INTEGRATING REINFORCEMENT LEARNING INTO A PROGRAMMING LANGUAGE

A Dissertation
Presented to
The Academic Faculty

By

Christopher L. Simpkins

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Interactive Computing

Georgia Institute of Technology

August 2017

INTEGRATING REINFORCEMENT LEARNING INTO A PROGRAMMING LANGUAGE

Approved by:

Professor Charles L. Isbell, Jr., Advisor
College of Computing
Georgia Institute of Technology

Dr. Douglas Bodner
Tennenbaum Institute
Georgia Institute of Technology

Professor Mark Riedl
School of Interactive Computing
Georgia Institute of Technology

Dr. Spencer Rugaber
School of Computer Science
Georgia Institute of Technology

Professor Andrea Thomaz
Electrical and Computer Engineering
University of Texas at Austin

Date Approved: May 25, 2017

A language that doesn't change the way you think about programming isn't worth knowing.

Alan Perlis

To my children, Isaac and Meredith, who motivated my decision to give up flying and enter academia.

ACKNOWLEDGEMENTS

First I must thank my parents. My mother was there for me every day. My father worked multiple jobs to support his family and taught me to work hard for everything I received. Without their love and support I would not have taken this path.

I started my Ph.D. journey over a decade ago while working at Georgia Tech Research Institute. Ashwin Ram, my first advisor, guided me through the Ph.D. application and gave me some of my earliest research opportunities. Sheila Isbell connected me to the right lab for me at GTRI, ITTL (now ICL). Terry Hilderbrand and Margaret Loper helped me access GTRI's Ph.D. support, and supported me as I balanced my GTRI duties with doctoral studies.

Charles Isbell accepted me as his student just when I needed him. Becoming Charles's student and joining his lab was invigorating. Just working near, conversing with, and meeting with Michael Holmes, Chip Mappus, David Roberts, Peng Zang, Sooraj Bhat, Luis Carlos "Luisca" Cobo Rus, Liam Mac Dermed, Arya Irani, Mark Nelson, Jon Scholz, Ryan Curtin, Kaushik Subramanium, Pushkar Kolhe, Ashley Edwards and Himanshu Sahni did a great deal to shape my development as a researcher. And they were as warm as they were brilliant. My children have as many fond memories of my lab mates as I do, from Nerf fights to cookouts to chess matches to "funking out" Charles's office when he was awarded tenure.

My work builds on the earlier work of my lab mate and friend, Sooraj Bhat. Sooraj helped me get started in language-integrated reinforcement learning and shared my interest in advanced programming languages.

As a graduate student I was inspired by Ashok Goel's Artificial Intelligence and Cognitive Science classes, Charles's Machine Learning class, and Spencer Rugaber's Programming Language Design class.

I am grateful that Mark Riedl and Charles invited me to join their funded project on computational interactive narratives. Working with them was a joy. After that project ended, Charles and my current boss, Bill Leahy, gave me the opportunity to become a lecturer. I am grateful to Bill for welcoming me into a group of outstanding teaching professionals that will likely be my home for the rest of my professional life.

I did most of my writing during Fall 2016. My CS 1331 Head TA, Taylor Hartman, took on a great deal of responsibility to allow me more time to write. CS 1331 could not have functioned so well without her leadership and the groundwork laid by past Head TAs like Keith Cartledge, Thomas Shields, Aaron Friesen, and Stefano Fenu. My CS 2316 Head TA, Kate Unsworth also took on additional tasks during a semester in which I was the new CS 2316 instructor following Jay Summet's departure. Joshua Diaddigo and Keith Cartledge wrote the web site and IntelliJ plug-in used in my programmer study and joined me in several late-night hacking sessions to iron out the kinks and dry-run the programmer study. Bob Waters covered several CS 2340 lectures at crucial times that allowed me to make progress writing. Without their help I could not have finished.

My committee has been outstanding. Spencer guided me with great care through the software engineering portions of my thesis. Doug helped me learn agent-based simulation and apply an early version of AFABL to behavior modeling. An expert in AI for games and interactive narratives, Mark helped me place AFABL in context. Andrea Thomaz encouraged me to conduct the programmer study, which distinguishes my work from other work in reinforcement learning in programming languages.

I am especially grateful to my advisor, Charles Isbell, for supporting me through years of personal struggle. Everyone knows that Charles is a brilliant researcher and leader. I know first hand that he is also an outstanding human being. No matter how difficult the challenges or how likely it seemed that I would fail, Charles always told me he believed I could finish. I am happy to have proved him right. I could not have finished without Charles's unwavering support and encouragement.

I met John Cortese when he gave a talk on quantum computing as part of his interview with GTRI in 2004. I asked a question he thought was insightful and after he was hired he sought me out to discuss it further. Over the ensuing years he became my best friend. John is not only one of the smartest people on earth (he's the only person in history to earn two Ph.D.s from Caltech, one in Electrical Engineering and one in Theoretical Physics), he has a gift for explaining advanced concepts to lay people, and he is one of the best human beings I know. Two years ago I was in despair. It seemed that the cumulative effects of family challenges and full-time work would prevent me from ever finishing my Ph.D. Without my reporting any of this to John – as if he had some sort of sixth sense – he called me out of the blue and offered to help me pay my bills so I could quit my job and finish my Ph.D. The next semester I went part-time and redoubled my efforts. Although I later went full-time again, John's support had reinvigorated me. It is no exaggeration to say that I owe this Ph.D. to John.

Finally I want to thank my wife, Caroline, who met me before I passed my qualifier, and after I became a full-time single father. She has endured years of being a "Ph.D. widow" while taking on the role of stepmother. Caroline has helped me in a million small and not so small ways as I struggled to manage a young family in distress. No matter how discouraged I became, no matter how sullen my mood, every time our eyes met she smiled. It is difficult to overstate the impact of such a seemingly small gesture. I would not be here without Caroline.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Reinforcement learning is a promising solution to the intelligent agent problem, namely, given the state of the world, which action should an agent take to maximize goal attainment. However, reinforcement learning algorithms are slow to converge for larger state spaces and using reinforcement learning in agent programs requires detailed knowledge of reinforcement learning algorithms.

One approach to solving the curse of dimensionality in reinforcement learning is decomposition. Modular reinforcement learning, as it is called in the literature, decomposes an agent into concurrently running reinforcement learning modules that each learn a "selfish" solution to a subset of the original problem. For example, a bunny agent might be decomposed into a module that avoids predators and a module that finds food. Current approaches to modular reinforcement learning support decomposition but, because the reward scales of the modules must be comparable, they are not composable – a module written for one agent cannot be reused in another agent without modifying its reward function.

This dissertation makes two contributions: (1) a command arbitration algorithm for modular reinforcement learning that enables composability by decoupling the reward scales of reinforcement learning modules, and (2) a Scala-embedded domain-specific language – AFABL (A Friendly Adaptive Behavior Language) – that integrates modular reinforcement learning in a way that allows programmers to use reinforcement learning without knowing much about reinforcement learning algorithms. We empirically demonstrate the reward comparability problem and show that our command arbitration algorithm solves it, and we present the results of a study in which programmers used AFABL and traditional programming to write a simple agent and adapt it to a new domain, demonstrating the promise of language-integrated reinforcement learning for practical agent software engineering.

CHAPTER 1
INTRODUCTION

If you don't know where you are going, any road will take you there.

– Lewis Carroll, paraphrased from Alice's Adventures in Wonderland

This chapter sets the stage for the work presented in this dissertation, an overview of its contributions, and concludes with a road-map of the following chapters.

1.1 The Dream of AI

Artificial intelligence was one of the first grand promises of computing. Almost as soon as the field of computer science was born the pioneers of AI were promising machines that could think and act as well as humans within the "visible future" (Herbert Simon, quoted in 1958). Early research in AI focused on machines that could "think" like humans and employed symbolic computation to create constraint solvers, planners, and exert systems that used truth maintenance systems to maintain sets of facts and inference rules that produced new knowledge. We had computer programs that solved algebra equations, found paths, played games, and diagnosed illnesses based on user-reported symptoms. Symbolic, or knowledge-based AI hit a bottleneck in the late 1980s – commonly called the knowledge acquisition bottleneck – and the early promise of developing systems that *replaced* humans faded. But AI in general did not fade. AI reinvented itself. Instead of creating systems of rules and inference engines based on encoded knowledge, modern AI applies well-developed models from mathematics and engineering – vector space models for text retrieval, hidden Markov models for speech recognition, neural networks (though originally invented within AI but then abandoned and developed by electrical engineers) for image recognition – to problems traditionally considered part of AI. In modern AI the emphasis is on rationality – achieving optimal results without trying to explicitly mimic human cognition – and in developing systems that assist humans in some way. In modern AI every approach must have a performance measure that is being optimized – speech phoneme recognition rate, generalization error in image classifiers, etc. – and AI algorithms are not accepted if they do not either define a new category rigorously, or rigorously compare their performance to existing algorithms using accepted performance criteria. Most AI algorithms today are concerned with focused problems and find application in software that assists humans, such as helping humans find the right books to buy using collaborative filtering or finding photos of their babies using facial image recognition. Reinforcement learning is notable for applying modern AI approaches to the age old intelligent agent problem: given the state of the world and all the actions an agent *could* take, which action *should* the agent take, that is, which action results in the agent accumulating the greatest long-term reward.

One can think of reinforcement learning (RL) as a machine learning approach to planning, that is, a way of finding a sequence of actions that achieves a goal. The RL problem formulation is this: an agent's world is described by a set of states, the agent can execute an action from a prescribed set of actions in each state, and the agent is rewarded to greater or lesser degrees for each state-changing action it executes. The performance measure being optimized by reinforcement learning algorithms is long-term expected reward. So a reinforcement learning agent learns delayed gratification. For example, eating cake now provides high immediate reward but low long-term reward. A reinforcement learning agent learns to choose salad over cake (unless the agent is Ron Swanson). For the software engineer who would like to employ reinforcement learning without becoming an expert the most important thing to understand is that the world of an agent can be modeled in terms of states, actions, and one-step rewards. Our work shows that if you can specify the states, actions, and rewards for an agent our algorithms can work behind the scenes to develop a control policy.

1.2 The Challenges of Software Engineering

Software engineering has struggled to keep pace with the growing size and complexity of the systems being demanded by users. Over time the field of software engineering, both in academia and industry, has developed a well-defined set of practices and design guidelines that result in software systems that are maintainable, reliable and extensible. Programming languages have been a primary means by which research in software engineering and formal computer science has been brought to bear for the working programmer. From structured programming to object-oriented programming to powerful modern type systems, important advances in computing research have real impact when they are incorporated as features in practical programming languages. In the same way that, say, formal methods are used by the modern programmer in the form of static type systems without requiring the programmer to know much about formal methods, AFABL's goal is to allow the programmer to use reinforcement learning without knowing much about reinforcement learning algorithms.

1.3 The Promise of Adaptive Partial Programming

Our work in integrating reinforcement learning into a programming language is based on the idea of partial programming developed by researchers in hierarchical reinforcement learning. Partial programming is a framework for programming in which a programmer or designer specifies the structure of certain parts of the system while leaving other portions unspecified, such that a learning system can learn how to perform them. Hierarchical reinforcement learning defines closed-loop policies that group action sequences into logical units – subroutines – that achieve intermediate goals. For example, if the ultimate goal of an agent is to leave a building by exiting a room and walking down an hall, and the agent's primitive actions are MOVE-FORWARD, MOVE-LEFT, TURN-RIGHT and so-on, then one of the agent's subroutines might be FIND-DOOR, which achieves the intermediate goal of exiting the room. Partial programming system allows the agent designer to specify intermediate goals, write code to achieve certain goals, and let reinforcement learning algorithms learn how to achieve the others.

Hierarchical reinforcement learning decomposes the reinforcement learning problem temporally, that is, it breaks action sequences into subsequences. Another kind of adaptive programming – known as modular reinforcement learning (MRL) – is based on concurrent problem decomposition in which an agent may take only one action at a time but must pursue several goals simultaneously. MRL is especially well suited to continuing problems in which an agent is never "done" pursuing certain goals. For example, for the entire time it is operating a self-driving car will need to avoid other vehicles, optimize speeds for road conditions subject to speed limits, maximize fuel economy, minimize travel time, and so on.

MRL is a developing field with few algorithms and a small number of programming systems attempting to make MRL usable for working programmers. Currently available algorithms for MRL and programming systems based on them suffer from a reward coupling problem: modules used within the same agent must be authored together to ensure reward scales are comparable. This reward coupling is an impediment to module reuse in practical software engineering because an existing module can not simply be reused in a new context without re-engineering its reward function. Our work solves this problem with a reformulation of MRL and an algorithm that makes module reuse possible, and integrates this new kind of MRL in a practical programming language.

1.4 Contributions

The work presented in this dissertation marries artificial intelligence and software engineering in a way that advances both fields. The needs of practical software engineering for reuse and composability inspires a new AI algorithm for modular reinforcement learning. Integrating this new formulation of modular reinforcement learning and associated algorithms into a programming language enables a new kind of software engineering: modular adaptive agent programming. In particular, this work makes the following contributions:

- We explain a problem with the current state of the art in modular reinforcement learning, namely, that performance degrades if modules have differing, incomparable reward scales. While there may be other incompatibility problems, such as synchronization and impedance mismatch, we focus on reward incomparability because our goal is to enable independently authored modules.
- We empirically demonstrate the performance degradation of modular reinforcement learning agents whose modules have incomparable reward scales.
- We present an analysis of the shortcoming of current approaches to modular reinforcement learning based on Arrow's Impossibility Theorem for social choice in order to frame our solution.
- We reformulate the modular reinforcement learning problem as one of *command arbitration* instead of merging MDPs or Q-functions.
- We present a command arbitration algorithm – Arbi-Q – that uses our theoretically grounded reformulation of modular reinforcement learning.
- We empirically demonstrate that modular reinforcement learning agents using Arbi-Q exhibit no performance degradation when modules have incomparable

reward scales.

- We present a Scala-embedded domain-specific language – AFABL – that integrates modular reinforcement learning and our Arbi-Q command arbitration algorithm.

- We demonstrate and quantify the value of integrating modular reinforcement learning into a programming language to practical software engineering in a programmer study applying AFABL in a synthetic agent programming domain.

- We apply AFABL to a practical problem in psychology-based human agent modeling to demonstrate AFABL's practical potential.

1.5 Outline

In the following chapters we present the two major contributions of this dissertation: command arbitration for modular reinforcement learning and a practical agent programming language that integrates modular reinforcement learning: AFABL. For both major contributions there is a chapter providing the necessary background, then a chapter presenting our contributions. Related work is provided in each chapter where it is most relevant rather than gathered in a single place.

Chapter 2 provides background information in modular reinforcement learning and existing approaches to modular reinforcement learning.

Chapter 3 presents an empirical demonstration of the performance degradation of modular reinforcement learning agents whose modules have incomparable reward scales. Arrow's Impossibility Theorem for social choice provides an explanation for the failure of existing approaches to modular reinforcement learning and a framework for our solution. We present our solution, the Arbi-Q command arbitration algorithm, and empirically demonstrate that it does not exhibit the same performance degradation as existing approaches to modular reinforcement learning.

Chapter 4 provides background information on software engineering that motivates the use of modular reinforcement learning in building practical software systems, and the integration of modular reinforcement learning into a programming language.

Chapter 5 presents a programming language, AFABL, which integrates modular reinforcement learning. AFABL, a domain-specific language embedded in the Scala language, allows programmers to write adaptive software agents in a declarative style using elements of modular reinforcement learning: modules with states, actions, and rewards. We present the results of a programmer study that shows the value of integrating reinforcement learning into a programming language: AFABL agents are less complex, easier to write, and easier to adapt to changes in the environment.

Chapter 8 presents a practical application of AFABL that further demonstrates the usefulness of integrating modular reinforcement learning into a programming language.

Chapter 9 concludes the dissertation by reviewing how the central theses of this dissertation were confirmed and the present work's context, limitations, and consequences and discuss directions for future work.

CHAPTER 2

BACKGROUND IN REINFORCEMENT LEARNING

The field of reinforcement learning is large and growing. In this chapter we provide the background in reinforcement learning and modular reinforcement learning necessary to understand our work and place it in context.

2.1 Monolithic Reinforcement Learning

One can think of reinforcement learning (RL) [1, 2] as a machine learning approach to planning, that is, as a way of finding a sequence of actions that achieves a goal. 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 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 agent's 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. In short, a policy defines which action an agent should take in a given state to maximize its chances of reaching a goal.

2.1.1 Markov Decision Processes

The basic Markov decision process is a 3-tuple,

$$\langle S, A, T(s,a,s') \rangle \tag{2.1}$$

where

- S is a set of states,
- A is a set of actions, and
- $T(s,a,s')$ is a transition function which gives the probability that executing action a in state s will result in s'.

Most definitions of MDPs include a reward function, $R(s,a,s')$ which specifies the reward that the world provides to an agent for taking action a in state s and arriving in state s' or, equivalently, a reward for arriving in state s, $R(s)$. Some definitions of MDPs include an initialization function, $I(s)$, which specifies the probability the the agent will start in some state $s \in S$, others specify a particular state from S as the start state.

We prefer to think of the basic 3-tuple MDP as representing the states and state transition dynamics of a world, and an agent solving a Markov Decision *Problem* which adds the reward function, initialization function, and discount factor – the rate at which long-term rewards are discounted compared to short-term rewards, usually 0.9. This distinction between Markov Decision *Processes* and Markov Decision *Problems* leads to a more natural expression of worlds and agents for programmers who are not familiar with the underlying theory of reinforcement learning. For example, most people would not include a single universal reward function as part of the "world" because different agents may value states differently. Similarly, different agents may have a shorter or longer term view of decision optimality and thus different discount factors. Separating the world dynamics from the agent's use of the world dynamics doesn't change the reinforcement learning algorithms, but as we will see in Chapter 5, separating worlds from agents is useful in practical programming.

The solution to a Markov Decision Problem is a policy, denoted π, which is a mapping from states to actions. The policy says, for any state, what is the best action to take in order to maximize the agent's long term reward.

2.1.2 Long Term Reward

Acting optimally in the world means maximizing your long term reward. In the MDP setting there are many ways to compute long term reward depending on whether you consider rewards gained during a finite or infinite time horizon, can depend on there being a terminal state the agent is guaranteed to reach, or whether the agent's task is episodic or continuing. The most commonly used formulation of long term reward is geometrically discounted rewards over an infinite time horizon, which works for episodic tasks, continuing tasks, and results in stationary policies, that is, policies that depend only on the current state no matter the time step in which the state is visited:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.2}$$

where R_t is the long-term sum of rewards at time step t, r_{t+k+1} is the one-step reward for a particular step in the future, and γ is a discount factor less than 1 which causes short-term rewards to be more important than rewards received in the future. We use this conception of long term reward to calculate the values of states.

2.1.3 Value Functions

A value function assigns a number to each state indicating how good is it for an agent to be in that state. If you know the values of all the possible next states of your current state, then the action you should take is the action most likely to get you into the next state having the highest value.

The value of a state (or state, action pair) under a particular policy π is the expected long term reward an agent would get starting from state s if it followed π. The value is computed from the rewards and state transition dynamics of a world as follows:

$$V^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right] \tag{2.3}$$

State value functions form the basis of dynamic programming approaches to solving MPDs. The reinforcement learning algorithms we discuss in Section 2.1.6 use action value functions, which assign values directly to the actions available in a given state.

2.1.4 Optimal Policies

Since some states have higher values than others, and some actions have higher probabilities of causing transitions to higher-valued successor states, there is an optimal action in each state, namely, the action most likely to cause a transition to the highest-valued successor state. The optimal actions in each state is the optimal policy:

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}} \sum_{s'} T(s, a, s') V(s') \tag{2.4}$$

There may be many optimal policies for a given MDP, but without loss of generality we usually simply say "the optimal policy."

2.1.5 Solving MDPs with Dynamic Programming

If you have the full MDP – you know the transition function and reward function – you can solve for the value of each state using value iteration, from which the optimal policy is derived, or find the policy directly using policy iteration. Although a practical reinforcement learning agent usually does not have the full MDP model, the theory of dynamic programming is useful in understanding reinforcement learning algorithms.

Value Iteration

Previously we defined the value of a state under a particular policy. If we assume the optimal policy, we can define the maximum value of a state in terms of optimal actions:

$$V(s) = R(s) + \max_{a \in A} \sum_{s'} T(s, a, s') V(s') \tag{2.5}$$

This equation is called the Bellman optimality equation [3, 4]. There is one Bellman equation for each state – n equations in n unknowns (the values) for a state space of size n. However, since the *max* operator is nonlinear we cannot solve the system of simultaneous Bellman equations using linear algebra. One solution is to use an iterative dynamic programming approach: value iteration. The value iteration algorithm initializes each state's value to a random value, then iteratively update these values by turning the Bellman equation into an update rule (the Bellman update):

$$V_{i+1}(s) \leftarrow R(s) + \max_{a \in A} \sum_{s'} T(s, a, s') V_i(s') \tag{2.6}$$

These updates are applied at the same time for all states, i.e., the values in iteration $i + 1$ are calculated from the values in iteration i. The value iteration algorithm is shown in Algorithm 2.1

Algorithm 2.1 Value Iteration

```
$V \leftarrow$ random initial values
**repeat**
    $V' \leftarrow V$
    **for** each $s \in S$ **do**
        $V'(s) \leftarrow R(s) + \max_{a \in A} \sum_{s'} T(s,a,s')V(s')$
    $V \leftarrow V'$
**until** $V$ changes by a sufficiently small amount
```

If we apply the Bellman update infinitely often, then value iteration reaches an equilibrium at which point the values of the states are solutions to the Bellman equations, that is, they are optimal. A policy derived from these values is an optimal policy.

In practice we may iterate until the updated values change by a sufficiently small amount. If all we care about is finding the optimal policy, then it does not matter that we find the optimal values, only that the values we find lead to the same optimal policy.

Policy Iteration

In policy iteration [5] we start with a random initial values and policy and alternate between two steps for each iteration i:

- **Policy evaluation.** Use policy π_i to calculate the values of each state using the discounted current values of their successor states. Since we are calculating the values under a particular policy, we drop the *max* operator:

$$V_{i+1}(s) = R(s) + \gamma \sum_{s'} T(s,a,s')V(s') \tag{2.7}$$

- **Policy improvement.** Calculate policy p_{i+1} using the values calculated in the previous step.

When policy improvement does not change the policy, an optimal policy has been found and policy iteration terminates.

Note that since the update equation used in policy evaluation is linear, we can use linear algebra to solve the set of simultaneous linear equations in $O(n^3)$. This method works fine for smaller state spaces but may be too expensive for large state spaces. A solution to this problem is known as modified policy iteration [6, 7], which combines policy iteration with value iteration by using a bounded number of Bellman updates to perform the policy evaluation step.

2.1.6 Learning Policies via Reinforcement Learning

If the agent does not have the model, then the agent must learn a behavior policy by interacting with the world by taking actions, observing the effects of these actions (state transitions and reward signals), and using these observations to update some model of the world that can be used to guide future behavior. Unlike the MDP solvers discussed previously which calculate values for every state in every iteration, reinforcement learning agents choose which states to visit and often do not visit every state, leading to a central issue in reinforcement learning: exploration vs. exploitation. Much of the reinforcement learning information in this section can be found in the excellent introductory book by Sutton and Barto [1], the survey by Kaelbling and colleagues [2] or even a general AI textbook such as the excellent book by Russell and Norvig [8]

Exploration versus Exploitation

Exploration means visiting some state of the world you have not yet visited. Eating at a restaurant or ordering a dish you have not tried to see if you like it is an example of exploration. Exploitation means using knowledge already learned. Eating a dish you know you like at a restaurant you know you like is an example of exploitation.

The exploration versus exploitation question is one of risk versus reward: exploration risks wasting effort on a possibly bad outcome, exploitation trades that risk for a known reward but risks missing a discovery of something better.

In the case of reinforcement learning exploration means taking an action in a state that may take the agent to a state not yet visited. Exploitation means using the agent's current value estimates to choose an action. We call exploitation *greedy* action selection because it chooses an action that is likely to result in higher reward. There is a saying in machine learning: the less you know the less you should trust your knowledge, the more you know the more you should trust your knowledge. A reinforcement learning agent should favor exploration early in its learning process and exploitation after its model of the world stabilizes.

