailed accounts in the literature for the interested reader.

2.1 AI Planning

An intelligent agent maximizes goal attainment given available information. In knowledge-based AI, a variety of techniques are used to solve problems. Typical one-step problem-solving scenarios include board games, where an agent must decide on the best move given the current board state. Planning algorithms are used in environments where an agent must find a *sequence* of actions in order to satisfy its goals. Like most Good Old-Fashioned AI (GOFAI), classical planning algorithms rely on deterministic representations; that is, they are not designed to handle probabilistic settings where certain parts of the state space are hidden and some actions don't always result in exactly the same state change. As we will see in the next sections, machine learning addresses such partially-observable, probabilistic environments directly. For a more detailed discussion of AI in general, and planning in particular, see (Russell and Norvig 2003).

2.2 Machine Learning

Machine learning algorithms improve their performance on some task as they gain experience. Learning problems specify a task, a performance metric, and a source of training experience. It is important that the training experience provide some feedback so that the learning algorithm can improve its performance. Sometimes the feedback is explicit, as in the case of supervised learning. In supervised learning, an algorithm is presented with a set of examples of a target concept, and the algorithm's performance is judged by how well it judges new instances of the class. For example, a character recognition system can be trained by presenting it with a large number of examples of the letters of the alphabet, after which it will be able to recognize new examples of alphabetic characters. Some commonly known techniques for such tasks are neural networks, support vector machines, and k-nearest neighbor.

Such learning tasks are said to be *batch-oriented* or *offline* because the training is separate from the performance. In supervised learning, the learner – such as a neural network – is presented with examples of target concepts and its performance task is to recognize new instances of the concepts. A supervised learner learns a mapping from instance features to classes by being presented with example mappings from instances to classes. In online virtual and real environments, an agent does not have such training available. It is not presented with example mappings of states to actions. Instead, it is presented with mappings from states to

rewards, and it must learn a mapping from states to actions (which is precisely the task of a reinforcement learning algorithm). Additionally, in *online* learning an agent must perform at the same time it is learning, and the feedback here is obtained by exploration – acting in the world and succeeding or failing. As we will see in the next section, reinforcement learning algorithms represent this type of algorithm and are particularly well-suited to the construction of intelligent autonomous agents.

For a more detailed discussion of machine learning, see (Mitchell 1997).

2.3 Reinforcement Learning

One can think of reinforcement learning (RL) as a machine learning approach to planning. In RL, problems of decision-making by agents interacting with uncertain environments are usually modeled as Markov decision processes (MDPs). In the MDP framework, at each time step the agent senses the state of the environment, and chooses and executes an action from the set of actions available to it in that state. The agent's action (and perhaps other uncontrolled external events) cause a stochastic change in the state of the environment. The agent receives a (possibly zero) scalar reward from the environment. The agents goal is to find a *policy*; that is, to choose actions so as to maximize the expected sum of rewards over some time horizon. An optimal policy is a mapping from states to actions that maximizes the long-term expected reward. Many RL algorithms are guaranteed to converge to the optimal policy in the limit (as time increases), though in practice it may be advantageous to employ suboptimal yet more efficient algorithms. Such algorithms find *satisficing* policies—that is, policies that are "good enough"—similar to how real-world agents (like humans) act in the world.

Many RL algorithms have been developed for learning good approximations to an optimal policy from the agent's experience in its environment. At a high level, most algorithms use this experience to learn value functions (or Q-values) that map state-action pairs to the maximal expected sum of reward that can be achieved by starting from that state-action pair and then following the optimal policy from that point on. The learned value function is used to choose actions. In addition, many RL algorithms use some form of function approximation (parametric representations of complex value functions) both to map state-action features to their values and to map states to distributions over actions (*i.e.*, the policy).

We direct the interested reader to any introductory text on reinforcement learning. There are several such texts, including (Sutton and Barto 1998; Kaelbling et al. 1996).

2.4 Modular Reinforcement Learning

Real-world agents (and agents in interesting artificial worlds) must pursue multiple goals in parallel nearly all of the time. Thus, to make real-world partial programming feasible, we must be able to represent the multiple goals of realistic agents and have a learning system that handles them acceptably well in terms of computation time, optimality, and expressiveness. Typically, multiple-goal RL agents are modeled as collections of RL sub-agents that share an action set. Some arbitration is performed to select the sub-agent action to be performed by the agent. In contrast to hierarchical reinforcement learning, which decomposes an agent's subgoals temporally, we use a formulation of multiple-goal reinforcement learning which decomposes the agent's subgoals *concurrently*. This concurrent decompositional formulation of multiple-goal reinforcement learning, called modular reinforcement learning (MRL), is better suited to modeling the multiple concurrent goals that must be pursued by realistic agents. A more in-depth overview of modular reinforcement learning is available in (Sprague & Ballard 2003).

