Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

PICKLE: A FLEXIBLE ABMS FRAMEWORK FOR DYNAMICALLY GENERATING SERIALIZABLE INTELLIGENT AGENTS

Terrance Medina
Maria Hybinette

Department of Computer Science
The University of Georgia
415 Boyd Graduate Studies Research Center
Athens, GA 30602-7404, USA

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 always 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 simplifying the description of agent controllers are inadequate for that challenge, so we present Pickle, an ABMS platform that generates complete simulations and agents using behavior-based controllers from simple XML file descriptions.

1 INTRODUCTION

Ethology, the science of studying animal behavior, can be a time-consuming occupation. Ethologists spend hours observing animals, sometimes in the lab and sometimes 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 is a difficult and a time consuming process, and when the ethologist uses an executable model - a piece of software that runs in a simulated environment - as a research tool, the task becomes even more difficult (Balch, Dellaert, Feldman, Guillory, Isbell, Khan, Pratt, Stein, and Wilde 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 (Ozik and Collier 2014) or MASON (Luke, Cioffi-Revilla, Panait, Sullivan, and Balan 2005). This is not only time and labor intensive,, but could also present significant problems for animal researchers who are not experienced programmers. We believe this presents an opportunity to simplify or even automate the process of producing executable models of animal behavior.

The research fields of multiagent systems and model-driven engineering have 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 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. But 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. NetLogo is a very popular simulation framework that explicitly targets non-programmers(Wilensky 2015). NetLogo has a full-featured GUI interface, but its models must still be programmed using Logo, a functional programming language. Furthermore, its model-sharing functionality is based on pre-compiled applets, a dead technology, for which NetLogo has not yet found a suitable alternative (Tisue and Bertsche 2015).

In this paper 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.

Finally, 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.

2 PREVIOUS WORK

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, Hybinette, and Balch 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 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, Kraemer, Xiong, Matthews, and Ahmed 2006). SASSY uses a middle-layer API to wed an ABMS framework to a high-performance Parallel Discrete Event Simulation (PDES) kernel. But 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.

2.3 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, Arkin, and Cameron 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 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. But 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.

3 BEHAVIOR-BASED CONTROLLERS

Agent-based Modeling and Simulation is an M & S approach that uses autonomous software processes to generate emergent behavior and complex systems. These autonomous software entities are called Agents, and their defining feature is that they follow a *"sense-think-act"* operational life cycle. At a given step of a simulation, an agent takes input from its environment (sensing), processes it in some manner (thinking)

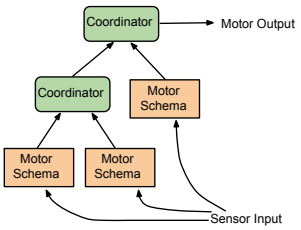

Figure 1: Reactive controllers emphasize tight coupling of sensor inputs and motor outputs, with minimal deliberation in between.

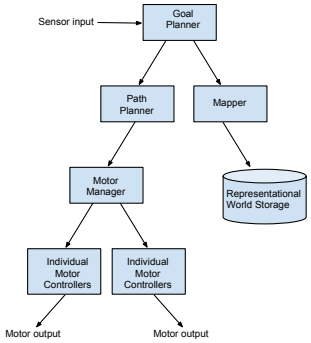

Figure 2: A hypothetical deliberative-style control architecture. Sensor input is passed to a long-term goal planner, which filters the information down to subordinate processes like path planning and mapping, as necessary. Unlike reactive controllers, there is typically a lot of computation between the sensor input and the motor output.

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.

Much of the research around agent control architectures has emerged from the robotics community. Over the last several decades, robot control architectures have gone from hierarchical controllers, which are plan-intensive and deliberative, to lightweight and dynamic reactive controllers. Later, the emergence of behavior-based controllers combined the two previous approaches.

In a hierarchical controller, a task is broken down into subtasks and distributed to subcomponents of the architecture. (See Fig. 2). Several successful robots from the 1970's followed this method, such as Shakey built at Stanford Research Institute (SRI).