Implementing an exploration strategy in a reinforcement learning algorithm can be done using softmax action selection such as a Boltzmann distribution, which uses a decaying temperature parameter similar to annealing to gradually decrease the randomness of action selection, or by a simpler method known as ϵ-greedy action selection. ϵ is a real number in $(0, 1)$ that represents the probability of choosing an action randomly (exploring). ϵ is decayed over time so that action selection favors exploration early in the learning process and exploitation later. We use ϵ-greedy action selection in our algorithm implementations.

Q-Learning

An alternative value function assigns values to state-action pairs instead of states. Such a function is called a *Q-function*, and the optimal Q-function is defined as:

$$Q^*(s,a) = R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a') \tag{2.8}$$

which means that the value of taking action a in state s is the reward received from state s plus the discounted probabilistic sum of the Q-values of successor states assuming that optimal successor action a' are taken in the successor state s'.

An important consequence of learning a Q-function instead of a state-value function is that, although Q-values can be defined as above, a reinforcement learning agent can approximate the Q-function using temporal difference learning which does not require the model. A reinforcement learning agent only needs to know the Q-values because the purpose is only to learn a policy which is derived directly from the Q-function. This is the idea behind the Q-learning algorithm [9], which uses the following temporal difference update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{2.9}$$

α is a learning rate parameter. Temporal difference algorithms like Q-learning use the differences between Q-values in successive states to update Q-values in each iteration. Like value and policy iteration, Q-learning has been shown to converge to $Q*$ in the limit with infinite exploration. In practice developing a close approximation to $Q*$ isn't necessary as long as the same policy results.

Algorithm 2.2 Q-Learning

```
$Q \leftarrow$ random initial values
**for** each episode **do**
    $s \leftarrow$ world.initialState()
    **repeat**
        $a \leftarrow \epsilon-$greedy action for $s$ from $\pi$ derived from $Q$
        Execute $a$, observe effects $r$ and $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha[R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$
    **until** $s$ is terminal
```

Sarsa

Because it uses the best successor Q-value in its update rule, Q-learning is an off-policy learning algorithm, meaning that it does not use the policy being followed by the agent. A close relative of Q-learning is the Sarsa algorithm [10], which is an on-policy algorithm that uses the following update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s) + \gamma Q(s',a') - Q(s,a))] \tag{2.10}$$

Sarsa gets its name from the elements of its update rule: s, a, r, s', a'. The key difference is that the $Q(s',a')$ from the agent's policy is used in the update rule.

Algorithm 2.3 Sarsa

```
$Q \leftarrow$ random initial values
**for** each episode **do**
    $s \leftarrow$ world.initialState()
    $a \leftarrow \epsilon-$greedy action for $s$ from $\pi$ derived from $Q$
    **repeat**
        Execute $a$, observe effects $r$ and $s'$
        $a' \leftarrow \epsilon-$greedy action for $s'$ from $\pi$ derived from $Q$
        $Q(s,a) \leftarrow Q(s,a) + \alpha[R(s) + \gamma Q(s',a') - Q(s,a))]$
        $s \leftarrow s'$
        $a \leftarrow a'$
    **until** $s$ is terminal
```

Q-learning has the advantage of being able to learn an optimal policy even if the control policy during learning is poor. The advantage of Sarsa over Q-learning is that if the agent does not have full control over action selection, Sarsa will learn a policy that is optimal in the presence of external control over action selection, such as other (sub)agents. For this reason, Sarsa is used in modular reinforcement learning.

2.2 Decompositional Reinforcement Learning

Decomposition is an important tool in designing software systems in general, and as we will see in Section 2.2.2 a necessity for larger state spaces likely to be encountered in real-world problems. Hierarchical reinforcement learning (HRL), discussed first below, has had as a primary goal improved authoring based on temporal subgoals that allows intermediate tasks to be modeled as higher-level actions. HRL has also formed the basis of current approaches to RL-based programming systems. Modular reinforcement learning (MRL) decomposes the original problem concurrently, modeling an agent as a set of concurrently running reinforcement learning modules. MRL has been used primarily to model multiple-goal problems and to deal with large state spaces. This work is the first to use MRL as the basis of a practical programming system and validate its advantages to working programmers scientifically.

2.2.1 Hierarchical Reinforcement Learning

Current implementations of partial programming are based on hierarchical reinforcement learning (HRL), which exploits a temporal decomposition of the Q function. In the case of HRL, the designer typically specifies a delegation hierarchy of components with points of adaptation where a policy is learned to perform the delegation. The designer programs the policies of some of these components, which constitute a partial specification of the agent's behavior, and some of the components use reinforcement learning to adapt to the hierarchy by learning the control policies for their parts of the problem. The adaptive components relieve the designer from writing the parts of the program that are hard to specify, or require difficult to write adaptivity, and the partial program constrains the learning problem faced by the adaptive components, which speeds convergence. Components can be reused in other contexts, providing for modularity in temporal problem decompositions.

The current state of the art in HRL is based on the theory of **Semi-Markov Decision Processes** (SMDPs). In an SMDP some actions are allowed to take more than one time step. These multi-step actions, sometimes called macros, subroutines, options, or hierarchical machines represent a form of procedural abstraction that may allow a programmer to write reinforcement learning agents in a manner similar to writing other kinds of computer programs. The SMDP-based thread of research in HRL also includes work in developing programming languages and systems that exploit HRL models, making this line of research particularly relevant to our work.

Precup's **Options** [11, 12, 13] framework develops a detailed theory of temporal abstraction based on SMDPs. Dietterich's **MAXQ** [14, 15] and **Hierarchical Abstract Machines** [16] models decompose an MDP into a hierarchies of smaller MDPs. These smaller MDPs represent *subroutines* in the MAXQ framework or *HAMs* in the **Hierarchical Abstract Machines** framework. Andre's **Programmable Hierarchical Abstract Machines** [17, 18] develops a programming language for HAMs called ALisp (Adaptive Lisp), and Marthi and colleagues [19] extend ALisp (Concurrent ALisp) for single reinforcement learning agents with multiple simultaneous actions, such as as robots with multiple effectors or a controller for multiple characters in a computer game.

2.2.2 The Curse of Dimensionality in Reinforcement Learning

As with other kinds of machine learning, reinforcement learning must deal with the curse of dimensionality. In reinforcement learning the curse of dimensionality manifests itself primarily in the size of the state space, namely, the state space grows exponentially in the number of state features. As an example, consider the 5×5 grid of the bunny world in Figure A.1.

The bunny, food, and wolf can be in one of 25 possible locations. If the task of the bunny were only to reach a particular location, then the number of states would be 25. But if you add the task of avoiding a wolf that pursues the bunny, then the state space grows to $25^2 = 625$. Add the food that reappears in a new location after it is found and "eaten" and the state space grows to $25^3 = 15625$. If you model this problem as two separate modules, one in which the bunny avoids the wolf and another in which the bunny finds food, then each module solves a problem with a state space of $25^2 = 625$ states (bunny plus wolf and bunny plus food).

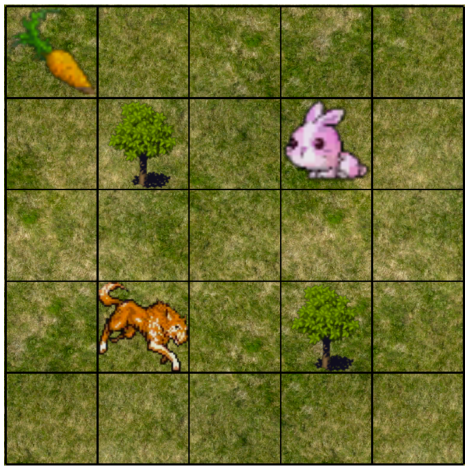

Figure 2.1: In the grid world above, the bunny must pursue two goals simultaneously: find food and avoid the wolf. The bunny may move north, south, east, or west. When it finds food it consumes the food and new food appears elsewhere in the grid world, when it meets the wolf it is eaten and "dies."

2.2.3 Modular Reinforcement Learning

A second kind of decomposition in reinforcement learning, which is somewhat confusingly referred to as modular reinforcement learning (MRL) [20, 21], decomposes the original problem *concurrently* rather than temporally. Instead of composing sequential actions into subsequences, a MRL agent is decomposed into several modules each of which is concurrently learning a different subgoal of the original, complex, multiple-goal learning problem. The architecture of an MRL agent is reminiscent of Brooks's subsumption architecture [22], where an agent is decomposed into several modules that each receive sensor input and each independently suggest an action for the overall agent to take (Brooks refers to these modules as layers). As we will soon discuss, our approach uses Brooks's idea of a central arbitrator to choose the agent's actions from the suggestions of its modules.

There have been two major kinds of approaches to modular reinforcement learning: merging MDPs [23], and merging Q functions [21, 20]. Since we are interested in applying reinforcement learning in practical domains where we can not assume, or do not want to require, complete knowledge of the world's state transition dynamics, we focus on approaches to MRL that merge Q-functions.

In an MRL agent each module observes the action taken by the agent, the state transition and a reward signal specific to the module. On each occasion that a decision for what action to take is needed, the agent combines the action preferences of the modules to compute the output of the joint policy. The current state of the art in modular reinforcement learning uses the the Q-values of the modules directly to effect this action selection. The joint policy is derived from a joint Q-function in which the Q-values for each module are added.

$$Q_{joint}(s,a) = \sum Q_i(s,a) \tag{2.11}$$

Russel's and Zimdars' Q-decomposition [20] is equivalent to Sprague and Ballard's GM-Sarsa [21]. In this work we will refer mostly to GM-Sarsa. Both of these investigations of Q-function (de)composition showed that Sarsa is a better choice for agent modules than Q-learning because Sarsa is on-policy. As we discussed earlier, Sarsa updates its Q-function based on the policy being followed by the agent where as Q-learning updates its Q-function assuming the optimal policy will be followed. Because modules do not have direct control over the policy being followed, on-policy learning algorithms like Sarsa perform better. For this reason, and so we can compare directly to GM-Sarsa, we also use Sarsa in our modules.

There is clearly a desire to model agents with multiple goals represented as reinforcement learning modules. We list only a few examples here. Sprague and Ballard have applied GM-Sarsa to problems in eye movement scheduling [24, 25]. Konidaris and Barto applied an algorithm derived from GM-Sarsa to adaptive robot control [26]. Aissani and colleagues used GM-Sarsa to develop a system for dynamic scheduling of maintenance tasks in the petroleum industry [27, 28]. Rowe and colleagues used GM-Sarsa for interactive narrative planning [29]. All of the work applying MRL has been done by single teams of researchers applying MRL to research problems. Thus, modules were authored together with comparable reward scales. To support reusability in a software engineering sense we need to support the separate authoring of modules. Separately authored modules may use reward scales that are internally consistent within modules but incomparable to the rewards used in other modules. In the next chapter we show that existing approaches to MRL degrade when modules use incomparable reward scales and present an algorithm that does not exhibit the same degradation.

CHAPTER 3
ROBUST COMMAND ARBITRATION FOR MODULAR REINFORCEMENT LEARNING

In this chapter we demonstrate the performance degradation of modular reinforcement learning agents whose module have incomparable reward scales. Arrow's Impossibility Theorem for social choice provides an explanation for the failure of existing approaches to modular reinforcement learning and a framework for our solution. We present our solution, the Arbi-Q command arbitration algorithm, and empirically demonstrate that it does not exhibit the same performance degradation as existing approaches to modular reinforcement learning.

3.1 Modular reinforcement learning

In the previous chapter we explained the state of the art in modular reinforcement learning as a decomposition of an implicitly global Q-function in to additive modules. To support software engineering we would like these components to be truly modular. In particular, we would like the components to be reusable and easily understood by agent authors. Unfortunately, current approaches implicitly require a global, monolithic reward signal, which detracts from these properties. In particular, it detracts from the ability of the agent author to locally define the reward for a component because the reward scales of other modules must be taken into account.

Like any software engineering, agent programming would benefit from modularity. Truly modular reinforcement learning would facilitate speed of learning convergence, state abstraction, and transfer – a module written in one context can be reused in another context because the component is dependent only on its own local reward signal.

In this section we discuss the difficulties that current approaches face in achieving true modularity and present a new formulation which solves the core problem in Section 3.2.

3.1.1 Merging local signals

Difficulty arises when multiple single goals are being combined in a larger, multi-goal learning problem. Take AvoidWolf and FindFood for instance; it is fairly straightforward to code an internally consistent reward signal for each. However, it is unclear how to combine the two into the larger task of LiveLong. For example, if there is no penalty for failing to eat within a certain time period (starvation), then the obvious policy is to avoid the predator and ignore the food. Such a degenerate case could also happen if the reward signal for one learning module were scaled without re-engineering the other reward signals in the system, a problem explained by Bhat, et., al. [30]. For example, one could imagine swapping-in a new AvoidWolf module with a reward several times higher than the previous module, such that avoiding the predator always carries higher reward than finding the food. In this case, a delegating agent would favor AvoidWolf to the near exclusion of FindFood. The ability to substitute learning modules without modifying the rest of the system is one of the primary benefits of true modularity, and this modularity is difficult to achieve if local reward signals must be merged.

3.1.2 Ideal Action Selection is Impossible

Aside from the practical challenges cited above, Bhat, et., al. [30] showed that ideal action selection is impossible in full generality because the problem reduces to Arrow's Paradox [31]: an agent is a "society" of modules, and action selection is social choice. The problem is that we want the action selection mechanism to have the following reasonable properties:

- **Universality**: the ability to handle any possible set of modules.
- **Unanimity**: guarantee that if every module prefers action A, action A will be selected.
- **Independence of Irrelevant Alternatives**: each module's preference for actions A and B are independent of the availability of any other action C. This property prevents any particular module from affecting the global action choice by dishonestly reporting its own preference ordering.
- **Scale Invariance**: ability to scale any module's Q-values without affecting the arbitrator's choice. This is the crucial property that allows separately authored modules with incomparable reward signals.
- **Non-Dictatorship**: no module gets its way all the time.

[32]

According to Arrow's Paradox, if $|A| \geq 3$, then there does not exist an arbitration function that satisfies each of the properties listed above. So even simple agents with more than three actions are too complex for theoretically ideal arbitration. This dissertation contributes a novel formulation of MRL and an algorithm that implements it, which we discuss in Section 3.3.

3.2 Reformulating MRL

Bhat, et., al., [30] argued for a "benevolent dictator", a special module executing a command arbitration function for action selection but left the arbitration algorithm unspecified. Here we present a command arbitration algorithm embodying the ideas in [30] and show that it performs competitively with other MRL algorithms and shows superior robustness to module modification. This robustness to module modification is the chief enabler of truly modular reinforcement learning in which modules can be transferred from one system to another without having to re-engineer the reward signals to fit the new host system.

This formulation relaxes the non-dictatorship requirement of ideal action selection if you think of the arbitrator as a special module. By Arrow's theorem, other properties will still hold. In the next section we present our Arbi-Q algorithm based on this practicalized arbitrator-based framework.

3.2.1 Formalization

Our reformulation of MRL adds a *command arbitrator* [22], Bhat's "benevolent dictator" module. The arbitrator has a state space that may be the same as or different from the modules' state spaces, an action set that represents choosing a module, $A_{CA} = 1...n$, and a reward signal that represents the "greater good." The arbitrator's reward function, $R_a(s)$, is independently defined rather than being derived from the module rewards. It is now another part of the problem specification; in the partial programming setting, this corresponds to $R_a(s)$ being human-authored. Note that $R_a(s)$ may or may not be equal to the sum of the rewards of the agent's modules. In fact, the module rewards $R_i(s, a)$ may not have any correlation with the arbitrator's reward $R_a(s)$.

The agent's policy is defined indirectly by the arbitrator's policy, $\pi_{CA}(s, a)$, which assigns probabilities to the selection of each module's preferred action for each state.

For the agent author, this formulation adds the requirement of authoring a dedicated reward signal for the arbitrator. For our bunny agent, this is LiveLongProsper:

- *Why* avoid predator, *why* eat? To live longer.
- Encodes the tradeoffs between modules – perhaps food is more important to some bunnies.

- The arbitrator could be hand-authored, or could be another RL agent.

For the small cost of authoring a reward signal that represents the "greater good" you get true modularity, that is, the ability to combine separately authored modules with incomparable rewards. This new reward signal is now the metric we use to measure the performance of the agent.

In our MRL framework an agent is an arbitrator plus a list of modules. Formally, an agent consists of the following elements:

1. A reward function for the command arbitrator, $R_{CA}(s)$,
2. An action set A for the agent as a whole, shared by each module,
3. A set of reinforcement learning modules, M
4. A state abstraction function, $moduleState_i$ for each module m_i
5. A reward function, $reward_i$ for each module m_i

In the next section we present a reinforcement learning-based command arbitration algorithm.

3.3 The Arbi-Q Command Arbitration Algorithm

Our reformulation of MRL is based on an independently specified arbitrator [22] that is itself a reinforcement learner. The state space for the arbitrator is the world state – no state abstraction is used for the arbitrator. The arbitrator's action set, A_{CA}, is a set of integer indexes to the agent's list of modules (we use CA in subscript to refer to the command arbitrator and numbers or i to refer to modules). As with any reinforcement learner, the arbitrator learns a policy. In the case of the arbitrator this policy, π_{CA} is a mapping from states to modules. The modules' policies are mappings from abstracted module state to actions in the world, that is, the agent's actions. The policy defines which module chooses the agent's action in a particular state.

Arbi-Q uses the Sarsa Q-Learning algorithm to learn the arbitrator's policy. At each time step the arbitrator uses its policy to select a module, then the module uses its local policy to select an action that the agent executes. The results of executing the action are communicated to the arbitrator as a consequence of the module selection, and to the modules as a consequence of action selection. Each module uses a state abstraction function to transform the world state into the the subset of the state relevant to the module, and a reward function that is based on the module's state abstraction. In this way the modules are coupled to the world in which they operate – the modules can only operate in worlds which contain the state features expected by its state abstraction function – but the modules are not coupled to other modules or to an arbitrator. The Arbi-Q algorithm is detailed in Algorithm 3.1

Algorithm 3.1 Arbi-Q

```
$Q_{CA} \leftarrow$ random initial values
**for** each module $i$ **do**
    $Q_i \leftarrow$ random initial values
**for** each episode **do**
    $s \leftarrow$ world.initialState()
    $m \leftarrow \epsilon-$greedy action for $s$ from $\pi_{CA}$ derived from $Q_{CA}$        ▷ choose module
    $s_m \leftarrow moduleState(s)$                                  ▷ abstract state for module
    $a \leftarrow \epsilon-$greedy action for $s_m$ from $\pi_m$ derived from $Q_m$
    **repeat**
        Execute $a$, observe effects $r_{CA}$ and $s'$
        $m \leftarrow \epsilon-$greedy action for $s$ from $\pi_{CA}$ derived from $Q_{CA}$    ▷ choose module
        $s_m \leftarrow moduleState(s)$                              ▷ abstract state for module
        $a' \leftarrow \epsilon-$greedy action for $s'$ from $\pi$ derived from $Q$
        $Q_{CA}(s,a) \leftarrow Q_{CA}(s,a) + \alpha[R_{CA}(s) + \gamma Q_{CA}(s',a') - Q_{CA}(s,a)]$
        **for** each module $i$ **do**
            $s'_i \leftarrow moduleState(s')$                        ▷ abstract state for module
            $r_i \leftarrow reward(s_i)$                             ▷ module-specific reward
            $Q_i(s,a) \leftarrow Q_i(s,a) + \alpha[r_i + \gamma Q(s'_i,a') - Q(s_i,a)]$
        $s \leftarrow s'$
        $a \leftarrow a'$
    **until** $s$ is terminal
```

3.4 Experiments

Our principal claim is that Arbi-Q is robust to modules with incomparable reward scales, which would be an authoring error in existing MRL approaches. Our experiments show that GM-Q/Q-decomposition degrades when modules are modified to have incomparable reward scales and that Arbi-Q is robust to such modification.

3.4.1 Bunny-Wolf World

We use a world derived from Sprague and Ballard [21]. In Bunny-Wolf world, our agent is a bunny that must eat food and avoid being eaten by a wolf. The bunny world is a continuing world rather than an episodic world. There is no specified start state and there is no termination of episodes. When the bunny finds and eats food, a new food item appears elsewhere. When the wolf eats the bunny the bunny "respawns" in a new location, similar to video games. We can represent such a bunny agent in our formulation as follows:

- Module 1: FindFood. The bunny agent must find food in order to continue living. When the bunny finds food it gets a reward of 1.0. In each step that it does not eat the bunny gets a reward of -0.1 to represent increasing hunger.
- Module 2: AvoidWolf. The bunny agent must avoid the wolf. Meeting the wolf gives the bunny a reward of -1.0. In each time step that the bunny avoids the wolf the bunny receives a reward of 0.1.
- Agent's overall goal (implemented in arbitrator): LiveLongProsper – get as much food per time step as possible, which will require balancing food finding with wolf avoidance. The arbitrator's reward function is 0.0 for meeting the wolf, 1.0 for finding food, and 0.5 for each step in which the wolf is avoided but no food is eaten. This is the same as the score used to evaluate algorithm

performance (discussed below). Using the score makes sense because the score is the overall goal of the agent.

To facilitate comparison between Arbi-Q and GM-Sarsa we use a performance metric – a score – that is independent of the reward received by any modules or agents as a whole. The learning of the modules is still guided by their reward functions, but an independent score is necessary for comparison between algorithms to avoid coupling their reward scales. The score we use is 0.0 for meeting the wolf, 1.0 for finding food, and 0.5 for each step in which the wolf is avoided but no food is eaten.

We validate Arbi-Q's performance by comparing it with Greatest Mass Sarsa, which is a Q-decomposition algorithm that orders actions by their summed Q-value, $X_a = \sum_j Q_j(s, a)$. We evaluate each algorithm similarly to Sprague and Ballard [21]. We run each learning algorithm for n steps, suspending learning every $n/100$ steps to evaluate performance. Performance is evaluated by running the greedy policy in the world for 1000 episodes and calculating the average score per time step. Each algorithm used a discount rate of 0.9 and ϵ-greedy action selection during training with ϵ linearly discounted from 0.4, as in Sprague and Ballard's experiments.

For baselines, GM and Arbi-Q algorithms used modules with similarly scaled rewards. For robustness validation, we scaled the AvoidWolf module reward by 10 to simulate the swapping out of separately-authored learning modules. We believe that a truly modular arbitrator function should handle such module modification without serious degradation of performance. Otherwise, any time a learning module were modified, the arbitrator, and possibly all the other modules, would need to be modified to ensure compatibility.

3.5 Results

Empirical results show that the performance of GM-Sarsa degrades when the reward scales of the modules are not comparable. The learning curves depicted in Figure 3.5 show that GM-Sarsa bunny agent with incomparable reward scales for its modules converges to a lower score than with comparable rewards.

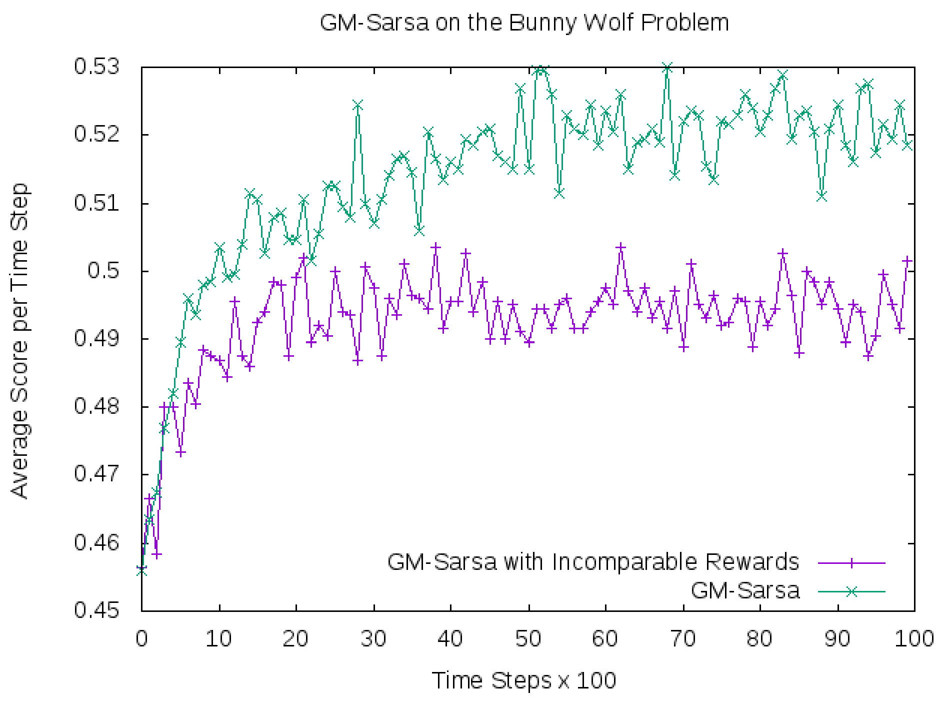

Figure 3.1: Performance of GM-Sarsa/Q-decomposition on the bunny-wolf problem. The learning curves show that Greatest Mass command arbitration degrades significantly when its module rewards are incomparable.