3. A Behavior Language (ABL)

ABL represents the cutting edge of implemented agent modeling languages (Mateas and Stern 2004). ABL is a reactive planning language with Java-like syntax based on the Oz Project believable agent language Hap (Loyall and Bates 1991). It has been used to build actual live interactive games and dramas, such as Facade (Mateas and Stern 2003). In Facade, developed by Andrew Stern and Michael Mateas, the player is asked to deal with a relationship between an arguing couple. It is a single act drama where the player must negotiate her way through a minefield of personal interactions with two characters who happen to be celebrating their ten-year marriage.

An ABL agent consists of a library of sequential and parallel behaviors with reactive annotations. Each behavior consists of a set of steps to be executed either sequentially or in parallel. There are four basic step types: acts, subgoals, mental acts and waits. Act steps perform an action in the world; subgoal steps establish goals that must be accomplished in order to accomplish the enclosing behavior; mental acts perform bits of pure computation, such as mathematical computations or modifications to working memory; and wait steps can be combined with continually-monitored tests to produce behaviors that wait for a specific condition to be true before continuing or completing.

The agent dynamically selects behaviors to accomplish specific goals and attempts to instantiate alternate behaviors to accomplish a subgoal whenever a behavior fails. The current execution state of the agent is captured by the active behavior tree (ABT) and working memory. Working memory contains any information the agent needs to monitor, organized as a collection of working memory elements (WMEs). There are several one-shot and continually-monitored tests available for annotating a behavior specification. For instance, preconditions can be written to define states of the world in which a behavior is applicable. These tests use pattern matching semantics over working memory familiar

from production rule languages; we will refer to them as *WME tests*.

In the remainder of this paper, we will discuss the development of agents in ABL, point out the issues with writing agents in ABL, and show how A^{2}BL addresses these issues. We will then implement the same agent using A^{2}BL to show the benefits to the programmer of integrating true adaptivity into the programming language itself. We conclude with a discussion of the state of A^{2}BL development and some research issues to be addressed in its future development.

3.1 The `Predator-Food` World

To provide a concrete grounding for our discussion, we will analyze two different implementations of an agent for the `Predator-Food` world. The `Predator-Food` world is a grid where there are two main activities: avoiding the predator and finding food. At every time step, the agent must pick a direction to move. Food appears randomly at fixed locations, and there is a predator in the environment who moves towards the agent once every other time step.

3.2 The `Predator-Food` Agent as a Reactive Planning Problem

Recall from Section 2 that a plan is a sequence of actions that accomplishes a goal. In the `Predator-Food` world, an agent has two goals: finding food and avoiding the predator. Accomplishing each of these goals requires a sequence of actions. In a reactive planning agent, the sequence of actions is determined in *reaction* to percepts from the environment. For example, if the food is sensed in a certain direction, the agent reacts by planning movements in that direction. Note that there may be many plans that accomplish a goal, and in a dynamic environment, constant replanning may be needed. The reactive planning approach naturally replans in response to such changes. In the next section we show how to code a reactive planning agent for the `Predator-Food` world in ABL.

3.3 A `Predator-Food` Agent in ABL

Below we present ABL code for a reactive planning agent that operates in the `Predator-Food` world.

Lines 1–6 of Figure 1 define an agent and its principal behavior, `LiveLongProsper`. `LiveLongProsper` is defined as a `parallel behavior` to reflect the fact that both of its subgoals must be pursued in parallel in order for the enclosing behavior to succeed.