Like ABMS agents, deliberative robot architectures operate on a sense-think-act cycle. And like every other attempt at agent controller generation, deliberative architectures are confined to a single decision-tree pattern of processing the sensor input and selecting an appropriate output for the actuators. But these hierarchical planners proved to be very cumbersome. They were too resource intensive for the early hardware on which they ran, and they were not able to adapt well to highly dynamic environments.

In the early 1980's, some roboticists started using so-called "reactive" controllers. These emphasized a tight coupling between sensor inputs and actuator outputs, with minimal planning or deliberation in between (See Fig. 1). In a reactive view, the presence of a 'god-like' controller is both unnecessary and ineffective in generating robust behavior. This point of view is summarized well in Rodney Brooks' 1987 memo "Planning is just a way of avoiding figuring out what to do next"(Brooks 1987).

Brooks later developed the Subsumption architecture, which inverted the flow of information in the controller. Rather than flowing from top (a master controller) to bottom (subordinate processes) as in an hierarchical controller, a subsumption architecture operates from bottom to top. Specifically, rather than just one master process, sensor inputs are consumed by many independent processes. Each of these processes 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.

Behavior-based architectures represent a compromise between the hierarchical and subsumptive approaches. They take the notion of bottom-up information flow from subsumptive controllers, combined with the representational world knowledge and 'sense-think-act' approach of hierarchical controllers, to create planning robot controllers that can adapt to uncertain and dynamic physical environments. We use Arkin's

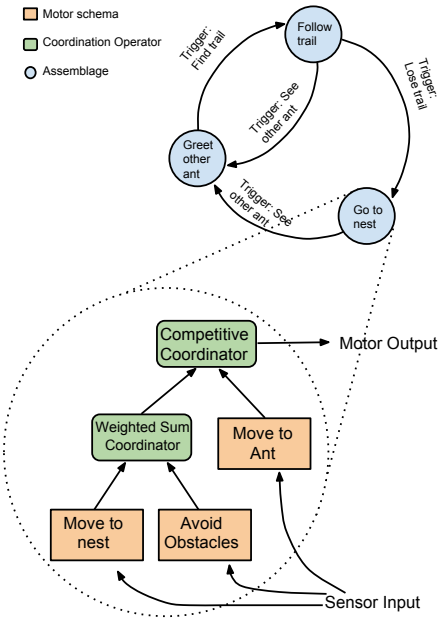

Figure 3: An example of a hypothetical hybrid controller. A temporal coordinator, or finite state machine, controls the behavior of the agent, while each state of the machine is a reactive controller, or behavioral assemblage.

Theory of Societal Agents as a conceptual model for our behavior-based experiments(Arkin 1998). This model itself derives from Marvin Minsky's Society of Mind(Minsky 1988), which envisioned intelligence as a phenomenon that emerged from a society of competing and cooperating interests.

4 SYSTEM DESIGN

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

4.1 Design Objectives

When designing Pickle, we had three design 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, and 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. This will enable future support for multiple simulation kernels, visualization methods and semi-structured data representations.

Third, everything must be serializable. Full serializibility 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.

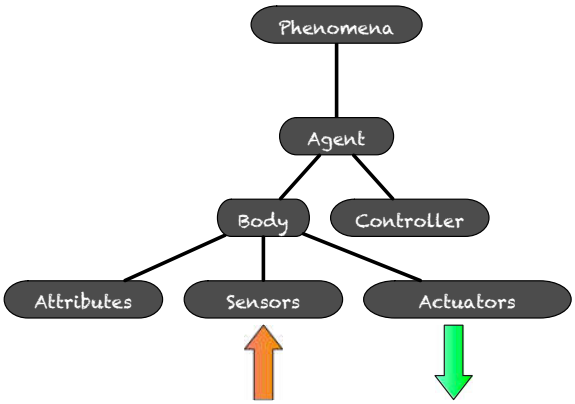

Figure 4: A description of the pickle taxonomy. Agents are distinguished from other Phenomena by the presence of Sensors, which consume information about the simulated environment, Actuators that effect changes to the environment, and a Controller that binds sensors with actuators.