3.5.1 How GM-Sarsa Degrades with Incomparable Rewards

To illustrate how GM-Sarsa degrades when modules have incomparable reward scales, consider a simplified example of the composite Q-values computed by GM-Sarsa with comparable rewards (B is for bunny, F is for food, W is for wolf):

		B	W	F

FindFood with Comparable Reward Scales

With comparable rewards the Q-value of moving right for FindFood would be (we use deterministic state transition dynamics here for simplicity)

$$Q(s, Right) = R(s) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(s', a')$$
$$= -0.1 + 0.9(1.0)$$
$$= 0.8$$

because the max next action would find the food. The value of moving left would be

$$Q(s, Left) = R(s) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(s', a')$$
$$= -0.1 + 0.9(0.8)$$
$$= 0.72$$

because the max next action would be Right, to get closer to the food.

AvoidWolf with Comparable Reward Scales

With comparable rewards the Q-value of moving right for AvoidWolf would be

$$\begin{aligned} Q(s, Right) &= R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a') \\ &= 0.5 + 0.9(-1.0) \\ &= -0.4 \end{aligned}$$

because the next state meets the wolf. The value of moving left would be

$$\begin{aligned} Q(s, Left) &= R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a') \\ &= 0.5 + 0.9(0.5) \\ &= 0.95 \end{aligned}$$

because the max next action would again avoid the wolf.

Composite GM-Sarsa Q-values with Comparable Reward Scales

Given the module Q-values above, the composite Q-values for the Right and Left actions would be

$$\begin{aligned} Q(s, Right) &= 0.8 - 0.4 = 0.4 \\ Q(s, Left) &= 0.72 + 0.95 = 1.67 \end{aligned}$$

.

Given these composite Q-values the next action decided by GM-Sarsa would be Left, which is correct because it avoids getting eaten by the wolf.

FindFood with Incomparable Reward Scales

If we scale the FindFood module's rewards by 10, the Q-values for moving right and left would be

$$\begin{aligned} Q(s, Right) &= R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a') \\ &= -1.0 + 0.9(10.0) \\ &= 8.0 \end{aligned}$$

and

$$\begin{aligned} Q(s, Left) &= R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a') \\ &= -1.0 + 0.9(8.0) \\ &= 6.2 \end{aligned}$$

Composite GM-Sarsa Q-values with Incomparable Reward Scales

Using the same AvoidWolf values as above and the scaled FindFood Q-values using incomparable rewards the composite Q-values would be

$$\begin{aligned} Q(s, Right) &= 8.0 - 0.4 = 7.6 \\ Q(s, Left) &= 6.2 + 0.95 = 7.15 \end{aligned}$$

.

and the bunny would move right and get eaten by the wolf.

This example demonstrates how scaling the FindFood module's rewards causes the preferences of FindFood to dominate action selection, resulting in the bunny getting eaten and not getting to the food.

3.5.2 How Arbi-Q does not Degrade with Incomparable Rewards

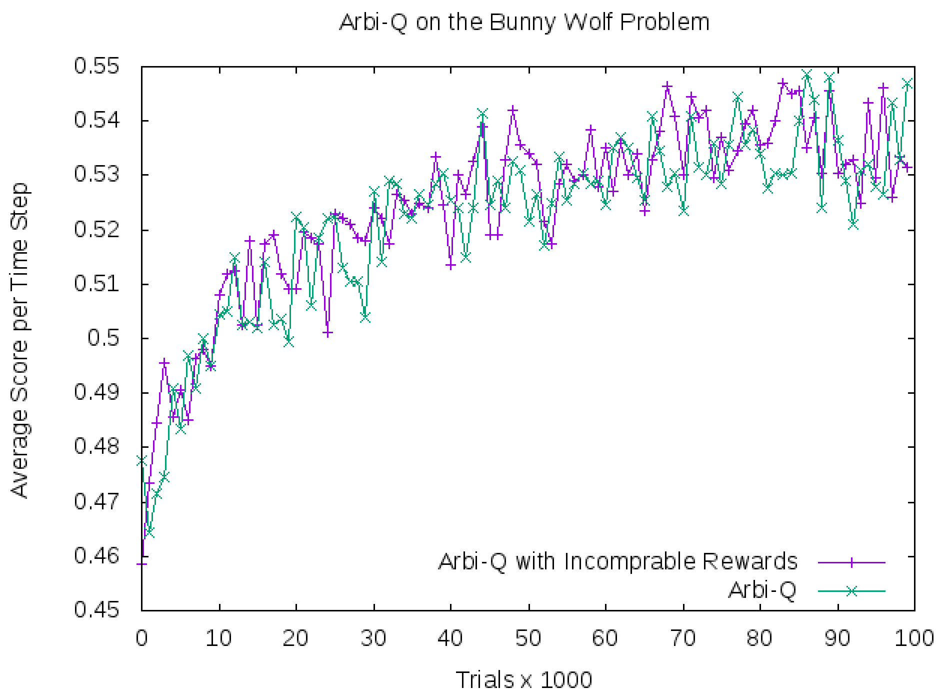

Figure 3.2: Performance of Arbi-Q on the bunny-wolf problem. Arbi-Q converges to similar scores as GM-Sarsa and shows no degradation in performance when modules have incomparable rewards, suggesting that it is amenable to “swappable” modules.

As Figure 3.5.2 shows, Arbi-Q does not exhibit any performance degradation when the agent's modules have incomparable reward scales. Arbi-Q does not use the Q-values of its modules directly. Instead, Arbi-Q learns when it should listen to a particular module. More precisely, Arbi-Q develops a probability distribution for each state which says which module has the best advice in that state. Using the example above with incomparable reward scales, the modules would learn the same local policies using the same Q-values as above, but the arbitrator would learn a policy based on Q-values for selecting modules that chose actions that resulted in particular rewards *for the agent as a whole.* Our LiveLongProsper reward function assigned 0 for getting eaten by the wolf, 1.0 for finding food, and 0.5 for avoiding the wolf but not finding food. The resulting Q-values for the arbitrator for its actions of choosing FindFood and AvoidWolf would be

$$\begin{aligned} Q(s, FindFood) &= R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a') \\ &= 0.5 + 0.9(0) \\ &= 0.5 \end{aligned}$$

and

$$\begin{aligned} Q(s, AvoidWolf) &= R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a') \\ &= 0.5 + 0.9(0.5) \\ &= 0.95 \end{aligned}$$

and in this state Arbi-Q would delegate to the AvoidWolf module, which would move left, which is correct.

So Arbi-Q learns that when the wolf is close AvoidWolf should decide the bunny agent's action, and when the wolf is comfortably distant FindFood should decide the bunny agent's action.

3.6 Related Work

Zhang, Song and Ballard [33] build on Sprague's and Ballard's earlier work on GM-Sarsa to develop three algorithms for deriving a global policy from independent modules. Their first algorithm is simply GM-Sarsa/Q-decomposition. The second chooses the action of the module with the greatest "weight", where the weight of a module is defined as the standard deviation of the modules Q-values for a state. The intuition is that a module with higher standard deviation more strongly prefers its max action to alternative actions. Their third algorithm is based on voting. Each module votes for its optimal action, but the vote is weighted by the weight measure calculated from its Q-values' standard deviation. All of these algorithms still rely on the internal Q-values of the modules and thus require comparable reward scales.

Rohanimanesh and Mahadevan extended the options HRL framework to concurrent settings in which multiple agents executing multiple simultaneous actions [34, 35]. Their work differs from ours in that their framework applies to a single agent taking multiple actions or multiple agents taking simultaneous actions, whereas we are concerned with a single agent executing a single action that is decided by multiple reinforcement learning modules.

Marthi and colleagues [19] suggest extending their work in concurrent ALisp to include the Q-decomposition algorithm of Russel and Zimdars [20], but this line of research was not pursued. Lau and colleagues developed a modular reinforcement learning system that uses a central coordinator for multiple concurrent MPDs [36]. Lau's work differs form ours in that they develop a constraint system in the central coordinator that limits the allowable actions of the component reinforcement learners, thereby constraining their learning. Our approach does not require the arbitrator to know details of component learners, and component learners require no explicit or implicit knowledge of the arbitrator or the other components.

Due to the curse of dimensionality, abstraction of various kinds has long been an active area of research in reinforcement learning. One thread in abstraction is to use examples to guide abstraction. Zang and colleagues used examples of nearly optimal action sequences, or trajectories, to dynamically discover options from data, delivering speedups of up to 30 times in some cases [37]. Learning from demonstration [38] uses human input to improve reinforcement learning performance. Zang and colleagues developed a value function approximation algorithm that leveraged human input to speed convergence for function approximation-based reinforcement learning algorithms [39]. Cobo Rus and colleagues' Abstraction from Demonstration technique uses human demonstrations to infer state abstractions and builds policies based on those state abstractions [40, 41, 42].

Another thread in abstraction seeks to use models from the physical world to create abstractions of (simulated) physical state spaces. Cobo Rus and colleagues created abstractions of state spaces by organizing state spaces into classes of objects and using non-optimal Q-functions to estimate the risk of ignoring certain classes of objects. Cobo Rus's Object-Focused Q-Learning (OFQ) achieved exponential speedups in some cases [43]. Scholz and colleagues developed Physics-Based Reinforcement Learning [44], which uses computational physics engines such as Box2D [45] as model representations, resulting in more sample-efficient learning compared to traditional object-oriented MDP approaches. Physics-based reinforcement learning was then applied successfully in robotic mobile manipulation [46] and robot navigation [47] applications.

3.7 Conclusion

In a software engineering sense, modularity means compositionality and reusability in different contexts. To our knowledge, no other approach to modular reinforcement learning permits general reuse of separately authored modules due to the requirement of reward scale comparability. In addition to state and reward function abstraction, the primary contribution of our reformulation of MRL and the Arbi-Q command arbitration algorithm is the reusability afforded by reward scale decoupling. The Arbi-Q command arbitration algorithm makes it possible for modules written by different programmers with different reward scales to be used together within the same modular reinforcement learning agent. As we will see in the next chapter, reuse is an essential part of modern software engineering. With our MRL framework and the Arbi-Q algorithm it is now possible to integrate modular reinforcement learning into *practical* programming systems.

CHAPTER 4
BACKGROUND IN SOFTWARE ENGINEERING

Software engineering is the process of creating software that is correct, reliable, and maintainable. This chapter provides background in software engineering that relates to our work. In particular, we claim in the next chapter that AFABL provides two benefits that have been central issues in software engineering: AFABL facilitates reuse and reduces complexity. Here we discuss the issue of reuse in software engineering and relate reuse to domain-specific languages, then briefly discuss complexity measurement in software engineering. Finally, we close with a discussion of adaptive programming.

4.1 Software Reuse

Software reuse was identified as a primary tool in improving software engineering practice since the birth of the field of software engineering in 1968 [48]. Software reuse means using an existing software artifact in a new software system, ideally without modifying the original artifact [49, 50]. Reusable artifacts may be source code libraries, components, programming languages, and application frameworks [51] as well as concepts such as software schemas, architectures, and design patterns. The benefits of software reuse seem obvious, and empirical studies have indeed shown that reuse reduces defect rates, reduces refactoring costs, and increases productivity [52, 53]. In the next chapter we also quantify the reduction in code complexity afforded by the AFABL DSL. DSLs, as we discuss below, are a particular kind of reusable artifact.

Krueger presents a useful framework for understanding and assessing reuse techniques. Of particular interest to our work, he discusses reuse techniques in terms of cognitive distance, which he defines as an intuitive measure of the effort required to use a reusable software artifact in the process of turning the concept of a software application into a working system. The smaller the cognitive distance between a reusable software artifact and the concept of the application program in which it is to be reused, the more successful the reuse. Thus, abstraction is crucial to software reuse, the higher the level of abstraction the better. Krueger proposes three techniques for minimizing the cognitive distance in reusable software artifacts: "(1) using fixed and variable abstractions that are both succinct and expressive, (2) maximizing the hidden parts of the abstractions, and (3) using automated mappings from abstraction specifications to abstraction realizations" [49]. Krueger was the first to recognize that high level languages such as C and Java are themselves examples of software reuse – language constructs are abstraction specifications, assembly language or byte code are abstraction realizations. DSLs (or VHLLs – Very High Level Languages – as he called them) are also examples of software reuse that raise the level of abstraction even higher, offering abstraction specifications that are entities in some problem domain such as set theory or circuit design. In the next chapter we will analyze AFABL according to Krueger's framework.

Gacek argues for creating domain-specific reference architectures to facilitate reuse[54]. A domain-specific application architecture identifies all of the components that comprise a software application for a particular domain and the interactions between the components. A domain-specific language can be seen as a domain-specific architecture, whose components are modeled as language abstractions. AFABL, in this sense, is a domain-specific architecture for agents with multiple continuing goals.

4.1.1 Domain-Specific Languages

A domain-specific language (DSL) is a language that provides constructs and semantics tailored to a specific problem domain. Specialized programming languages were already widespread when Landin proposed the first unified framework for designing domain specific languages in 1966 [55]. In Landin's framework the design of a DSL consists of two independent parts: the written form of the language, and the kinds of abstractions that can be expressed in the language. Every language has an abstract syntax, axiomatization, and an underlying abstract machine. Today many DSLs are in widespread use for various application domains such as typesetting and manuscript preparation (TeXand LaTeX), circuit design (VHDL), web page authoring (HTML and CSS), data exchange (JSON and XML), and many more.

Perhaps the most successful DSL, with which every reasonably literate software engineer or computer scientist is familiar, is Structured Query Language (SQL). SQL was originally presented as SEQUEL in 1974 by Chamberlin and Boyce of IBM Research [56]. Today SQL is used in all significant relational databases and its ANSI/ISO standard is on its third version. SQL has succeeded so completely because it provides exactly the right abstractions and semantics for using relational databases.

Domain-specific languages provide two primary benefits in software engineering: improving programmer productivity and improving communication with domain experts [57]. In Chapter 5 we show that AFABL improves programmer productivity by reducing the effort required to write agents and reducing the complexity of agent code. In Chapter 8 we present an application of AFABL to the domain of personality modeling in psychology to demonstrate AFABL's usefulness as a tool for bridging specialist knowledge from a non-computing domain with computational models.

Hudak argued that a domain-specific language is the "ultimate abstraction," providing abstractions and semantics tailored to a particular application domain, but that languages are typically difficult to implement and difficult to evolve as the domain is better understood and changes need to be made to the language [58]. To solve this problem Hudak argued for and demonstrated domain-specific *embedded* languages, that is, DSLs embedded in a general-purpose host language, inheriting the tooling, syntax, and semantics of the host language while adding domain-specific constructs. Hudak used Haskell and showed how higher-order typed languages were particularly well-suited for hosting DSLs [59]. For AFABL we used Scala, which has similarities to Haskell, as we discuss in the next chapter.

Ward [60] proposed Language Oriented Programming as a way of organizing software development. Instead of developing reusable libraries in a general purpose language the software engineer develops a formal domain-specific specification of the application, then implements this specification as a DSL. He calls the resulting software development process "middle-out" development, where the DSL is created first (the middle layer) then the DSL is implemented (the lower layer) and specific applications are developed using the DSL (the upper layer). He presents several examples of this approach, including LaTeX[61] as a collection of TeX[62] macros and Emacs [63], which is essentially a Lisp interpreter with addressable memory buffers – what users think of as the editor is actually implemented as a collection of Emacs Lisp functions, making Emacs infinitely extensible. Neighbors was the first to propose *domain languages* specifically for the purpose of reuse [64]. Lorenz and colleagues tie together the concept of reuse with Language Oriented Programming in software engineering [65], comparing the effort and benefits of implementing and using internal versus external DSLs. They performed a case study in which they created an external DSL (using a language workbench system called Meta-Programming System MPS [66]) and internal DSL using the experimental LOP language Cedalion [67] for the same task (calculator software product line). They found that both approaches achieved their code reuse goals but, while both DSLs took similar effort to use, the external

DSL took four times longer to implement. They concluded that internal DSLs should be favored over external DSLs for most software development projects. Rosenan argues that host languages themselves can be designed with internal DSL creation in mind – a kind of "Language-Oriented Programming language" – and presents a LOP language as a proof of concept [67]. As we discuss in the next chapter, while not explicitly advertised as an LOP language, hosting DSLs has been a design goal of the Scala language that we use to host AFABL.

There are downsides to using DSLs in application development. Creating a DSL takes time and effort. If productivity is a concern, then a DSL should only be created if it will be used in a sufficient number of projects that the sum of the cost savings in those projects outweighs the cost of creating the DSL. Related this this concern, implementing DSLs, whether internal or external, requires significantly more programming expertise than application or even traditional library development. Finally, DSLs require programmers to learn the DSL, and for external DSLs a new tool chain. Internal DSLs that needlessly create new syntax often frustrate programmers. For example, in Scala DSL designers can create operators, which are simple method calls, with just about any symbols. Scala's support for internal DSL construction has spurned a tremendous number of DSLs, and there is a tremendous amount of push-back from developers tired of learning odd new operator symbols instead of simply using familiar method calls with readable names.

Writing AFABL as a DSL is justified because (1) for the cost of learning the DSL the programmer is freed from learning details of reinforcement learning algorithms, and (2) AFABL is a shallow DSL using familiar Scala language constructs and idioms.

4.2 Software Complexity

A second goal of AFABL is reducing the complexity of agent code. For our purposes we define complexity as a measure of the effort required to understand, modify, or test a piece of code. Many kinds of complexity measures have been proposed, including function point analysis [68], Halstead's "software science" [69], information flow [70] and many others. But perhaps the most widespread complexity measure is McCabe's cyclomatic complexity [71, 72]. Roughly speaking, the McCabe cyclomatic complexity number is a measure of the number of unique paths through a program. McCabe's metric is a measure of control-flow complexity.

Research in software complexity has continued and evolved in the age of object-oriented programming. Object-oriented programs tend to have far fewer control flow structures, but these are traded for classes, methods, and inheritance hierarchies that contribute in their own ways to the cognitive burden of a programmer trying to use an object-oriented library or maintain an object-oriented system. Chidamber and Kemerer [73] developed a suite of six metrics to capture the complexity of an object-oriented program and analyzed them using Weyuker's software metrics evaluation principles [74]. Briand and colleagues placed Chidamber and Kemerer's work and many other metrics within a framework of property-based measurement which abstracts over language constructs [75]. Basili and colleagues experimentally validated Chimader and Kumarer's metrics on software projects and found their metrics to be useful predictors of software quality [76].

While the modern object-oriented metrics are better suited to modern large-scale software development, they are less well-suited to the study we report in Chapter 6. The Scala-based agent programs only used one class and grew in the number of decision structures to handle more complex agent problems. Even using a general framework like Briand's which allows consideration of helper methods as modules, the Scala agent programs simply used helper methods to thematically group decision structures, so modular complexity simply reduces to McCabe cyclomatic complexity. In the case of AFABL agents, each agent uses the same agent and module classes. Agent's differ in their reward functions, which are typically implemented with if statements. Again, cyclomatic complexity better captures the differences in the complexities of AFABL agent programs. For these reasons, in this work we evaluate program complexity using McCabe's cyclomatic complexity measure.

Though there are critiques of cyclomatic complexity [77] and proposed modifications [74], McCabe's cyclomatic complexity measure is still in widespread use and has been shown to be a useful and valid measure. Curtis and colleagues found that both Halstead and McCabe complexity metrics correlate with psychological complexity of code, especially for less experienced programmers [78]. Finally, McCabe's cyclomatic complexity number has a simple method of calculation which makes it particularly appealing. We explain McCabe's cyclomatic complexity in Chapter 6.

4.3 Adaptive Programming

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 [79].

In this work 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. Due to its integration of reinforcement learning, AFABL provides this kind of adaptivity, making the construction of adaptive agents much easier.

4.3.1 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. AFABL integrates two of Norvig's key technologies: agent technology and reinforcement learning.

4.3.2 The Partial Programming Paradigm

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. AFABL, being a domain-specific language (DSL) [58] embedded in Scala [80, 81], is effectively multi-paradigmatic, supporting functional and object-oriented programming through its direct use of Scala, 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. An AFABL programmer writes a partial agent consisting of modules with states and reward functions, and the action-selection logic of the agent is handled by the integrated reinforcement learning algorithms. Thus 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. AFABL 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 AFABL 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. AFABL 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 AFABL, 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 AFABL's integrated reinforcement learning. This ability to program partial behaviors relieves a great deal of burden from the programmer and greatly simplifies the task of writing adaptive agents. In the next chapter we will see how AFABL implements its support for adaptivity and partial programming.

4.3.3 Related Work in Adaptive Programming

There is already a body of work in integrating reinforcement learning into programming languages, mostly from Stuart Russell and his group at UC Berkeley [82, 18]. Their work is based on *hierarchical reinforcement learning* [16, 14], 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) [82] was integrated into a programming language in the form of a set of Lisp macros (ALisp) [18]. 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 except that AFABL integrates modular, rather than hierarchical reinforcement learning, and we validate the software engineering benefits through a programmer study.

Bauer's Ph.D. work in adaptation-based programming [83] is the closest to ours in its focus on the practical application of adaptive programming. Bauer implemented automated adaptation as a Java library [84] and as a Haskell embedded DSL [85]. These systems used Q-learning internally but did not use modular reinforcement learning. Bauer also did not conduct empirical software engineering studies of programmers to quantify and qualify the benefits of integrating reinforcement learning into a programming language. In the next chapter we present our language, AFABL, which integrates modular reinforcement learning and report the results of a programmer study that demonstrates its value.

CHAPTER 5
AFABL: A FRIENDLY ADAPTIVE BEHAVIOR LANGUAGE

AFABL is an internal domain-specific language (DSL) shallowly embedded in the Scala programming language. In this chapter we explain why we chose to implement AFABL as a Scala-embedded DSL, present the basic elements of AFABL with examples, and report the results of a programmer study which confirm and quantify the usefulness of integrating reinforcement learning into a programming language.

5.1 Why an embedded DSL?

We chose to implement AFABL as a shallowly embedded domain-specific language because of the exploratory nature of this research. Our goal at this point is to confirm our expectation that integrating reinforcement learning is useful to programmers writing agents, to explore the nature of this integration, and to get qualitative feedback from programmers on their experience using a language that integrates reinforcement learning. Writing a full language with its own lexical and syntactic structure, internal representations, and tools (interpreters, compilers, linkers, etc.) would distract from the core questions we are trying to answer. As we discuss in Section 9.3.4, creating a full independent language is a direction for future research which will be guided by the results we present here, for which an embedded DSL is sufficient.

5.2 Why Scala?

Hosting DSLs is a primary design goal of the Scala programming language. Scala is an expressive and concise language which already enables the expression of domain models with little syntactic baggage. By employing just a few Scala language features that are designed for writing expressive and convenient libraries, we can create a DSL that Scala programmers will find familiar and non-Scala programmers can use to encode adaptive agents. Indeed, the nature of Scala's syntax and language features is such that many Scala libraries that are not explicitly labeled as DSLs qualify as shallow embedded DSLs. In Scala a DSL is simply a library which takes advantage of Scala's language features and idioms that makes code using the library look like a custom language.