Lines 9–14 define the `FindFood` subgoal as a `sequential behavior`. Each of the subgoals—`MoveNorthForFood`, `MoveSouthForFood`, `MoveEastForFood`, and `MoveWestForFood`—must be performed in a particular sequence if the agent is to succeed in finding food. Note that, because some subgoals will not be selected for execution in any given time step, the subgoals must be annotated with `ignore_failure` to prevent the enclosing behavior from failing. The agent will only move in one direction in each time step, so three of the subgoals will fail because their preconditions will not be satisfied.

```
1  behaving_entity FurryCreature
2  {
3    parallel behavior LiveLongProsper() {
4      subgoal FindFood();
5      subgoal AvoidPredator();
6    }
7
8    // subgoal 1
9    sequential behavior FindFood() {
10     with (ignore_failure) subgoal MoveNorthForFood();
11     with (ignore_failure) subgoal MoveSouthForFood();
12     with (ignore_failure) subgoal MoveEastForFood();
13     with (ignore_failure) subgoal MoveWestForFood();
14   }
15
16   // subgoal 2
17   sequential behavior AvoidPredator() {
18     with (ignore_failure) subgoal
           MoveNorthAwayFromPredator();
19     with (ignore_failure) subgoal
           MoveSouthAwayFromPredator();
20     with (ignore_failure) subgoal
           MoveEastAwayFromPredator();
21     with (ignore_failure) subgoal
           MoveWestAwayFromPredator();
22   }
23
24   sequential behavior MoveNorthForFood() {
25     precondition {
26       (FoodWME x::foodX y::foodY)
27       (SelfWME x::myX y::myY)
28       ((foodY - myY) > 0) // The food is north of me
29     }
30
31     // Code for moving agent to the north elided
32   }
33
34   // ...
35
36   sequential behavior MoveNorthAwayFromPredator() {
37     precondition {
38       (PredatorWME x::predX y::predY)
39       (SelfWME x::myX y::myY)
40       (moveNorthIsFarther(myX,myY,predX,predY))
41     }
42
43     // Code for moving agent to the north elided
44   }
45 }
```

Figure 1. An ABL agent for the `Predator-Food` world.

Lines 24–32 define `MoveNorthForFood`. The `precondition` block defined at the beginning of the behavior defines the circumstances under which ABL's run-time planning system may select this behavior for execution, that is, the agent may *react* to this set of preconditions by selecting this behavior. Line 26 assigns the `x` property of the `FoodWME` to the local variable `foodX`, and the `y` property of the `FoodWME` to the local variable `foodY`. These local variables are then used in the boolean condition `((foodY - myY) > 0)` to define the precondition, which states that if the food is north of the agent's position, the agent should move north. A WME is a global variable defined by the environment which represents a thing that an agent can perceive. An agent perceives a particular aspect of the environment by inspecting its working memory for the appropriate WME.

Thus, if an agent has sensed the food, it will have a `FoodWME` that reports the position of the food.

The precondition for `MoveNorthForFood` defines the desirability of moving north in search of food, but ignores the predator. We define the behavior of moving north away from the predator in lines 36–44. As in the `MoveNorthForFood` behavior, the conditions under which `MoveNorthAwayFromPredator` may be selected for execution are defined in a `precondition` block. Note that we have factored the code for computing whether the precondition has been met into a utility function, `moveNorthIsFarther`. Similar subgoal behavior would be defined for each direction of movement, and for each reason for such movement. The full code (with details elided) is given in Figure 1.

While ABL's reactive-planning paradigm and declarative system make it possible to define complex autonomous agents, there are several problems. First, each subgoal behavior assumes that the position of both the food and the predator are known. Second, if there is a conflict between subgoals, the programmer must write code to resolve this conflict. For example, what should the agent do if the FindFood subgoal wants to move north to get to the food, but the AvoidPredator subgoal wants to move south to get away from the predator?

The biggest problem with this ABL agent is that low-level agent actions (movement) and the reasons for selecting those actions are coupled. Because of this coupling, movement behaviors must be duplicated for each possible reason the movement might be executed. Thus, moving north for food and moving north to avoid the predator must be represented separately and the preconditions for each carefully specified. While the movement action itself could be factored into a separate function called by each behavior, there is still a considerable cognitive burden on the programmer who must consider each combination of agent action and reason for action. Note that any programming language that does not provide a means for separating the concerns of what must be done and how it is to be accomplished will impose a similar cognitive burden on agent programmers.

Another problem with the ABL version of the Predator-Food agent is that the programmer must fully specify the agent's behavior. If there is a part of the agent's behavior that the programmer does not know, he must implement his best guess. This becomes difficult in the typically ill-specified and dynamic environments where we would want to deploy intelligent agents, such as massively multi-player games.

As we will see in the next sections, integrating adaptivity into the programming language not only reduces the amount of code required to implement an agent, but more importantly allows the programmer to think about *what* the agent's goals are and leave the agent to figure out *how* to achieve them. This separation of concerns is enabled by partial programming, in which the programmer need only specify what he knows, leaving the run-time system to figure out the rest.

4. An Adaptive Behavior Language (A^2BL)

Our solution to the problems described in the previous section is to provide built-in language support for adaptivity. In A^2BL, adaptivity is achieved by integrating reinforcement learning directly into the language. In the following sections we show how to model a `Predator-Food` agent as a reinforcement learning problem, how this model maps to adaptive behaviors, and finally how to implement an adaptive `Predator-Food` agent in A^2BL.

