Behavior-based code generation for robots and autonomous agents

Terrance Medina
The University of Georgia
Athens, Georiga
medinat@cs.uga.edu

Maria Hybinette
The University of Georgia
Athens, Georiga
maria@cs.uga.edu

Tucker Balch
Georgia Institute of Technology
Atlanta, Georgia
tucker@cc.gatech.edu

ABSTRACT

Constructing accurate animal behavior models often requires close interaction between a diverse group of domain experts in fields such as biology, ecology, modeling and simulation. To ease the effort and time involved in creating these models, we propose *on-the-fly automatic behavior model generation.*

Automatic code generation is a well-established software engineering technique. It has proven valuable in GUI generation, web service specification and in multi-agent systems design. But while these techniques have served programmers well, there is still a lack of tools that are targeted towards domain specialists utilizing modeling and simulation frameworks.

To address this deficiency we present a tool that leverages robot control architectures to provide automatic code generation of animal models using an intermediate language that is readable and writable by both human and machine.

Categories and Subject Descriptors

I.6.5 [**Simulation and Modeling**]: Model Development Automatic code generation for intelligent agents

General Terms

Design, Languages

Keywords

Code generation, robot controllers, modeling and simulation

1. OVERVIEW

An *ethologist*, or animal behavior researcher, who wants to create an executable model of animal behavior to run in a simulation faces a daunting challenge. First, he or she will have to spend hours observing the animal with a notebook, recording their observations in minute detail. Next these observations must be translated into an *ethogram*, a graphical depiction of the behavior that resembles a finite state machine or Markov model. Finally, this ethogram must 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 can also present significant problems for animal researchers who are not also experienced programmers.

In fact, this problem is not unique to the field of ethology. In specialized domains such as animal behavior research, business process management and graphic design, software creation has largely moved out of the hands of software engineers and into the hands of domain specialists. This has fueled a demand for automatic code generation frameworks, which allow the creation of specialized software with little or no actual code writing on the part of the programmer. For example, a business manager might use a graphical flowchart to generate process management software to automate business transactions. Graphic designers use GUI web development kits to generate interactive web pages. Automatic code generation is even used by software engineers to ease the burden of writing highly redundant "boilerplate" code like remote procedure calls and web services. And increasingly, researchers in sociology, economics and social animal behavior are turning to Agent-based Modeling and Simulation (ABMS) tools to conduct their research.

There have been some attempts to meet this need in the field of Multi Agent Systems (MAS) research, by constructing toolkits to automatically generate agent controllers from visual specifications. But these frameworks either rely on a UML type of model to describe agents, which is not intuitive to domain specialists, or they support sequential, procedural types of agent controllers, which are often inadequate to describe complex animal behaviors.

We present a code-generation framework that uses behavior-based robot control architectures as a model for agent controllers. Furthermore, to maximize code flexibility, we choose the XML as a language-neutral intermediate language, to allow our agent controllers to be not only machine readable, but machine writeable – an opportunity we intend to exploit in future work.

A popular technique for automatic code generation is the use of templates. Template-based code generation relies on predefined translation descriptions. For instance, a template might indicate that the presence of a certain keyword or textual pattern as input should produce a segment of code as output. This is identical to the problem of programming language compilation, and indeed the YACC and GNU Bison programs are well-known template-based code generators.

SIMUTools March 03 – 05, 2014, Lisbon, Portugal

Simutools 2014, March 17-19, Lisbon, Portugal

DOI 10.4108/icst.simutools.2014.254853

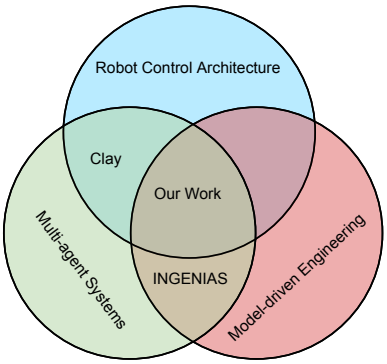

Figure 1: This work combines approaches from three disciplines: Multi-agent systems, Model-driven engineering and Robot control architectures.

The problem we face is the use of such templates to produce an executable program from a description of behavior, and to make that executable model dynamically available at run-time. Our solution is to combine two well-established paradigms - behavior-based robot controllers, and an automatic code-generation framework - in a novel way. This allows us to generate agent controllers that respond to dynamically changing environments.

But while the use of automatic code-generation frameworks is well established, there have been several different approaches to describe the models for those frameworks. For instance, the MAS community typically uses UML diagrams to model the configuration parameters of the system itself, while the control logic of the agents is created as either a procedural workflow, or a hierarchical decision making system in which tasks are subdivided into component objectives and delegated to subordinate processes by a master controller. The same approach has been carried over to the ABMS community with some success, as evidence by the "Agent Modeling Platform" (AMP) framework. But sequential and hierarchical controllers can become problematic in dynamic, unpredictable environments. Furthermore the use of UML to describe agent behavior, while close to the domain representation for software systems, is typically different from behavioral representations used by the domain specialists who are likely to make use of ABMS tools in their research.