5.3 AFABL Concepts

AFABL is a language for encoding adaptive (intelligent) agents. Before we present the AFABL language, we review basic adaptive agent concepts that can be encoded in AFABL.

5.3.1 Agent Architecture

An AFABL agent is a behavioral agent that is composed of reusable behavior modules. We use behavioral agent in the sense common in agent programming literature [86] – an agent receives a percept from the environment and executes an action in response. Each behavior module is itself an agent that has a preferred action for each state. An AFABL agent performs command arbitration to choose one of its modules' recommended actions for each state. The behavior modules recommend actions in each state, and the arbitrator chooses which module to ”listen to” in each state.

5.3.2 Behavior Modules

Behavior modules, sometimes called subagents in the modular reinforcement learning literature, are agents that are meant to be combined to form larger agents. Behavior modules are similar to the layers of Brooks's subsumption architecture with an important difference: autonomy. The internal working of a behavior module is never altered externally. A behavior module defines a state abstraction that converts the state observation it is given to a (possibly) simpler state that is used internally for decision making and learning. Module state abstractions that are simpler (contain fewer features) than the global state help to speed the convergence of the underlying reinforcement learning algorithms. The decision making and adaptation mechanisms inside a module remain completely under the module's control. Interaction with the module consists entirely of reporting a state observation to the module, asking the module for an action, and reporting to the module the effect of executing an action.

5.3.3 Adaptive Modules

An adaptive module employs learning algorithms under the hood to achieve automatic adaptivity. By adaptive we mean two things: (1) adaptation to new worlds, and (2) run-time adaptation. A module that is programmed to work for worlds with a given state representation will work with any world that provides the same state representation, even if the dynamics of the worlds differ. An adaptive module need simply be retrained for the new world. Once an adaptive module is running in an active agent, the module may continue to tune its internal learning models as the agent acts in the world, providing for run-time adaptation.

5.3.4 Command Arbitrators

A command arbitrator takes as input the state of the world and the action preferences of a set of modules, and selects one of the modules or actions to be executed by the agent.

5.4 The AFABL Language

AFABL is a DSL implemented as a library in the Scala programming language designed for writing adaptive agents. An AFABL agent operates in a world, is composed of one or more modules, and has an agent level reward function that it uses to learn a command arbitration policy.

5.4.1 Worlds

Every AFABL module and agent is designed to operate in a world. A world defines the states, actions and state transition dynamics for a given set of states and actions. Details are discussed below.

States

The states of a world can be represented with any kind of Scala class. Case classes are good for representing states because of their concise syntax and built-in equality methods. Figure 5.4.1 shows a case class for a state with three state variables: the locations of a bunny, wolf, and food.

```scala
case class Location(x: Int, y: Int)

case class BunnyState(
  bunny: Location,
  wolf: Location,
  food: Location
)
```

Figure 5.1: Scala code to represent states in the bunny world.

Actions

Actions are represented by objects which can be instances of any class. As with states, case classes make a good choice for implementing actions. Figure 5.4.1 shows actions for the bunny world implemented as a Scala enumeration.

```scala
object BunnyAction extends Enumeration {
  val Up = Value("^")
  val Down = Value("v")
  val Left = Value("<")
  val Right = Value(">")
}
```

Figure 5.2: Scala code to represent the actions that the bunny agent can take in the bunny world.

World Dynamics

An agent executes actions in a world, and those actions potentially change the state of the world. Having defined Scala representations for states and actions, we can define a world. Figure 5.4.1 shows the abstract class which defines the basic interface of world objects, which are instances of subclasses of `World`. As we discuss in Sections 5.4.2 and 5.4.3, all modules and agents are defined to act in a particular instance of a world. As with states and actions, world representations make no advanced use of the Scala programming language.

```scala
abstract class World[S, A] {
  def init(): S
  def resetAgent(): S
  def states: Seq[S]
  def actions: Seq[A]
  def act(action: A): S
}
```

Figure 5.3: The abstract superclass of all world classes for AFABL agents.

Figure 5.4.1 shows some of the code for the Bunny World.

```
class BunnyWorld(val width: Int = 5, val height: Int = 5)
  extends World[BunnyState, BunnyAction.Value] with LazyLogging {

  // Initialize the world state
  var state = init()

  // In Scala, defs can be overridden with vals
  val states = {
    // Calculate every possible combination of locations for the
    // bunny, wolf, and food
  }

  // This line returns all the values of the BunnyAction enumeration
  val actions = BunnyAction.values.toSeq

  def init(): BunnyState = {
    // Calculate initial locations for the bunny, wolf, and food.
    //
  }

  def resetAgent(): BunnyState = {
    // "Respawn" the bunny at a new location, update the world state
    // and return the new state
  }

  def act(intendedAction: BunnyAction.Value): BunnyState = {
    // Code to calculate the actual action due to uncertainty in the
    // environment and update the state of the world based on the
    // actual action.
  }
  // Helper functions ...
}
```

Figure 5.4: Parts of the bunny world class showing important aspects of the implementation of the `World` abstract class.

5.4.2 Modules

Figure 5.4.2 shows the complete code for an AFABL implementation of a behavior module that represents the goal of finding food. First is the definition of a case class, `FindFoodState`, to represent the state abstraction for FindFood modules. `FindFoodState` includes only two of the three state variables in the bunny world.

```
case class FindFoodState(bunny: Location, food: Location)

val findFood = AfablModule(
  world = new BunnyWorld,

  stateAbstraction = (worldState: BunnyState) => {
    FindFoodState(worldState.bunny, worldState.food)
  },

  moduleReward = (moduleState: FindFoodState) => {
    if (moduleState.bunny == moduleState.food) 1.0
    else -0.1
  }
)
```

Figure 5.5: AFABL code for a module that represents the goal of constantly finding food.

```
case class FindFoodState(bunny: Location, food: Location)
```

We store a reference to an `AfablModule` for FindFood in `findFood`.

```
val findFood = AfablModule(
```

The `AfablModule` factory method takes three arguments: an instance of a `World` that the module can act and learn in, a `stateAbstraction` function, and a `moduleReward` function.

The first argument to `AfablModule` is the world:

```
world = new BunnyWorld
```

The `world` and `=` are optional, but if included must be verbatim, i.e., considered part of the AFABL language.

The second argument is a state abstraction function that takes a world-state object as a parameter and returns an instance of our state abstraction class:

```
stateAbstraction = (worldState: BunnyState) => {
  FindFoodState(worldState.bunny, worldState.food)
}
```

The `stateAbstraction` and `=` are optional, but if included should be considered part of the AFABL language. `worldState` is a user-chosen name, `BunnyState` must match the state type defined for the world in which the module and agent operate, in this case it is the first type parameter to `World` in the `BunnyWorld` code in Figure 5.4.1. The last expression in the body of the `stateAbstraction` function must be an instance of a module state, in this case `FindFoodState`.

The third and final argument to the `AfablModule` factory method is a module reward function that takes an instance of our state abstraction class and returns the reward this module receives for being in that state:

```
moduleReward = (moduleState: FindFoodState) => {
  if (moduleState.bunny == moduleState.food) 1.0
  else -0.1
}
```

The `moduleReward` and `=` are optional, but if included should be considered part of the AFABL language. `moduleState` is a user-chosen name, but the parameter type, `FindFoodState` in this example, must match the return type of the `stateAbstraction` function. The last expression in the body of the `moduleReward` function must be a `Double` value. In this case, which is typical, the body of the `moduleReward` function is an `if` expression which simply returns the reward based on state predicates. This example is another case where we could have implemented DSL-specific syntax, such as a list of predicates and values, but the syntactic overhead of Scala's `if` expression is minimal and the code is crystal clear to any Scala programmer.

These three components – world, state abstraction and module reward – define a module specific learning problem on a subset of the world in which the module (and agent containing the module) may act. Internally, AFABL uses these components to instantiate a Sarsa learning algorithm using the algorithm parameters discussed in Chapter 3, but the programmer need not be aware of any details of reinforcement learning *algorithms*. The AFABL programmer need only be familiar with the reinforcement learning *problem*.

This module example shows the value of splitting the world dynamics from the agent module's reward function. We can think of the world and the agent independently. In essence, the definition of a full MDP is split across the definition of a world, and the definition of an agent that acts in that world.

Here we also begin to see syntactic conveniences afforded by the AFABL DSL. There are only two type annotations and one control structure (in the reward function). The rest of the types are inferred by Scala's type inferencer thanks to the way we wrote the factory method that creates `AfablModule`s. It's worth noting that we could have refined the DSL to further strip the few Scala syntactic artifacts (like the `if` statement and the anonymous functions for `stateAbstraction` and `moduleReward`) but the syntactic overhead is minimal and there is a tradeoff between writing specific DSL syntax and using Scala's built-in syntax directly. Creating unique syntax for a DSL imposes cognitive burden on programmers who are proficient in the host language. The benefit of the unique syntax must outweigh this cognitive burden. Here we hope to strike the right balance between convenient domain-specific syntax and familiarity to programmers. Thanks to Scala's already concise and expressive language and idioms, although this looks like a DSL there is no special syntax in this example. This code is a good example of shallow DSL embedding.

5.4.3 Agents

An AFABL agent is an agent that acts in a particular world, is composed of independent behavior modules pursuing their own continuing goals, and has a central command arbitrator that uses an agent level reward function to learn when it should listen to each module. As the code in Figure 5.4.4 shows, an AFABL agent allows programmers to express these components concisely, with very little cognitive distance between the concepts that make up the agent and the code that represents them. As with modules, the `AfablAgent` class uses features of the Scala programming language to make the syntax more convenient. For example, there is only one explicit type annotation in the AFABL bunny agent code in Figure 5.4.4, but behind the scenes a carefully written factory method in the companion object allows Scala's static type inferencer to infer type parameters of the `AfablAgent` constructor, return types for anonymous functions, and assign a concrete type value to a path-dependent abstract type variable. Figuring out all this stuff and wrestling with Scala's type checker directly is not easy. Writing an AFABL agent is easy.

```
val bunny = AfablAgent(
  world = new BunnyWorld,

  modules = Seq(findFood, avoidWolf),

  agentLevelReward = (state: BunnyState) => {
    if (state.bunny == state.wolf) 0.0
    else if (state.bunny == state.food) 1.0
    else 0.5
  }
)
```

Figure 5.6: An AFABL agent that acts in a world, contains behavior modules, and has an agent level reward.

5.4.4 A Complete AFABL Bunny

A complete bunny agent using the AFABL DSL is shown in Figure 5.4.4. This code would typically fit in a single editor window and represents a tremendous amount of functionality. This agent pursues two goals simultaneously and prioritizes them based on the relative locations of the bunny, the food, and the wolf.

```
val bunnyWorld = new BunnyWorld

case class FindFoodState(bunny: Location, food: Location)
val findFood = AfablModule(
  world = bunnyWorld,
  stateAbstraction = (worldState: BunnyState) => {
    FindFoodState(worldState.bunny, worldState.food)
  },
  moduleReward = (moduleState: FindFoodState) => {
    if (moduleState.bunny == moduleState.food) 1.0
    else -0.1
  }
)

case class AvoidWolfState(bunny: Location, wolf: Location)
val avoidWolf = AfablModule(
  world = bunnyWorld,
  stateAbstraction = (worldState: BunnyState) => {
    AvoidWolfState(worldState.bunny, worldState.wolf)
  },
  moduleReward = (moduleState: AvoidWolfState) => {
    if (moduleState.bunny == moduleState.wolf) -0.1
    else 0.1
  }
)

val bunny = AfablAgent(
  world = new BunnyWorld,
  modules = Seq(findFood, avoidWolf),
  agentLevelReward = (state: BunnyState) => {
    if (state.bunny == state.wolf) 0.0
    else if (state.bunny == state.food) 1.0
    else 0.5
  }
)
```

Figure 5.7: A complete bunny agent in the AFABL DSL. Code for the modules is repeated from previous figures to give a sense of the full quantity of code required to write an agent with two behavior modules.

5.5 Conclusion

In the next Chapter we provide a quantitative evaluation of the benefits of integrating reinforcement learning into a programming language.

CHAPTER 6

AFABL PROGRAMMER STUDY

AFABL supports a declarative style in which the agent programmer specifies which states are desirable and undesirable, but not how the agent should choose actions to pursue or avoid those states. Action selection logic is derived automatically by reinforcement learning algorithms that the AFABL programmer never sees. AFABL also provides agent-based software abstractions that permit code to be reused in new domains. This reuse is one of the things we mean by adaptivity: existing AFABL code can adapt to new domains without modifying the code. In this chapter we report the results of a programmer study to quantify the value of AFABL's agent programming abstractions.

6.1 Experiments

Programmers were randomly assigned to two equally-sized groups based on their demographics (see Appendix A for details): one group used Scala without AFABL first – the Scala-first group – and the other group used AFABL first – the AFABL-first group. Each group completed two programming tasks using Scala and AFABL in the order determined by their group. For each task the programmers were asked to write elegant code that meets the requirements of the task as quickly as possible, balancing the quality of their solutions with time. The idea was to get a good solution quickly, not a perfect solution in a long time.

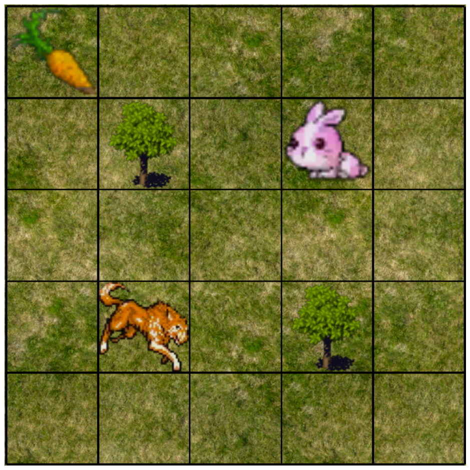

Figure 6.1: In the grid world above, the bunny must pursue two goals simultaneously: find food and avoid the wolf. The bunny may move up, down, left, or right. When it finds food it consumes the food and new food appears elsewhere in the grid world, when it meets the wolf it is eaten and "respawns" elsewhere.

6.1.1 Task 1: Bunny-Food-Wolf

In this task each programmer wrote an agents that control a bunny character in a simple world, depicted in Figure A.1. The bunny world works as follows:

- The bunny world is a discrete grid of cells. The bunny, wolf, and food each occupy one cell.
- During each time step the bunny may move north, south, east, or west – this is the bunny agent's action set.
- Every two time steps the wolf moves towards the bunny.
- If the bunny moves to the cell currently occupied by the food, the agent should be written to recognize this fact and give the agent an appropriate reward signal. In any case the simulation assumes food is "eaten" and new food appears elsewhere.
- If the wolf moves to the cell currently occupied by the bunny it eats the bunny

and the bunny "respawns" in a new location.

Programmers were asked to write bunny agents that meet the wolf as little as possible and eat as much food as possible. The bunny's percepts are complete state descriptions: the locations of the bunny and the wolf.

6.1.2 Task 2: Mating Bunny

In this task each programmer wrote a bunny agent for a world that is identical to the world in Task 1 except that the bunny must also find mates. This world includes one static potential mate that behaves similarly to the food. When the bunny finds the potential mate, the simulation assumes that the bunny has "mated," the mate disappears, and another potential mate appears elsewhere. The simulation runs as in Task 1, and the scorer additionally keeps track of how many mates the bunny finds. As in Task 1, programmers were asked to write bunny agents that meet the wolf as little as possible, eat as much food as possible, and find as many mates as possible. As in Task 1, the bunny's percepts are complete state descriptions: the locations of the bunny, the wolf and the mate.

6.1.3 Provided Code

Study participants were given starter code so they could focus on writing the behavior of their agents. We provided the world for each task and the files in which to write their code. Participants were also given general documentation for AFABL but assumed to already be proficient in Scala.

Figure 6.1.3 shows the code given to participants for the Scala bunny on Task 1. Figure 6.1.3 shows the code given to participants for the AFABL bunny on Task 1. The `BunnyWorld`, `BunnyState`, and `BunnyAction` classes were also provided. It was up to participants to write state abstraction classes if they chose to do so.

```
class ScalaBunny1 extends Agent[BunnyState, BunnyAction.Value]
    with Task1Scorer {

  // Your code goes in the body of this method. This method defines
  // your agent's behavior, that is, what action it takes in a given
  // state. The last expression in this method must be a
  // BunnyAction.  You may create as many helper functions as you
  // like, but please do not alter any of the provided code.
  def getAction(state: BunnyState, shouldExplore: Boolean = false) = {

    // This is a stub to make the code compile. Please
    // replace this with your code.
    BunnyAction.Up
  }
}
```

Figure 6.2: Starter Scala code provided to participants for Task 1.

```
object AfablTask1 {

  // Use this val in your agent definitions.
  val bunnyWorld = new BunnyWorld

  // Please place all of your AFABL code for Task 1 in this singleton
  // object.


  // Your solution must assign your AFABL bunny agent for Task 1 to
  // the val afablBuny1.
  val afablBunny1 = ???
}
```

Figure 6.3: Starter AFABL code provided to participants for Task 1.

The provided code for Task 2 was identical to the provided code for Task 1, except for the names of the files. For Task 2 participants were encouraged to copy code from Task 1 if they found it helpful, or to use any objects defined for Task 1 that would be helpful, such as behavior modules. As we discuss below, reusing code was straightforward for the AFABL agents but not for the Scala agents. As in Task 1, it was up to participants to write state abstraction classes if they chose to do so, but they could reuse any state abstraction classes written for Task 1.

Each task had a main method which ran the agents in the world to evaluate their performance.

6.2 Quantitative Analysis

We analyzed the submissions of study participants to compare Scala agents to AFABL agents in terms of code size, time spent writing Scala versus AFABL agents, the complexity of Scala versus AFABL agent code, and the performance of the agents on the assigned tasks.

6.2.1 Code Size

The size of a code base is often correlated with the level of effort required to write or understand the code. We computed the number of lines of code for each agent, not including comments.

6.2.2 Time

Study participants used the IntelliJ IDEA IDE with a plug-in that we wrote especially for this study. The plug-in recorded timestamps each time the editor tab with Task 1 or Task 2 files gained or lost the focus. We processed these logs to add up the time deltas between gaining and losing focus as an indication of the time programmers spent writing the code for each bunny agent.

6.2.3 Cyclomatic Complexity

We computed a complexity measure for all the submitted bunny agents. For Scala code we employed the simplified McCabe cyclomatic complexity measure [71]:

$$v = \pi + 1 \tag{6.1}$$

where v is the complexity score and π is the number of predicates in decision structures.

6.2.4 Performance

Each programmer's Scala bunny and AFABL bunny were run for 1000 time steps and their average scores recorded. The score is based on how much food the bunny eats and how many times the bunny is eaten by the wolf for Task 1, and additionally how many times the bunny mates for Task 2. The score is normalized by the number of steps to indicate a sort of "happiness index," a ratio of rewards to lifespan. This score happens to correspond to the measure used to validate Arbi-Q in Chapter 3 to facilitate comparison between programmer authored agents and the performance of the algorithms we implemented for our reformulated MRL. It is important to note, however, that our aim here is not to achieve optimal performance but to create a programming system that makes it easy to write agents that achieve good performance.

6.2.5 Typical Task 1 Submissions

Figure 6.2.5 shows a typical Scala submission for Task 1. The action selection strategy is about as simple as possible. For example, there is no determination of where the wolf is in relation to the bunny and food other than distance. If the wolf is closer to the food than the bunny, move away from the wolf, otherwise move toward to the food. The agent does not distinguish between cases where the wolf is between the wolf and the food, or if the wolf is closer but sufficiently far away. One could imagine writing code to calculate the projection of the wolf's position onto the line between the bunny and the food to determine a safe closure rate for the wolf. However, this simple strategy is all that is needed to achieve nearly optimal results. This Scala-only bunny agent achieves an average performance score of 0.54, the same as the AFABL bunny.

```
class ScalaBunny1 extends Agent[BunnyState, BunnyAction.Value]
    with Task1Scorer {

  def getAction(state: BunnyState, shouldExplore: Boolean = false) = {
    if (wolfNearFood(state))
      moveAwayFromWolf(state)
    else
      moveTowardFood(state)
   }

  def wolfNearFood(state: BunnyState) = {
    val wolfToFood = sqrt(pow(state.food.x - state.wolf.x, 2) +
                          pow(state.food.y - state.wolf.y, 2))
    val bunnyToFood = sqrt(pow(state.food.x - state.bunny.x, 2) +
                           pow(state.food.y - state.bunny.y, 2))
    wolfToFood < bunnyToFood
  }

  def moveTowardFood(state: BunnyState) = {
    if (state.food.x > state.bunny.x)
      BunnyAction.Right
    else if (state.food.x < state.bunny.x)
      BunnyAction.Left
    else if (state.food.y < state.bunny.y)
      BunnyAction.Up
    else
      BunnyAction.Down
  }

  def moveAwayFromWolf(state: BunnyState) = {
    if (state.wolf.x < state.bunny.x)
      BunnyAction.Right
    else if (state.wolf.x > state.bunny.x)
      BunnyAction.Left
    else if (state.wolf.y > state.bunny.y)
      BunnyAction.Up
    else
      BunnyAction.Down
  }
}
```

Figure 6.4: Typical Scala submission for Task 1.

Figure 6.2.5 shows a typical AFABL submission for Task 1. As in our earlier examples, the AFABL bunny is composed of behavior modules for finding food and avoiding the wolf. The AFABL documentation contained tips for reward authoring in modules and at the agent level.

```
case class FindFoodState(bunny: Location, food: Location)
val findFood = AfablModule(
  world = bunnyWorld,
  stateAbstraction = (worldState: BunnyState) => {
    FindFoodState(worldState.bunny, worldState.food)
  },
  moduleReward = (moduleState: FindFoodState) => {
    if (moduleState.bunny == moduleState.food) 1.0
    else -0.1
  }
)

case class AvoidWolfState(bunny: Location, wolf: Location)
val avoidWolf = AfablModule(
  world = bunnyWorld,
  stateAbstraction = (worldState: BunnyState) => {
    AvoidWolfState(worldState.bunny, worldState.wolf)
  },
  moduleReward = (moduleState: AvoidWolfState) => {
    if (moduleState.bunny == moduleState.wolf) -0.1
    else 0.1
  }
)

val afablBunny1 = AfablAgent(

  world = bunnyWorld,

  modules = Seq(findFood, avoidWolf),

  agentLevelReward = (state: BunnyState) => {
    if (state.bunny == state.wolf) 0.0
    else if (state.bunny == state.food) 1.0
    else 0.5
  }
)
```

Figure 6.5: Typical AFABL submission for Task 1.

The AFABL solution to Task 1 contains 31 lines of code and has a cyclomatic complexity of 5 (4 predicates in decision structures). The Scala solution to Task 1 uses 34 lines of code and has a cyclomatic complexity of 8 (7 predicates in decision structures). Both programs achieve the same nearly optimal level of performance with scores of 0.54. Optimal performance was determined by running GM-Sarsa/Q-decomposition on the problem, which has been shown by Russell and Zimdars to be optimal [20].

6.2.6 Typical Task 2 Submissions

```scala
class ScalaBunny2 extends Agent[BunnyState, BunnyAction.Value]
    with Task2Scorer {

  def getAction(state: BunnyState, shouldExplore: Boolean = false) = {

    if ((distance(state.wolf, state.food) < distance(state.food, state.bunny))
      || distance(state.wolf, state.mate) < distance(state.mate, state.bunny))
      moveAwayFromWolf(state)
    else if (distance(state.bunny, state.food) < distance(state.bunny,
    state.mate))
      moveToward(state.bunny, state.food)
    else
      moveToward(state.bunny, state.mate)
  }

  def distance(a: Location, b: Location) = {
    sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2))
  }

  def moveToward(from: Location, to: Location) = {
    if (to.x > from.x)
      BunnyAction.Right
    else if (to.x < from.x)
      BunnyAction.Left
    else if (to.y > from.y)
      BunnyAction.Up
    else
      BunnyAction.Down
  }

  def moveAwayFromWolf(state: BunnyState) = {
    if (state.wolf.x < state.bunny.x)
      BunnyAction.Right
    else if (state.wolf.x > state.bunny.x)
      BunnyAction.Left
    else if (state.wolf.y > state.bunny.y)
      BunnyAction.Up
    else
      BunnyAction.Down
  }
}
```

Figure 6.6: Typical Scala submission for Task 2.