4.1 The `Predator-Food` Agent as a Reinforcement Learning Problem

In reinforcement learning, agents and the worlds in which they operate are modeled by states, actions, and rewards. Goals are represented implicitly by rewards. Each state in the world provides an agent with a scalar reward – positive or negative – that precisely specifies the desirability of being in that state. In the `Predator-Food` world, meeting the predator carries a large negative reward, finding the food carries a large positive reward, and other states carry zero reward. The job of a reinforcement learning agent is to maximize long-term reward by moving to states that carry higher rewards. In each state an agent has a set of available actions that take the agent to another state. A reinforcement learning algorithm explores the state space (finding where the higher rewards lie) to learn a policy, that is, a function that maps states to actions. The sequence of actions specified by a policy is much like a plan, except that the policy is *learned* automatically rather than deduced by analyzing the preconditions and postconditions of the available actions. Specifying the rewards given by each state is far less cumbersome and error-prone than specifying pre- and post-conditions for each action.

4.2 The `Predator-Food` Agent in A^2BL: Mapping a Reinforcement Learning Problem to Language Constructs

A^2BL provides language constructs to model reinforcement learning agents without having to think about the details of reinforcement learning. When a behavior is marked as `adaptive`, A^2BL employs a reinforcement algorithm "under the hood" to determine how to select the actions within the adaptive behavior. In a `Predator-Food` agent, for example, marking the `FindFood` behavior as `adaptive` tells A^2BL's runtime system to learn how to employ the actions specified within the behavior. No hand-coding of preconditions is necessary. Within adaptive behaviors, `reward` and `state` constructs provide the reinforcement learning algorithm with the information it needs to perform its learning task. For example, the `FindFood` behavior would have a `reward` construct that defines a large positive reward for finding food. A `state` construct within the behavior would specify how to map percepts from the environment (modeled by WMEs) to objects that can be used in computa-

tions, such as grid coordinates. These constructs will be explained in more detail in the next section, which presents a `Predator-Food` agent coded in A^{2}BL.

The value of adaptive behaviors is that it enables *partial programming*. An adaptive behavior models part of the solution to a problem, namely, the actions available to reach a particular goal. The rest of the solution – which of the actions to select and the order in which to select them – are learned by the run-time reinforcement learning system. Note that the programmer specifies a reinforcement learning *problem* using A^{2}BL's adaptive language constructs, but does not deal directly with the reinforcement learning algorithms used internally by the A^{2}BL run-time system.

4.3 The `Predator-Food` Agent In A^{2}BL

In Section 3.3 we showed a Predator-Food agent coded in ABL. The ABL code for this agent had to deal with many low-level issues of action selection, essentially hand-coding a policy. In this section we show that, with adaptivity built into the language, it is possible for the programmer to think at a much higher level, reducing the cognitive burden significantly. Using the state, reward, and action model of reinforcement learning, the programmer can simply say "these are the agent's goals (in terms of rewards), and these are the actions available to achieve these goals." The reinforcement learning system learns the states under which given actions should be selected.

The full code (minus irrelevant details of movement implementation) is given in Figure 2. The first difference between the ABL agent and the A^{2}BL agent is that the principal enclosing behavior, `LiveLongProsper` is defined as an `adaptive collection behavior`. This tells the A^{2}BL run-time system to treat the enclosed adaptive behaviors as sub-agents in the MRL framework. Each sub-agent behavior then defines a set of relevant actions (designated using the `subgoal` annotation inherited from ABL), and the action set of the agent as a whole is the union of all sub-agent action sets. Note that each sub-agent contains exactly the same actions. There is no need to define different action subgoals and the conditions under which they are selected – the learning algorithms built into A^{2}BL automatically handle these tasks.