In the remaining sections of this paper, we will provide a survey of the existing approaches, and present details of our framework. In section 2 we will review similarities and differences between multi-agent systems (MAS) and agent-based modeling and simulation (ABMS), while contrasting significant work in model-based engineering. In section 4, we will discuss the emergence and benefits of behavior-based robot control within the robotics community. In section 7 we will give technical details of our proof-of-concept implementation, and in the final section we will outline future directions and goals for this approach.

2. MULTI AGENT SYSTEMS

We define a multi-agent system as a software system consisting of autonomous processes known as agents, which perceive features of their environment and carry out actions to influence that environment. This is typically referred to as a *sense-think-act* cycle.

The notion of multi-agent systems has been applied to software engineering in the form of Agent-Oriented Systems Design (AOSD). This approach models a software system (e.g., a web server) as a set of processes within an environment. These processes have a hierarchical relationship with one another, an internal algorithm or control logic, and a mental state. The presence of a mental state is the differentiating feature between AOSD and the more conventional object-oriented design. Mental states often use Michael Bratman's *Beliefs, Desires, Intents* (BDI) model [3], in which 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. This model follows Allen Newell's notion of rational agents [8], where an agent selects from a variety of possible actions whether such an action will accomplish its goal.

In the next section, we will examine how the agent concept has been applied to modeling and simulation systems.

3. AGENT-BASED 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.

Agent-based Modeling and Simulation (ABMS) uses autonomous software entities called agents, that follow the *"sense-think-act"* operational life cycle. Most importantly, ABMS uses autonomous software processes to generate emergent behavior and complex systems.

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.

4. ROBOT CONTROL ARCHITECTURES

Much of the research around agent control architectures has emerged from the robotics community. Over the last several decades, robot control architectures have evolved 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, the way a general might relay orders to his or her lieutenants on a battlefield. (Fig. 3) Several successful robots from the 1970's followed this method, such as Shakey built at Stanford Research Institute (SRI). But 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.

As a reaction to hierarchical planners, 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. (Fig. 2)

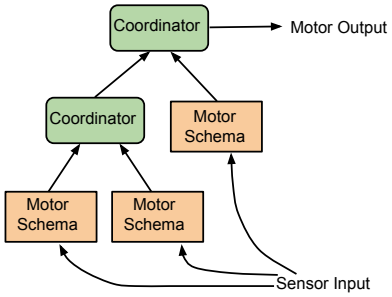

Figure 2: Reactive controllers emphasize tight coupling of sensor inputs and motor outputs, with minimal deliberation in between.

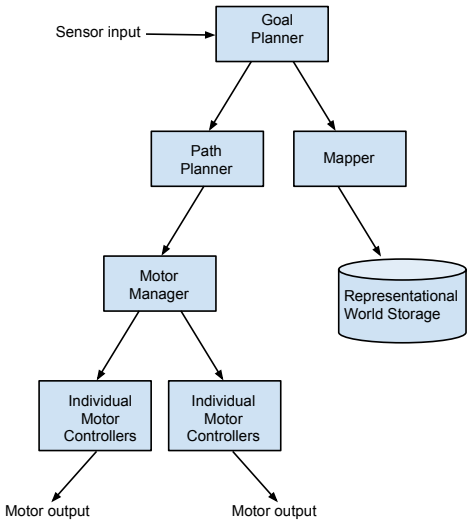

Figure 3: A hypothetical deliberative-style control architecture. Sensor input is passed to a long-term goal planner, which filters the information down to subordinate processes like path planning and mapping, as necessary. Unlike reactive controllers, there is typically a lot of computation between the sensor input and the motor output.

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 (if irreverently) in Rodney Brooks' 1987 memo "Planning is just a way of avoiding figuring out what to do next" [4].

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, 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.

Behavior-based architectures represent a compromise between the hierarchical and subsumptive approaches. They

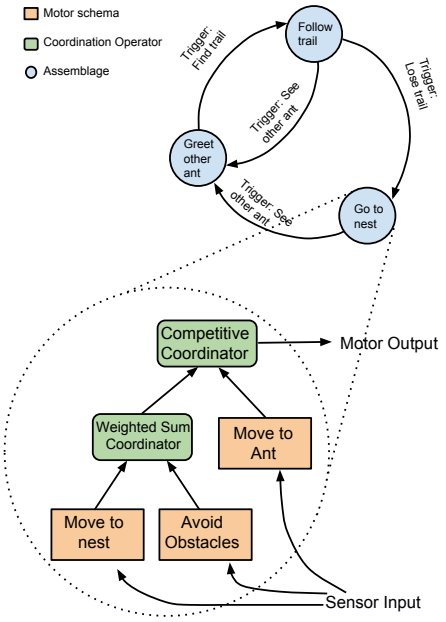

Figure 4: An example of a hypothetical hybrid controller. A temporal coordinator, or finite state machine, controls the behavior of the agent, while each state of the machine is a reactive controller, or behavioral assemblage.

