Pickle: Evolving Serializable Agents for Collective Animal Behavior Research

by

Terrance Nelson Medina

(Under the direction of Maria Hybinette)

Abstract

Agent-based Modeling and Simulation has become a mainstream tool for use in business and research in multiple disciplines. Along with its mainstream status, ABMS has attracted the attention of practitioners who are not comfortable developing software in Java, C++ or any of the scripting languages commonly used for ABMS frameworks. In particular, animal behavior researchers, or ethologists, require agent controllers that can describe complex animal behavior in dynamic, unpredictable environments. But the existing solutions for expressing agent controllers require complicated code. We present Pickle, an ABMS platform that automatically generates complete simulations and agents using behavior-based controllers from simple XML file descriptions. This novel approach allows for new approaches to understanding and reproducing collective animal behavior, by controlling and evolving the fundamental behavioral mechanisms of its agents, rather than only the physical parameters of their sensors and actuators.

Index words: Behavior-based Robot control architecture Ethology Modeling Simulation XML Scala

Pickle: Evolving Serializable Agents for Collective Animal Behavior Research

by

Terrance Nelson Medina

B.Mus., University of Cincinnati, 1998

B.S., University of Georgia, 2009

A Dissertation Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Master of Computer Science

Athens, Georgia

2015

Pickle: Evolving Serializable Agents for Collective Animal Behavior Research

by

Terrance Nelson Medina

Approved:

Major Professor:	Maria Hybinette
Committee:	John Miller Tucker Balch

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
August 2015

Pickle: Evolving Serializable Agents for Collective Animal Behavior Research

Terrance Nelson Medina

July 17, 2015

Dedication

This work is dedicated to my wife, Amanda Knisely-Medina, for putting up with the many late nights when I was locked in my office. To my father, Dr. Augusto Medina, who taught me to never stop trying. And to my mother, Norma Dinell Medina, whose last words to me were "I love you, now go finish your research."

Contents

List of Figures viii

List of Tables xi

1 Introduction 1

1.1 Background . . . 2

1.2 Modeling and Simulation . . . 4

1.3 Agent Controllers . . . 5

1.4 Evolutionary Algorithms . . . 12

1.5 Ethology . . . 14

1.6 Collective animal behavior . . . 15

1.7 Summary . . . 21

2 Pickle 22

2.1 Design of the Simulation Framework . . . 22

2.2 Comparison with Previous Work . . . 32

2.3 Implementation of the Simulation Framework . . . 34

3 The Genetic Programming Framework and Experimental Platform 37

3.1 Design . . . 38

3.2 Implementation of the Experimental Platform . . . 39

4 Experimental Results **47**

4.1 Comparison of Code Complexity 56

5 Conclusion **59**

Bibliography **61**

List of Figures

1.1 A hypothetical deliberative-style control architecture. Sensor input is passed to a long-term goal planner, which filters the information down to subordinate processes like path planning and mapping, as necessary. Unlike reactive controllers, there is typically a lot of computation between the sensor input and the motor output. 7

1.2 Reactive controllers emphasize tight coupling of sensor inputs and motor outputs, with minimal deliberation in between. 9

1.3 Visualization of a hybrid controller 11

1.4 A school of fish in the Florida Keys National Marine Sanctuary. These fish are assuming a "funnel" shape. Photo by Dr. Jiangang Luo, courtesy of the National Oceanographic and Atmospheric Administration. 16

1.5 Typical schooling behavior in small fish. These are schooling over the Geyer Bank in the Gulf of Mexico. Photo courtesy of the National Oceanographic and Atmospheric Administration. 17

1.6 A depiction of the zones of attraction, orientation and repulsion, described by Iain Couzin in his work on collective flocking. 18

2.1 A high-level view of Pickle's architecture. Pickle is essentially an application layer that reads input from a Configuration Layer consisting of XML descriptions for simulation parameters, Agents and Controllers. It uses a set of drivers as a translation interface to run on multiple simulation kernels. . 23

2.2 A description of the Pickle taxonomy. Agents are distinguished from other phenomena by the presence of sensors, which consume information about the simulated environment, actuators that effect changes to the environment, and a controller that binds sensors with actuators. 24

2.3 A view of how pie slice regions for sensors work. Each Sensor operates within a single pie-slice region, and Agents may have multiple sensors. 26

2.4 The interface for the simulation driver in our prototype implementation. A suitable driver implementation would need to fulfill these functions using its simulation kernel. 31

3.1 The predator controller used in our genetic programming experiments. This controller consists of a priority or subsumption coordination operator that chooses between repulsion from pool edges, repulsion from other predators, attraction to prey and random motion. 41

3.2 The boid-like controller that we hand-coded as the control in our experiments. It consists of a priority coordination operator that chooses between avoiding pool edges, avoiding predators, closing in on other prey, avoiding collisions with other prey and random motion. 43

4.1 A frequency distribution of the fitness evaluations for the hand-coded flocking controller. 48

4.2 A screen shot of the flocking behavior displayed by our control group of "hand-coded" boid-like agents. Notice that the flocks grouping are fairly compact. 48
4.3 A frequency distribution of the fitness evaluations for the evolved flocking controller. This controller shows a slight improvement over the hand-coded flocking controller. 49
4.4 A screen shot of the evolved flocking behavior displayed by the most fit individual in generation 50 of the fourth set of evolutionary experiments. Notice that the flocks appear looser, more expansive and larger. 49
4.5 A frequency distribution of the fitness evaluations for the evolved "swarming" controller. This controller outperforms the hand-coded flocking controller by 30%. 49
4.6 A screen shot of the evolved "swarming" behavior displayed by the most fit individual in generation 12 of the fifth set of evolutionary experiments. . . . 49
4.7 An evolved controller that exhibits distinctive flocking behavior. 51
4.8 An evolved controller that enables superior predator evasion, but does not use flocking. 52
4.9 A side-by-side comparison of the fitness of the hand-coded, evolved flocking and evolved swarming controllers. The evolved swarming controller outperforms the hand-coded flocking controller by 30%. 52
4.10 A plot of the "most fit" members from the fourth set of evolution trials. This shows a steady improvement in the maximum fitness of the population. This population produced flocking behavior starting at generation 28. 53
4.11 A screen capture of Minnows that showed promising flocking behavior. Note that they clump together, making them easy targets for predators. 54

List of Tables

3.1 Physical attributes and sensors of the predators ("Shark"s) and prey ("Minnow"s). 40

3.2 Description of the evolution parameters for five different sets of experiments. 45

4.1 Description of outcomes from each set of experiments. 48

4.2 Average fitness and relative frequency of individual motor schema "genes". These numbers were produced by parsing all of the evolved individuals from Experiment Set 4. 55

4.3 Results from our complexity evaluation of Pickle. We found that Pickle produced accurate agent behaviors with less complex code as measured by the number of lines and number of files required. 58

Chapter 1

Introduction

In this chapter, we will introduce the fundamental concepts behind our research. First, we will frame the problem that we are addressing and introduce our solution. In the remaining sections, we will define the fields of Agent-based Modeling and Simulation, including their heritage from discrete event based simulation and parallel simulation [Section 1.2]. We will discuss agent control strategies, with special emphasis on behavior-based robotics. We will also include a brief history of robot control architectures to show the logical progression from hierarchical to reactive and ultimately hybrid controllers [Section 1.3]. Next we will give a brief overview of Ethology, with emphasis on its relationship to research in collective animal behavior [Section 1.5]. We will present the design and implementation of Pickle, our platform for modeling agents using robot control architectures [Section 2.1]. Finally, we will demonstrate the usefulness of Pickle by using it to show a contribution to the field of collective animal behavior research [Section 3].

1.1 Background

Ethology, the science of animal behavior, can be a time-consuming occupation. Ethologists spend hours observing animals, sometimes in the lab, but preferably in the field. They use a notebook to record their observations in minute detail, and then translate those observations into an *ethogram*, or a graphical depiction of the animal's behavior that resembles a finite state machine or Markov model.

This can be a difficult and time-consuming process, and if the ethologist wants to use an executable model - a piece of software that runs in a simulated environment - as a research tool, the task becomes even more difficult [Balch et al., 2006]. The ethogram would have to be programmed by hand into executable code in a programming language like Java or C++, that targets a particular simulation platform such as Repast or MASON. This is not only time and labor intensive, but could also present significant problems for animal researchers who are not experienced programmers [Minar et al., 1996]. We believe this presents an opportunity to simplify or even automate the process of producing executable models of animal behavior.

The research field of multiagent systems has provided some approaches to simplify controller generation by constructing toolkits that automatically generate agent controllers from visual specifications. Some frameworks, like the Agent Modeling Platform (AMP) [Parker, 2015] for Eclipse, allow users to visually construct hierarchical controllers that specify sequences of actions for agents to take and conditions under which they should execute them. Unfortunately, these sequential approaches are often inadequate for describing the behaviors of living creatures in dynamic, unpredictable environments. Repast Simphony (sic) features the Statecharts modeling tools, which allow a user to graphically depict states and transitions between them, but the actual behavior functions of each state must still be hand-coded by the developer [Ozik and Collier, 2014]. Other frameworks, such as easyABMS [Garro and

Russo, 2010] or INGENIAS [Pavón and Gómez-Sanz, 2003] use the Unified Modeling Language (UML) or a similar software-oriented modeling language to allow users to visually diagram an agent in terms of views and relationships. However, UML is not likely to be a familiar language for animal behavior researchers, who would benefit from an intuitive interface that more closely resembles the ethogram depictions that they already use.

In this work we present Pickle, an Agent-based modeling framework that uses behavior-based robot control architectures as a basis for describing agent controllers. Behavior-based models, also known as hybrid controllers, incorporate the advantages of sequential and UML-based controllers, while overcoming many of their shortcomings. Similarly to sequential controllers, behavior-based controllers allow agents to operate under different rules in different circumstances, but whereas sequential controllers are tied to rigid sequences of actions, behavior-based controllers allow for emergent behavior driven by a stream of input from their sensors. This allows them to be more adaptable to changes in their environment.

Pickle simulations use semi-structured data files to describe not only the simulation parameters, but also the agents themselves including their behavior controllers. This allows the entire simulation to be not only machine readable, but machine writable. This means that we can arbitrarily modify semi-structured data and even randomly generate controllers from scratch, given a schema description of what those controllers should include. It also means that entire simulations and agents may be serialized for transmission, collaboration and storage, independently of the source code for the simulation environment in which they run.

We have designed Pickle to be easily extendible by experienced programmers, able to be run on multiple simulation kernels, and explicitly capable of supporting automatic generation and modification of agent XML descriptions.

Finally, to demonstrate the usefulness of our platform, we make a contribution to the field of collective animal behavior research by using genetic programming techniques to evolve

controllers that avoid predators in a contained space. Previous work has focused on evolving the sensor parameters of prey agents, but using Pickle's serializable controllers, we are able to evolve the actual control architecture itself. Using Pickle, we show that flocking behavior does in fact evolve in response to predation, but that, particularly in confined spaces, it is not always the best solution to predator evasion. Furthermore, we are able to quantify the relative importance of the primary behavioral traits of our controllers with respect to predator evasion.

1.2 Modeling and Simulation

Modeling and simulation systems rely on descriptions of an entity within an environment (a model), to try to predict the state of the environment, or system, at a given offset from the initialization time. The two conventional approaches to M&S are continuous and discrete event systems.

Continuous simulation relies on sets of differential equations as models to predict the state of a system at any arbitrary time t, where the values of t are uncountably infinite.

Discrete-event simulation (DES) is used to predict the state of a system at discrete points in time. The system changes state only at the edges of those discrete time points. Models in a DES typically use random variables (e.g., from the Poisson distribution) as models. For example, a Poisson distribution might be used to model the time between arrival times (inter-arrival time) of phone calls at a call center.

Process, or parallel simulation is a further refinement of DES, which keeps the features of a DES, but in addition, models actors within the simulation. These actors are typically implemented as interdependent threads which create and handle events as they run in the simulation environment. It is important to note that process simulation is used to model

the same kinds of systems as discrete event simulation. The difference is primarily one of implementation: multi-threaded and event based, versus single-threaded and synchronous.

Agent-based Modeling and Simulation (ABMS) can be seen as a logical extension of process simulation, but it is actually orthogonal to both process and discrete-event methodologies. That is, an ABMS could be implemented as either a single-threaded controller or a multi-threaded set of interacting processes. In ABMS, the actors, or agents, are autonomous software entities that follow the "sense-think-act" operational life cycle. Most importantly, ABMS uses autonomous software processes to model complex systems using emergent behavior. The key observation is that complex systems such as financial markets, social animal communities and airport runways can be difficult to model explicitly using differential equations or random variables [North and Macal, 2007]. The ABMS approach attempts to overcome this challenge by treating a simulation as a system of autonomous agents, each having some set of instructions that controls its interactions with other agents and the environment. This allows complex behaviors to emerge from the system, instead of having to explicitly model those complex behaviors in a set of threaded actors.

As mentioned previously, the defining characteristic of an agent is the "sense-think-act" cycle. At a given time step of a simulation, an agent takes input from its environment (sensing), processes it in some manner (thinking) and uses the result to enact some change in its environment, such as moving within it, or modifying its state (acting). Defining the middle part, *think*, is crucial to agent development, and constitutes what is typically referred to as the control architecture of an autonomous agent.