4.3.1 The `adaptive` Keyword

The most notable addition in A^{2}BL is the `adaptive` keyword, used as a modifier for behaviors. When modifying a sequential behavior, `adaptive` signifies that, instead of pursuing the steps in sequential order, the behavior should learn a policy for which step to pursue, as a function of the state of the world. Consider lines 9–22 of Figure 2; the `adaptive` modifier on this behavior tells the A^{2}BL run-time system to learn how to sequence the subgoals specified within the behavior as it interacts in the environment. The programmer codes a partial specification of the problem—the subgoals—and the system learns the rest, namely, how to sequence them optimally in a dynamic environment. Note that an `adaptive` *`sequential`* behavior will be handled by A^{2}BL with a single reinforcement learning algorithm, whereas an `adaptive` *`collection`* behavior specifies a set of behaviors, each of which is handled by a reinforcement learning algorithm (see Section 4.3.5) and whose outputs are combined by an arbitrator function that ultimately decides the agent's action in a particular state. We discuss arbitration functions in Section 4.3.6.

```
1  behaving_entity FurryCreature
2  {
3    adaptive collection behavior LiveLongProsper() {
4      subgoal FindFood();
5      subgoal AvoidPredator();
6    }
7
8    // subgoal 1
9    adaptive sequential behavior FindFood() {
10     reward {
11       100 if { (FoodWME) }
12     }
13     state {
14       (FoodWME x::foodX y::foodY)
15       (SelfWME x::myX y::myY)
16       return (myX,myY,foodX,foodY);
17     }
18     subgoal MoveNorth();
19     subgoal MoveSouth();
20     subgoal MoveEast();
21     subgoal MoveWest();
22   }
23
24   // subgoal 2
25   adaptive sequential behavior AvoidPredator() {
26     reward {
27       -10 if { (PredatorWME) }
28     }
29     state {
30       (PredatorWME x::predX y::predY)
31       (SelfWME x::myX y::myY)
32       return (myX,myY,predX,predY);
33     }
34     subgoal MoveNorth();
35     subgoal MoveSouth();
36     subgoal MoveEast();
37     subgoal MoveWest();
38   }
39
40   // ...
41 }
```

Figure 2. An A^{2}BL agent for the `Predator-Food` world.

4.3.2 The `state` Construct

As there could be a large amount of information in working memory (which is the agent's perception of the state of the world), we have introduced a `state` construct to allow the programmer to specify which parts of working memory the behavior should pay attention to in order to learn an effective policy. This allows for human-authored *state abstraction*, a fundamental concept in reinforcement learning. In this example, we specify the state as:

```
state {
   (FoodWME x::foodX y::foodY)
   (SelfWME x::myX y::myY)
   return (myX,myY,foodX,foodY);
}
```

This tells the A^{2}BL runtime system what comprises the state to be used in its RL algorithms for this particular behavior or task. The policy learned for food-finding will be predicated on this state. Note that the state contains no elements that are not needed for reasoning about finding food. This is an essential feature of modular behaviors, allowing them to be coded in a truly modular fashion.

4.3.3 The `success_condition` Condition

In ABL, a behavior normally succeeds when all its steps succeed. Because it is unknown which steps the policy will ultimately execute, adaptive behaviors introduce a new continually-monitored condition, the `success_condition`, which indicates that the goal of the behavior has been met. When the success condition becomes true, the behavior immediately succeeds. In our example agent, there is no such end-state goal. The agent must continually find food and avoid the predator.

4.3.4 The `reward` Construct

To learn a policy at all, the behavior needs a reinforcement signal. With the `reward` construct, authors specify a function that maps world states to reinforcement signals. Defining the reward that the environment gives to an agent in a given state is a straightforward way inject domain knowledge into an agent. Defining the rewards in this manner reduces the need to define complex preconditions in behaviors, which makes it possible for a domain expert who is not a programmer to participate directly in the construction of A^{2}BL agents. In natural analogy to existing ABL constructs, these new constructs make use of WME tests for reasoning and computing over working memory. Consider the following code:

```
reward {
  100 if { (FoodWME) }
}
```

The code above says that, if the agent finds the food, it gets a large positive reward (recall that WMEs are the mechanism by which an agent senses the world in ABL and in A^{2}BL). This reward is used by the RL algorithms to learn an action selection policy that maximizes long-term reward. Note that the numbers used for rewards only need to be internally consistent for a given task. For example, for the FindFood task, the programmer only need specify the relative desirability of finding food compared to not finding food (here implicitly zero). We could have written this reward as 10 or 1000. What matters is that it is relatively better than not finding food. With modular reinforcement learning (MRL), the rewards for each task are defined completely separately, and the arbitration function combines the relative preferences of each sub-agent (e.g., FindFood and AvoidPredator) to determine the agent's behavior. So we could define the rewards for FindFood on a 10 point scale and the rewards for AvoidPredator on a 100 point scale and the arbitrator would still "do the right thing" when determining the agent's behavior. This modularity allows different behaviors to be developed independently and combined in agents in various ways, greatly facilitating the engineering of large agent systems by multi-programmer teams.