Figure 6.2.6 shows a typical Scala solution for Task 2. While the Scala solution to Task 2 is more complex than the Scala solution to Task 1, it uses only one more line of code – 35 – due to refactoring of common logic. Of course, this refactoring took extra time and without the refactoring the line count and likely the cyclomatic complexity would have been higher.

```
case class FindMateState(bunny: Location, mate: Location)
val findMate = AfablModule(
  world = bunnyWorld,
  stateAbstraction = (state: BunnyState) => {
    FindMateState(state.bunny, state.mate)
  },
  moduleReward = (state: FindMateState) => {
    if (state.bunny == state.mate) 1.0
    else -0.1
  }
)

// Your solution must assign your AFABL bunny agent for Task 2 to
// the val afablBuny2.
val afablBunny2 = AfablAgent(

  world = bunnyWorld,

  modules = Seq(AfablTask1.findFood, AfablTask1.avoidWolf, findMate),

  agentLevelReward = (state: BunnyState) => {
    if (state.bunny == state.wolf) 0.0
    else if (state.bunny == state.food) 1.0
    else if (state.bunny == state.mate) 1.0
    else 0.5
  }
)
```

Figure 6.7: Typical AFABL submission for Task 2.

Figure 6.2.6 shows typical AFABL code for Task 2. Notice that the `findFood` and `avoidWolf` modules from Task 1 have been reused directly. This works because the world, `BunnyWorld`, is the same. In Task 1 the bunny was ignoring the mate. In Task 2 we adapt the bunny to find the mate, and all we need to do is add a `findMate` module and add a line to the `agentLevelReward` function so that the agent will also value finding mates.

The AFABL solution to Task 2 contains 21 lines of code due to reuse of modules from Task 1, and has the same cyclomatic complexity of 5 (4 predicates in decision structures) even though the agent is solving a more complex problem. Even with the refactoring of common logic in Task 2 the Scala solution has a higher cyclomatic complexity of 10 (9 predicates in decision structures), which McCabe says is the maximum allowable cyclomatic complexity for a testable, maintainable software module [71]. Finally, the performance of the Scala solution to Task 2 decreases to 0.48, while the AFABL solution continues to achieve the same nearly optimal 0.54. With additional work perhaps the Scala agent's performance on Task 2 could have been improved, but the point here is that AFABL agents are easier to write, easier to adapt to new domains, have less complex code, and perform well without requiring a great deal of effort beyond choosing reward signals.

6.2.7 Quantitative Results

Table 6.1: Quantitative results of Scala agent code versus AFABL agent code on Task 1. p-value is for comparison of means between samples of unequal variances (Welch's t-test [87]). A p-value of less than .05 mean that the difference in means is statistically significant at the 95% significance level, i.e. we reject $H_0 : \mu_1 = \mu_2$ and conclude that the means are different. Power is the probability that we reject the null hypothesis H_0 when it is in fact false, given the sample size, variance, and significance level of 95% ($\alpha = .05$).

Task 1	Scala Mean	AFABL Mean	p-value	Power
Time in seconds	1511.45	1780.91	0.47	0.11
Lines of code	39.33	31.20	0.22	0.85
Complexity	10.80	5.27	0.01	1.00
Performance	0.44	0.53	0.02	0.73

Overall results for Task 1 are summarized in Table 6.1. Overall results for Task 2 are summarized in Table 6.2. All but one of the study programmers created good

Table 6.2: Quantitative results of Scala agent code versus AFABL agent code on Task 2. p-value is for comparison of means between samples of unequal variances (Welch's t-test [87]). A p-value of less than .05 mean that the difference in means is statistically significant at the 95% significance level, i.e. we reject $H_0 : \mu_1 = \mu_2$ and conclude that the means are different. Power is the probability that we reject the null hypothesis H_0 when it is in fact false, given the sample size, variance, and significance level of 95% ($\alpha = .05$).

Task 2	Scala Mean	AFABL Mean	p-value	Power
Time in seconds	797.82	626.73	0.53	0.10
Lines of code	41.67	39.20	0.62	0.08
Complexity	11.27	8.20	0.05	0.95
Performance	0.48	0.54	0.03	0.98

AFABL agents. One programmer failed to understand the right way to write the reward function and therefore got very poor performance with their AFABL agents. The data do not show statistically significant differences in time or lines of code, but for the same amount of time and number of lines of code AFABL solutions were less complex and performed better. We also believe that the AFABL results for time are unreliable because programmers left their editors open while they were reading documentation on AFABL. So the time measures for AFABL solutions include programming time and the time spent learning AFABL. Given this fact, the similarity in time between Scala and AFABL solutions indicates that AFABL programs are indeed quicker to write, but we cannot say that with statistical certainty.

Programmer Demographics

We collected demographic data from each programmer, including professional programming experience (work experience), the largest program personally written (code experience), agent programming proficiency, Scala proficiency, level of education, and major. Almost all participants were undergraduate students, and all participants were computer science majors except for one electrical engineering major who was an experienced professional programmer proficient in Scala. The only categories in which there was a nearly 50% split were code experience and agent programming experience, so we compared these splits to get a sense of how programming proficiency affected their results.

Interestingly, the only metric on which experienced and novice coders had statistically significant difference was in lines of code for the Scala solution to Task 1. This result makes some sense given that Scala is an expressive language with terse idioms available to experienced programmers. The lack of differences in other metrics suggest that the agent programming problem was simply to small to reveal differences in programmer proficiency.

Table 6.3: Comparison of novice vs. experienced coder results. p-value is for comparison of means between samples of unequal variances (Welch's t-test [87]). A p-value of less than .05 mean that the difference in means is statistically significant at the 95% significance level, i.e. we reject $H_0 : \mu_1 = \mu_2$ and conclude that the means are different. Power is the probability that we reject the null hypothesis H_0 when it is in fact false, given the sample size, variance, and significance level of 95% ($\alpha = .05$).

Afabl Task 1	Coding Novice Mean	Experienced Coder Mean	p-value	Power
Time in Seconds	2071.00	1539.17	0.39	0.12
Lines of Code	31.43	31.00	0.69	0.07
Complexity	5.00	5.50	0.35	0.24
Performance	0.55	0.51	0.30	0.26
Afabl Task 2	Coding Novice Mean	Experienced Coder Mean	p-value	Power
Time in Seconds	470.20	757.17	0.30	0.18
Lines of Code	40.43	38.12	0.62	0.08
Complexity	8.29	8.12	0.75	0.06
Performance	0.55	0.54	0.45	0.11
Scala Task 1	Coding Novice Mean	Experienced Coder Mean	p-value	Power
Time in Seconds	1487.60	1531.33	0.92	0.05
Lines of Code	25.71	51.25	0.04	0.67
Complexity	8.14	13.12	0.13	0.33
Performance	0.44	0.44	0.99	0.05
Scala Task 2	Coding Novice Mean	Experienced Coder Mean	p-value	Power
Time in Seconds	470.20	757.17	0.30	0.18
Lines of Code	40.43	38.12	0.62	0.08
Complexity	8.29	8.12	0.75	0.06
Performance	0.55	0.54	0.45	0.11

As the results in Table 6.4 show, agent programming experience had no effect on any of the programming metrics. This results suggests that simple agent problems are readily soved with AFABL, and that the agent problems in the study were not sufficiently complex to bring out differences in agent programming experience.

Table 6.4: Comparison of results from programmers no agent programming experience vs programmers with some agent programming experience. p-value is for comparison of means between samples of unequal variances (Welch's t-test [87]). A p-value of less than .05 mean that the difference in means is statistically significant at the 95% significance level, i.e. we reject $H_0 : \mu_1 = \mu_2$ and conclude that the means are different. Power is the probability that we reject the null hypothesis H_0 when it is in fact false, given the sample size, variance, and significance level of 95% ($\alpha = .05$).

Afabl Task 1	Agent Novice Mean	Some Agent Exp Mean	p-value	Power
Time in Seconds	1770.00	1787.14	0.98	0.05
Lines of Code	31.83	30.78	0.45	0.14
Complexity	5.67	5.00	0.36	0.29
Performance	0.55	0.51	0.34	0.19
Afabl Task 2	**Agent Novice Mean**	**Some Agent Exp Mean**	**p-value**	**Power**
Time in Seconds	552.50	669.14	0.71	0.06
Lines of Code	40.67	38.22	0.57	0.08
Complexity	8.17	8.22	0.92	0.05
Performance	0.55	0.54	0.54	0.09
Scala Task 1	**Agent Novice Mean**	**Some Agent Exp Mean**	**p-value**	**Power**
Time in Seconds	1717.50	1393.71	0.56	0.09
Lines of Code	40.83	38.33	0.84	0.05
Complexity	11.17	10.56	0.87	0.05
Performance	0.43	0.44	0.81	0.06
Scala Task 2	**Agent Novice Mean**	**Some Agent Exp Mean**	**p-value**	**Power**
Time in Seconds	552.50	669.14	0.71	0.06
Lines of Code	40.67	38.22	0.57	0.08
Complexity	8.17	8.22	0.92	0.05
Performance	0.55	0.54	0.54	0.09

6.2.8 Qualitative Results

Programmers responded to a questionnaire to give their impressions of agent programming in AFABL versus agent programming in Scala. Here we summarize the responses of the eight participants who were not TAs of the principal investigator. Figure 6.8 shows that programmers had a positive response to AFABL. The most

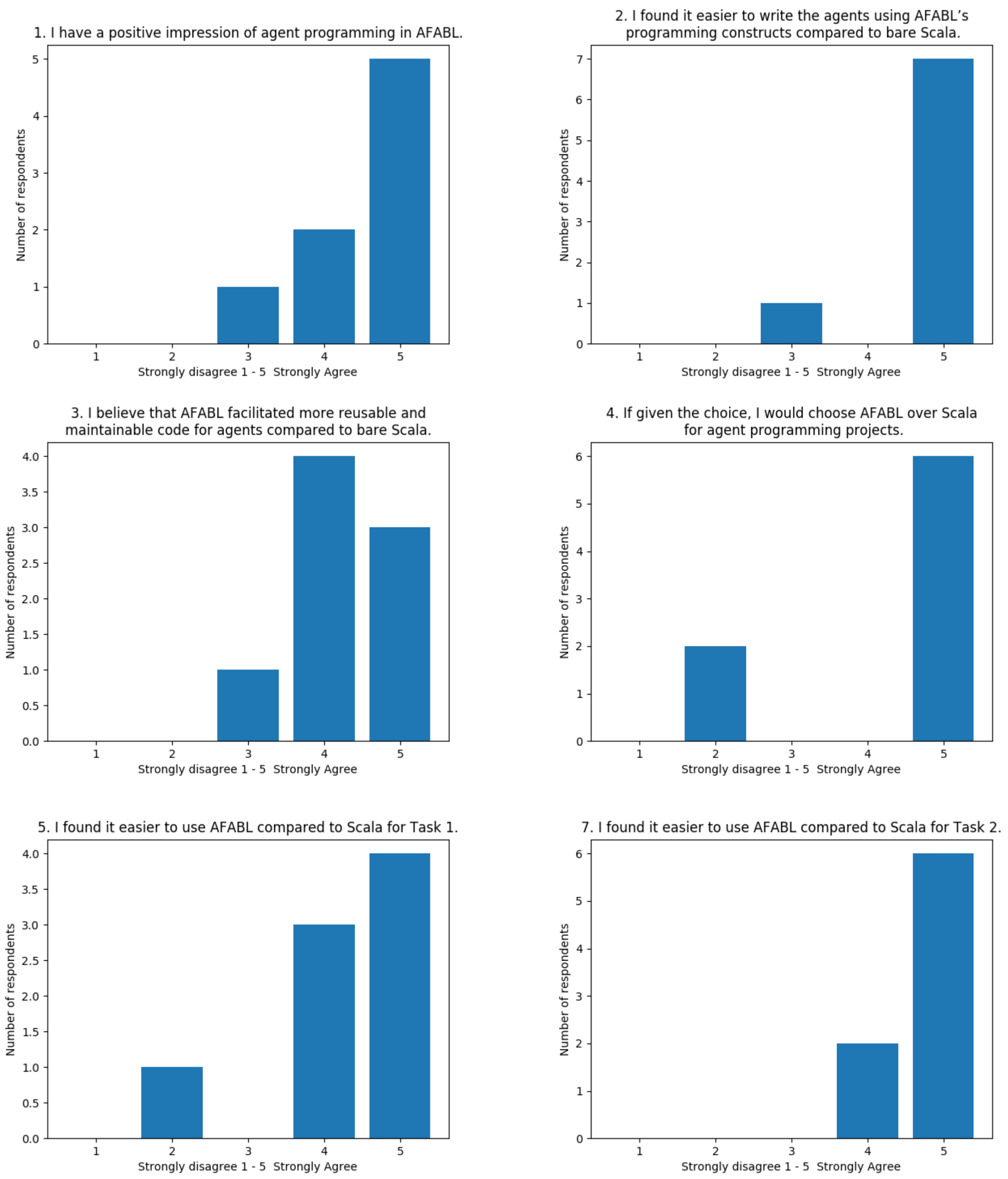

Figure 6.8: Responses to Likert-scale question on reflection survey.

interesting responses of these is for the question “If given the choice, I would choose AFABL over Scala for agent programming projects.” Although every programmer found AFABL to be easier than Scala for Task 2, and almost all programmers found AFABL easier for Task 1, some of the study participants were very experienced Scala programmers who would still prefer the familiarity of Scala. The programmer who reported that they would choose Scala over AFABL did report for Task 2 that they found it easier to adapt their AFABL agent to Task 2 after they had developed some basic knowledge of AFABL.

Internal Validity of Reflection Survey

The Cronbach alpha coefficient measures the correlation between the answers to questions that measure the same construct and is given by:

$$\alpha = \frac{k}{k-1} \times (1 - \frac{s_T^2 - \sum s_I^2}{s_T^2})$$

where

- s_T^2 is the total variance of all the items (questions) for a construct
- s_I^2 is the variance of an individual item, and
- k is the number of items.

We evaluated the internal consistency of the survey by calculating the Cronbach alpha coefficients for the following constructs:

1. User satisfaction with Scala for agent programming tasks.
    - Questions 1 and 3
    - Cronbach alpha: 1.38
2. User satisfaction with AFABL for agent programming tasks.

- Questions 2 and 4
- Cronbach alpha: 1.35

3. User preference for AFABL over Scala for agent programming tasks.

    - Questions 5 and 7
    - Cronbach alpha: 1.28

Since all Cronbach alpha scores were above 0.7, we may consider the reflection survey to be reliable.

Free Response Questions

Two of the questions on the survey allowed for free responses. These responses are listed below, again, only for non-TAs.

What was it about AFABL that made Task 1 easier or harder?

- I didn't have to put so much thought into the mechanics of moving and distance calculation, and didn't have to think about priorities.
- The choice of going towards to food or avoiding the wolf is tricky. Should you only avoid the wolf when it's adjacent? 2 Squares away? what exactly does it mean to avoid? Answering these questions is hard, but expressing the rewards and punishments for getting food and meeting the wolf is easy. AFABL did not require me to express these domain-specific problems, instead it learned based on the rewards and costs what policy is optimal.
- No need to explicitly compare locations, calculate distances or balance separate goals.
- Easier because it only asks for important parts of the specific domain, more difficult as I don't know how different goals interact and I don't know why my end score was lower in AFABL or what changes improve a score.

- While learning AFABL had some overhead for Task 1, being able to program in terms of rewards and punishments was much more intuitive than coding an algorithm from scratch that may or may not be correct.

- I've written agents similar to my Scala agent in the past, so it took much less new thought to develop it, whereas the AFABL agent was conceptually and structurally very different from anything I've written before, and were therefore harder to approach.

- You simply define success and failure scenarios; there's no dealing with weighting different values and that makes it MUCH easier to work with, as a programmer.

- There was no need to define the movement algorithm

What was it about AFABL that made Task 2 easier or harder?

- All I had to do was add a 3rd module which was very much like FindFood.

- Same arguments as 1, but now there's the mate. I actually found that I could treat the mate exactly as a food source by adding another module very similar to the food module. The problem then became how to maximize the total reward by playing around with the module-level rewards and the agent-level rewards. If it were very important to find good reward values, one could run some optimization program on the rewards. It's worth mentioning that if I were faced with this problem, I would feel more comfortable running an optimization algorithm than writing my own domain-specific agent behavior. AFABL would pair nicely with a solver that optimizes on rewards.

- The design of independent modules made adding an additional module/goal trivial.

- Both were pretty easy to add 1 more goal. But in Scala I copied and pasted. In AFABL it was really neat to just import modules from the other file with no effort to integrate them.
- Being able to just add in another module and tack it onto the agent with AFABL was much easier and more elegant than having to go in and modify existing methods and logic in Scala. Adding the additional functionality with AFABL was much more convenient in this respect.
- After understanding AFABL to some degree, it was quite easy to modify my existing agent to the new task and thereby receive a good score.
- You can much more clearly see the similarities between Task 1 and Task 2 in the AFABL version, for one thing. Second, it doesn't require modifying existing code nearly as much as the plain Scala version does. It's a delight to use, and as a programmer at a startup, I would much rather work with this format over what I have to do to work with AWS' Machine Learning program.
- Too many cases to work on in Scala, hard to figure out whether mating or eating takes importance, takes far less time with AFABL.

Many of the answers above mention the reduced time and effort of writing AFABL agents. These answers corroborate our belief that the time tracking in the quantitative section was unreliable due to programmers not following directions to move the focus off their editors while reading documentation, and possible malfunctions in the IntelliJ IDEA plug-in used to automatically track time.

6.3 Threats to Validity

The first threat to validity is sample size – there were only 16 study participants. Indeed the statistical power calculations for most comparisons indicate that results are inconclusive due to the small sample sizes and large variances in many metrics. However, two core metrics – complexity and performance – yielded very high statistical power, supporting our claim that AFABL agent programs are simpler to write than their non-AFABL equivalents, and that it takes less effort to get good performance from AFABL agents.

The second threat is problem size. The tasks that programmers were given were smaller than typical software engineering tasks. Given the difficulty of getting programmers to work for hours writing programs for a study, and the further challenge of finding Scala programmers, the problem size had to be limited. We believe that the insights are still valid because the chief differences between AFABL agent programs and traditional programs are the control-flow complexity and the additional effort required to adapt agent programs to more complex domains. As our results showed, the small tasks were sufficient to show these differences.

The final threat to validity is the fact that 8 of the 16 study participants were teaching assistants for the principal investigator. This fact may have biased them towards writing better AFABL solutions in order to help me show positive results. However, as Table 6.5 shows, the data do not show that the TAs' results differed from the non-TAs' results. Again, due to small samples sizes the statistical power of these comparisons is low, but nevertheless we cannot show a statistically significant difference between the results of TAs and non-TAs.

6.4 Conclusion

As you can see from the similarity of the submission in Figure 6.2.5 to our explanatory example, most AFABL bunny agents look the same. There is one obvious way to implement a bunny agent that must pursue multiple goals. This uniformity is desirable. As Tim Peters says in the Zen of Python [88], "There should be one– and

Table 6.5: Comparison of TA results versus non-TA results. p-value is for comparison of means between samples of unequal variances (Welch's t-test[87]). A p-value of less than .05 mean that the difference in means is statistically significant at the 95% significance level, i.e. we reject $H_0 : \mu_1 = \mu_2$ and conclude that the means are different. Power is the probability that we reject the null hypothesis H_0 when it is in fact false, given the sample size, variance, and significance level of 95% ($\alpha = .05$).

Afabl Task 1	TA Mean	Non-TA Mean	p-value	Power
Time in Seconds	1620.00	1733.33	0.86	0.05
Lines of Code	31.38	31.12	0.82	0.06
Complexity	5.00	5.50	0.35	0.26
Performance	0.51	0.54	0.42	0.19
Afabl Task 2	TA Mean	Non-TA Mean	p-value	Power
Time in Seconds	686.83	552.00	0.62	0.08
Lines of Code	41.75	38.25	0.47	0.11
Complexity	8.38	8.12	0.61	0.08
Performance	0.55	0.54	0.19	0.26
Scala Task 1	TA Mean	Non-TA Mean	p-value	Power
Time in Seconds	1520.67	1462.17	0.89	0.05
Lines of Code	36.88	40.12	0.80	0.06
Complexity	10.62	11.12	0.88	0.05
Performance	0.46	0.42	0.47	0.11
Scala Task 2	TA Mean	Non-TA Mean	p-value	Power
Time in Seconds	686.83	552.00	0.62	0.08
Lines of Code	41.75	38.25	0.47	0.11
Complexity	8.38	8.12	0.61	0.08
Performance	0.55	0.54	0.19	0.26

preferably only one –obvious way to do it." The similarity in most AFABL solutions to a particular modular agent programming problem is an indication that AFABL provides the right abstractions for adaptive agent programming. This regularity results from exploiting the structure of certain kinds of agent programming problems. In the next chapter we discuss the kinds of problems for which AFABL is well-suited and those for which it is less well-suited.

CHAPTER 7
AFABL IN CONTEXT

In this chapter we place AFABL in context. First we present an extended example that compares typical AFABL code to typical Scala code for a series of agent programs for increasingly complex but closely related task environments. This example demonstrates the sub-linear progression of complexity of AFABL code compared to a linear progression of complexity for traditional code on the same task progression. Finally we discuss the kinds of problems that are well-suited to AFABL and those that are not well-suited to AFABL.

7.1 AFABL Programs versus Traditional Programs

In this section we incrementally expand the Bunny world from previous chapters to show how AFABL programs and traditional programs grow in response to additional world dynamics. The sections below show typical submissions from the previous chapter and build on them using similar styles to handle increasingly complex world dynamics. These examples demonstrate the superior scalability of AFABL.

7.1.1 Bunny, Food

To help keep the code straight, we will number each task starting at 0. We start with the simplest task, that of finding food. Figure 7.1 shows the AFABL agent for Task 0.

Figure 7.2 shows the Scala code for Task 0.

```
object AfablTask0 {
  case class FindFoodState(bunny: Location, food: Location)

  val findFood = AfablModule(
    world = bunnyWorld,
    stateAbstraction = (worldState: BunnyState) => {
      FindFoodState(worldState.bunny, worldState.food)
    },
    moduleReward = (moduleState: FindFoodState) => {
      if (moduleState.bunny == moduleState.food) 1.0
      else -0.1
    }
  )

  val afablBunny0 = AfablAgent(

    world = bunnyWorld,

    modules = Seq(findFood),

    agentLevelReward = (state: BunnyState) => {
      if (world.bunnyEats()) 1.0
      else 0.5
    }
  )
}
```

Figure 7.1: An AFABL bunny agent that finds food.

7.1.2 Bunny, Food, Wolf

Task 1 adds the task of avoiding a wolf. As we see in Figure 7.3, the AFABL version looks quite similar – we simply add a module with a different state abstraction and reward function, and incorporate the added criterion of avoiding the wolf to the agent level reward.

Figure 7.4 shows the Scala code for Task 1, which adds logic for analyzing the relative distance of the wolf to the food in order to choose between pursuing food or running away from the wolf.

```
class ScalaBunny0 extends Agent[BunnyState, BunnyAction.Value] {

  def getAction(state: BunnyState) = {
      moveTowardFood(state)
   }

  def moveTowardFood(state: BunnyState) = {
    if (state.food.x > state.bunny.x)
      BunnyAction.Right
    else if (state.food.x < state.bunny.x)
      BunnyAction.Left
    else if (state.food.y < state.bunny.y)
      BunnyAction.Up
    else
      BunnyAction.Down
  }
}
```

Figure 7.2: A Scala bunny agent that finds food.

7.1.3 Bunny, Food, Wolf, Mate

Task 2 adds the task of finding a mate. The AFABL agent is able to reuse modules from afablBunny1 directly. ScalaBunny2 also benefits from the refactoring of common logic in ScalaBunny1.

In ScalaBunny2, shown in Figure 7.6, we follow the advice of Martin Fowler: if you need the same code in two places, copy and paste. If you need the same code in a third place, refactor the common logic in reusable program units. In ScalaBunny2 we factor out the code for finding distance and the code for moving towards something.