1.3 Agent Controllers

There are two main streams of thought in agent controller design: the "scruffy" and the "neat" [Minsky, 1991]. The "neat" philosophy grows out of classical Decision Theory and

Knowledge-based systems. It is based on Rational Agents, described first by Allen Newell, a Turing award recipient and one of the founding fathers of artificial intelligence research. In Newell's conception, a rational agent is equivalent to a knowledge-level system; it has some incomplete knowledge base about its domain, and a means of inferring information from the knowledge base about how to act in a given situation [Carley and Newell, 1994]. Rational agents are deliberative (Figure 1.1). When it is time to make a decision, they will consider all possible courses of action, compute the likely outcome given the current state of their environment, and take the action that is most likely to lead them to some desired state.

Some of the earliest robots used controllers that extended from this rational agent methodology. Shakey was a robot developed between the late 1960's and early 1970's by Stanford Research Institute (SRI), and is credited with being the first autonomous mobile robot that was able to analyze its commands and break them down into subtasks – a decidedly rationalist approach [Nilsson, 1984]. This also meant that Shakey's navigation algorithm had to evaluate and compare alternative routes to try to approximate the best available traversal path when moving through an obstacle course. This combination of prior knowledge and heuristic guessing was groundbreaking for its time, and is in fact the now-famous A* search algorithm. However, it also resulted in a robot that would move a little, stop and think a lot, then move a little again – an approach that is unsuitable for rapidly changing environments.

This approach has been refined by Michael Bratman into his Belief, Desires and Intents (BDI) model for agent design [Bratman et al., 1988]. In BDI, a "Belief" is information (possibly inaccurate) about the environment that is stored within an agent. "Desires" are goals that an agent tries to achieve, and which are selected for action according to its belief set. Once selected for execution a desire becomes an "Intent". The BDI model attempts to reconcile the need for long-range, deliberative planning with the real-world problem of finite

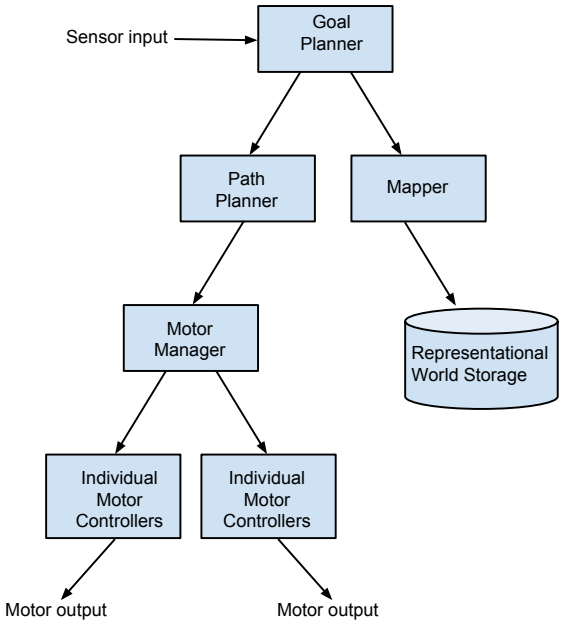

Figure 1.1: A hypothetical deliberative-style control architecture. Sensor input is passed to a long-term goal planner, which filters the information down to subordinate processes like path planning and mapping, as necessary. Unlike reactive controllers, there is typically a lot of computation between the sensor input and the motor output.

computational resources by limiting the agent's deliberation to the selection of completely or partially pre-computed plans.

The BDI approach has been adopted by some successful agent simulators such as the Procedural Reasoning System (PRS) developed at SRI during the 1980's [Cohen and Levesque, 1990] and Jack, a commercial MAS developed by researchers on the PRS [Howden et al., 2001].

The second stream of thought in agent design, the "scruffy", grew out of a radical rethinking of the nature of intelligence, which became popular with a rebel group of robotics researchers in the 1980's and 90's.

In the late 1960's an MIT researcher named Marvin Minsky was trying to make a robotic arm that could grasp children's building blocks and construct them to mimic an existing example structure [Minsky, 1988]. This was no easy task. In fact, while following the approach set down by Newell and the rationalists, Minsky quickly became overwhelmed by the seemingly endless nesting of subtasks within subtasks involved in something as seemingly simple as playing with children's blocks.

Along with Seymour Papert, then the director of the Artificial Intelligence Laboratory at MIT, Minsky began to develop a model of intelligence not as a single omnipotent process, but as a community of smaller heterogeneous processes that coordinate themselves cooperatively to produce a result. They named this approach The Society of Mind.

The success of this model in Minsky and Papert's experiments spurred a reaction to hierarchical planners. Now, researchers started using so-called "reactive" controllers, which emphasized a tight coupling between sensor inputs and actuator outputs, with minimal planning or deliberating in between (Figure 1.2). In the reactive view, the presence of a 'god-like' controller was both unnecessary and ineffective in generating robust behavior. This point of view is summarized in Rodney Brooks' 1987 memo "Planning is just a way of avoiding figuring out what to do next" [Brooks, 1987].

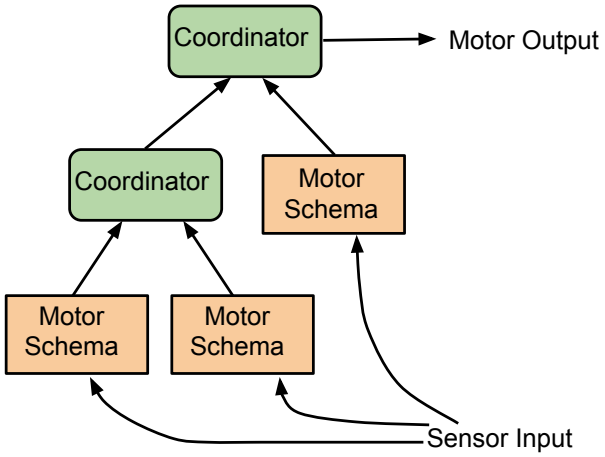

Figure 1.2: Reactive controllers emphasize tight coupling of sensor inputs and motor outputs, with minimal deliberation in between.

Brooks later developed the subsumption architecture, which expanded the reactive controller model by establishing a hierarchy of action impulses and allowing for some impulses to modify or completely override other impulses. Specifically, sensor inputs are consumed by many independent processes, each of which processes the perceived information according to its own rules. The results are then passed along to higher order processes, which subsume those micro decisions by either accepting or overriding them.

The subsumption architecture was implemented by Brooks in his robot Allen [Brooks, 1991], which used three levels of control impulses. The lowest level allowed it to avoid obstacles, the next level allowed it to wander aimlessly, and the third level allowed it to navigate toward a distant objective. When combined in the subsumption architecture, Allen was able to navigate through an obstacle course to a goal.

While reactive controllers succeeded in allowing robots to behave adaptively in dynamic environments, they lost something in the process. Whereas, the rational approach to intelligence emphasized explicit planning and world representation, the reactive approach consid-

ered these to be unnecessary for adaptive navigation. As a result, reactive controllers did not allow for any long-range planning or learning – a feature which greatly limited their utility. Ronald Arkin, attempting to bridge the two models into a unified approach that allowed for both dynamic adaptation and explicit long-range planning and goal-orientedness, developed the Theory of Societal Agents, also known as Behavior-based Robotics [Arkin, 1998].

Behavior-based architectures take the notion of bottom-up information flow from subsumptive controllers, combined with the representational world knowledge and long-range planning capabilities of hierarchical controllers (Figure 1.3) to create planning robot controllers that can adapt to uncertain and dynamic physical environments [Arkin and Balch, 1997].

The Societal Agents model uses *schema theory* to provide basic building blocks for complex agent controllers [MacKenzie et al., 1997]. Schema theory has roots as far back as Emmanuel Kant, but has been applied in the 20th century to neuroscience and the study of animal behavior. A schema defines the process by which a sensor input produces an impulse to action [Arbib, 1992]. It produces a "coarse-grained" model for behavior; it is not overly concerned with the details of its implementation (whether in code or in neurons), but rather provides a broad, symbolic description of its effect.

When a schema produces a result, or impulse to action, it does so in the form of a vector. A vector, as in physics, is simply a point in space with both direction and magnitude. To describe behavior, vectors can be used to indicate attraction to a goal (e.g., water, prey or home) or repulsion from obstacles or predators. Furthermore, vectors can be summed to produce result vectors. This leads to the use of vector fields for path navigation. For example, the combination of vectors that attract an agent to a goal, and repel it from an obstacle, create resultant vectors that lead the agent around the obstacle and to the goal. Such a use of vector fields as the basis for motor impulse decisions is well-established in neuroscience [Arkin, 1998] and ethology [Arbib, 2003].

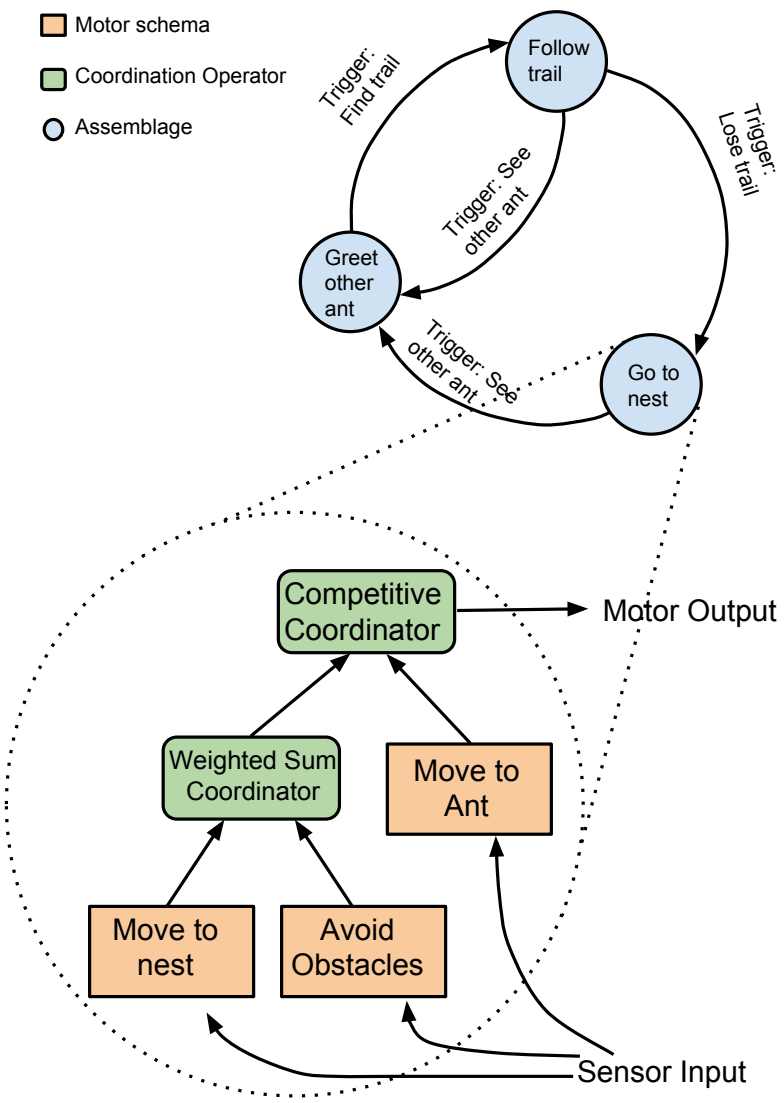

Figure 1.3: An example of a hypothetical hybrid controller. A temporal coordinator, or finite state machine, controls the behavior of the agent, while each state of the machine is a reactive controller, or behavioral assemblage.

The theory of societal agents considers three types of schemas: perceptual schemas, motor schemas and behavior schemas. Perceptual schemas provide sensor data about the world to the controller, and are embedded within motor schemas. Motor schemas describe basic stimulus and response processes; given information provided by its embedded perceptual schema, a motor schema produces a result vector. Motor schemas may be grouped together via coordination operators to form more complex behavior schemas, or *assemblages*.

Coordination operators take the result vectors of their subordinate motor schemas and produce a combined result vector. Theory of Societal Agents (TSA) describes three coordination operators, Continuous, Temporal and Competitive. Continuous coordination operators are the simplest – they produce a weighted vector sum of all of their subordinate schemas or assemblages. A temporal coordination operator is a finite state machine. Each coordinated assemblage is a state, and transitions between states are motivated by perceptual triggers. Competitive coordination operators function much the same as the subsumption controllers described previously. Assemblages are prioritized, with higher priority assemblages given the power to modify or override the result vectors of lower assemblages.

These assemblages are then selected by a planner of some kind. In MacKenzie, Arkin and Cameron's MissionLab simulator for goal-based robot teams [MacKenzie et al., 1997], planning is performed by a temporal coordination operator called the temporal sequencer The temporal sequencer chooses from a set of assemblages based on input from a perceptual trigger. This is essentially a finite state machine, where assemblages are states and perceptual triggers are edges.

1.4 Evolutionary Algorithms

We will now detour into a seemingly unrelated branch of artificial intelligence research – evolutionary algorithms. Evolutionary algorithms are an heuristic approach that applies

Darwin's Theory of Natural Selection to generally intractable optimization problems [Eiben and Smith, 2003]. The idea is to represent inputs to the problem as some kind of data structure (in the simplest genetic algorithms a list of values or even a bitstring is used). This is called the genome. At first the genomes of the initial population are randomly constructed. A simulation of the problem to be solved is run with the values from the genome as inputs – this could be as simple as plugging the values into a formula, or as complex as running an agent-based simulator. This is called the "fitness function," and after each run the genomes are ranked according to how well they solved the fitness function. Depending on the type of algorithm used, some may be randomly mutated, or changed arbitrarily, others may be combined together in a process called crossover to produce new offspring genomes. Some genomes, typically the worst performing ones, will be dropped from the next generation and others will be held over.