4.3.5 `collection` Behaviors

An adaptive collection behavior is specifically designed for modeling the concurrency of MRL. This type of behavior contains within it several adaptive sequential behaviors, which correspond to the sub-agents in the MRL framework. Consider the following code:

```
adaptive collection behavior LiveLongProsper() {
  subgoal FindFood();
  subgoal AvoidPredator();
}
```

This code defines the `LiveLongProsper` behavior as consisting of two concurrent subgoals – `FindFood` and `AvoidPredator`. A^{2}BL will attempt to pursue both of the goals concurrently while the agent is running.

4.3.6 Arbitration: Resolving Conflicts Between Subgoals

The exact manner in which arbitration functions will be defined by the programmer is an active area of research, depending partly on parallel work we are doing in modular reinforcement learning. Here we discuss some of the possibilities from the perspective of the agent programmer.

Once we have defined the two adaptive subgoals, we need to define an arbitration function on the enclosing goal, `LiveLongProsper`. In previous work, we showed that it is impossible to construct an ideal arbitration function automatically (Bhat et al. 2006), so we cannot employ the compiler to generate an all-purpose arbitration rule.[1] Instead, the programmer must define an arbitration function, either hand-authored or learned.

A hand-authored arbitration function encodes the tradeoffs the programmer believes to be true about the utilities of the subgoals. In this example, we may decide that the benefit of finding food equals the cost of running into a predator; given our reward signals, the arbitrator would select the action maximizing $\frac{1}{10}Q_1(s,a) + Q_2(s,a)$ (recall from Figure 2 that the reward for finding food is 100 and the reward for meeting the predator is -10). Alternatively, the hand-authored arbitration function could be independent of the sub-agent Q-values; to simply avoid starvation, for instance, one might consider round-robin scheduling.

Finally, we could try posing `LiveLongProsper`'s arbitration task as another reinforcement learning problem, with its own reward function encapsulating a notion of goodness

[1] Briefly, arbitration in MRL, as it has been typically defined, can be shown to be equivalent to finding an optimal social choice function and thus falls prey to Arrow's Impossibility Result. One can avoid this impossibility by having the programmer explicitly define the tradeoffs, essentially repealing the non-dictator property of a "fair" voting system.

for living well, as opposed to one that only makes sense for finding food or avoiding a predator. For example, the reward function might provide positive feedback for having more offspring; this would be an "evolutionary" notion of reward.

The reader may wonder why `FindFood` and `AvoidPredator` should have their own reward signals if one is available for `LiveLongProsper`. The reasons should be familiar: modularity and speed of learning. The reward signal for `FindFood`, for instance, is specifically tailored for the task of finding food, so the learning should converge more quickly than learning via an "indirect" global reward signal. Further, with the right state features, the behavior should be reusable in different contexts. Specifying a reward signal for each behavior allows the reward signals to embody what each behavior truly cares about: `FindFood` cares about finding grid squares with food, `AvoidPredator` cares about avoiding the predator, and `LiveLongProsper` cares about ensuring the future of the species.

4.4 A^{2}BL as a Model of Adaptive Programming

In the introduction, we listed the elements of Peter Norvig's model of adaptive programming (Norvig 1998). Here we discuss A^{2}BL's implementation of this model.

4.4.1 Functions and Classes versus Agents and Modules

A^{2}BL inherits the agent-orientation of ABL. The fundamental units of abstraction are agents and behaviors, where an agent is essentially a collection of behaviors. One could think of agents as analogous to classes/objects and behaviors as analogous to functions, but the analogy quickly breaks down. First, agents cannot be composed of other agents the way objects can be composed of other objects. Second, functions are called directly in a procedural fashion; behaviors are specified declaratively and selected for execution by ABL's runtime planning system only if and when those behaviors are needed to pursue some goal. ABL's declarative reactive planning paradigm, and A^{2}BL's adaptive model provide much better support for a style of programming that separates the *what* of agent behavior from the *how*.

4.4.2 Input/Output versus Perception/Action

In traditional programming, even to a large extent in event-driven object-oriented programming, programs are written and reasoned about in terms of input/output behavior. A function is given some input and produces some output. A class is given responsibility for some part of the application's data, responds to particular messages, and provides particular responses. In agent-oriented programming, on the other hand, the agent programmer thinks in terms of what an agent can perceive in the world, and what actions the agent can execute to modify the state of the world. In ABL and A^{2}BL, perception is modeled by WMEs that represent the agent's awareness of the world in which it is situated. Actions are procedural calls within behaviors that effect changes in whatever world the agent is operating in. The WMEs (perceptions) and actions constitute an API between agents and worlds, effectively decoupling agents from worlds.