7.1.4 Bunny, Wolf, Food, Mate, Spoiling Food

Task 3 adds spoilage to the food. If the bunny does not eat the food within `SPOIL_TIME` time steps the food disappears and new food respawns elsewhere. Here we begin to see the scalability benefits of AFABL. As we see in Figure 7.7, adapting afablBunny2 to Task 3 requires no new code whatsoever. The reinforcement learning algorithms underlying the AFABL code adjust to the new world dynamics automatically. The Scala

```
object AfablTask1 {
  case class FindFoodState(bunny: Location, food: Location)
  case class AvoidWolfState(bunny: Location, wolf: Location)

  val avoidWolf = AfablModule(
    world = bunnyWorld,
    stateAbstraction = (worldState: BunnyState) => {
      AvoidWolfState(worldState.bunny, worldState.wolf)
    },
    moduleReward = (moduleState: AvoidWolfState) => {
      if (moduleState.bunny == moduleState.wolf) -0.1
      else 0.1
    }
  )

  val afablBunny1 = AfablAgent(

    world = bunnyWorld,

    modules = Seq(AfablTask0.findFood, avoidWolf),

    agentLevelReward = (state: BunnyState) => {
      if (world.bunnyDies()) 0.0
      else if (world.bunnyEats()) 1.0
      else 0.5
    }
  )
}
```

Figure 7.3: An AFABL bunny agent that finds food and avoids a wolf.

version, on the other hand, adds a tremendous amount of logic to add consideration of spoiling food.

Figure 7.8 shows the Scala code for Task 3. ScalaBunny3 extends ScalaBunny2 to inherit its distance and moveTowards methods, but we must add logic to determine when it doesn't make sense to move towards the food because it will spoil before the bunny gets to it. This is a fairly straightforward approach but required some thought to come up with and the resulting code is slightly more complex due to the additional predicates in decision structures.

7.1.5 Bunny, Food, Wolf, Mate, Spoiling Food, Picky Mate

Task 4 adds a picky mate that won't mate unless the bunny has eaten within the last `HUNGER_LIMIT` time steps. Here again we see that no new code needs to be written for the AFABL agent in Figure 7.9. The reinforcement learning algorithms simply learns new behavior for the changing world dynamics.

ScalaBunny4, shown in Figure 7.10, requires much more logic than ScalaBunny3. ScalaBunny4 must keep a reference to the world to know when it has eaten, it must keep track of its hunger level, and the decision of whether to pursue the food or the mate is now more complicated.

7.1.6 Analysis

Figure 7.11 shows the complexity of AFABL programs is lower than equivalent programs in traditional programming languages, that the complexity of AFABL code grows more slowly, and that once AFABL code has been written for the basic elements of a task environment, the complexity does not increases as the dynamics of the world change. In contrast, agent programs in traditional languages increase in complexity as the task environment increases in complexity in a linear fashion. The complexity of AFABL programs grows in a sub-linear, often logarithmic fashion when the task environment changes can be handled automatically by AFABL's reinforcement learning algorithms. We discuss this further in the next section.

In Figure 7.12 we see a similar pattern in lines of AFABL code. The number of lines of code increases until the basic modules are defined, then the lines of code actually decrease as modules are reused. This reuse is one of the goals of AFABL. With the Scala code we see the benefit of refactoring code in traditional languages. While the Scala agents require far more code, the growth in lines of code slows as reusable functions are created. However, the complexity still grows due to the need to manually encode all the decision logic in the agent.

7.2 When to Use AFABL

AFABL shows the greatest scalability benefit when adapting agents to worlds with changing dynamics but the same state representation. As long as the state representation stays the same, the same AFABL code can be used in a world with different or changing dynamics. In fact, AFABL modules written separately with different reward structures can be used.

AFABL is a good choice for agent programs when:

- the agent pursue multiple subgoals simultaneously that are clearly separated and non-overlapping and thus can be modeled with behavior modules;
- the subgoals are sometimes at odds, that is, when one two or more modules would choose to go in different directions;
- there is an agent-level reward signal that represents goals that may or may not be represented in any module;
- there is a single action set shared by all the modules; and
- the world dynamics may change but retain the same state representation.

AFABL is not a good choice for all agent programs. In particular, AFABL is not a good choice when:

- the best action in some states is a compromise action, not the first choice of any module. In such cases AFABL's arbitration algorithm will choose a suboptimal action.
- the agent's goals cannot be well modeled with concurrent subgoals;

- there is no clear agent-level reward signal that can be used to automatically learn how to arbitrate modules' action choices; and
- the problem itself is not well-suited to reinforcement learning agents in general.

```
class ScalaBunny1 extends Agent[BunnyState, BunnyAction.Value] {

  def getAction(state: BunnyState) = {
    if (wolfNearFood(state))
      moveAwayFromWolf(state)
    else
      moveTowardFood(state)
   }

  def wolfNearFood(state: BunnyState) = {
    val wolfToFood = sqrt(pow(state.food.x - state.wolf.x, 2) +
                          pow(state.food.y - state.wolf.y, 2))
    val bunnyToFood = sqrt(pow(state.food.x - state.bunny.x, 2) +
                           pow(state.food.y - state.bunny.y, 2))
    wolfToFood < bunnyToFood
  }

  def moveTowardFood(state: BunnyState) = {
    if (state.food.x > state.bunny.x)
      BunnyAction.Right
    else if (state.food.x < state.bunny.x)
      BunnyAction.Left
    else if (state.food.y < state.bunny.y)
      BunnyAction.Up
    else
      BunnyAction.Down
  }

  def moveAwayFromWolf(state: BunnyState) = {
    if (state.wolf.x < state.bunny.x)
      BunnyAction.Right
    else if (state.wolf.x > state.bunny.x)
      BunnyAction.Left
    else if (state.wolf.y > state.bunny.y)
      BunnyAction.Up
    else
      BunnyAction.Down
  }
}
```

Figure 7.4: A Scala bunny agent that finds food and avoids a wolf.

```
object AfablTask2 {
  case class FindFoodState(bunny: Location, food: Location)
  case class AvoidWolfState(bunny: Location, wolf: Location)
  case class FindMateState(bunny: Location, mate: Location)

  val findMate = AfablModule(
    world = bunnyWorld,
    stateAbstraction = (state: BunnyState) => {
      FindMateState(state.bunny, state.mate)
    },
    moduleReward = (state: FindMateState) => {
      if (world.bunnyMates()) 1.0
      else -0.1
    }
  )

  val afablBunny2 = AfablAgent(

    world = bunnyWorld,

    modules = Seq(AfablTask0.findFood, AfablTask1.avoidWolf, findMate),

    agentLevelReward = (state: BunnyState) => {
      if (world.bunnyDies()) 0.0
      else if (world.bunnyEats()) 1.0
      else if (world.bunnyMates()) 1.0
      else 0.5
    }
  )
}
```

Figure 7.5: An AFABL bunny agent that finds food, avoids a wolf, and pursues a mate.

```scala
class ScalaBunny2 extends Agent[BunnyState, BunnyAction.Value] {

  def getAction(state: BunnyState) = {
    if ((distance(state.wolf, state.food) < distance(state.food, state.bunny))
      || distance(state.wolf, state.mate) < distance(state.mate, state.bunny))
      moveAwayFromWolf(state)
    else if (distance(state.bunny, state.food) < distance(state.bunny,
    state.mate))
      moveToward(state.bunny, state.food)
    else
      moveToward(state.bunny, state.mate)
  }

  def distance(a: Location, b: Location) = {
    sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2))
  }

  def moveToward(from: Location, to: Location) = {
    if (to.x > from.x)
      BunnyAction.Right
    else if (to.x < from.x)
      BunnyAction.Left
    else if (to.y > from.y)
      BunnyAction.Up
    else
      BunnyAction.Down
  }

  def moveAwayFromWolf(state: BunnyState) = {
    if (state.wolf.x < state.bunny.x)
      BunnyAction.Right
    else if (state.wolf.x > state.bunny.x)
      BunnyAction.Left
    else if (state.wolf.y > state.bunny.y)
      BunnyAction.Up
    else
      BunnyAction.Down
  }
}
```

Figure 7.6: A Scala bunny agent that finds food, avoids a wolf, and finds a mate.

```
object AfablTask3 {
  val afablBunny3 = AfablAgent(

    world = bunnyWorld,

    modules = Seq(AfablTask0.findFood, AfablTask1.avoidWolf, AfablTask2.findMate),

    agentLevelReward = (state: BunnyState) => {
      if (world.bunnyDies()) 0.0
      else if (world.bunnyEats()) 1.0
      else if (world.bunnyMates()) 1.0
      else 0.5
    }
  )
}
```

Figure 7.7: An AFABL bunny agent that finds food that spoils, avoids a wolf, and finds a mate.

```
class ScalaBunny3 extends ScalaBunny2 {

  val SPOIL_TIME = 10
  var stepCount = 0

  override def getAction(state: BunnyState) = {
    stepCount = stepCount + 1
    if ((distance(state.wolf, state.food) < distance(state.food, state.bunny))
      || distance(state.wolf, state.mate) < distance(state.mate, state.bunny))
      moveAwayFromWolf(state)
    else if (distance(state.bunny, state.food) < distance(state.bunny, state.mate)
      && distance(state.bunny, state.food) < stepCount % SPOIL_TIME)
      moveToward(state.bunny, state.food)
    else
      moveToward(state.bunny, state.mate)
  }

  def moveAwayFromWolf(state: BunnyState) = {
    if (state.wolf.x < state.bunny.x)
      BunnyAction.Right
    else if (state.wolf.x > state.bunny.x)
      BunnyAction.Left
    else if (state.wolf.y > state.bunny.y)
      BunnyAction.Up
    else
      BunnyAction.Down
  }
}
```

Figure 7.8: A Scala bunny agent that finds food that spoils, avoids a wolf, and finds a mate.

```
object AfablTask4 {
  val afablBunny4 = AfablAgent(

    world = bunnyWorld,

    modules = Seq(AfablTask0.findFood, AfablTask1.avoidWolf, AfablTask2.findMate),

    agentLevelReward = (state: BunnyState) => {
      if (world.bunnyDies()) 0.0
      else if (world.bunnyEats()) 1.0
      else if (world.bunnyMates()) 1.0
      else 0.5
    }
  )
}
```

Figure 7.9: An AFABL bunny agent that finds food that spoils, avoids a wolf, and finds a mate who rejects the bunny if the bunny hasn't eaten recently.

```
class ScalaBunny4(val world: BunnyWorld) extends ScalaBunny3 {

  val HUNGER_LIMIT
  val SPOIL_TIME = 10
  var stepCount = 0
  var hungerLevel = 0

  override def getAction(state: BunnyState) = {
    stepCount = stepCount + 1
    if (world.bunnyEats()) {
      hungerLevel = 0
    } else {
      hungerLevel = hungerLevel + 1
    }
    if ((distance(state.wolf, state.food) < distance(state.food, state.bunny))
      || distance(state.wolf, state.mate) < distance(state.mate, state.bunny))
      moveAwayFromWolf(state)
    else if ((distance(state.bunny, state.food) < distance(state.bunny,
    state.mate)
      && distance(state.bunny, state.food) < stepCount % SPOIL_TIME)
      || (hungerLevel + distance(state.bunny, state.mate) > HUNGER_LIMIT)
      moveToward(state.bunny, state.food)
    else
      moveToward(state.bunny, state.mate)
  }

  def moveAwayFromWolf(state: BunnyState) = {
    if (state.wolf.x < state.bunny.x)
      BunnyAction.Right
    else if (state.wolf.x > state.bunny.x)
      BunnyAction.Left
    else if (state.wolf.y > state.bunny.y)
      BunnyAction.Up
    else
      BunnyAction.Down
  }
}
```

Figure 7.10: A Scala bunny agent that finds food that spoils, avoids a wolf, and finds a mate who rejects the bunny if the bunny hasn't eaten recently.

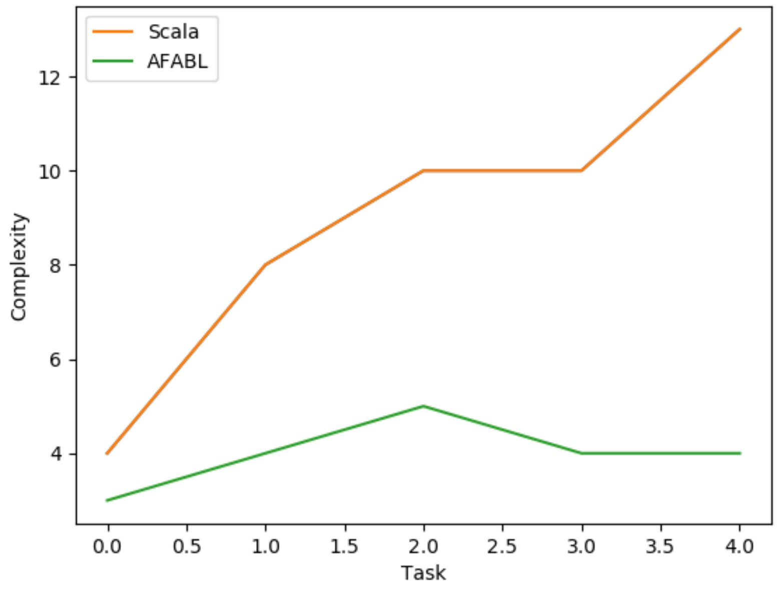

Figure 7.11: Growth in complexity of agent programs as the task becomes more complex.

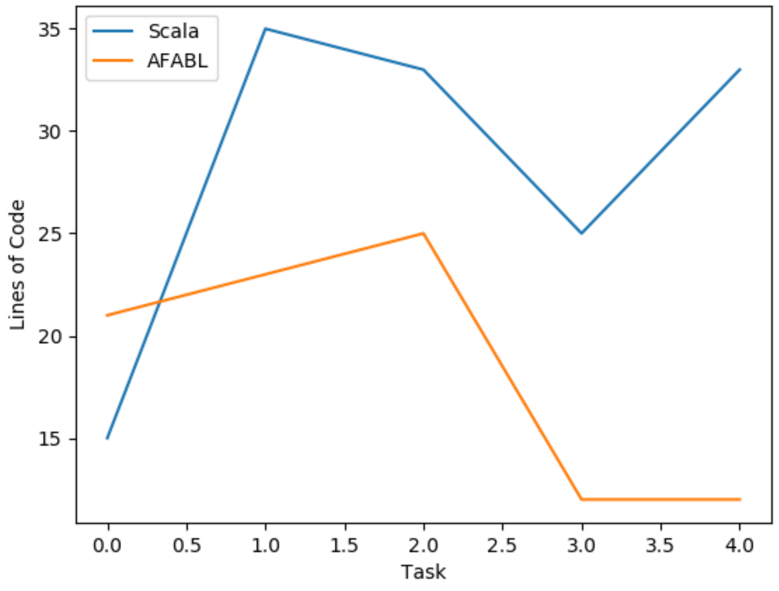

Figure 7.12: Growth in lines of code in agent programs as the task becomes more complex.

CHAPTER 8
AN EXAMPLE APPLICATION: DERIVING BEHAVIOR FROM PERSONALITY

In this chapter we present an application of language-integrated reinforcement learning to the problem of personality simulation. Creating artificial intelligent agents that are high-fidelity simulations of natural agents will require that behavioral scientists be able to write code themselves, not merely act as consultants with the ensuing knowledge acquisition bottleneck. However, translating personality models into the concrete behavior of an agent using currently available programming constructs would require a level of code complexity that would make the system inaccessible to behavioral scientists. What we need is a way to derive the concrete actions of an agent directly from psychological personality models. This chapter describes a reinforcement learning approach to solving this problem in which we represent trait-theoretic personality models as reinforcement learning agents. We validate our approach by creating a virtual reconstruction of a psychology experiment using human subjects and showing that our virtual agents exhibit similar behavior patterns. Note that this work was conducted and published before we finished the Arbi-Q command arbitration algorithm of Chapter 3, so the AFABL agents used the Greatest-Mass q-decomposition algorithm. We have also updated the code from our original work to use AFABL syntax.

8.1 Introduction

There is tremendous interest in creating synthetic agents that behave as closely as possible to natural (human) agents. Rich, interactive intelligent agents will advance the state of the art in training simulations, interactive games and narratives, and social science simulations. However, the programming systems for creating such rich synthetic agents are too complex, or rather too steeped in computational concepts, to be used directly by the behavioral scientists who are most knowledgeable in modeling natural agents. Engaging behavioral scientists more directly in the authoring of synthetic agents would go a long way towards improving the fidelity of synthetic agents.

What we need is a programming language that a behavioral scientist can use to write agent programs using concepts familiar to behavioral scientists. This task is complicated by the fact that the most popular and best understood personality models from behavioral science do not lend themselves to direct translation into computer programs. Requiring a behavioral scientist to specify behaviors in the detail required in even the most cutting edge purpose-built programming language would plunge the would-be behavioral scientist agent programmer right into a morass of complex computational concepts that lie outside the expertise of most dedicated behavioral experts. To solve this problem we need a way to get from personality models to behaviors, to derive specific agent actions in an environment from a personality model without having to program the derivation in great detail.

In this chapter, we describe a way to model motivational factors from trait-oriented personality theory with reinforcement learning modules. We describe a virtual agent simulation that reconstructs a human subject experiment from psychology, namely some of Atkinson's original work in achievement motivation and test anxiety, and show that our simulation exhibits the same general behavior patterns as the human subjects in Atkinson's experiments. First, we briefly discuss relevant personality research and provide some background.

8.1.1 Personality

Personality is a branch of psychology that studies and characterizes the underlying commonalities and differences in human behavior. Within psychology, there are two broad categories of personality theories: processing theories, and dispositional, or trait theories. Social-cognitive and information-processing theories identify processes of encodings, expectancies, and goals in an attempt to characterize the mechanisms by which people process their perceptions, store conceptualizations, and how those processes drive their interactions with others [89, 90, 91]. A strength of processing theories, especially from a computational perspective, is that they provide a detailed account of the cognitive processes that give rise to personality and drive behavior. This strength is also a drawback – processing theories tend to be detailed and often low-level (though not as low-level as cognitive architectures, which we will discuss below), and this makes them less intuitive and less suited to describing personality in broad, easily understood terms.

Trait theories [90], the most well-known example of which is the Five-Factor model [92], attempt to identify stable traits (sometimes called "trait adjectives") that can be measured on numerical scales and remain invariant across situations in determining behavior. A strength of the trait approach is that they are well-suited to describing individuals in broad, intuitive terms. Two drawbacks of the approach are that there is not yet widespread agreement on a set of truly universal traits (or how many there are), and it is not clear how trait models drive behavior. A promising line of research by Elliot and Thrash [93] is working towards solving these problems by integrating motivation into personality in a general way. The work of Elliot and Thrash particularly supports the approach we present here, as they show that approach and avoidance motivation underpins all currently popular trait theories.

While debate continues about the merits and drawbacks of the different approaches to personality, the psychology community is also attempting to unify personality and motivation theory [94]. While the work we present here is focused on bridging the gap between the descriptive power of trait-oriented models and the behavior that arise from them, we consider this work to be complementary to work in encoding information processing theories. In the future, rich computational agents may be built by combining approaches.

8.1.2 Modeling Personality with Reinforcement Learning

The essential idea behind modeling personality traits with reinforcement learning is that each motivational factor can be represented by a reinforcement learning module. In psychology, the inherent desirability or attractiveness of a behavior or situation is referred to as *valence*. For a person high in success approach motivation, behaviors or situations that provide an "opportunity to excel" will have high valence, while other behaviors will have lower valence. The notion of valence translates fairly directly into the concept of reward in reinforcement learning. Just as people with certain motivational factors will be attracted to high-valence behaviors, a reinforcement learner is attracted to high-reward behaviors. This is the basis for modeling motivational factors with reinforcement learning modules. By encoding the valence of certain behaviors as a reward structure, reinforcement learners can learn the behavioral patterns that are associated with particular motivational factors. This is a powerful idea, because it allows an agent author to write agent code using motivational factors while minimizing the need to encode the complex mechanisms by which such factors lead to concrete behavior.

A critical aspect of trait theory is that traits can have interactive effects. It is clear that a person who is high in achievement motivation will "go for it" when given the opportunity and that a person who is high in avoidance motivation will be more reserved. But what happens when a person is high in both motivations? Such interactive effects cannot be ignored in a credible treatment of personality, but it is hard to predict the behavioral patterns that will arise from given combinations of motivational factors. One can imagine the code complexity that might result from trying to model such interactive effects with production rules or other traditional programming constructs. As we demonstrate later, our reinforcement learning approach handles such interactive effects automatically.

It is important to note that we are not creating a new theory of personality. We are creating a computational means of translating existing theories of personality from *psychology* (not computer science) into actions executed by synthetic agents. We are also not committing to a particular theory from psychology, but rather supporting the general category of trait theories of personality which, until now, have not been directly realizable in computer agents.

In the remainder of this chapter we discuss some related work in agent modeling, present our virtual reconstruction of a human subject experiment using our reinforcement learning approach, and discuss the promising results and their implications for future work.

8.2 Related Work

There is a great deal of work in modeling all sorts of phenomena in synthetic agents. Cognitive architectures provide computational models of many low-level cognitive processes, such as memory, perception, and conceptualization [95, 96]. Cognitive architectures support scientific research in cognitive psychology by providing runnable models of cognitive processes, support research in human-computer interaction with detailed user models [97], and can serve as the "brains" of agents in a variety of contexts. The most notable and actively developed cognitive architectures are Soar [98] and ACT-R [99]. Recently, some effort has gone into integrating reinforcement learning into Soar [100]. While RL is used to improve the reasoning system in Soar, we are using RL to support new paradigms of computer programming for agent systems. In general, our work differs from and complements work in cognitive architectures in that we are drawing on psychological theory that is expressed at a much higher level of abstraction. Cognitive psychology and AI have often built on each other. Indeed, cognitive psychology is the basis of cognitive architectures in AI. Our work is an attempt to bring in mainstream personality psychology as a basis for building intelligent agents, which we hope will complement the detailed models of cognitive architectures in creating rich synthetic agents.

There is a large and rich body of work in believable agents. Mateas and Stern built on the work of the Oz project [101] in creating a programming language and reactive–planning architecture for rich believable agents. They implemented their theory in the computer game Facade, a one-act interactive drama in which the player interacts with computer simulated characters that provide rich social interactivity [102]. Gratch, Marsella and colleagues have a large body of work in creating rich simulations of humans for training simulations that incorporate models of appraisal theory and emotion [103, 104]. A distinctive feature of the work of both Mateas, et. al., and Gratch, et. al., is that they are dealing with the entire range of AI problems in creating believable agents that sense, act, understand and communicate in natural language, think, and exhibit human-like personalities. Our work differs from other work in personality modeling in that we are not attempting to simulate personality, but using definitions of personality to drive the behavior of synthetic agents. We want to derive behavior that is consistent with a given personality model, but not necessarily to ensure that the agent gives the appearance of having that personality.

8.3 Experiments

To test our claim that personality can be modeled by reinforcement learning modules, we created a population of simple two-module multiple-goal reinforcement learning agents and ran them in a world that replicated experiments carried out with humans by psychologist John Atkinson. First we describe Atkinson's original research, and then discuss our virtual reconstruction of his experiments.

8.3.1 Atkinson's Ring Toss Experiment

John Atkinson was among the first researchers to study the existence and role of approach and avoidance motivation in human behavior. Prior to Atkinson's work, it was believed that test anxiety was equivalent to low achievement motivation. However, Atkinson showed that test anxiety is actually a separate avoidance motivation, a "fear of failure" dimension that works against and interacts with achievement motivation [105]. To test his hypothesis, he administered standard tests of achievement motivation and test anxiety to a group of undergraduate psychology students and devised a series of experiments which examined the effort put forth in achieving success in tasks such as taking a final exam. It is important to note that he did not measure the outcomes of the task, but rather the effort put forth in doing well in them. Thus, his experiments examined the relationship between motivation and *behavior*, not necessarily competence. One of his experiments, a ring toss game, produced results that clearly show the interplay of approach and avoidance motivation and is particularly well-suited to computer simulation.