When successful, the evolutionary approach will, over the course of many generations, develop genomes the steadily improve their performance against the fitness function. While there is no guarantee in finding the optimal solution to a problem, the evolutionary approach has proven useful in engineering and manufacturing disciplines, especially when counterintuitive decisions are needed to solve design problems. For example, evolutionary algorithms have been used in the design of completely asymmetrical satellite booms that are 20,000% better at damping vibrations than other, more traditional designs [Keane, 1996].

One particular branch of evolutionary algorithms uses a tree data structure instead of a list or a bitstring. This is called genetic programming, because the original intent was to develop the parse trees for entire programs through the evolutionary process described above. In genetic programming, the mutation procedure typically consists of randomly modifying an arbitrary subtree of a given genome, while crossover implies swapping subtrees of two genomes.

Previous work in this vein has evolved programs using LISP, a natural choice due to the tree structure of pure s-expressions [Koza, 1989]. In this work, we employ genetic programming to evolve the behavior-based controllers of our agents by directly manipulating the hierarchy of motor schemas and coordination operators in the agent navigator. By using a machine-writable format for our controllers, we can generate and modify behaviors such as attraction to fellow prey agents or repulsion from predators and obstacles. By abstracting away the implementation details of the code, we can manipulate and evolve the controllers based on their semantic meaning, rather than on the syntactic details of their implementation.

1.5 Ethology

Ethology concerns the creation of quantitative models of individual animal behavior through observation of real animals in their natural environments [Gould, 1982]. As a principal tool of the ethologist, the ethogram is an encoding of the model as a catalogue of observed actions, accompanied by a frequency distribution for each observed action, or a state transition table that shows the probability of moving between any observed state and any other observed state [Eibl-Eibesfeldt, 1970].

Ethology as a field started in the 19th century, with the works of Darwin, whose discovery of natural selection prompted further inquiry into the nature of inborn, instinctive behaviors in animals. Karl von Frisch showed experimentally that bees use a sensitivity to ultraviolet light in their pollination routines, and further that they are particularly sensitive to carbon dioxide, relative humidity and the Earth's magnetic fields.

In the 1930's, the work of Konrad Lorenz and Niko Tinbergen is generally accepted as the beginning of ethology as a mature science. One of their most significant contributions was the recognition of "releaser" mechanisms, in which certain environmental stimuli would

automatically trigger a predetermined response in an animal, known as a fixed action pattern. For instance, the greylag goose has a distinctive behavior while incubating its eggs. If it sees that one of its eggs has rolled out of the nest, it will carefully roll the egg back up into the nest. Lorenz and Tinbergen showed that, far from being a carefully thought out solution to a problem, the behavior was entirely instinct driven. In fact the goose would roll anything remotely egg-shaped (e.g. batteries, lightbulbs and snail shells) into the nest and incubate it as though it were an egg.

Modern ethology concerns itself primarily with the understanding of how these innate release mechanisms, or IRMs, are genetically encoded, inherited and modified through the evolutionary process. It also attempts to understand how those primitive, inherited IRMs are shaped by environmental factors through imprinting, habituation, learning and teaching [Gould, 1982].

1.6 Collective animal behavior

Ethology emphasizes the study of individual animal behavior, but what about the behavior of animals as groups? The collective formation of schools of fish, flocks of birds and herds of cattle has fascinated and continues to inspire biologists and laymen alike with the aesthetic beauty of hundreds and even thousands of animals moving synchronously while avoiding obstacles, predators and each other (See Figures 1.4 and 1.5).

While many early ethologists and biologists worked on the presumption that collective behaviors emerged because of some benefit to the collective as a whole, D.W. Hamilton put forth the idea that such collective behaviors emerged from a competition amongst members of the group to avoid predation. He published this idea in his 1971 paper "Geometry of the Selfish Herd." [Hamilton, 1971] Here he postulated that positioning within the herd could be modeled with a Voronoi diagram, in which each member is enclosed within a convex polygon,

Figure 1.4: A school of fish in the Florida Keys National Marine Sanctuary. These fish are assuming a "funnel" shape. Photo by Dr. Jiangang Luo, courtesy of the National Oceanographic and Atmospheric Administration.

every point of which is closer to its center than it is to any other point on the polygon. Such a shape constitutes a "domain of danger" around the group member. Hamilton ran numerical simulations in one dimension to support his claim that selfish behavior could lead to aggregation, but left the behavior in two and three dimensions as a thought experiment.

Significantly, Hamilton modeled the collective behavior not as a whole, or even as a set of individual actions where the individual has some global knowledge of the current shape of the flock or even the existence of the flock, but rather as a simple behavioral process relative to each individual's nearby neighbors and the presence of a predator as an outside stimulus.

Ten years later, Ichiro Aoki expanded on this idea by conducting two-dimensional computer simulations of flocking behavior using a discrete event simulator [Aoki, 1982]. Similarly to Hamilton, Aoki modeled the behavior of each individual as a simple rule set in relation to its neighbors, but he also expanded on it by representing the movement of each individual as a stochastic vector, and by postulating three distinct behavioral rules as sufficient for flock formation: attraction to other individuals, collision avoidance and sympathetic orientation, in which individuals mimic the orientation of their nearby flock-mates. Using these rules, Aoki was able to show the creation of flock behavior without programming any explicit

Figure 1.5: Typical schooling behavior in small fish. These are schooling over the Geyer Bank in the Gulf of Mexico. Photo courtesy of the National Oceanographic and Atmospheric Administration.

knowledge of the flock into his agents or having any hierarchical leadership structure within the flock.

Aoki's approach was later expanded into three dimensions by Craig Reynolds with his well-known Boids model [Reynolds, 1987]. Using early three-dimensional computer animation, Reynolds used the same three basic behaviors to create flocks of bird-like creatures, or "boids" in the New York vernacular, that collectively avoid obstacles while maintaining group cohesion.

An important part of the model shared by Aoki and Reynolds is the use of zones in a pie-slice shape around the agents. These concentric zones constitute the areas over which each behavior is active, and each is prioritized. The inner circle is the highest priority – if another individual is seen within the inner zone, the agent will try to avoid it to prevent collisions. The next concentric circle represents the zone of orientation, and is also the next higher priority. When another similar agent is seen within this zone, the agent that sees it will try to orient itself in a direction similar to that agent. Finally, the outer zone is the

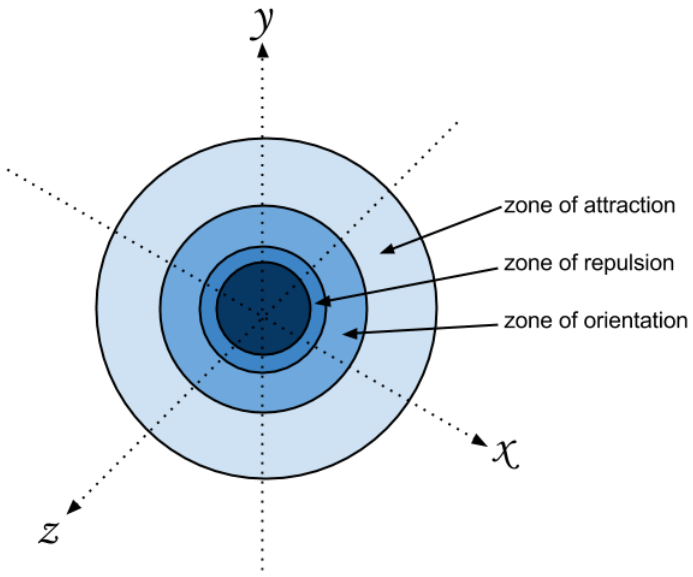

Figure 1.6: A depiction of the zones of attraction, orientation and repulsion, described by Iain Couzin in his work on collective flocking.

zone of attraction, in which an agent will be attracted towards any other similar agents it perceives in that zone (Figure 1.6).

In his 2002 work on flocking behaviors, Iain Couzin started with this model of individual behavior, but treated the actual width of each zone as an adjustable parameter in a three-dimensional simulation. He found that as he gradually adjusted the widths of the concentric zones, there were sudden changes in the collective behavior of the individuals [Couzin et al., 2002]. With a very small zone of orientation, the agents exhibited a swarming behavior in which they were actively drawn toward each other and avoided collisions, but there was no coordinated movement as in a flock. With a relatively small zone of orientation, the agents exhibited the so-called torus formation, in which they collectively swirl around in a funnel shape. This formation is sometimes observed in open-water fish populations. As the zone of orientation grows larger, the agents begin to take on weakly directed movement,

which Couzin refers to as a Dynamically Parallel Group, which moves about freely with significantly aligned orientation. Couzin observed this type of formation to occur with large zones of orientation and medium to large zones of attraction. Finally, with a very large zone of orientation, the agents showed highly directed, almost rigidly defined directionality, which Couzin termed the Highly Parallel Group.

There have been a few attempts to use genetic algorithms to generate these models. Wood and Ackland used predation as a stimulating factor in evolving prey agents in the style of Aoki and Couzin [Wood and Ackland, 2007]. They evolved their model based on a gene with five traits: 1) the angle of perception and 2) range of movement, which were made to be complementary to avoid an individual that simply maximizes both, 3) the width of the zone of orientation, 4) the amount of noise, and 5) the weight placed on the detection of external stimuli. Furthermore, the individuals were evolved as part of a heterogeneous population, with each individual having different values for the parameters than its fellow individuals. There were some interesting choices in their setup for the evolution. For instance, crossover selection was driven not by performance according to a fitness function but rather by ability of a single individual to survive a simulation run, as well as its ability to forage for food. Predators in their model have a finite lifespan, and are removed from the simulation as soon as either their lifespan has ended or they have caught a single prey. Using this system, they were able to evolve flocking agents that exhibited the torus-like behavior identified by Couzin, as well as the dynamically parallel grouping, against which predators were only successful 60-70% of the time. From this, they concluded that flocking behavior likely evolved as a defense against predation.

While Wood and Ackland extended Couzin's work on the effects of sensor thresholds in flocking agents, Reluga and Viscido have conducted similar work in which they attempt to validate Hamilton's selfish herd theory through genetic algorithmic techniques [Reluga and Viscido, 2005]. In their simulated environment, prey agents are distributed uniform

randomly in a two-dimensional environment. At each run of the simulation, each agent chooses a direction vector that is weighted and influenced by its neighbors in the simulation and advances a fixed distance in the direction of the vector. Next, a predator appears at a random place, similar to an ambush attack as described by Hamilton, and the prey nearest to the predator is eaten and removed from the simulation. After each simulation run, a survivor is chosen at random to generate a replacement via mutation for the eaten prey. Only a single gene was evolved, representing an influence factor of nearby neighbors on the acting agent's direction vector.

The implication is that individuals that are drawn primarily toward their closest neighbors end up on the periphery of the cluster, thereby having a larger domain of danger and being more susceptible to predation. On the other hand, individuals that are drawn more strongly to distant neighbors will "leapfrog" over their nearby neighbors, thus selfishly putting that neighbor on the periphery of the cluster and reducing the agent's own domain of danger. As expected the gene responsible for the more selfish strategy gained dominance over the course of Reluga and Viscido's experiments.

It is notable that these evolution experiments sought to evolve genes that were tied directly to parameters of the individuals' sensory or physical capabilities. In the case of Wood and Ackland, the width of the domain of orientation as well as range of motion, active sensor region and noise ratio were evolved, while they assumed as constant the necessity of the three behavioral drives: attraction to neighbors, collision avoidance and orientation with neighbors. Our work takes the converse approach. We hold as constant the types and attributes of sensors and physical capabilities, these being selected based on values that have been shown on our platform to allow for capable flock maneuvering, and attempt to evolve a set of behaviors as a controller that enables flocking to avoid predation. The description of our model, test platform and experimental setup and results follow in the proceeding sections.

1.7 Summary

In this chapter we discussed the various approaches to modeling and simulation, we looked at the historical roots of reactive and hybrid robot controllers, and we presented a brief overview on the field of Ethology and its relation to collective animal behavior research. In this section, we will briefly summarize the relevance of each of these subjects in relation to our framework, Pickle.

- **Agent-based:** Pickle is an agent-based modeling and simulation framework, which means that its fundamental units are autonomous processes that operate on a "sense-think-act" paradigm.

- **Behavior-based:** Pickle uses behavior-based robot controllers to drive the "think" functionality of its agents. This means that Pickle agents have the flexibility and adaptability of reactive agents, but are also capable of some higher level planning in the form of activating different behavioral states.

- **Evolutionary:** Pickle uses genetic programming to evolve its behavior-based controllers.

- **Collective animal behavior:** Pickle's novel approach to agent controller description allows for new approaches to understanding and reproducing collective animal behavior, by controlling and evolving the fundamental behavioral mechanisms of its agents, rather than only the physical parameters of their sensors and actuators.

Chapter 2

Pickle

In this chapter we will describe the overall system design of the Pickle agent-based platform. Later, we will discuss specific details about its implementation.

2.1 Design of the Simulation Framework

When designing Pickle, we had three objectives for our modeling framework: first, it must have explicit support for sense-think-act agents. This paradigm is fundamental to collaborative biological and simulation & modeling research. Our framework must enable researchers to create and modify both sensors and actuators easily. It should also be easy to connect them through a controller, while maintaining a consistent interface between sensors, actuators and controller.