4.4.3 Logic-based versus Probability-based

In traditional programming, selection logic (boolean tests and if/then constructs) is an important part of any non-trivial program. To a large extent, this is true even in ABL, where behaviors are selected based on logical preconditions. By integrating RL, A^{2}BL incorporates probabilistic reasoning into the core of the language: RL algorithms build probabilistic models of the world and of agent optimal behavior in that world. In this way, A^{2}BL provides explicit support for probabilistic reasoning without the programmer having to think explicitly about stochasticity.

4.4.4 Goal-based versus Utility-based

Goal attainment is a fundamental metaphor in ABL, and in agent programming in general. In A^{2}BL, goal attainment is represented explicitly in terms of rewards, or utilities. Every state in the world has an associated utility (often implicitly zero), and A^{2}BL's adaptive features seek to maximize the agent's utility automatically.

4.4.5 Sequential, single- versus Parallel, multi-

A^{2}BL inherits ABL's parallelism and extends it to support concurrent modular reinforcement learning.

4.4.6 Hand-programmed versus Trained (Learning)

With A^{2}BL's support for partial programming, the programmer can ignore low-level behavior that is either too poorly specified or too dynamic to encode explicitly and leave A^{2}BL's run-time learning system to learn the details.

4.4.7 Fidelity to designer versus Perform well in environment

In traditional software engineering, a program is good if it conforms to its specification. In adaptive partial programming, a program is good if it performs well in whatever environment it finds itself in. With A^{2}BL's explicit support for reward and state specification, and its automatic learning of policies, A^{2}BL agents are written to perform well in their environments even when design specifications are vague.

4.4.8 Pass test suite versus Scientific method

Closely related to the previous point, test suites are written to test a program's conformance to design specifications; however, a certain amount of experimentation is often necessary to determine just what exactly is the right thing to do in given situations. Yet there is always some imperative to act given whatever information you have at the moment. As a technical matter, reinforcement learning makes explicit this tradeoff between the exploration of environments and the exploitation of already gained knowledge. A^{2}BL inherits this

principled approach to the exploration/exploitation tradeoff by using RL to implement adaptivity. In a sense, RL algorithms learn by experimentation.

5. Research Issues and Future Directions

Currently, we have implemented an ANTLR-based parser for A^{2}BL, and we have tested several reinforcement learning algorithms for use in A^{2}BL agents. In particular, we have tested Q-Learning and Sarsa algorithms for single-goal agents and are working to design a general arbitration algorithm, that is, to develop the theory of modular reinforcement learning. Current reinforcement learning algorithms work acceptably well on individual goals, like FindFood or AvoidPredator, but we have not yet successfully implemented an acceptable arbitration mechanism, which is a major focus of ongoing work. Aside from designing an arbitration algorithm, the major remaining tasks in implementing A^{2}BL—and by far the major portion of the work—are to integrate the reinforcement learning algorithms with the A^{2}BL run-time system and add to the code generation phase of the compiler the logic necessary to place calls to the run-time learning routines at the appropriate places in the generated code.

Many challenging and important issues need to be addressed to realize our vision for A^{2}BL. These issues range from foundational RL theory to pragmatic software engineering considerations. We discuss some of these below.

5.1 Adaptive Software Engineering

Ultimately, an agent is a kind of computer program running in a run-time environment. Whatever language features A^{2}BL supports, computer programs will need to be written and debugged. Given the complexity of individual agents and our desire to support real world-scale multi-agent system modeling, the task of writing A^{2}BL agents and multi-agent systems is likely to be a significant effort, akin to that of a large software engineering project. We will therefore need to address many of the same issues as traditional software engineering:

- Are there effective visual metaphors for agent behavior that would enable the effective use of a visual programming environment for A^{2}BL?
- What does it mean to "debug" an intelligent agent or multi-agent system?
- Can some of the mechanisms for structuring large software systems, such as objects and modules, be transferred effectively to an agent-authoring domain? What new kinds of structuring mechanisms need to be invented?
- Can the A^{2}BL language, compiler, and run-time environment be designed in such a way that the agent author need not be concerned with efficiency or optimization? If not, are we resigned to requiring expert programmers to author intelligent agents?