4.2 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. 4.

4.3 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. But 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 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. 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.

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

4.5 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, 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 not on any previous steps. 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.

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. This sensor data is identified in 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.

4.6 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 is 'linear', 'quadratic' or 'exponential', and a weight value or priority value, to be used by its Coordination Operator.

4.7 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. 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 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.

4.8 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.

4.9 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.

4.10 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

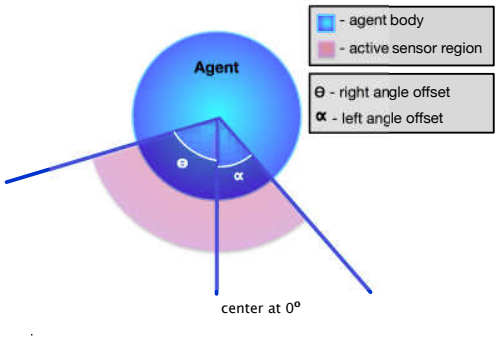

Figure 5: A view of how pie slice regions for sensors work. Each Sensor operates within a single pie-slice region, and Agents may have multiple sensors.

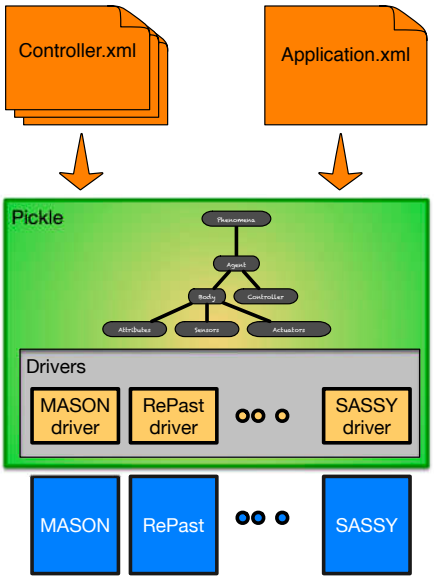

Figure 6: A high-level view of Pickle's architecture. Pickle is essentially an application layer that reads input from a Configuration Layer consisting of XML descriptions for simulation parameters, Agents and Controllers. It uses a set of drivers as a translation interface to run on multiple simulation kernels.

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.

5 IMPLEMENTATION

5.1 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.

5.2 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.

5.3 Simulation Kernel

We have implemented our prototype driver to support the MASON simulation kernel from George Mason University(Luke, Cioffi-Revilla, Panait, Sullivan, and Balan 2005). MASON operates on a single thread that uses a time-stepped event queue to poll the agents in the simulation sequentially. But we should reiterate that the design of Pickle is not restricted to either time-stepped or being 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 may be a single-thread that manages the underlying event queue and environment data structures, or it may 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.

5.4 Visualization

Currently, we use MASON's built-in Java Swing-based visualization layer to view our simulations. This is just a convenient stopgap decision. Our future plans are to implement a WebGL-based visualization layer for viewing the simulation as well as for designing the Agent XML. In fact, our next iteration of Pickle will be an entirely web-based application, in which the engine described here is run from a server, and sends positional data to a client, which renders the visualization remotely.

6 INITIAL PLATFORM EXPERIMENTS

For an initial stage of testing, we evaluated the success of Pickle in terms of 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 is evaluated by observing whether the specified behavior generates 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. We found that in both scenarios, Pickle yielded correct models of the agent behavior, using a substantially less complex description compared to handwritten code.

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

Table 1: Results from our initial testing 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

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 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).

7 LIMITATIONS AND FUTURE WORK

While our initial testing of Pickle is promising there is much work remaining. Our next step is to use Pickle as a platform for evolutionary programming, by machine-generating XML controller descriptions and modifying them as part of the evolution process. With genetic programming we hope to achieve qualitatively convincing flocking behavior from controllers that evolve for survival in the presence of a predator.