Second, the same separation of concerns between sensors, actuators and controller should be a feature of the entire framework. The simulation kernel itself should have a consistent interface to the application layer, and the configuration layer, which comprises the XML descriptions, should be loosely coupled with the application layer (Figure 2.1). This will enable

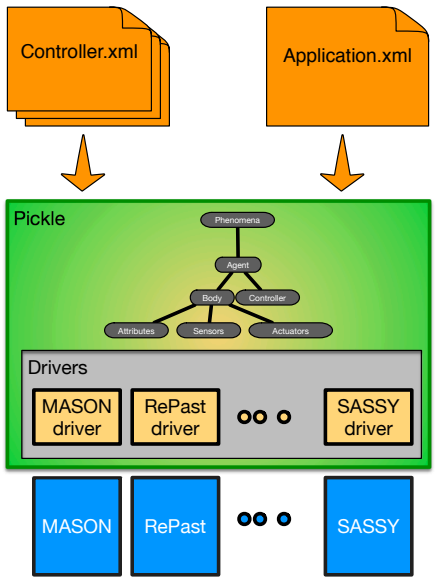

Figure 2.1: A high-level view of Pickle's architecture. Pickle is essentially an application layer that reads input from a Configuration Layer consisting of XML descriptions for simulation parameters, Agents and Controllers. It uses a set of drivers as a translation interface to run on multiple simulation kernels.

future support for multiple simulation kernels, visualization methods and semi-structured data representations.

Third, everything must be serializable. Full serializability enables both the collaboration of non-programmers, and the automatic generation of agents. To enable the collaboration of non-programmers, there must be a way for agent and controller definitions to be generated, stored and shared independently of the executable source code that is used to actually run them. This in turn enables the automatic generation and modification of agents for our experiments in genetic programming.

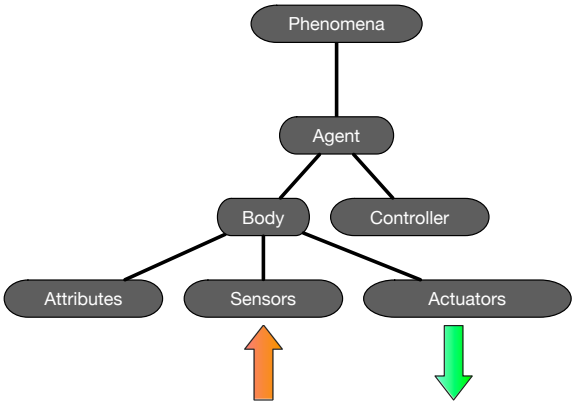

Figure 2.2: A description of the Pickle taxonomy. Agents are distinguished from other phenomena by the presence of sensors, which consume information about the simulated environment, actuators that effect changes to the environment, and a controller that binds sensors with actuators.

2.1.1 Taxonomy of a Pickle Simulation

The notion of a physical world in which agents can perceive objects, process those perceptions and then take actions to effect change in their physical environment is central to Pickle and to agent-based modeling and simulation as a whole. In a Pickle simulation, anything that is perceivable through a sensor is called a "Phenomenon". Phenomena also have bodies with sets of free-form attributes such as size and color.

There are two basic types of Phenomena: Obstacles, which are inanimate objects and Agents, which are animated through a sense-think-act cycle. The things that set an Agent apart from any other Phenomenon in a Pickle simulation are: Sensors, which collect data about neighboring Phenomena, Actuators, which attempt to modify the simulated world in some way, and a Controller, which connects the Sensors to the Actuators. A diagram of the Pickle Taxonomy is depicted in Fig. 2.2.

2.1.2 Sensors

Every agent has a set of Sensors that collect information about nearby Phenomena in the simulated world and pass that information on to the Controller. For example, a simple sensor might return all Phenomena within a certain distance from the sensing agent's current location. However, there are several ways that we can refine that information. First, it is probably more useful for a sensor to return information about a particular type of Phenomenon. For instance, a minnow might have a sensor dedicated specifically to detecting predators, like sharks, another sensor dedicated to other minnows, and still another sensor dedicated to obstacles like coral reefs. In Pickle, this is accomplished by defining a list of filters for each sensor.

Another way in which we can filter the information returned by a sensor is by specifying an active region for the sensor. Every sensor has an active region that is essentially a pie slice with respect to the sensing agent (Figure 2.3). Phenomena that fall within the pie slice are perceived by the sensor, while those that fall outside the pie slice are ignored.

Whenever a sensor is activated, resulting data is passed to the behavior controller, which in turn activates the agent's actuators.

2.1.3 Actuators

For an agent to interact with its environment, it must have actuators. These allow the agent to move through the environment, to grab or eat other agents, and to modify the state of their environment in any meaningful way. Actuators receive their instructions from the agent's controller as a list of Action objects, each of which is marked as belonging to one of the agent's actuators.

For example, a Navigation Action marked with the name of the agent's navigation actuator will contain a vector that is given to the actuator. The Navigation actuator in turn makes

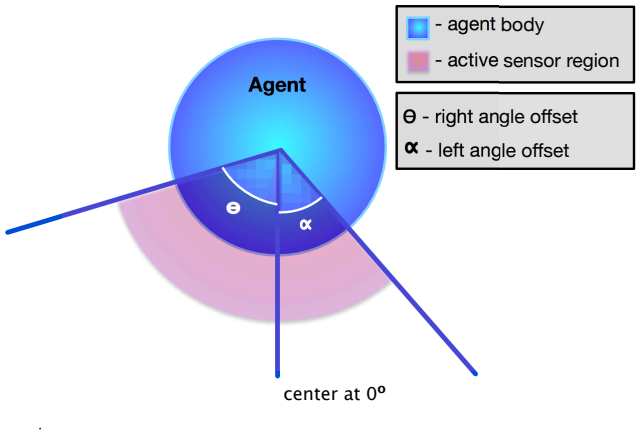

Figure 2.3: A view of how pie slice regions for sensors work. Each Sensor operates within a single pie-slice region, and Agents may have multiple sensors.

a request to the Simulation Driver to move its owning agent to a point in space specified by the vector.

2.1.4 Controller

A Pickle Controller is a memoryless finite state machine that binds sensors to actuators. It does this by consuming sensor data (a list of items called Perceptual Schemas) provided by the agent's sensors, and producing a list of actions which are processed by the agent's actuators. A Controller is basically a set of schemas, each of which subscribes to exactly one Perceptual Schema. The use of the Schema nomenclature comes from our previous work in BioSim [Medina et al., 2014], and more broadly from the work of Arkin and Arbib [Arbib, 2003].

By "memoryless" we mean that at any given step (event or time based), the result of the controller computation depends only on its sensor inputs for this step and the current state,

but not on any previous steps or state. This creates some limitations for our agents. For instance if a predator is closing in on a prey, and the prey moves outside of the predator's sensor range, the predator has effectively forgotten about the prey and will stop tracking it until it comes back into its sensor range. This is in keeping with the underlying reactive nature of our controllers. Long-term planning in the style of Arkin's AuRA controllers is provided through the State Change Schemas described in Section 2.1.7.

When a Perceptual Schema arrives at the Controller, the Controller Schemas are polled to see if any of them subscribe to it. If so, that Controller Schema is "fired" by which we mean that it is given the data attached to the Perceptual Schema as input, and called upon to produce some result.

For example, an agent may be attracted to a resource such as a food pellet. Such an agent will have a sensor which detects nearby food pellets. When that sensor detects a food pellet, it sends a reference to the nearest pellet along with the pellet's point in space relative to the agent as sensor data to the controller as a Perceptual Schema called "nearestPellet". The controller would then have a schema that subscribes to the "nearestPellet" Perceptual Schema, that when fired, would produce a vector pointing toward the food pellet. That vector is then passed along to the agent's Navigation Actuator, which asks the Simulation Driver to move the agent accordingly.

2.1.5 Motor Schemas

In fact, this describes a Motor Schema, which is one of three kinds of Controller Schemas, each of which is distinguished by the kind of result it is expected to produce. Motor Schemas produce a vector, which signals a desire to move the Agent in some corresponding direction, with a given magnitude. Motor schemas are defined by the Perceptual Schema that they react to, the type of reaction, either 'attraction' or 'repulsion', the response curve of the

reaction, which may be 'linear', 'quadratic' or 'exponential', and a weight value or priority value, to be used by its Coordination Operator.

2.1.6 Action Schemas

An Action Schema is tied directly to one of the Agent's non-navigation actuators. When an Action Schema is fired, it sends the corresponding sensor data to the Actuator. For example, a Shark Agent might have a sensor called bumpedMinnow. When that sensor is activated, a Perceptual Schema called "bumpedMinnow" is sent to the Shark's controller. If the Shark has an Action Schema called "chompMinnow" that is tied to a Chomp Actuator, the sensor data (i.e., the x and y coordinate, or "Point" in shark-space and the reference to the minnow itself) are sent to the shark's Chomp actuator, which in turn can tell the Simulation Driver "I want to chomp this minnow".

It is important to note here the Simulation Driver's role as an arbiter of the simulated world. An agent may request to move to a point in space, and the Simulation Driver may move it there, or deliver the bad news that there is an obstacle or another agent in the way, and leave the agent where it is. By the same token, an agent may signal a desire to eat another agent, and the Simulation Driver may either act accordingly or, with some random probability allow the prey to escape.

2.1.7 State Change Schemas

Motor Schemas and Action Schemas are grouped together in the Controller as a unified state called an Agent Schema. An Agent Controller may have multiple Agent Schemas, along with a way to transition in and out of these states. This brings us to Pickle's third kind of Controller Schema, the State Change Schema. Each State Change Schema belongs to an Agent Schema and is also bound to some other Agent Schema. When fired, a State Change

Schema changes the currently active state from the Agent Schema to which it belongs, to the Agent Schema to which it is bound.

A controller is a finite state machine, consisting of discrete states and transitions between those states. At each step, sensor data is ingested by the controller. First, State Change Schemas are checked against the sensor data, and if one is fired, then the active Agent Schema changes immediately. Next, the Motor Schemas are checked and produce a result vector. Finally, the Action Schemas are checked. Once the Action Schemas have returned their results, they are put into a list along with the navigation vector and passed to the actuators for action.

2.1.8 The Navigator

In the controller, Motor Schemas are grouped hierarchically into an abstract syntax tree, where the Motor Schemas are the leaf nodes and Coordination operators are the interior nodes.

Every node of the Navigator produces a vector. Motor Schemas (the leaf nodes) produce vectors when fired as detailed above. If a Motor Schema is not fired, it produces a zero vector. Weighted sum operators evaluate all of their children and scale their output according to a weight value assigned to each child, and return the vector sum of all of them. Priority, or subsumption operators, evaluate their children and choose exactly one result to return. The choice is made by assigning a priority to each child, so that the child that has both fired and has the highest priority is selected as the result vector.

For example, imagine a scenario with a shark that currently sees two things in its environment: a minnow, to which it is attracted, and an obstacle, from which it is repulsed. With a summation coordination operator, the attractive and repulsive vectors are summed into a result vector which, over successive time steps will guide the shark around the obstacle and toward the minnow, as demonstrated by Arkin. Alternatively, the subsumption operator

will completely suppress the output of one schema in the presence of another schema. So in the above example, the repulsive vector would be completely disregarded in favor of the attractive vector toward the minnow. The end result could be that the shark takes a more direct route toward its prey, or that it stumbles blindly into an obstacle.

A more practical use of the subsumption operator can be given in the implementation of Boid mechanics. In Boid mechanics, agents attempt to maintain an ideal distance between each other in a flock. This comes from the artificial life research of Craig Reynolds [Reynolds, 1987]. Boid agents have an inner zone and an outer zone and neighboring agents will try to stay within the outer zone without entering the inner zone and risking a collision. This can be implemented using a subsumption operator as follows: agents have both an inner zone sensor and an outer zone sensor. While the outer zone sensor fires, the agents are attracted to their neighbor, but as soon as the inner zone sensor fires, it suppresses the attraction to the neighbor and returns only a repulsion from the neighbor (Figure 1.6).

2.1.9 The Simulation Driver

With so many autonomous processes roaming throughout the simulated world, the need for an arbitrator, or referee becomes apparent. This is the job of the translation layer, or Simulation Driver. When agents use their actuators to effect some change in the simulated world, they send the request to a simulation driver, which collects actuator requests from all the agents in a particular timestep, and may or may not implement those requests in the simulated world. The simulation driver is in charge of arbitrating collision detection between both agents and inanimates, maintaining the physics of motion for the agents, deciding who gets eaten and who escapes. The use of a single layer for this task supports our notion of loose coupling between the simulation kernel and the application layer (Figure 2.4).

```
package edu.uga.pickle.drivers;
import edu.uga.pickle.body.Point
import edu.uga.pickle.application.{Phenomenon, Agent}

abstract class SimulationDriver
{
  val width: Int
  val height: Int
  def put(p: Phenomenon, l: Point)
  def markReady(p: Agent)
  def randomDouble: Double
  def getRough(caller: Agent): List[Phenomenon]
  def get(filters: List[(Phenomenon) => Boolean], range: Int, center: Int,
      offsetL: Int, offsetR: Int )(caller: Agent, rough: List[Phenomenon]) :
      List[(Point,Phenomenon)]
  def getRandom(freq: Double, active: Double, center: Double)(caller: Agent,
      rough: List[Phenomenon]) : List[(Point,Phenomenon)]
  def getWithFilters(filters: List[(Phenomenon) => Boolean])(caller: Agent, bag:
      List[Object]) : List[Phenomenon]
  def getNearest(filters: List[(Phenomenon) => Boolean], range: Int, center:
      Int, offsetL: Int, offsetR: Int )(caller: Agent, rough: List[Phenomenon]) :
      List[(Point,Phenomenon)]
  def runWithGUI(bgColor: String, showIDs: Boolean)
  def removePhenomenon(p: Phenomenon)
  def bumpSensor(target: String)(caller: Agent, empty: List[Phenomenon]):
      List[(Point,Phenomenon)]

}
```

Figure 2.4: The interface for the simulation driver in our prototype implementation. A suitable driver implementation would need to fulfill these functions using its simulation kernel.