In Atkinson's ring toss experiment, subjects played a ring toss game in which players attempted to toss a ring from a specified distance onto a peg. Subjects made 10 tosses from any distance they wished, from 1 through 15 feet, and the distance at which each subject made each toss was recorded. For analysis, subjects were divided into four groups according to their measures of achievement motivation and test anxiety so that the relationship between these motivations and their behavior could be analyzed. For each of the two measures – achievement motivation and test anxiety – subjects were classified as either high or low, with the dividing line between high and low set at the median scores in each measure. (For example, a H-L subject is high in achievement motivation and low in test anxiety). Subjects were divided into four groups – H-L, H-H, L-L, and L-H – and the percentage of shots taken at each distance by each group was recorded. We discuss his results and our simulation below.

8.3.2 Computational Models of Atkinson's Subjects

We reconstructed Atkinson's ring toss experiment in a computer simulation. We created 49 virtual agents that corresponded to each of the 49 human subjects in Atkinson's experiments, with the same distribution of high and low measures of achievement motivation and test anxiety. Simplified code for a representative student subject is presented in Figure 8.1. Since we did not have access to Atkinson's source data, we modeled high motivation measures as having a mean of 1.5 and low motivation with a mean of 0.5, both with standard Normal distributions (mean $= 0$, variance $= 1$) scaled by $\frac{1}{2}$, so virtual test subjects did not all have the same measures.

As discussed earlier, each of the motivational dimensions of the virtual subjects was implemented with reinforcement learning modules that learned to satisfy the preference for perceived valence of behaviors (modeled as reward). For example, in the achievement motivation module (see Figure 8.2), the greater the distance from the peg, the greater the reward because it represents greater achievement. Similarly, in the test anxiety module (see Figure 8.3), greater reward is given to closer distances, because they minimize, or "avoid" the chance of failure from a long-distance toss.

Internally, each personality module is implemented with the standard Q-learning

```
val motivatedStudent = GmAgent(
  world = RingTossWorld,

  // Sequence of pairs where the second element of each pair
  // is the weight of the pair, corresponding to the personality
  // trait measure
  modules = Seq((achievementMotivation, 1.5 + X ~ N(0, 1) / 2),
                (testAnxiety, .5 + X ~ N(0, 1) / 2))
}
```

Figure 8.1: An agent representing a success-oriented student in Atkinson's ring toss experiment, containing two RL modules representing high achievement motivation and low test anxiety. The code snippets presented here are simplified versions of the Scala code we used to run our experiments.

```
val achievementMotivation = AfablModule(

  world = RingTossWorld,

  moduleReward = (state: RingTossState) => state match {
    case OneFootLine => 1,
    case TwoFootLine => 2,
    ...
    case FifteenFootLine => 15
  }

)
```

Figure 8.2: A reinforcement learning module representing achievement motivation.

algorithm [1]. The ring toss world consists of 16 states – a start state and one state for each of the 15 distances, and 15 actions available in each state that represent playing (making a toss) from a particular distance. Each reinforcement learning module used a step-size parameter of $\alpha = 0.1$, a discount factor of $\gamma = 0.9$ (though discounting wasn't important given that the 15 states representing playing lines were terminal states, since each play was a training episode), and employed an ϵ-greedy action selection strategy with $\epsilon = 0.2$. (Readers familiar with reinforcement learning will also notice that this game is equivalent to a 15-armed bandit problem.) We emphasize that the details of the reinforcement learning algorithms are not essential to modeling motivational factors, and those details are hidden inside the implementation of the

```
val testAnxiety = AfablModule(

  world = RingTossWorld,

  moduleReward = (state: RingTossState) => state match {
    case OneFootLine => 15,
    case TwoFootLine => 14,
    ...
    case FifteenFootLine => 1
  }

)
```

Figure 8.3: A reinforcement learning module representing Test Anxiety (‘avoidance motive, a.k.a. “fear of failure”). Note that the rewards are inverted from the achievement motivation module, that is, the valence of avoiding achievement is higher.

modules. Indeed a major goal of our work is to simplify the task of writing synthetic agents by taking care of such details automatically.

Recall that reinforcement learning algorithms learn an action value for each action available in a given state. An action value for a state represents the expected total reward that can be achieved from a state by executing that action and transitioning to a successor state. For each of the modules – Achievement and TestAnxiety – the action values represent the learned utility of the actions in serving the motivational tendencies the modules represent. The Student agents take into account the preferences of the modules – represented by action values – by summing their action values weighted by their module weights to get a composite action value for each action in a given state. If we denote each module’s action value by $Q(s,a)$ and the weights by W, then the composite, or overall, action value is:

$$Q_{student}(s,a) = W_{Achievement}Q_{Achievement}(s,a) + \tag{8.1}$$

$$W_{TestAnxiety}Q_{TestAnxiety}(s,a) \tag{8.2}$$

For the virtual experiments, each module – Achievement and TestAnxiety – was run to convergence and then the student agents simulated 10 plays of the ring toss game, just as in Atkinson's experiment. We discuss the results of the experiment below.

8.4 Model Validation

A model is a set of explicit assumptions about how some system of interest works [106]. In psychology the system of interest is (usually) a human or group of humans. Our virtual reconstruction of Atkinson's experiments constitutes a computational representation of Atkinson's two-factor model of personality. Thus, our agents are simulation models of Atkinson's subjects (the students in his ring toss experiment). While the work presented here is only a proof of concept, we do hope to achieve a high level of validity as we refine our approach, so it will be useful to validate our models using techniques from simulation science [106].

As we described earlier, Atkinson divided his subjects into four groups according to their measures (high or low) on achievement motivation and test anxiety. For each of these four groups – H-L, H-H, L-L, L-H – he recorded the percentage of shots that each group took from each of the 15 distances. We ran 10 replications of our simulation and recorded the mean percentages for each group and distance. For each percentage mean we calculated a 95% confidence interval. We consider a model to be valid if the confidence intervals calculated on the simulation percentage means contain the percentages obtained by Atkinson in his experiments with human subjects.

The validation results are presented in Table 8.1. Atkinson analyzed his experimental data by aggregating the shots taken by subjects into three "buckets" representing low, medium, and high difficulty. In Atkinson's analyses the dividing lines between the three buckets were set in four different ways with each yielding similar

Table 8.1: Validation Results. For each subject group the percentage of shots taken by Atkinson's human subjects and by our simulation from each of three ranges is presented along with a 95% confidence interval for the mean percentage of shots in 10 simulated replications of Atkinson's experiment.

Achievement:	High	High	Low	Low
Test Anxiety:	Low	High	Low	High
	Atkinson	Atkinson	Atkinson	Atkinson
	Simulation	Simulation	Simulation	Simulation
Range	Conf. Int.	Conf. Int.	Conf. Int.	Conf. Int.
1-7	**11**	26	18	32
	7.7	14.0	5.6	8.5
	(4.0, 11.4)	(5.6, 22.4)	(1.4, 9.7)	(4.4, 12.5)
8-12	**82**	60	58	48
	75.4	69.0	74.4	80.0
	(65.1, 85.7)	(61.1, 76.9)	(62.0, 86.9)	(74.1, 85.9)
13-15	7	**14**	**24**	20
	16.9	**17.0**	**20.0**	11.5
	(8.8, 25.0)	**(9.4, 24.6)**	**(8.3, 31.7)**	(6.9, 16.2)

results. For brevity we present the division obtained by using both geographical distance and distribution of shots about the median shot of 9.8 ft, in other words, the dividing line one would choose by inspecting the histogram for distinct regions. This strategy resulted in the three buckets listed in the left column of Table 8.1. Each cell of the four subject groups – H-L, H-H, L-L, L-H - contains the percentage of shots taken by Atkinson's subjects, the mean percentage obtained by running 10 replications of our simulation of Atkinson's experiment, and a 95% confidence interval for the mean percentage. While our model did not achieve formal validation, the general patterns of behavior are quite similar to Atkinson's human subject experiment, as shown in Figure 8.4, and we consider these results to be a good proof of concept. We discuss some reasons behind these results and strategies for improvement below.

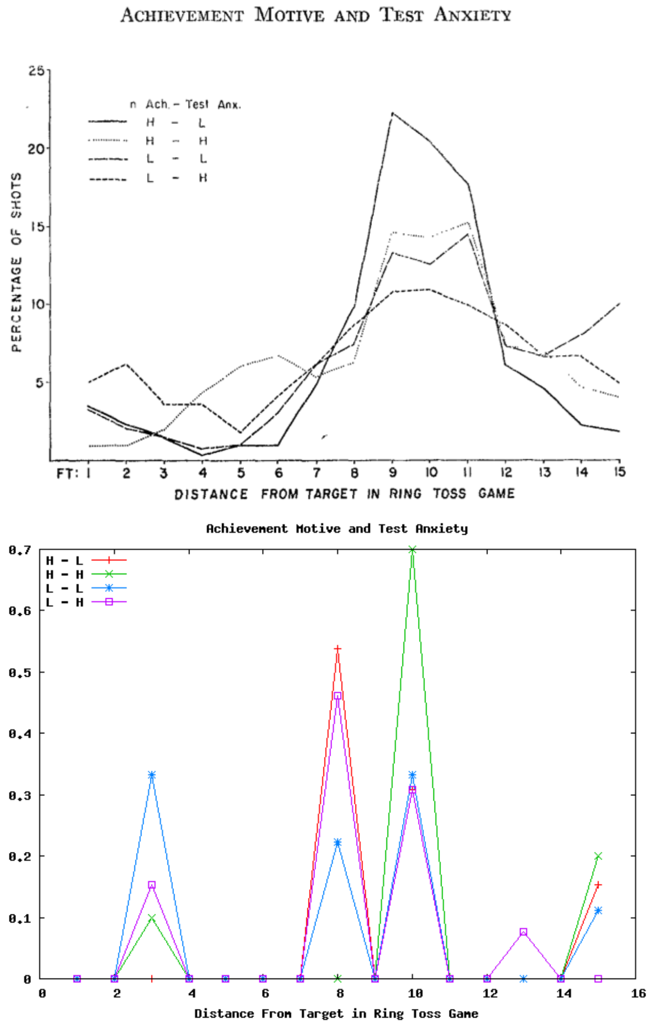

Figure 8.4: The top plot shows the behavior patterns of human subjects in Atkinson's Ring Toss experiment. The bottom plot shows the behavior patterns of our synthetic agents that re-created Atkinson's experiment. Note that Atkinson's plot is smoothed, while ours is not.

8.5 Discussion

We made several assumptions in our models that affected the validation results. First, because we did not have access to Atkinson's original data, only summary presentations, we did not know the exact distribution of motivational factors among his subjects, or even the scales used in his measures. We assumed normally distributed measures and tried several different scales before settling on the values used in the simulations reported here. Second, it is not clear how the valence of behaviors should be translated into reward structures for RL agents. We chose a simple linear reward structure in hopes that the system would be robust to naive encodings. To make our approach widely useful we will need to address the manner in which reward structures are determined.

We chose the Atkinson ring toss experiment on the advice of psychologists who recommended it as a well-known example of trait-oriented behavior theory, and because of its simplicity. However, our goal is to create large agent systems, so future work will need to address scalability – to greater numbers of trait factors and more complex worlds – and generalizability, or transferability, to other domains.

Finally, notice that the example code presented in this paper contains no logic for implementing behavior. The agents and the modules are defined declaratively by specifying a state space, an action set, and a reward structure. The run-time system derives the concrete behavior of the agents automatically from these specifications. This technique, sometimes called partial programming or adaptive programming[107], is a key concept that increases the usability of agent programming by allowing programmers to specify *what* an agent is to do without getting mired in *how* the agent should do it.

8.6 Conclusions and Future Work

Much work remains to make accessible personality-based agent programming systems a reality, and our work is progressing on three paths. First, the integration of reinforcement learning into agent programming systems needs to be studied further so that we know when it is useful and how much detail can be hidden from the agent programmer. This dissertation has confirmed what we already know, namely, that authoring reward functions is not straightforward. We need to be able to specify modules in simpler terms and let the reward structure be derived automatically (we will discuss this further in Chapter 9. Second, the examples presented here were written together so that the reward signals of each agent were directly comparable, which allowed us to use the Greatest-Mass q-decomposition algorithm for combining the modules. Now that we have an arbitration algorithm that is robust to incomparable reward scales, we can either use a `GmAgent` for personality modeling, as we have done here, or extend AFABL with weighting to enable trait-oriented personality modeling. Finally, while AFABL is currently able to handle the personality modeling presented in this chapter, AFABL is still a shallowly-embedded Scala DSL and therefore beyond the programming capabilities of most psychologists. We will need to make AFABL simpler to use. Nevertheless, reinforcement learning provides a promising approach to modeling personality traits and motivational factors in synthetic agents. In particular, it provides us with a means to create agent programming systems that are at least comprehensible by behavioral scientists and harness their knowledge directly while minimizing the need for complex programming.

CHAPTER 9

CONCLUSION

9.1 Review of Major Contributions

This dissertation has reported on two primary contributions: a command arbitration algorithm for robust modular reinforcement learning, and a domain-specific language that supports modular reinforcement learning, AFABL. Our new command arbitration algorithm solves a problem that existed in previous approaches to modular reinforcement learning, namely, that reward scales for all modules had to be comparable. Supporting modules with incomparable rewards allows modules to be written separately, which supports modular agent software engineering, towards which AFABL is a first step. We demonstrated the benefits of language-integrated reinforcement learning in a study of programmers' solutions to agent programming tasks using AFABL compared to solutions using a traditional programming language. AFABL agents are easier to write, are expressed in less complex code, and have more readily reused components than agents written in traditional programming languages. In the remainder of this chapter we discuss some implications of AFABL (as a representative first step in language-integrated modular reinforcement learning), limitations of the current work, and directions for future work.

9.2 Limitations of Current Work

9.2.1 Reward Authoring

As many other researchers have noted, reward authoring is not straightforward for programmers not trained in reinforcement learning. Study participants spent much of their AFABL writing time trying out different reward structures in an effort to improve their agents' performance. Although we provided documentation with hints on how to author reward structures, writing good reward functions is too opaque for most programmers. In the next section we discuss a possible improvement to AFABL which would relieve programmers from writing the reward functions of modules.

9.2.2 Training

Using any reinforcement learning-based programming system requires the availability of a simulation environment to train the learning modules before being used "in production." Using an untrained reinforcement learning agent and accepting that it will perform poorly until it learns is not practical because reinforcement learning algorithms typically require hundreds or thousands of iterations to reach an acceptable level of performance. Separate modules with local state abstractions and reward functions help speed up training, but finding good factorizations into modules is a potentially steep burden to place on the programmer for larger agents.

9.2.3 Host Language Limitations

Writing AFABL agents is writing Scala code, so AFABL programmers must have at least basic competence with Scala and the Scala tool chain. Since it was outside the scope of the present work, we did not try to determine how much of the Scala tooling can be hidden or automated for AFABL programmers. We required study participants to use a recent version of IntelliJ IDEA and provided a pre-configured Scala/AFABL project and an IntelliJ plug-in to automate the time tracking and submission process. Still, several participants had trouble running the study code smoothly, as is often the case with development tools. Many study participants who did not participate in a group session simply abandoned the study. We advertised the study to the Atlanta Scala Meet-up, a group of local software engineers either using Scala professionally or interested in learning. Approximately 15 Scala Meet-up members started the study and only one finished. Due to the number of individual software issues we needed to help participants solve – differing operating systems, IntelliJ versions, etc – we believe many of these dropouts were due to simple software setup issues.

In addition to Scala tooling issues, AFABL programmers must deal with the Scala programming language. For example, when a programmer makes a mistake in their code the error messages come from the Scala compiler and run-time system, not AFABL. Luckily, in the study few people had such issues with AFABL code itself. With more complex agents problems are more likely to occur, and the programmer may be faced with the famous complexity of the Scala type system. The AFABL programmer who is not also a competent Scala programmer has little hope of debugging non-trivial errors.

9.3 Directions for Future Work

9.3.1 Refined Module Types

AFABL currently supports a narrow definition of an agent: a behaving entity with a set of states that must constantly be pursued or avoided. In reinforcement learning these kinds of modules are called continuing tasks, as opposed to episodic goals. Previous versions of AFABL supported a greater set of features but we removed them to focus on AFABL's core for the purpose of this work. With a cleaner core AFABL we could re-implement some of these features and more, which we discuss below. The following features would provide a richer set of agent modeling tools for programmers.

Drives A Drive is a behavior module that runs throughout the life of its containing agent and represents a state that an agent should constantly seek.

Aversions An Aversion module is a behavior module that runs throughout the life of its containing agent and represents a state that an agent should constantly avoid. It is a constraint in the sense that, in certain states, a constraint module will identify actions that should *not* be executed.

Objectives An objective is a short-term goal state that generates a drive module that is active until its goal is achieved. The command arbitrator gives objective modules priority over drive modules, but all modules are constrained by constraint modules.

Tasks A task is a temporally-extended action, a ”mini-policy” that achieves a subgoal. Tasks are equivalent to subtasks (MaxQ), abstract machines (PHAM), or options from hierarchical reinforcement learning. Tasks could be manually authored, or algorithms from hierarchical reinforcement learning could be integrated into AFABL.

9.3.2 Simplified Syntax

The features listed above may make it possible to automatically author reward functions for modules. For example, the Bunny agent for Task 2 from Chapter 6 could be written with Drives and Aversions as shown in Figure 9.3.2

Instead of writing code to specify modules, the programmer specifies states that are to be constantly sought or avoided – expressed as state predicates – and the modules are derived from them automatically. Note also that this proposed syntax

```
val afablBunny2 = AfablAgent(

  world = bunnyWorld,

  drives = Drives(state: BunnyState) {
    (state.bunny == state.food),
    (state.bunny == state.mate)
  },

  aversions = Aversions(state: BunnyState) {
    (state.bunny == state.wolf)
  }

  agentLevelReward = (state: BunnyState) => {
    if (state.bunny == state.wolf) 0.0
    else if (state.bunny == state.food) 1.0
    else if (state.bunny == state.mate) 1.0
    else 0.5
  }
)
```

Figure 9.1: Simplified AFABL syntax with drives and aversions.

does not include state abstraction functions in modules because they could be derived automatically from the states that are to be sought or avoided.

Drama Manager Support

The features discussed above would go along way toward supporting drama managers for intelligent interactive narratives. In addition, a drama manager would need to be able to activate and deactivate modules and inject new objectives to support particular story goals.

9.3.3 General Agent Architecture

The current version of AFABL focuses on integrated reinforcement learning but could easily be extended to support integrated intelligence, that is, mixing of agent modules that employ different kinds of AI algorithms. Because an AFABL agent performs command arbitration over modules that support a behavioral interface (providing an action given a state observation) as opposed to merging elements of reinforcement learners (like Q-values), the modules themselves can employ any mechanism to decide on actions given a state. This information hiding means that AFABL agents could be composed of a mixture of modules that use many different kinds of AI, including statistical learning, rule-based reasoning, or (reactive) planning. In this sense AFABL would be an integrated intelligence architecture.

Knowledge-Based Arbitrators In addition to the modules the arbitrator itself could employ different kinds of algorithms for command arbitration. A knowledge-based arbitrator could use hand-coded logic to decide from among the actions recommended by an agent's modules. Simple arbitrators with few modules to arbitrate can often be coded quite simply as knowledge-based arbitrators.

Hierarchical Decomposition Because modules are themselves agents, modules can contain other modules and perform command arbitration over those modules just as the top-level agent does. Agents can thus be decomposed recursively into behavioral subsystems. This recursive behavior module decomposition would provide the agent designer with great flexibility. Recursive module composition is somewhat similar to the levels of competence in Brooks's subsumption architecture with an important difference: the internal workings of modules are never altered externally. Modules are treated as black-boxes. Command arbitration accomplishes the same result that output suppression does in classic subsumption.

9.3.4 Independent (Non-Embedded) Language

Finally, once the additions to the language are integrated into the internal DSL and studied and refined sufficiently, an external DSL could be considered. Although an external DSL is far more work to implement, the benefits could justify the cost. A stand-alone version of AFABL would have its own set of development tools, report agent-oriented error messages to the user, and potentially run faster than equivalent internal DSL code.

Appendices

APPENDIX A
AFABL PROGRAMMER STUDY

A.1 Experiments

Programmers were randomly assigned to two equally-sized groups: one group used Scala without AFABL first – the Scala-first group – and the other group used AFABL first – the AFABL-first group. Each group completed two programming tasks using Scala and AFABL in the order determined by their group. For each task the programmers were asked to design and implement elegant code that meets the requirements of the task as quickly as possible, balancing the quality of their solutions with time. The idea is to get a good solution quickly, not a perfect solution in a long time.

A.1.1 Task 1: The Bunny-Wolf Domain

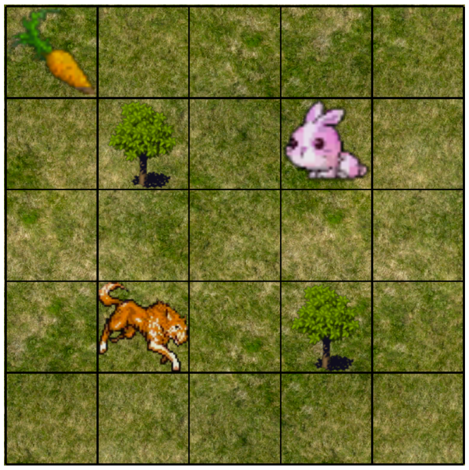

Figure A.1: In the grid world above, the bunny must pursue two goals simultaneously: find food and avoid the wolf. The bunny may move north, south, east, or west. When it finds food it consumes the food and new food appears elsewhere in the grid world, when it meets the wolf it is eaten and "dies."

In this task each programmer wrote an agent that controls a bunny character in a simple world, depicted in Figure A.1. The bunny world works as follows:

- The bunny world is a discrete grid of cells. The bunny, wolf, and food each occupy one cell.
- During each time step the bunny may move north, south, east, or west.
- Every two time steps the wolf moves towards the bunny.
- If the bunny moves to the cell currently occupied by the food, the bunny eats the food, receives a signal from the simulation that it has eaten the food, and new food appears elsewhere.
- If the wolf moves to the cell currently occupied by the bunny it eats the bunny and the episode ends.

The simulation runs several episodes, keeping track of how much food the bunny eats and how long (how many time steps) the bunny "lives" in each episode. Programmers will be asked to write bunny agents that live as long as possible and eat as much food as possible.

A.1.2 Task 2: Mating Bunny

In this task each programmer will write a bunny agent for a world that is identical to the world in Task 1 except that the bunny must also find mates. The world will include one static potential mate that behaves similarly to the food. When the bunny finds the potential mate, the bunny receives a signal that it has "mated," the mate disappears (because it goes off to have babies), and another potential mate appears elsewhere. The simulation runs as in Task 1, additionally keeping track of how many mates the bunny finds. As in Task 1, programmers will be asked to write bunny agents that live as long as possible, eat as much food as possible, and find as many mates as possible.

A.2 Data Collection

Data were collected from programmers in three ways: directly through questionnaires and surveys, automatically with a custom-developed IntelliJ IDEA plug-in that tracked time and submitted solutions, and independently through static and dynamic analysis of programmers' submitted solutions to the programming tasks [108]. The following sections discuss each of the data collection methods.

A.2.1 Programmer Demographics Survey

The demographics survey was be given to participants after they read and agreed to the consent form and before they were given task instructions and assigned to groups.

Purpose The purpose of this survey was to place the programmers in groups according to their programming experience and skills. In particular, the web application used this survey to assign participants to the Scala-first and AFABL-first groups so that each group had similar distributions of programmer experience and skill levels. Survey results were also used to develop other quantitative and qualitative measures relating demographic information to task performance.

Questionnaire

1. What is your level of education?
   (a) High School
   (b) Associate Degree, or currently enrolled in Bachelor degree program
   (c) Bachelor Degree

(d) Master Degree

(e) Doctoral Degree

Rationale: education level can affect programming proficiency.

2. What is your most applicable college major? By most applicable we mean the major you are using most in your profession. For example, if you got a B.S. in electrical engineering and a M.S. in computer science and work as a software engineer, then your most applicable major is computer science.