5.2 OOP in A^{2}BL

ABL does not currently support inheritance. It seems natural to model agents with an inheritance hierarchy similar to OO modeling in modern software engineering; however, supporting inheritance in agents may not be as simple as borrowing the body of existing theory from OOP. Agents are more than objects, and their behavior is stochastic. What would it mean for an agent to be a subtype of another agent? Would we call this an "is-a" relationship? Would we ascribe all of the semantics that OOP ascribes to "is-a" relationships? In particular, how do we model preconditions and postconditions in a stochastic agent? Because type inheritance, or some related form of reuse, seems useful for supporting large-scale, real-world agent programming, it is worthwhile to develop the theory necessary to implement an inheritance mechanism that (1) supports the design of large systems of agents and (2) supports reuse mechanisms for A^{2}BL.

5.3 Usability

Because "behavior" is a part of the ABL acronym, one might believe that ABL is designed for experts in human behavior, such as psychologists or sociologists. While ABL can support the needs of such designers, ABL is a complex language that exposes many technical details to agent authors, making it suitable mainly for programming experts. So far, mainly senior undergraduate and graduate students in computer science have been productive with ABL. Given that we envision A^{2}BL as a tool for non-programming experts, and A^{2}BL is based on ABL, we must consider several important questions:

- What kinds of abstractions and language features are required by behavior experts such as psychologists to effectively encode their domain knowledge in A^{2}BL?
- Can such non-programmer-oriented language features subsume the advanced features that lead to ABL's complexity without losing the power they bring to ABL?
- Noting Alan Perlis's epigram—"a programming language is low level when its programs require attention to the irrelevant"—what is irrelevant when modeling intelligent agents?
- Is it desirable to have both programmer-oriented, and domain expert-oriented language features in A^{2}BL so that an agent author can choose to "get down and dirty" sometimes and maintain a higher level of abstraction at other times?
- Is it realistic to expect psychologists or sociologists to adopt a form of computer programming as a basic part of their methodological tool kit? How should we go about making that happen?

6. Conclusions

In this paper we have presented A^{2}BL, a language that integrates reinforcement learning into a programming language. We have argued that it implements many of the features necessary for partial programming while specifically using programming features that have proven useful for designing large adaptive software agents.

We believe that while there is a great deal of work to do in proving convergence and correctness of various machine learning algorithms in the challenging environments we envision, this is in some sense a straightforward exercise. The more difficult task is to understand how one would build useful development and testing environments, and to understand the software engineering principles that apply for scalable partial programming.

7. Acknowledgments

We are grateful for the generous support of DARPA under contract number HR0011-07-1-0028, and NSF under contract numbers IIS-0644206 and IIS-0749316.

References

David Andre and Stuart Russell. Programmable reinforcement learning agents. In *Advances in Neural Information Processing Systems*, volume 13, 2001. URL citeseer.ist.psu.edu/article/andre00programmable.html.

David Andre and Stuart Russell. State abstraction for programmable reinforcement learning agents. In *AAAI-02*, Edmonton, Alberta, 2002. AAAI Press.

Sooraj Bhat, Charles Isbell, and Michael Mateas. On the difficulty of modular reinforcement learning for real-world partial programming. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-06)*, Boston, MA, USA, July 2006.

Thomas G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proc. 15th International Conf. on Machine Learning*, pages 118–126. Morgan Kaufmann, San Francisco, CA, 1998. URL citeseer.ist.psu.edu/dietterich98maxq.html.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. URL citeseer.ist.psu.edu/kaelbling96reinforcement.html.

A. B. Loyall and J. Bates. Hap: A reactive adaptive architecture for agents. Technical Report CMU-CS-91-147, 1991. URL citeseer.ist.psu.edu/loyall91hap.html.

Michael Mateas and Andrew Stern. Facade: An experiment in building a fully-realized interactive drama. In *Game Developers Conference: Game Design Track*, San Jose, CA, March 2003.

Michael Mateas and Andrew Stern. *Life-like Characters. Tools, Affective Functions and Applications*, chapter A Behavior Language: Joint Action and Behavioral Idioms. Springer, 2004. URL http://www.interactivestory.net/papers/MateasSternLifelikeBook04.pdf.

Tom Mitchell. *Machine Learning*. McGraw-Hill, 1997.

Peter Norvig. Decision theory: The language of adaptive agent software. Presentation, March 1998. URL http://www.norvig.com/adaptive/index.htm.

Peter Norvig and David Cohn. Adaptive software, 1998. URL http://norvig.com/adapaper-pcai.html.

Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998. URL citeseer.ist.psu.edu/parr97reinforcement.html.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prenticce Hall, Upper Saddle River, NJ, 2003.

R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998. URL citeseer.ist.psu.edu/sutton98reinforcement.html.

Sprague, N., and Ballard, D. 2003. Multiple-Goal Reinforcement Learning with Modular Sarsa(0). In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*. Workshop paper.