2.2 Comparison with Previous Work

2.2.1 BioSim

In previous work with the BioSim platform, we detailed methods for dynamically generating controllers from semistructured data descriptions by generating and compiling Java source code on the fly [Medina et al., 2014]. We accomplished this through the use of XSLT transformations to produce the text of the source code, and then invoking the Java ClassLoader to compile the code and inject it into an already running environment. This entire process was controlled by a dynamically generated ANT build script, and we used the Java Architecture for XML Binding (JAXB) to generate a document object model for our controllers, which allowed us to randomly generate and arbitrarily modify our XML controller descriptions.

The Java source code that was generated was specifically targeted for the BioSim modeling and simulation framework, which uses the MASON simulation kernel and the Clay robot control architecture library [Balch, 1998].

While these efforts were successful and promising, the present work improves upon them in several important ways. First our approach to BioSim became complicated because it was necessary to overcome inherent limitations in the simulation framework. Second, the BioSim framework required pulling together different technologies in sometimes counterintuitive ways. For instance, in BioSim there is no explicit support for a sense-think-act cycle with a separation of concerns between each step in the cycle.

By designing Pickle from the ground up with separation of concerns and explicit support for sense-think-act agents in mind, we are now able to specify not just controllers, but entire agents, including body attributes, sensors and actuators using XML. Rather than generating and compiling code at runtime, we can simply use the XML specification to populate instances of Pickle classes and carry out the internal wiring of those classes.

2.2.2 SASSY

Our use of a middle translation layer to maintain separation between the application layer and the simulation kernel has been partly inspired by previous work on the SASSY Agent-based Modeling and Simulation framework [Hybinette et al., 2006]. SASSY uses a middle-layer API to wed an ABMS framework to a high-performance Parallel Discrete Event Simulation (PDES) kernel. However, SASSY was never intended to be a multi-kernel architecture; it essentially provides its own application layer and its own PDES kernel. Furthermore SASSY has no support for serializable agents; a SASSY user would need to be a capable programmer to create an application. Pickle has been designed with the intent to function on multiple kernels, given appropriately written Simulation Drivers for each kernel. In future development, we intend to incorporate more high-performance elements into Pickle, such as a middle layer that can use GPU acceleration for massively parallel agent computations.

2.2.3 MissionLab

Our controller model, and the field of behavior-based robotics in general, owes a particular debt to the work of Ronald Arkin, who formalized much of its groundwork. Along with Douglas MacKenzie and Jonathan Cameron, Arkin produced MissionLab, a robotics simulator that uses behavior-based robot controllers [MacKenzie et al., 1997]. MissionLab features a graphical user interface to specify the robot controllers, which produces code in the Configuration Description Language (CDL), a domain-specific language (DSL) developed specially for MissionLab. CDL describes high-level features of robot controllers, including behavior primitives and how those primitives combine to form assemblages, but it does not define the implementation of behavior primitives themselves. Rather it presumes that the CDL primitives will be bound to some existing library of primitives designed for a particular physical platform. This makes sense for MissionLab, since the intent was to compile a robot

controller, test it in a simulated environment, then take that same controller and put it onto an actual robot.

Pickle extends on these ideas in three ways. First, we abandon the use of special purpose domain-specific languages in favor of semi-structured data representations, or XML in our case. This allows us to maintain a language-neutral environment, and also makes our controllers completely machine generatable and modifiable, a feature that is not easily supported with a domain specific language.

Second, Pickle allows users to create and modify an agent's sensors and actuators as part of the XML description. In MissionLab, sensor and actuator hardware are simulated through a server application, which supports very specific models of hardware sensors and actuators which may be found on the target hardware robotics platforms. By removing the strict dependency on hardware availability and describing sensors and actuators in terms of other Phenomena (anything that is perceivable through a Sensor) in the simulated world, Pickle offers a set of capabilities better suited towards an ABMS for animal behavior research.

Finally, MissionLab is based around the idea of specifying teams of robots that work together toward a common goal. However, for animal behavior research, it is just as necessary to specify agents that will be working at odds, as in a predator and prey scenario. As such, while MissionLab offers no support for complex behaviors such as killing, consuming or otherwise removing agents from the world, Pickle supports these actions directly through its actuators and Simulation Driver model.

2.3 Implementation of the Simulation Framework

2.3.1 Scala and XML

We have written our prototype implementation of Pickle in the Scala programming language, a hybrid object-oriented and functional language that compiles to produce bytecode for the

Java Virtual Machine. Scala enables us to define all of our agents' behaviors at runtime while keeping a relatively small and manageable code base. We exploit its functional programming features to both enable a thin programmer user interface and to dynamically generate anonymous functions. While the same end results are possible in a pure Java implementation, (i.e., anything in Scala has an equivalent implementation in Java), such an implementation would require thick, intrusive interfaces, hand written code and result in a code base that is more bloated and less maintainable and extendable than our Scala implementation.

Furthermore, our prototype implementation uses XML (Extensible Markup Language) as the semi-structured data format to serialize simulations, agents and controllers. While it is convenient that the Scala programming language features native support for creating and parsing XML as literal values, there is no reason why a similar implementation could not support JSON or any other semi-structured language as well, and this is an extension that is marked for future work.

2.3.2 Sensor implementation

In practical terms, a sensor is a curried lambda function, generated at runtime that queries a data structure maintained by the simulation kernel and accessed through the Simulation Driver. When the sensor is initially created, it is given a list of filter functions and a numeric range with offset angles to define the range of the sensor, or its effective "pie slice." When it is called, the function is provided with the current position of the calling agent.

Each filter is an anonymous Boolean function. For instance a filter that returns all minnows within the given range would be type = "Minnow", which performs a string comparison on the perceived Phenomenon's type value. Similar comparisons can be made on any of a Phenomenon's attributes. So for example, if an agent needed a sensor specialized for all green food pellets versus all red food pellets, or all minnows of size greater than 5, this

would be specified in the XML as a Sensor with two filters: one that matches on the type ("minnow" or "food pellet") and one that matches on the attribute ("color = green").

The pie slice region can be defined in the XML as a center angle relative to the agent's straight-ahead or 0° heading. It has a left-offset to define the inner-angle of the left edge of the pie slice with the center and a right-offset for the right edge. Finally, it has a range parameter to specify the radius from the agent's center. Pie slice regions do not need to be symmetrical.

In future work, we plan to provide support for a sensor decay function, to allow users to specify a curve to describe a decrease in sensor reliability as a factor of distance from the sensing agent.

2.3.3 Simulation Kernel

We have implemented our prototype driver to support the MASON simulation kernel from George Mason University [Luke et al., 2005]. MASON operates on a single thread that uses a time-stepped event queue to poll the agents in the simulation sequentially. However, we should reiterate that the design of Pickle is not restricted to being either time-stepped or single threaded; it allows for both Discrete Event Simulation, or a multithreaded Process-based Simulation. Pickle simply inherits the simulation paradigm of the implementation of the Simulation Driver.

When polled and given sensor data, Pickle Agents use their Actuators to send requests to the Simulation Driver. This could be a single-thread that manages the underlying event queue and environment data structures (as in our prototype implementation), or it could be a thread manager that dispatches requests to a subordinate process running in parallel. In a Pickle simulation, the application remains insulated from the implementation details of the underlying kernel.

Chapter 3

The Genetic Programming Framework and Experimental Platform

To demonstrate the usefulness of our approach to simulation and agent controller specification, we have implemented a genetic programming framework for Pickle and used it to conduct experiments in evolving XML-serialized controllers. Through these experiments, we have explored the following questions: does flocking behavior evolve as a way for prey agents to avoid predation? Are there other successful strategies besides flocking to avoid predation? What fundamental behaviors are necessary to induce flocking as a way to escape predation? If there are valid solutions besides flocking, what behavior mechanisms do they use? To help us answer these questions, our genetic programming framework generates, scores and modifies Pickle controllers by directly manipulating their XML representation. The details of our experimental design are given in Section 3.1, and the results of our experiments are given in Chapter 4.

3.1 Design

Our genetic programming framework is designed to be easily controllable through an XML configuration file that specifies parameters of the evolution such as the population size, the number of generations to evolve, which application to run as the fitness function for the evolution and for which agent of the application we should be generating controllers.

Pickle applications are normally launched through a shell script; launching pickle with the "evolve" parameter causes the application to parse its evolution parameters and begin the genetic programming process. Each individual of each generation is recorded as an XML file that includes its controller, ranked amongst the other individuals of its generation, a unique, randomly generated identifier and a "family history" of parents if the individual is the product of crossover or mutation. These XML documents may later be searched through and individually replayed to examine the behavior of any particular individual from the evolution.

The initial population for an evolution consists of randomly constructed controllers. These are built by parsing the application file to create a list of available sensors and actuators for the agent whose controller is being generated. First, a Navigator is generated by recursively generating coordination operators and populating them with motor schemas. When creating a coordination operator, the type ("sum" or "priority") is selected with a uniform random distribution. When creating a motor schema, the type ("attraction," "repulsion," or "mimic") and the response curve ("linear," "exponential" or "logarithmic") are chosen uniform randomly. Next the perceptual schema for the motor schema is selected randomly from the list of available sensors. Finally, weights or priorities are assigned to each coordinator or motor schema based on the type of its parent node in the tree hierarchy. The numerical value for each weight or priority is generated randomly within a predefined window of values. This allows for the possibility of duplicate weight or priority values, but poses no

difficulty for weights as it simply indicates that the vectors produced by the motor schemas have the same weight when summed together during controller evaluation. For priorities however, this introduces a conflict. When the controller is evaluated, the conflict is resolved by choosing the first of two motor schemas that have the same priority for inclusion in the result vector.

This randomly constructive process is carried out recursively. At each level of the recursion, a random decision is made to either stop the recursion or continue. To avoid arbitrarily deep controller trees, a maximum depth parameter is specified in the evolution parameters XML file. In the same manner, the maximum number of motor schemas and coordination operators that may be generated as the children of a parent coordination operator is configurable via the XML file.

Crossover between two controllers is implemented by first selecting a receiver and a donor controller. Next, an interior node (excluding the root) is selected at random from both the receiver and donor. Finally, the donor's selected node and it's entire subtree are used to replace the selected node in the receiver. The receiver controller is then returned as the "offspring" of the two controller individuals.

Mutation of a single controller is implemented by selecting an interior node randomly from the individual, generating a new random subtree and replacing the selected node with the newly generated subtree.

The evolution continues until either the process is killed by the user, or the maximum number of generations specified in the configuration file is reached.

3.2 Implementation of the Experimental Platform

Our simulation consisted of a predator and prey scenario with 60 prey in a rectangular shaped simulation environment, and three predators which attempted to catch and eat the

Table 3.1: Physical attributes and sensors of the predators ("Shark"s) and prey ("Minnow"s).

Physical parameters		
	Shark	Minnow
Size	20px	10px
Speed	8px	10px
Turning radius	40°	30°
Shark Sensors		
Name	Range	Active Zone (on each side)
getNearestMinnow	550px	180°
getNearestShark	50px	180°
getNearestEdge	20px	180°
getRandom		70°
Minnow Sensors		
Name	Range	Active Zone
getNearestMinnowOuter	100px	100°
getNearestMinnowInner	50px	180°
getNearestMinnowCollide	20px	180°
getNearestShark	100px	180°
getNearestEdge	100px	120°
getRandom		70°

prey. We chose an aquatic environment, and so we used the LazyNavigation actuators for all agents with a world friction level of 0.7. The size of the simulated pool was 1280 by 720 pixels. We named our predator and prey agents Sharks and Minnows respectively, although this should not be construed as a direct reference to the actual biological creatures. Rather, these were convenient labels for abstract predator and prey agents.

The "Sharks" were slightly larger in size and slightly slower than the "Minnows", but with a slightly wider turning radius. Their sensors were designed for finding prey within the pool. All sensors were active for the full 360° around the predator, with the sensor range of the prey sensor set at a very large 550 pixels, the sensor for other predators set at a 50 pixel

```
<controller>
   <AgentSchema name="hunt" initial="true">
      <Navigation actuator="LazyNavigation">
            <CoordinationOperator type="priority" weight="1">
               <MotorSchema type="repulsion" curve="linear" priority="4"
                  PerceptualSchema="getNearestEdge" />
               <MotorSchema type="repulsion" curve="linear" priority="3"
                  PerceptualSchema="getNearestShark" />
               <MotorSchema type="attraction" curve="linear" priority="2"
                  PerceptualSchema="getNearestMinnow" />
               <MotorSchema type="attraction" curve="linear" priority="1"
                  PerceptualSchema="getRandomPoint" />
            </CoordinationOperator>
      </Navigation>

      <Action actuator="eatMinnow">
         <ActionSchema PerceptualSchema="bumpedMinnow" />
      </Action>
   </AgentSchema>
</controller>
```

Figure 3.1: The predator controller used in our genetic programming experiments. This controller consists of a priority or subsumption coordination operator that chooses between repulsion from pool edges, repulsion from other predators, attraction to prey and random motion.

range, and the sensor for the pool edges set at a conservative 20 pixel range. The predators were equipped with a "Chomp" actuator which fires the "eatMinnow" action when triggered.