   Rationale: college major can affect programming proficiency. Additionally, we may code the data to distinguish between technical (but not necessarily computer science) and non-technical majors.

3. Number of years of professional programming experience. You may include years spent working on graduate research projects. Count a semester as .5 years.

   Rationale: programming experience can affect programming proficiency.

4. How proficient are you at game/agent programming?

   (a) Not proficient

   (b) Familiar (have done tutorials or simple examples)

   (c) Proficient (can write programs with multiple objects and files)

   (d) Expert

   Rationale: game/agent programming proficiency is not common in the general programmer population. Knowing programmers proficiencies in game/agent programming will help randomize the experiment groups and allow additional inferences relating game/agent programming proficiency and proficiency with AFABL.

5. How proficient are you at Scala programming?

    (a) Not proficient

    (b) Familiar (have done tutorials or simple examples)

    (c) Proficient (can write programs with multiple objects and files)

    (d) Expert

    Rationale: Scala is our baseline language, and AFABL is embedded in Scala. Scala proficiency will have a profound effect on task proficiency and thus needs to be accounted for in randomizing the experiment groups.

A.2.2 Reflection Survey

The purpose of the reflection survey is to develop a qualitative assessment of programmer satisfaction with AFABL.

Questionnaire For each question, select the degree to which you agree with the statement based on the agent programming tasks you completed for this experiment.

1. I have a positive impression of agent programming in AFABL.

    (a) Strongly disagree

    (b) Disagree

    (c) Neutral

    (d) Agree

    (e) Strongly Agree

    Rationale: programmers impression of Scala will provide a baseline for evaluating programmers impression of AFABL.

2. I found it easier to write the agents using AFABLs programming constructs compared to bare Scala.

   (a) Strongly disagree

   (b) Disagree

   (c) Neutral

   (d) Agree

   (e) Strongly Agree

   Rationale: the point of AFABL is to facilitate agent programming, so programmers should have a more positive impression of AFABL for agent programming.

3. I believe that AFABL facilitated more reusable and maintainable code for agents compared to bare Scala.

   (a) Strongly disagree

   (b) Disagree

   (c) Neutral

   (d) Agree

   (e) Strongly Agree

   Rationale: answers to this question should correlate with answers to Question 1.

4. If given the choice, I would choose AFABL over Scala for agent programming projects.

   (a) Strongly disagree

   (b) Disagree

(c) Neutral

(d) Agree

(e) Strongly Agree

Rationale: answers to this question should correlate with answers to Question 2.

5. I found it easier to use AFABL compared to Scala for Task 1.

    (a) Strongly disagree

    (b) Disagree

    (c) Neutral

    (d) Agree

    (e) Strongly Agree

    Rationale: in addition to objective analyses of task submissions, we want to know whether programmers subjectively prefer AFABL.

6. What was it about AFABL that made the Task 1 easier or harder?

    Rationale: we want to get open-ended feedback for things we did not anticipate.

7. I found it easier to use AFABL compared to Scala for Task 2.

    (a) Strongly disagree

    (b) Disagree

    (c) Neutral

    (d) Agree

    (e) Strongly Agree

Rationale: in addition to objective analyses of task submissions, we want to know whether programmers subjectively prefer AFABL.

8. What was it about AFABL that made the Task 2 easier or harder?

   Rationale: we want to get open-ended feedback for things we did not anticipate.

A.3 Evaluation

Using results from the pilot study, we evaluated the internal consistency of the survey by calculating the Cronbach alpha coefficients for the following constructs:

1. User satisfaction with Scala for agent programming tasks.
   - Questions 1 and 3
2. User satisfaction with AFABL for agent programming tasks.
   - Questions 2 and 4
3. User preference for AFABL over Scala for agent programming tasks.
   - Questions 5 and 7

The Cronbach alpha coefficient measures the correlation between the answers to questions that measure the same construct and is given by:

$$\alpha = \frac{k}{k-1} \times (1 - \frac{s_T^2 - \sum s_I^2}{s_T^2})$$

where

- s_T^2 is the total variance of all the items (questions) for a construct
- s_I^2 is the variance of an individual item, and

- k is the number of items.

Typically a construct is considered valid if its Cronbach alpha coefficient is at least 0.7. As we report in Chapter 6, all Cronbach alpha coefficients were greater than 1.

A.4 Source Code Analysis

The code submitted for each task was analyzed to determine:

- How much code was required for each task with and without AFABL.
- How consistent the solutions were between programmers in each task with and without AFABL. Did AFABL lead to more consistent designs?
- How well the programmers understood the problem.

A.5 Run-time Analysis

The performance of the solutions submitted for each task was recorded for comparison. The purpose of this comparison was to determine how much effort was required to get good performance, not to get the best possible performance.

A.5.1 Logistics

The experiments were conducted online to provide easy access to the greatest number of participants. Participants:

1. Register online with (optional) confidential identifying information.
2. Complete demographics survey. The web application used the demographic survey to place participants in the Scala-first or AFABL-first groups.
3. Downloaded code and task instructions.

4. Completed Task 1 in Scala or AFABL.

5. Complete Task 2 in Scala or AFABL.

6. Submitted their solutions using our IntelliJ IDEA plug-in.

7. Completes a reflection survey.

REFERENCES

[1] R. Sutton and A. Barto, *Reinforcement learning: An introduction.* Cambridge, MA: MIT Press, 1998.

[2] L. P. Kaelbling, M. L. Littman, and A. P. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[3] R. E. Bellman, *Dynamic programming.* Princeton University Press, 1957.

[4] D. P. Bertsekas, *Dynamic programming and optimal control*, 3rd ed. Athena Scientific, 2013, vol. 1,2.

[5] R. Howard, "Dynamic programming and markov processes," PhD thesis, 1960.

[6] J. Van Nunen, "A set of successive approximation methods for discounted markovian decision problems," *Zeitschrift fuer operations research*, vol. 20, no. 5, pp. 203–208, 1976.

[7] M. L. Puterman and M. C. Shin, "Modified policy iteration algorithms for discounted markov decision problems," *Management Science*, vol. 24, no. 11, pp. 1127–1137, 1978.

[8] S. Russell and P. Norvig, *Artificial intelligence: A modern approach.* Upper Saddle River, NJ: Prenticce Hall, 2003.

[9] C. J. Watkins, "Models of delayed reinforcement learning," PhD thesis, Ph. D. thesis, Cambridge University, 1989.

[10] G. A. Rummery and M. Niranjan, *On-line q-learning using connectionist systems.* University of Cambridge, Department of Engineering, 1994.

[11] D. Precup, R. S. Sutton, and S. Singh, "Theoretical results on reinforcement learning with temporally abstract options," in *Machine Learning: ECML-98: 10th European Conference on Machine Learning Chemnitz, Germany, April 21–23, 1998 Proceedings*, C. Nédellec and C. Rouveirol, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 382–393, ISBN: 978-3-540-69781-7.

[12] R. S. Sutton, D. Precup, and S. P. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[13] D. Precup, "Temporal abstraction in reinforcement learning," PhD thesis, University of Massacheusetts Amherst, 2000.

[14] T. G. Dietterich, "The MAXQ method for hierarchical reinforcement learning," in *Proc. 15th International Conf. on Machine Learning*, Morgan Kaufmann, San Francisco, CA, 1998, pp. 118–126.

[15] ——, "Hierarchical reinforcement learning with the maxq value function decomposition," *Journal of Artificial Intelligence Research*, vol. 13, pp. 227–303, 2000.

[16] R. Parr and S. Russell, "Reinforcement learning with hierarchies of machines," in *Advances in Neural Information Processing Systems*, M. I. Jordan, M. J. Kearns, and S. A. Solla, Eds., vol. 10, The MIT Press, 1998.

[17] D. Andre and S. Russell, "Programmable reinforcement learning agents," in *Advances in Neural Information Processing Systems (NIPS)*, vol. 13, MIT Press, 2000.

[18] ——, "State abstraction for programmable reinforcement learning agents," in *AAAI-02*, Edmonton, Alberta: AAAI Press, 2002.

[19] B. Marthi, S. Russell, D. Latham, and C. Guestrin, "Concurrent hierarchical reinforcement learning," in *IN PROCEEDINGS IJCAI-2005*, 2005, pp. 779–785.

[20] S. Russell and A. L. Zimdars, "Q-decomposition for reinforcement learning agents," in *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)*, Washington, D.C., 2003.

[21] N. Sprague and D. Ballard, "Multiple-goal reinforcement learning with modular sarsa(o)," in *IJCAI'03: Proceedings of the 18th international joint conference on Artificial intelligence*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, pp. 1445–1447.

[22] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986.

[23] S. Singh and D. Cohn, "How to dynamically merge markov decision processes," in *Advances in Neural Information Processing Systems*, vol. 10, 1998.

[24] N. Sprague and D. Ballard, "Eye movements for reward maximization," in *Advances in neural information processing systems*, 2003, None.

[25] N. Sprague, D. Ballard, and A. Robinson, “Modeling embodied visual behaviors,” *ACM Transactions on Applied Perception (TAP)*, vol. 4, no. 2, p. 11, 2007.

[26] G. Konidaris and A. Barto, “An adaptive robot motivational system,” in *International Conference on Simulation of Adaptive Behavior*, Springer, 2006, pp. 346–356.

[27] N. Aissani, B. Beldjilali, and D. Trentesaux, “Dynamic scheduling of maintenance tasks in the petroleum industry: A reinforcement approach,” *Engineering Applications of Artificial Intelligence*, vol. 22, no. 7, pp. 1089–1103, 2009.

[28] T. Chaari, S. Chaabane, N. Aissani, and D. Trentesaux, “Scheduling under uncertainty: Survey and research directions,” in *Advanced Logistics and Transport (ICALT), 2014 International Conference on*, IEEE, 2014, pp. 229–234.

[29] J. P. Rowe and J. C. Lester, “A modular reinforcement learning framework for interactive narrative planning,” in *Proceedings of the 9th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013, pp. 57–63.

[30] S. Bhat, C. Isbell, and M. 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, 2006.

[31] K. J. Arrow, *Social choice and individual values*, 2nd. John Wiley and Sons, 1963.

[32] K. W. Roberts, “Interpersonal comparability and social choice theory,” *The Review of Economic Studies*, pp. 421–439, 1980.

[33] R. Zhang, Z. Song, and D. H. Ballard, “Global policy construction in modular reinforcement learning.,” in *AAAI*, 2015, pp. 4226–4227.

[34] K. Rohanimanesh and S. Mahadevan, “Decision-theoretic planning with concurrent temporally extended actions,” in *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, Morgan Kaufmann Publishers Inc., 2001, pp. 472–479.

[35] ——, “Learning to take concurrent actions,” in *Advances in neural information processing systems*, 2002, pp. 1619–1626.

[36] Q. P. Lau, M. L. Lee, and W. Hsu, “Coordination guided reinforcement learning,” in *Proceedings of the 11th International Conference on Autonomous*

Agents and Multiagent Systems-Volume 1, International Foundation for Autonomous Agents and Multiagent Systems, 2012, pp. 215–222.

[37] P. Zang, P. Zhou, D. Minnen, and C. L. Isbell, "Discovering Options from Example Trajectories," in *Proceedings of the Twenty-sixth International Conference on Machine Learning (ICML)*, (ICML), 2009.

[38] P. Zang, R. Tian, C. L. Isbell, and A. Thomaz, "Batch versus interactive learning by demonstration," in *Proceedings of the Ninth International Conference on Development and Learning (ICDL)*, (ICDL), 2010.

[39] P. Zang, A. Irani, P. Zhou, C. L. Isbell, and A. Thomaz, "Using Training Regimens to Teach Expanding Function Approximators," in *Proceedings of the Ninth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, (AAMAS), 2010.

[40] L. C. C. Rus, P. Zang, C. L. Isbell, and A. Thomaz, "Automatic State Abstraction from Demonstration," in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI)*, (IJCAI), 2011.

[41] L. C. C. Rus, C. L. Isbell, and A. Thomaz, "Automatic Decomposition and State Abstraction from Demonstration," in *Proceedings of the Eleventh International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, (AAMAS), 2012.

[42] L. C. C. Rus, K. Subramanian, C. L. Isbell, A. Lanterman, and A. Thomaz, "Abstraction from Demonstration for Efficient Reinforcement Learning in High-Dimensional Domains," *Artificial Intelligence*, vol. 216, no. 0, pp. 103–128, 2014.

[43] L. C. C. Rus, C. L. Isbell, and A. Thomaz, "Object Focused Q-learning for Autonomous Agents," in *Proceedings of the Twelfth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, (AAMAS), 2013.

[44] J. Scholz, M. Levihn, C. L. Isbell, and D. Wingate, "A Physics-Based Model Prior for Object-Oriented MDPs," in *Proceedings of the 31st International Conference on Machine Learning (ICML)*, (ICML), 2014.

[45] E. Catto. (2013). Box2d physics engine, (visited on 2013).

[46] J. Scholz, M. Levihn, C. L. Isbell, H. Christensen, and M. Stilman, "Learning Non-Holonomic Object Models for Mobile Manipulation," in *Proceedings of the the 2015 IEEE International Conference on Robotics and Automation (ICRA)*, (ICRA), 2015.

[47] J. Scholz, J. Nehchal, M. Levihn, and C. L. Isbell, "Navigation Among Movable Obstacles with Learned Dynamic Constraints," in *Proceedings of the the 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (IROS), 2016.

[48] M. D. McIlroy, J Buxton, P. Naur, and B. Randell, "Mass-produced software components," in *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, sn, 1968, pp. 88–98.

[49] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, pp. 131–183, Jun. 1992.

[50] W. B. Frakes and K. Kang, "Software reuse research: Status and future," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 529–536, 2005.

[51] G. Polančič, M. Heričko, and I. Rozman, "An empirical examination of application frameworks success based on technology acceptance model," *Journal of Systems and Software*, vol. 83, no. 4, pp. 574–584, 2010.

[52] V. R. Basili, L. C. Briand, and W. L. Melo, "How reuse influences productivity in object-oriented systems," *Commun. ACM*, vol. 39, no. 10, pp. 104–116, Oct. 1996.

[53] P. Mohagheghi and R. Conradi, "An empirical investigation of software reuse benefits in a large telecom product," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 3, 13:1–13:31, Jun. 2008.

[54] C. Gacek, "Exploiting domain architectures in software reuse," *SIGSOFT Softw. Eng. Notes*, vol. 20, no. SI, pp. 229–232, Aug. 1995.

[55] P. J. Landin, "The next 700 programming languages," *Communications of the ACM*, vol. 9, no. 3, pp. 157–166, 1966.

[56] D. D. Chamberlin and R. F. Boyce, "Sequel: A structured english query language," in *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, ACM, 1974, pp. 249–264.

[57] M. Fowler, *Domain specific languages*. Addison-Wesley, 2011.

[58] P. Hudak, "Building domain-specific embedded languages," *ACM COMPUTING SURVEYS*, vol. 28, 1996.

[59] ——, "Modular domain specific languages and tools," *Proceedings of the Fifth International Conference on Software Reuse*, pp. 134–142, 1998.

[60] M. P. Ward, "Language-oriented programming," *Software-Concepts and Tools*, vol. 15, no. 4, pp. 147–161, 1994.

[61] L. Lamport and A LaTEX, *Document preparation system.* Addison-Wesley Reading Mass, 1986.

[62] D. E. Knuth and D. Bibby, *The texbook.* Addison-Wesley Reading, 1984, vol. 3.

[63] R. M. Stallman, *The gnu emacs 24.4 reference manual.* Samurai Media Limited, 2014.

[64] J. M. Neighbors, "The draco approach to constructing software from reusable components," *IEEE Transactions on Software Engineering*, no. 5, pp. 564–574, 1984.

[65] D. H. Lorenz and B. Rosenan, "Code reuse with language oriented programming," in *Top Productivity through Software Reuse*, ser. Lecture Notes in Computer Science, K. Schmid, Ed., vol. 6727, Springer Berlin / Heidelberg, 2011, pp. 167–182.

[66] S. Dmitriev, "Language oriented programming: The next programming paradigm," *JetBrains onBoard*, vol. 1, no. 2, pp. 1–13, 2004.

[67] B. Rosenan, "Designing language-oriented programming languages," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ACM, 2010, pp. 207–208.

[68] A. J. Albrecht, "Measuring application development productivity," in *Proceedings of the joint SHARE/GUIDE/IBM application development symposium*, vol. 10, 1979, pp. 83–92.

[69] M. H. Halstead, *Elements of software science.* Elsevier New York, 1977, vol. 7.

[70] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE transactions on Software Engineering*, no. 5, pp. 510–518, 1981.

[71] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[72] T. J. McCabe and C. W. Butler, "Design complexity measurement and testing," *Communications of the ACM*, vol. 32, no. 12, pp. 1415–1425, 1989.

[73] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[74] E. J. Weyuker, "Evaluating software complexity measures," *IEEE transactions on Software Engineering*, vol. 14, no. 9, pp. 1357–1365, 1988.

[75] L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68–86, 1996.

[76] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on software engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[77] G. K. Gill and C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *IEEE transactions on software engineering*, vol. 17, no. 12, pp. 1284–1288, 1991.

[78] B. Curtis, S. B. Sheppard, P. Milliman, M. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics," *IEEE Transactions on software engineering*, no. 2, pp. 96–104, 1979.

[79] P. Norvig and D. Cohn. (1998). Adaptive software, (visited on 1998).

[80] M. Odersky, L. Spoon, and B. Venners, *Programming in scala*, 1st ed. Artima, 2008.

[81] M. Odersky and M. Zenger, "Scalable component abstractions," in *OOPSLA '05: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[82] D. Andre and S. J. Russell, "Programmable reinforcement learning agents," *Advances in neural information processing systems*, pp. 1019–1025, 2001.

[83] T. Bauer, "Adaptation-based programming," PhD thesis, Oregon State University, 2013.

[84] T. Bauer, M. Erwig, A. Fern, and J. Pinto, "Adaptation-based programming in java," in *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, ACM, 2011, pp. 81–90.

[85] ——, "Adaptation-based programming in haskell," *arXiv preprint arXiv:1109.0774*, 2011.

[86] M. Mateas and A. Stern, "Life-like characters. tools, affective functions and applications," in, H. Prendinger and M. Ishizuka, Eds. Springer, 2004, ch. A Behavior Language: Joint Action and Behavioral Idioms.

[87] B. L. Welch, "The generalization ofstudent's' problem when several different population variances are involved," *Biometrika*, vol. 34, no. 1/2, pp. 28–35, 1947.

[88] T. Peters. (2004). Pep 20 – the zen of python, (visited on 2016).

[89] C. S. Dweck and E. L. Leggett, "A social-cognitive approach to motivation and personality," *Psychological Review*, vol. 95, no. 2, pp. 256–273, 1988.

[90] D. Cervone and L. A. Pervin, *Personality: Theory and research*. John Wiley and Sons, 2009.

[91] D. Cervone and Y. Shoda, "The coherence of personality: Social-cognitive bases of consistency, veriability, and organization," in, D. Cervone and Y. Shoda, Eds. New York: Guilford Press, 1999, ch. Social-cognitive Theories and the Coherence of Personality, pp. 3–33.

[92] R. R. McCrae and J. Paul T. Costa, "Handbook of personality: Theory and research," in, O. John, R. Robins, and L. Pervin, Eds. New York: Guilford, 2008, ch. The Five-Factor Theory of Personality, pp. 159–181.

[93] A. J. Elliot and T. M. Thrash, "Approach-avoidance motivation in personality: Approach and avoidance temperaments and goals," *Journal of Personality and Social Psychology*, vol. 82, no. 5, pp. 804–818, 2002.

[94] W. Mischel and Y. Shoda, "Handbook of personality: Theory and research," in, O. John, R. Robins, and L. Pervin, Eds. New York: Guilford, 2008, ch. Toward a Unified Theory of Personality: Integrating Dispositions and Processing Dynamics within the Cognitive-Affective Processing System, pp. 208 –241.

[95] R. M. Jones, "An introduction to cognitive architectures for modeling and simulation," in *Proceedings of the Interservice/Industry Training/Simulation and Education Conference*, 2005.

[96] P. Langley, J. E. Laird, and S. Rogers, "Cognitive architectures: Research issues and challenges," *Cognitive Systems Research*, 2008.

[97] B. E. John, "Cognitive modeling for human-computer interaction," in *Invited paper in the Proceedings of Graphics Interface '98*, Canadian Human-Computer Communications Society, Vancouver, British Columbia, Canada, 1998.

[98] J. E. Laird, “Extending the soar cognitive architecture,” in *Proceedings of the First Conference on Artificial General Intelligence (AGI-08)*, 2008.

[99] J. R. Anderson, D. Bothell, and M. D. Byrne, “An integrated theory of the mind,” *Psychological Review*, vol. 111, no. 4, pp. 1036–1060, 2004.

[100] S. Nason and J. E. Laird, “Soar-rl: Integrating reinforcement learning with soar,” in *6th International Conference on Cognitive Modeling*, Pittsburgh, PA, 2008.

[101] A. B. Loyall and J. Bates, “Hap: A reactive adaptive architecture for agents,” Tech. Rep. CMU-CS-91-147, 1991.

[102] M. Mateas and A. Stern, “Life-like characters. tools, affective functions and applications,” in, H. Prendinger and M. Ishizuka, Eds. Springer, 2004, ch. A Behavior Language: Joint Action and Behavioral Idioms.

[103] J. Gratch and S. Marsella, “Lessons from emotion psychology for the design of lifelike characters,” *Journal of Applied Artificial Intelligence (special issue on Educational Agents - Beyond Virtual Tutors)*, vol. 19, no. 3-4, pp. 215–233, 2005.

[104] W. Swartout, J. Gratch, R. Hill, E. Hovy, S. Marsella, J. Rickel, and D. Traum, “Toward virtual humans,” *AI Magazine*, vol. 27, no. 1, 2006.

[105] J. W. Atkinson and G. H. Litwin, “Achievement motive and test anxiety conceived as motive to approach success and motive to avoid failure,” *Journal of Abnormal and Social Psychology*, vol. 60, no. 1, pp. 52–63, 1960.

[106] A. M. Law, *Simulation modeling and analysis*, 4th. McGraw-Hill, 2007.

[107] C. Simpkins, S. Bhat, C. Isbell, and M. Mateas, “Towards adaptive programming: Integrating reinforcement learning into a programming language,” in *OOPSLA ’08: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Onward! Track*, Nashville, TN USA, 2008.

[108] J. Singer, S. E. Sim, and T. C. Lethbridge, “Software engineering data collection for field studies,” in, F. Shull, J. Singer, and D. I. Sjøberg, Eds. Springer, 2008, ch. 1, pp. 9–34.

VITA

Chris Simpkins is the oldest of four brothers. An Air Force brat born in Biloxi, Mississippi at Keesler AFB, he lived in Germany from ages 6 through 13. He speaks German and French and enjoys returning to Europe to teach in study abroad programs and see old friends. After his dad retired from the Air Force Chris moved with his family to Shady Spring, West Virginia.

Chris began his professional life in the U.S. Air Force where he was a pilot, software engineer, and instructor. He was selected as the first ever first-assignment space instructor upon graduation from Undergraduate Space Training (UST) in 1992, where he taught basic Newtonian mechanics. He returned to flying after three years as a classroom instructor and interactive courseware developer to fly the KC-135, rising to the highest co-pilot position (Stan/Eval), being one of only two co-pilots in his squadron to earn special-ops qualification, and winning an outstanding crew award while deployed in Saudi Arabia. He finished his Air Force career doing one of the things he love most: teaching, this time as a T-37 instructor pilot at Columnbus AFB, MS. He left the Air Force in 2000 to pursue his other professional passion: computing.

Chris is currently a Lecturer in Computer Science at the Georgia Institute of Technology. He completed his MS in Computer Science in 2004 specializing in Artificial Intelligence. A 1990 graduate of the United States Air Force Academy, his background includes research, software engineering, flying, and teaching. During 15 years as a professional software engineer in private industry, the military, and applied research, he built and delivered dozens of successful enterprise-scale and single-user systems, mostly as chief architect or lead software engineer. As a researcher, he has applied machine learning to text analysis, radio emmiter identification, automated antenna design, and adaptive agent technology.