Writing XML by hand is a fairly cumbersome and error-prone process. Future versions of Pickle will feature a GUI-based front end for creating the agents, specifying the simulation parameters and generating the XML. Furthermore, we plan to expand Pickle into a web application, with a WebGL frontend for both constructing the agents and viewing the simulations. This would allow the application framework and kernel to be run on a remote server, potentially taking advantage of a high-performance kernel while allowing researchers to access the platform from cheaper notebooks or mobile devices.

The prototype implementation is not a high-performance framework, and as such does not scale well to large numbers of agents. We plan to implement a high-performance driver for Pickle in the near future.

Finally while we are distributing an initial implementation of Pickle there are still features important to ABMS research that are planned in later releases. For example, there is currently no integrated reporting feature for tracking simulation results aside from the GUI view. There is also no steering mechanism for human-in-the-loop changes to the running simulation. These however, will be addressed in future versions of the Pickle platform.

REFERENCES

Arbib, M. A. 2003. "The Handbook Of Brain Theory And Neural Networks". The MIT press.

Arkin, R. C. 1998. "Behavior-based Robotics". MIT press.

Balch, T. 1998. "Behavioral Diversity In Learning Robot Teams". Ph. D. thesis, Georgia Institute of Technology.

Balch, T., F. Dellaert, A. Feldman, A. Guillory, C. L. Isbell, Z. Khan, S. C. Pratt, A. N. Stein, and H. Wilde. 2006. "How Multirobot Systems Research Will Accelerate Our Understanding Of Social Animal Behavior". *Proceedings of the IEEE* 94 (7): 1445–1463.

Brooks, R. A. 1987, September. "Planning is Just a Way of Avoiding Figuring Out What To Do Next".

Garro, A., and W. Russo. 2010. "EasyABMS: A Domain-expert Oriented Methodology For Agent-based Modeling And Simulation". *Simulation Modelling Practice and Theory* 18 (10): 1453–1467.

Hybinette, M., E. Kraemer, Y. Xiong, G. Matthews, and J. Ahmed. 2006. "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*, 926–933. IEEE.

Luke, S., C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. 2005. ""MASON": A Multiagent Simulation Environment". *SIMULATION* 81:517–527.

MacKenzie, D. C., R. C. Arkin, and J. M. Cameron. 1997. "Multiagent Mission Specification and Execution". *Auton. Robots* 4 (1): 29–52.

Medina, Terrance 2015. "Pickle Video Demonstration". https://vimeo.com/terrancemedina/introtopickle.

Medina, T., M. Hybinette, and T. Balch. 2014. "Behavior-based Code Generation For Robots And Autonomous Agents". In *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, 172–177. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Minsky, M. 1988. "Society Of Mind". SimonandSchuster. com.

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

Parker, Miles T. 2015. "Eclipse, Agent Modeling Platform". https://eclipse.org/amp/.

Pavón, J., and J. Gómez-Sanz. 2003. "Agent Oriented Software Engineering With INGENIAS". In *Multi-Agent Systems and Applications III*, 394–403. Springer.

Reynolds, C. W. 1987. "Flocks, Herds And Schools: A Distributed Behavioral Model". *ACM Siggraph Computer Graphics* 21 (4): 25–34.

Tisue, Seth and Bertsche, Jason 2015. "NetLogo Applet Information". https://github.com/NetLogo/NetLogo/wiki/Applets.

Wilensky, U. 2015. "NetLogo". http://ccl.northwestern.edu/netlogo/.

AUTHOR BIOGRAPHIES

TERRANCE MEDINA is a Master's Degree student at the University of Georgia with a research emphasis on Agent-based Modeling and Simulation. His email address is medinat@cs.uga.edu.

MARIA HYBINETTE is an Associate Professor of Computer Science at the University of Georgia, performing research in high performance simulation systems. Her e-mail address is maria@cs.uga.edu.