The predator controller (Figure 3.1) was a simple "priority" type coordination operator. The highest priority went to avoiding the pool edges so as not to get stuck. The next highest priority behavior was to avoid other nearby predators. The next highest was to seek out prey, and finally if none of these perceptual schemas are fired, the predator moved randomly.

The prey were smaller than the predators, measuring 10 pixels in diameter, with a faster top speed of 10 pixels per turn and a slightly less navigable 30° turning radius. Their sensors were designed for spotting predators. The nearest predator sensor was active for 360° around the prey agent with a range of 100 pixels. The prey agent also had a set of sensors for other

prey that are analogous to the zones proposed by Aoki and Couzin. The outermost zone, named "getNearestMinnowOuter" had a range of 100 pixels and an active zone of 100° off of center for each side. The next, "getNearestMinnow" inner had a range of 50 pixels, with an active zone of 360° around the Minnow, and the innermost, "getNearestMinnowCollide," had a range of just 20 pixels and an active zone of 360° around the Minnow. Finally, the Minnow had a pool edge sensor with a range of 100 pixels and an active zone of 120° from center on either side of the agent.

To validate that flocking behavior was achievable with this setup for our Minnows, we "hand-coded" a Pickle controller description along the lines of those used by Aoki, Reynolds and Couzin (Figure 3.2). This controller had a priority coordination operator that selected between (in descending priority) avoiding the pool edges, avoiding Sharks, avoiding collisions with other Minnows, attraction to nearby Minnows and finally, random movement when no other sensor was fired. There are two refinements worth mentioning: the impulse to avoid nearby Minnows in the inner zone was tempered by summing the repulsion vector with a "mimic" vector that duplicates the current direction of the sensed Minnow. Likewise the attraction to Minnows in the outer zone was summed with a mimic vector so that the end result is for the acting Minnow to move in a heading that would put it just in front of the Minnow being sensed, rather than attempting to swim directly into it. This was in keeping with the selfish herd theory, in which prey animals attempt to move ahead of other animals. This controller exhibited dynamic flocking, with acting in coordinated maneuvers to avoid each other, pool edges and predators.

Each simulation ran for 1,000 time steps. During this time, the Sharks and Minnows were set free in the simulated pool environment. As Minnows were eaten by the Sharks, they were removed from the simulation, and at the end of the run, the total number of surviving agents was recorded. This included the three Sharks, so there is a constant bias of three. Thus,

```
<controller>
   <AgentSchema name="Boid">
      <Navigation init="true" actuator="LazyNavigation">
         <CoordinationOperator type="priority">

            <MotorSchema type="repulsion" curve="linear" priority="6"
               PerceptualSchema="getNearestEdge" />

            <MotorSchema type="repulsion" curve="linear" priority="5"
               PerceptualSchema="getNearestShark" />

            <MotorSchema type="repulsion" curve="linear" priority="4"
               PerceptualSchema="getNearestMinnowCollide" />

            <CoordinationOperator type="sum" priority="3">

               <MotorSchema type="mimic" curve="linear" weight="3"
                  PerceptualSchema="getNearestMinnowInner" />

               <MotorSchema type="repulsion" curve="linear" weight="1"
                  PerceptualSchema="getNearestMinnowInner" />

            </CoordinationOperator>

            <CoordinationOperator type="sum" priority="2">

               <MotorSchema type="mimic" curve="linear" weight="3"
                  PerceptualSchema="getNearestMinnowOuter" />

               <MotorSchema type="attraction" curve="linear" weight="1"
                  PerceptualSchema="getNearestMinnowOuter" />

            </CoordinationOperator>

            <MotorSchema type="attraction" curve="linear" priority="1"
               PerceptualSchema="getRandomPoint" />

         </CoordinationOperator>
      </Navigation>
   </AgentSchema>
</controller>
```

Figure 3.2: The boid-like controller that we hand-coded as the control in our experiments. It consists of a priority coordination operator that chooses between avoiding pool edges, avoiding predators, closing in on other prey, avoiding collisions with other prey and random motion.

a Minnow controller that behaved "perfectly" and was never caught by any of the Sharks would score a 63, while a controller in which all Minnows were eaten would score a 3.

The scores generated by a given controller tended to be normally distributed. In earlier experiments we used the outcome of a single simulation run as our fitness function. This introduced more error into the result scoring, as the score could potentially be at the low or high end of the normal curve. However, we believe that error was not necessarily detrimental to the outcome of the experiments, because in practice it served to allow more genetic mixing and less elitism amongst the individuals of the population. Furthermore, the average fitness of the population showed steady improvement despite the potential for error. However, in later experiments, we ran each simulation multiple times and used the mean of the outcomes as the score for that individual. Simulations were run until either the standard error ($\sigma/\sqrt{n}$) fell below a threshold of 1.5, or the maximum of 10 iterations was reached. Another valid approach would be to check the relative precision of the results.

The search space for this problem is very large. In general, the number of priority motor schemas is $n = numberOfPerceptualSchemas * numberOfCurveTypes * maximumPriority * numberOfTypes$, or 1,890 in our setup. The number of possible weighted motor schemas may be calculated similarly. Given the maximum number of motor schemas allowed in a coordination operator $= p$, we can say that the number of possible priority coordination operators is $\binom{n}{p}$ or 1.99×10^{14} for our setup. The same can be said for sum coordination operators, so the total number of possible coordination operators is 3.98×10^{14}. Finally, with a maximum tree depth of k and a limit of two coordination operators as children of any other coordination operator (i.e., a tree arity of two) the maximum number of coordination operators in a controller is $2^{k+1} - 1$, or 63 in our case. So the number of possible controllers for our problem is $\sum_{k=1}^{5} \binom{3.98 \times 10^{14}}{2^{k+1}-1}$. Furthermore, the fitness of each controller is a random variable. As such, we cannot directly calculate the fitness value of each dimension, but in-

Table 3.2: Description of the evolution parameters for five different sets of experiments.

Set	Size	Carryover	Crossover Pool	Mutation Pool	Replaced	Dropped
1	12	top 50%	top 50%	mid 25%	bottom 25%	0%
2	24	0%	top 50%	mid 25%	25%	bottom 50%
3	24	top 25%	top 50%	mid 25%	25%	bottom 25%
4	12	top 25%	top 50%	mid 25%	25%	bottom 25%
5	24	top 25%	top & bottom 25%	mid 25%	25%	bottom 25%

stead have to rely on multiple independent trials to generate a confidence interval for the mean value of each controller.

We ran five sets of evolution experiments with different parameters (Table 3.2). Our first attempt was to achieve quick convergence. Our prototype implementation of Pickle is not optimized for speed, and so a single simulation run of 1,000 time steps typically takes around five minutes to complete. This being the case, we designed our first set of experiments to converge quickly by having a high degree of generational elitism and a strict penalty for poorly performing controllers. These experiments used a population size of 12 individuals. After each run of the generation, the individuals were ordered by fitness. The top six controllers were held over for the next generation. These top six were also randomly paired together to produce three offspring. The next best-performing three individuals were selected for mutation. Their mutated versions were carried over into the next generation. Finally, the worst three individuals were dropped from the population and replaced by the three newly generated offspring from the top six.

Our second set of experiments was designed to allow more genetic mixing amongst the population. For this, we used a larger population size of 24 individuals. After each generational run, the top 50% were selected for crossover, but were not held over to the next

generation. The next 25% were selected for mutation and the bottom 25% were dropped from the population and replaced by newly generated random controllers, along with an additional 25% random controllers.

The third set of experiments split the difference between sets one and two. These experiments held over the top 25%, used the top 50% for crossover, the next 25% for mutation, replaced the bottom 25% with the new offspring and introduced 25% newly generated controllers. The population size for this set was 24 individuals.

A fourth used the mean from a series of simulation runs for the fitness function, as described previously. Because it took up to ten times longer to evaluate each individual from the population, we used a smaller population size, 12 individuals as in the first set, but with the carryover proportions the same as in set three.

Finally, we ran a set of experiments with parameters similar to the fourth set of experiments, but with a larger population size of 24 individuals, and a crossover pool that consists of both the top 25% of individuals from the previous generation and the bottom 25% of individuals from the previous generation. This was intended to promote better genetic mixing and prevent premature convergence.

The results from the evaluation of each controller were recorded into an XML file along with a UUID generated as an identifier for the controller, a copy of the controller itself, its rank and generation, fitness score and a list of its "heritage," or all of its parents' UUIDs.

In the next section, we discuss our findings from these experiments.

Chapter 4

Experimental Results

We found that flocking behavior does evolve as a valid solution to predator evasion, but that it is not the only valid solution. We found at least one controller that does not exhibit flocking behavior (Figures 4.5, 4.6 and 4.8), but performs as well and slightly better on average than the hand coded flocking controller (Figures 4.1, 4.2 and 3.2) and evolved flocking controller (Figures 4.3, 4.4 and 4.7). Finally, we found that while temporary flocking can be achieved with only some of the fundamental flocking behaviors mentioned previously, sustained flocking that avoids predation has only been observed in controllers that coordinate all of the fundamental behaviors. A brief summary of results from each experiment set is listed in Table 4.1.

In the first set of experiments, designed for quick convergence and high elitism, we achieved flocking very quickly after 11 generations in one population. However, successive runs failed to achieve either flocking behavior, or fitness scores as high as those we have observed with flocking controllers. Instead, they tended to converge too quickly around a lower local optimum.

Table 4.1: Description of outcomes from each set of experiments.

Set	Results
1	Successfully evolved flocking in one population. Others converged too early.
2	No improvement in population fitness.
3	Evolved swarming as a solution to predator evasion.
4	Successfully evolved flocking in one population. Others converged too early.
5	Successfully evolved a very high quality swarming controller.

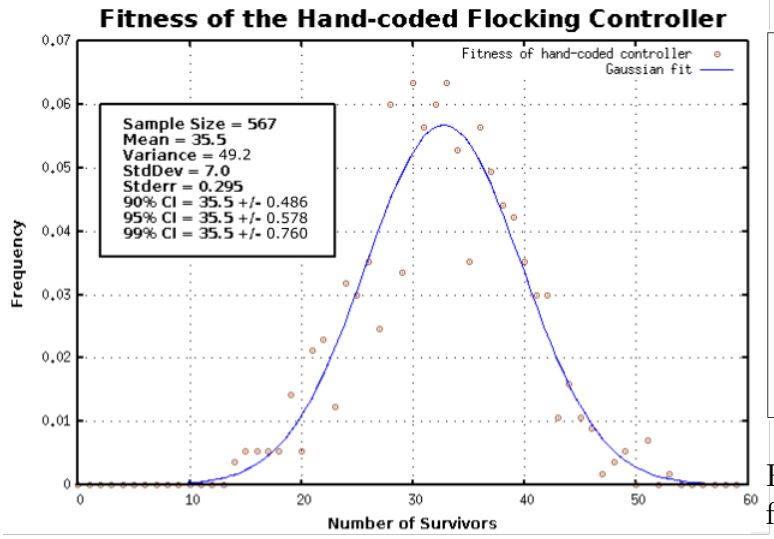

Figure 4.1: A frequency distribution of the fitness evaluations for the hand-coded flocking controller.

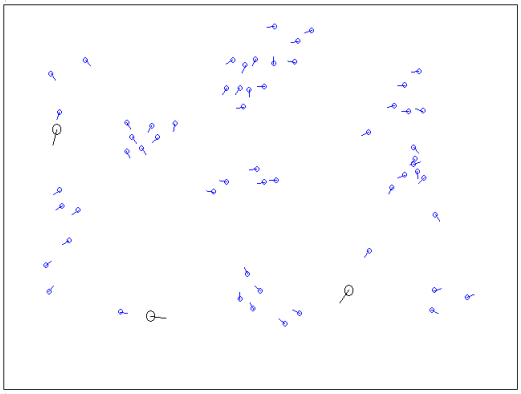

Figure 4.2: A screen shot of the flocking behavior displayed by our control group of "hand-coded" boid-like agents. Notice that the flocks grouping are fairly compact.

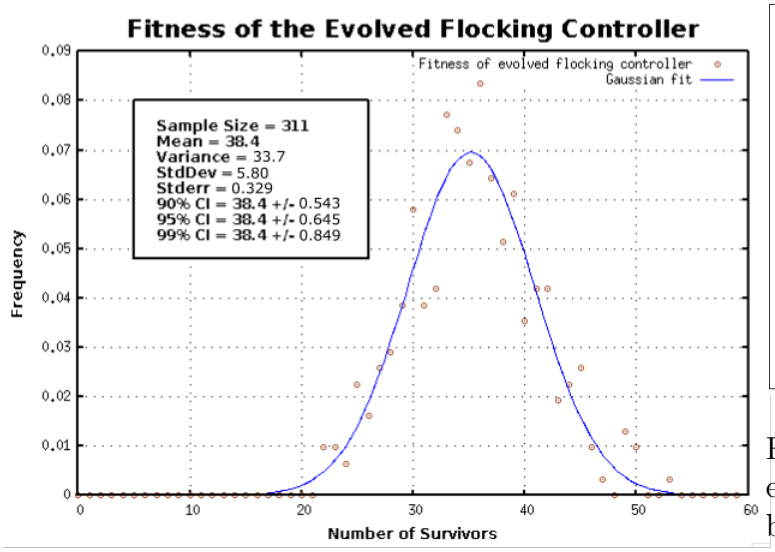

Figure 4.3: A frequency distribution of the fitness evaluations for the evolved flocking controller. This controller shows a slight improvement over the hand-coded flocking controller.

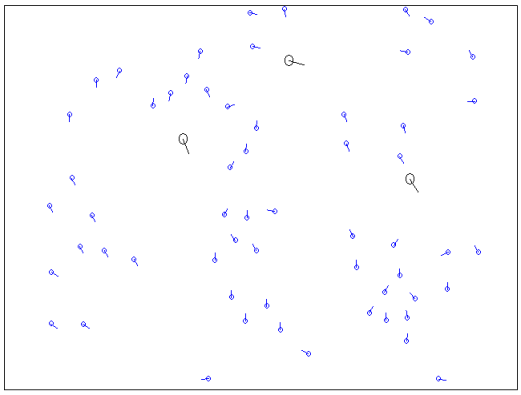

Figure 4.4: A screen shot of the evolved flocking behavior displayed by the most fit individual in generation 50 of the fourth set of evolutionary experiments. Notice that the flocks appear looser, more expansive and larger.

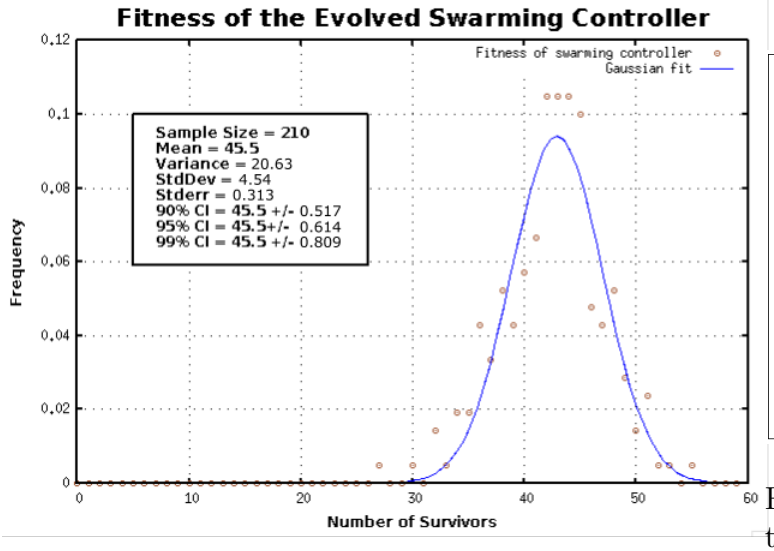

Figure 4.5: A frequency distribution of the fitness evaluations for the evolved "swarming" controller. This controller outperforms the hand-coded flocking controller by 30%.

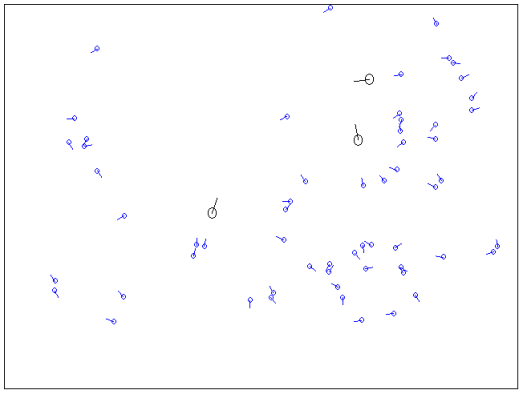

Figure 4.6: A screen shot of the evolved "swarming" behavior displayed by the most fit individual in generation 12 of the fifth set of evolutionary experiments.

The second set of experiments yielded no useful results. Without holding over any of the previous generation, the zero elitism approach, the fitness of the population never improved beyond the random walk.

The third set of experiments yielded steady growth in fitness population, but we did not observe flocking as the best solution. Instead, in all three of these evolution cycles, a controller that produced behavior more similar to swarming emerged. This controller behaved as well or better than either the hand coded flocking controller or the evolved flocking controller.

In the fourth set of experiments, in which we used the mean as a more stable fitness function, we again observed flocking evolve in one of the controllers starting at generation 28 (Figure 4.3). Once flocking was introduced into the gene pool, it came to quickly dominate the top 50% of individuals. Furthermore, the individuals from this population showed steady improvement over the course of the experiment (Figure 4.10). However, the other two runs from this experimental set failed to reach a fitness level as high as that of the population that achieved flocking, having reached plateaus similar to those of the first experimental set.

Finally, the fifth set of experiments produced a controller that performed significantly better than either the hand-coded or evolved flocking controllers (Figure 4.5). This controller emphasized predator avoidance and attraction to other prey to produce a "swarming" effect, but without the presence of mimic genes that create a coordinated flocking behavior. This controller performed 30% better than the hand-coded flocking controller and 18% better than the evolved flocking controller (Figure 4.9).

In examining and replaying the results from some of these experimental runs, we can reach some interesting conclusions about the nature of predator evasion in this simulated environment. First, we can learn about the requirements of flocking by examining controllers that exhibit some degree of flocking, but that failed to be successful at predator evasion. For instance, in one controller, flocking was observed, but the Minnows failed to avoid collisions

```
<controller id="627d0fb9-382e-42b6-a4d0-c6e0f7113877">
  <AgentSchema name="agentschema_1" init="true">
    <Navigation actuator="LazyNavigation">

      <CoordinationOperator type="sum">

        <MotorSchema type="repulsion" curve="logarithmic" weight="1"
               PerceptualSchema="getNearestEdge"></MotorSchema>
        <MotorSchema type="repulsion" curve="exponential" priority="2"
               PerceptualSchema="getNearestEdge"></MotorSchema>
        <MotorSchema type="repulsion" curve="logarithmic" weight="3"
               PerceptualSchema="getNearestMinnowInner"></MotorSchema>

        <CoordinationOperator type="priority" weight="1">

          <CoordinationOperator type="priority" weight="1">
            <MotorSchema type="repulsion" curve="logarithmic" weight="3"
                     PerceptualSchema="getNearestMinnowInner"></MotorSchema>
            <MotorSchema type="mimic" curve="logarithmic" priority="2"
                     PerceptualSchema="getNearestMinnowOuter"></MotorSchema>
            <MotorSchema type="attraction" curve="logarithmic" weight="2"
                     PerceptualSchema="getNearestEdge"></MotorSchema>
            <MotorSchema type="repulsion" curve="logarithmic" priority="1"
                     PerceptualSchema="getNearestShark"></MotorSchema>
            <MotorSchema type="repulsion" curve="exponential" priority="2"
                     PerceptualSchema="getNearestEdge"></MotorSchema>
          </CoordinationOperator>

          <MotorSchema type="mimic" curve="logarithmic" priority="2"
                   PerceptualSchema="getNearestMinnowOuter"></MotorSchema>
          <MotorSchema type="attraction" curve="logarithmic" weight="2"
                   PerceptualSchema="getNearestEdge"></MotorSchema>
          <MotorSchema type="repulsion" curve="logarithmic" priority="1"
                   PerceptualSchema="getNearestShark"></MotorSchema>
          <MotorSchema type="repulsion" curve="logarithmic" priority="1"
                   PerceptualSchema="getNearestShark"></MotorSchema>

        </CoordinationOperator>
      </CoordinationOperator>
    </Navigation>
    <StateChange></StateChange>
  </AgentSchema>
</controller>
```

Figure 4.7: An evolved controller that exhibits distinctive flocking behavior.

```
<controller id="29c793c7-7ac1-47ab-b5dd-3814fcdbd7f5">
  <AgentSchema name="agentschema_1" init="true">
    <Navigation actuator="LazyNavigation">
      <CoordinationOperator type="sum">
        <MotorSchema type="attraction" curve="exponential"
               weight="1" PerceptualSchema="getNearestMinnowOuter"/>
        <MotorSchema type="repulsion" curve="linear"
               weight="8" PerceptualSchema="getNearestShark"/>
        <CoordinationOperator type="sum" weight="1">
          <MotorSchema type="attraction" curve="exponential"
                  weight="3" PerceptualSchema="getNearestMinnowInner"/>
          <MotorSchema type="repulsion" curve="exponential"
                  weight="5" PerceptualSchema="getNearestEdge"/>
          <MotorSchema type="repulsion" curve="linear"
                  weight="7" PerceptualSchema="getNearestMinnowCollide"/>
          <MotorSchema type="repulsion" curve="logarithmic"
                  priority="4" PerceptualSchema="getNearestEdge"/>
        </CoordinationOperator>
      </CoordinationOperator>
    </Navigation>
    <StateChange></StateChange>
  </AgentSchema>
</controller>
```

Figure 4.8: An evolved controller that enables superior predator evasion, but does not use flocking.

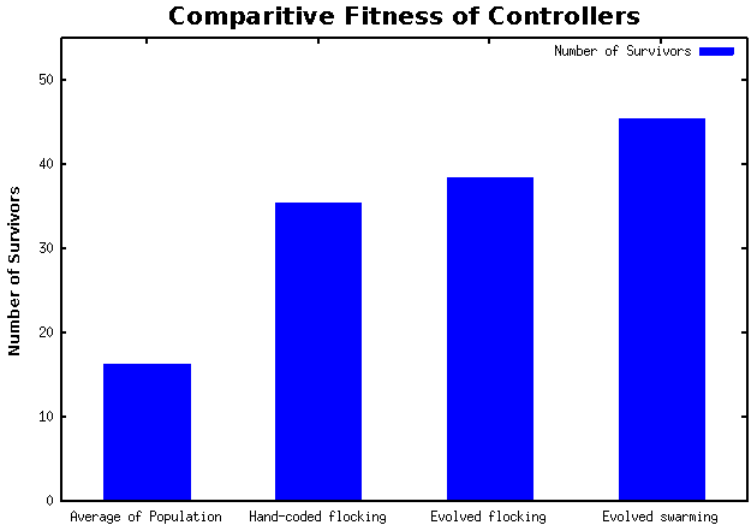

Figure 4.9: A side-by-side comparison of the fitness of the hand-coded, evolved flocking and evolved swarming controllers. The evolved swarming controller outperforms the hand-coded flocking controller by 30%.

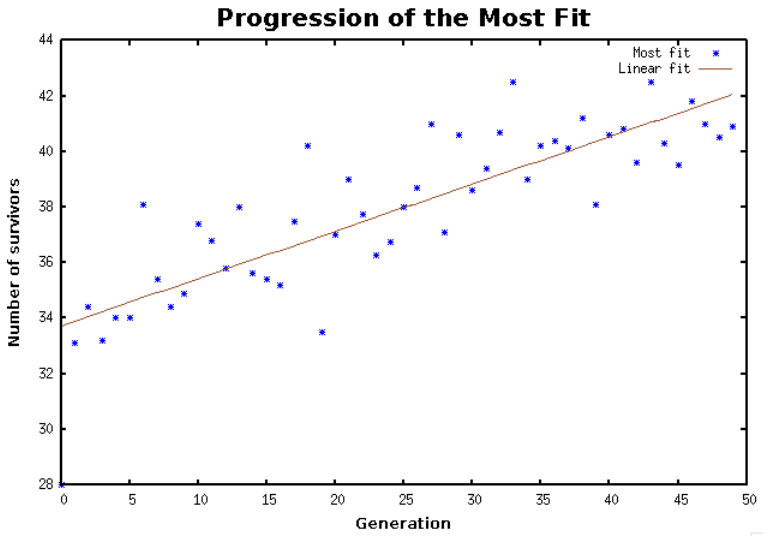

Figure 4.10: A plot of the "most fit" members from the fourth set of evolution trials. This shows a steady improvement in the maximum fitness of the population. This population produced flocking behavior starting at generation 28.

with each other. This left them in vulnerable clumps, which were devoured by the Sharks (Figure 4.11).

In another generation, the Minnows learned to flock very well, but failed to avoid obstacles, namely the pool edges. Once they got stuck on the pool edges, the Sharks were able to swim along the edges and eat them all.

Common to all successful controllers was the behavior to avoid predators, as might be expected, but also a motivation to group via an attraction to similar prey was also commonly observed. Finally, the ability to avoid obstacles proved to be another key to successful predator evasion. However, coordinated flocking behavior, while not a hindrance to predator evasion, was also not a requirement.

There are a few possible explanations for this. The first obvious one involves the limitations of the simulated environment. The predators used are somewhat naive in their implementation, and the simulated world is a fairly confined space, unlike the open water

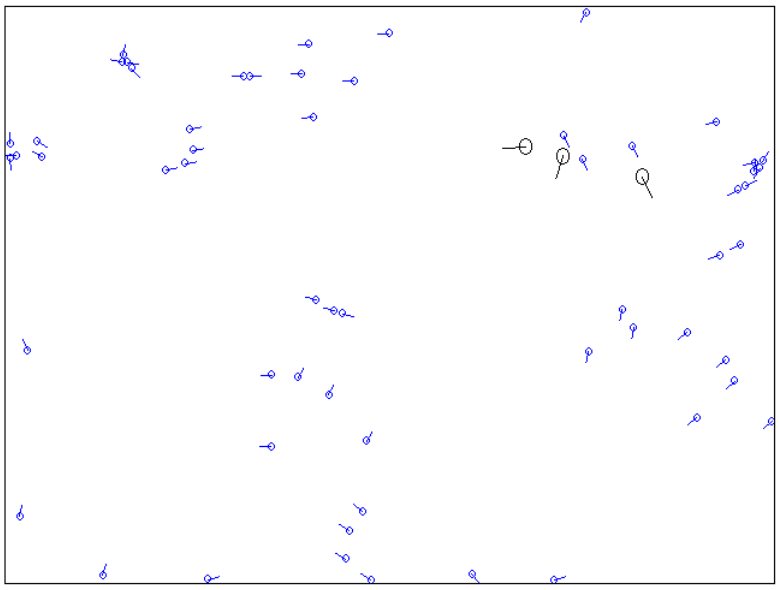

Figure 4.11: A screen capture of Minnows that showed promising flocking behavior. Note that they clump together, making them easy targets for predators.

areas in which flocking generally occurs. Strategies that work well in this environment, such as swarming near the edges of the pool, might work well in more constrictive spaces such as littorals and shoals, but fail in large open water, where schooling typically occurs. It could also suggest that flocking provides benefits in addition to predator evasion in the natural world. These could include easier movement through the environment by relying on neighbor motion to reduce drag, whether in aquatic or aerial environments. It could also include improved foraging through the group transmission of information.

Table 4.2 shows a list of all combinations of pairings of motor schema types and perceptual schemas. This table illustrates the relative importance of each trait pairing to predator evasion. The first column shows the average fitness of all individuals that possessed that pairing anywhere in their controller. The second column shows the percentage of the individuals in the sample set that possessed the trait pairing. For reference, the first column

Table 4.2: Average fitness and relative frequency of individual motor schema "genes". These numbers were produced by parsing all of the evolved individuals from Experiment Set 4.

Gene	Avg. Fitness	Frequency
All	16.37	100.0%
repulsion-getNearestEdge	23.42	37.4%
repulsion-getNearestShark	23.21	41.8%
repulsion-getNearestMinnowInner	21.87	37.8%
mimic-getNearestShark	19.50	32.7%
mimic-getNearestMinnowOuter	15.67	22.7%
attraction-getNearestEdge	14.43	17.6%
attraction-getNearestMinnowCollide	13.74	16.3%
repulsion-getNearestMinnowCollide	9.23	11.0%
mimic-getNearestEdge	8.49	14.0%
repulsion-getNearestMinnowOuter	7.44	11.9%
mimic-getNearestMinnowCollide	7.34	14.5%
attraction-getNearestMinnowInner	6.69	13.8%
mimic-getNearestMinnowInner	5.84	13.7%
attraction-getNearestShark	5.69	10.5%
attraction-getNearestMinnowOuter	5.45	10.9%

shows the average fitness of all individuals in the sample set. For this table, the sample set consisted of all individuals generated in the fourth set of experimental trials.

These tabular results reinforce our intuition that predator avoidance and pool-edge avoidance are key factors in successful predator evasion. This is followed closely by collision avoidance with other prey. Next comes a trait that causes prey to mimic, or essentially flock with nearby predators – an interesting strategy that performs surprisingly well. Next comes the trait that causes prey to flock in the outer zone of other prey. The prevalence of the mimic trait on the outer zone, as opposed to the inner zone could explain the looser flock groupings of our evolved flocking controller as opposed to the tighter flocks of our hand-written controller. Furthermore, the superior fitness shown by mimicking the outer

zone minnows versus the inner zone minnows could explain the improved performance of the evolved flocking controller over the hand-written flocking controller.

Two puzzling results are the relatively good performance of attraction to pool edges and the relatively equivalent performance of attraction and repulsion from the collision zone of other prey. The latter could signal a canceling out of the two traits, showing that the collision zone sensor is, on the whole, relatively unimportant. The former trait could be the result of noise in the results, since the relative structure of the genes within a controller is not taken into account in this analysis.

Finally, the table shows us an expected correlation between average fitness and frequency in the population. Traits that are more associated with successful controllers tend to stick around in the gene pool for longer and be passed along to more offspring.

These results reaffirm the work of previous researchers like Wood and Viscido that flocking does evolve as a response to predation. However, we expand on those previous efforts by showing that it is evolvable not only with respect to the parameters of agent sensors, but also with respect to the actual behavior mechanisms of a controller. Furthermore, this work shows that while flocking is a successful strategy to avoid predation, it is not the only one, and not always the most successful one. This suggests the possibility that coordinated flocking motion may serve multiple purposes, rather than being solely driven by predator evasion.

4.1 Comparison of Code Complexity

To further evaluate Pickle, we tested its ability to generate a variety of intelligent agents correctly from the XML descriptions as described above. We also compared the complexity of Pickle's XML descriptions against the complexity of Java-based examples of similar scenarios in Repast and MASON. We did this by building two scenarios, a terrestrial navigation simulation, suitable for land-bound animals like ants, and an aquatic simulation suitable for

studying swarming and schooling behaviors. Next we ran the simulations to check the agent behaviors for correctness. Correctness was evaluated by observing whether the specified behavior generated the expected behavior in simulation (e.g., if the specification calls for ants to go to green food pellets, do the simulated ants indeed go to green food pellets in the resulting simulation). Finally, we used the number of lines of code that a user would be expected to produce as a way to measure the complexity of the solution (Figure 4.3). We found that in both scenarios, Pickle yielded correct models of the agent behavior, using a substantially less complex description compared to handwritten code. In future work, it may be possible to reduce the lines of code even further by implementing Pickle to use an interpreted domain-specific language (DSL).

The terrestrial simulation generates a single agent, which navigates through a set of randomly placed obstacles to arrive at a food pellet. The simulation description, including all agents, their actuators and sensors, as well as the obstacles and general simulation parameters is represented by less than 100 lines of XML, and the controller for the agents, which has two MotorSchemas operating within a single AgentSchema, is represented by just twelve lines of XML. We compared this to the Keep-away Soccer demo from MASON, which features two mobile agents kicking a soccer ball. This simulation uses 574 lines of Java code in five classes. Similarly, the "Statechart Zombies" demo that ships with Repast occupies eight files and 299 lines of code.

Our second simulation was an aquatic simulation suitable for studying swarming and schooling behaviors. It consisted of 30 prey agents with Boid mechanics implemented as described previously, and two larger predator agents that seek out and consume the prey. This simulation also validated the friction setting of the simulated world. By setting the friction to a low value, the agents appeared to float in a single direction until their actuators push them in a new direction. As the simulation progressed, we could see the prey agents begin to cluster into small schools as they tried to avoid the roaming predators. The controller for the

Table 4.3: Results from our complexity evaluation of Pickle. We found that Pickle produced accurate agent behaviors with less complex code as measured by the number of lines and number of files required.

Scenario 1: Terrestrial Simulation			
Framework	MASON	Repast	Pickle
Lines of code	574	299	100
Number of files	5	8	2
Scenario 2: Aquatic Simulation			
Framework	MASON	Repast	Pickle
Lines of code	658	697	200
Number of files	6	8	3

prey agents consisted of five motor schemas: two to represent the attraction and repulsion of the Boid Mechanics, one to avoid the predators and one each to avoid obstacles and edges. The simulation and controllers for both the predator and prey comprised around 200 lines of XML spread out in three different files. We compared this against the Virus Infection demonstration that ships with MASON and uses similar predator and prey mechanics with a similar number of agents. The Virus Infection simulation used 658 lines of Java code in six different files. A similar simulation in Repast, the Flock demo, used 697 lines of code in eight different class files.

A video demonstration of these simulations is available for viewing on Vimeo [Medina, 2015].

Chapter 5

Conclusion

In this thesis, we have introduced a new simulation and modeling framework, Pickle, that uses a novel technique for representing agent controllers as user-defined XML descriptions. We have accomplished this by taking the notion of behavior-based controllers from robotics research and applying it to multi-agent simulations.

Furthermore, we have demonstrated the usefulness of this approach by carrying out new research in the field of collective animal behavior. We used genetic programming techniques to investigate the behavioral foundations, rather than the physical enablers, of flocking behavior. This work would not have been possible without the use of the machine readable and writable controller description that Pickle provides.

Finally, while Pickle is already showing its usefulness, there are many exciting possibilities for future expansion of this work. First, it would benefit greatly from a GUI designer that can generate the required XML files in the background before sending them to the simulator. It would also be worthwhile to enable the simulation viewer as a remote client that receives a stream of data and renders it as an animation. This would enable the simulations to be created and viewed through a lightweight client device, while a powerful server actually runs the simulations. Lastly, a high-performance driver, possibly utilizing

GPU computing, or possibly implemented using a driver for the SASSY framework, would be highly advantageous, particularly for performing genetic programming runs with much larger populations.

Bibliography

I Aoki. A simulation study on the schooling mechanism in fish. *Bulletin of the Japanese Society of Scientific Fisheries (Japan)*, 1982.

Michael A Arbib. Schema theory. *The Encyclopedia of Artificial Intelligence*, 2:1427–1443, 1992.

Michael A Arbib. *The handbook of brain theory and neural networks*. The MIT press, 2003.

Ronald C Arkin. *Behavior-based robotics*. MIT press, 1998.

Ronald C Arkin and Tucker Balch. Aura: Principles and practice in review. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2-3):175–189, 1997.

Tucker Balch. *Behavioral diversity in learning robot teams*. PhD thesis, Georgia Institute of Technology, 1998.

Tucker Balch, Frank Dellaert, Adam Feldman, Andrew Guillory, Charles L Isbell Jr, Zia Khan, Stephen C Pratt, Andrew N Stein, and Hank Wilde. How multirobot systems research will accelerate our understanding of social animal behavior. *Proceedings of the IEEE*, 94(7):1445–1463, 2006.

Michael E. Bratman, David Israel, and Martha E. Pollack. Plans and resource-bounded practical reasoning. *Computational intelligence*, 4(4):349–355, 1988.

Rodney A. Brooks. Planning is just a way of avoiding figuring out what to do next. September 1987.

Rodney A Brooks. How to build complete creatures rather than isolated cognitive simulators. *Architectures for intelligence*, pages 225–239, 1991.

Kathleen Carley and Allen Newell. The nature of the social agent*. *Journal of mathematical sociology*, 19(4):221–262, 1994.

Philip R Cohen and Hector J Levesque. Intention is choice with commitment. *Artificial intelligence*, 42(2):213–261, 1990.

Iain D Couzin, Jens Krause, Richard James, Graeme D Ruxton, and Nigel R Franks. Collective memory and spatial sorting in animal groups. *Journal of theoretical biology*, 218 (1):1–11, 2002.

Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer Science & Business Media, 2003.

Irenaus Eibl-Eibesfeldt. Ethology: The biology of behavior. 1970.

Alfredo Garro and Wilma Russo. easyabms: A domain-expert oriented methodology for agent-based modeling and simulation. *Simulation Modelling Practice and Theory*, 18(10): 1453–1467, 2010.

James L Gould. *Ethology: The mechanisms and evolution of behavior*, volume 85. WW Norton New York, 1982.

William D Hamilton. Geometry for the selfish herd. *Journal of theoretical Biology*, 31(2): 295–311, 1971.

Nick Howden, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. Jack intelligent agents-summary of an agent infrastructure. In *5th International conference on autonomous agents*, 2001.

Maria Hybinette, Eileen Kraemer, Yin Xiong, Glenn Matthews, and Jaim Ahmed. Sassy: a design for a scalable agent-based simulation system using a distributed discrete event infrastructure. In *Simulation Conference, 2006. WSC 06. Proceedings of the Winter*, pages 926–933. IEEE, 2006.

AJ Keane. The design of a satellite beam with enhanced vibration performance using genetic algorithm techniques. *The Journal of the Acoustical Society of America*, 99(4):2599–2603, 1996.

John R Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *IJCAI*, pages 768–774. Citeseer, 1989.

Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. "MASON": A multiagent simulation environment. *SIMULATION*, 81:517–527, 2005.

Douglas C. MacKenzie, Ronald C. Arkin, and Jonathan M. Cameron. Multiagent mission specification and execution. *Auton. Robots*, 4(1):29–52, 1997.

Terrance Medina. Pickle video demonstration. https://vimeo.com/terrancemedina/introtopickle, 2015.

Terrance Medina, Maria Hybinette, and Tucker Balch. Behavior-based code generation for robots and autonomous agents. In *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, pages 172–177. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014.

N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system, a toolkit for building multi-agent simulations, 1996. URL citeseer.ist.psu.edu/minar96swarm.html.

Marvin Minsky. *Society of mind*. Simon and Schuster, 1988.

Marvin L Minsky. Logical versus analogical or symbolic versus connectionist or neat versus scruffy. *AI magazine*, 12(2):34, 1991.

Nils J Nilsson. Shakey the robot. Technical report, DTIC Document, 1984.

Michael J North and Charles M Macal. *Managing business complexity: discovering strategic solutions with agent-based modeling and simulation*. Oxford University Press, 2007.

Jonathan Ozik and Nick Collier. Repast statecharts guide. http://repast.sourceforge.net/docs/Statecharts.pdf, 2014.

Miles T. Parker. Eclipse, agent modeling platform. https://eclipse.org/amp/, 2015.

Juan Pavón and Jorge Gómez-Sanz. Agent oriented software engineering with ingenias. In *Multi-Agent Systems and Applications III*, pages 394–403. Springer, 2003.

Timothy C Reluga and Steven Viscido. Simulated evolution of selfish herd behavior. *Journal of Theoretical Biology*, 234(2):213–225, 2005.

Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM Siggraph Computer Graphics*, volume 21, pages 25–34. ACM, 1987.

Andrew J Wood and Graeme J Ackland. Evolving the selfish herd: emergence of distinct aggregating strategies in an individual-based model. *Proceedings of the Royal Society of London B: Biological Sciences*, 274(1618):1637–1642, 2007.