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 Theory of Societal Agents as a conceptual model for our behavior-based experiments[2]. This model itself derives from Minsky's Society of Mind[7].

5. SCHEMA THEORY

The Societal Agents model uses *schema theory* to provide basic building blocks for complex agent controllers. Schema theory has roots as far back as Emmanuel Kant, but has been applied in the 20th century to neuroscience e.g., by Ron Arkin. A schema defines the process by which a sensor input produces an impulse to action. 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[2] and ethology[1].

We consider 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 the MacKenzie, Arkin and Cameron's MissionLab simulator for goal-based robot teams [6], 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.

Finally, we discuss Model-Driven Engineering, a software-engineering technique that uses abstract models of systems to represent the structure and functions of actual executable code. Unlike the models discussed previously which represented physical phenomena or entities via mathematical functions or executable software controllers, these are graphical models that visually capture salient features of a software system or module. The idea is that abstract representations of the actual programming code will better represent the meaning of the software modules being represented in a condensed form that might be obscured by having to read and understand the actual code base. A good abstract model will both encapsulate the important features of the things it models and hew closely to domain-specific representations. For instance, software engineers might make good use of the Unified Modeling Language (UML) as an abstract model for object-oriented code design, while Business Process Managers might be more comfortable working with flow diagrams to describe the sequence of transactions that constitute a business process.

These abstract models might be used strictly for planning and communication between teams, as in the case with waterfall-style software engineering, or they might be used to automatically generate some or all of the executable code by an MDE framework.

In the next section, we will discuss how Model-driven Engineering has been applied to multi-agent systems.

6. PREVIOUS WORK

While there has been extensive work on automatic code generation in the field of Agent-oriented Systems Design (AOSD), there has been relatively little targeted towards the Agent-based Modeling and Simulation (ABMS) community. This is particularly surprising because many of the users of ABMS toolkits are domain specialists, with little experience in software engineering. The MDE work generated by the AOSD community tends to focus on the total software development lifecycle, while the agent controllers are typically described by sequential process diagrams or deconstructive, hierarchical task managers. In this section, we will discuss the significant work on applying Model-Driven Engineering (MDE) to AOSD, as well as some work from the ABMS and robotics communities.

The INGENIAS framework[9], developed by the grasia! research group in Madrid, constitutes a total software lifecycle approach to Agent-oriented software development, with a visual specification language and the MetaEdit+ development tool. The visual specification language is built around the popular GOPRR meta-modeling language. GOPRR (Graphs, Objects, Properties, Relationships, Roles) is commonly used to describe data flow, processes diagrams and use-case diagrams. Like UML, it excels at defining relationships between objects.

The INGENIAS method breaks a MAS down into *views*, each of which describes one of 1) single agents, 2) how agents interact with each other, 3) relationships among goals and tasks, 4) an organizational model that describes grouping and hierarchy of system components or 5) the things an agent perceives in its environment. It is the first of these - the agent model, that is of interest to us here.

Another framework that brings MDE capabilities to the ABMS community is easyABMS[5] developed at the University of Calabria, Italy. Like the MDE tools developed for AOSD systems, easyABMS uses UML and process flow diagrams to describe agent behavior and simulation parameters. The framework generates code that integrates with the popular Repast Simphony simulation engine. An easyABMS agent controller is defined by a set of behaviors that may be triggered based on pre-defined parameters. Each behavior is essentially a sequence of actions with control flow statements (loops, branches, etc.) As discussed previously, this type of controller, while conceptually straightforward, and well-suited to software systems with little uncertainty, can be problematic in situated environments, which are often unpredictable and dynamic.

AMP, or Agent Modeling Platform , is a similar effort that makes use of the Eclipse Modeling Tools to generate essentially sequential agent controllers and configurations.

Our controllers are based on Arkin's Theory of Societal Agents. In MissionLab, Arkin developed his own Model-based Engineering approach, with automatic code generation based on a graphical user interface. MissionLab uses the Configuration Description Language (CDL), to describe high-level features of robot controllers, including behavior primitives and how those primitives combine to form assemblages. CDL is a quasi-functional language with support for recursively defined agents (i.e., agents that are combinations of other agents). CDL does not define the implementation of behavior primitives, but rather presumes that CDL primitives will be bound to some existing library of primitives designed for a particular physical platform.

7. IMPLEMENTATION

For our MDE framework we use the Extensible Markup

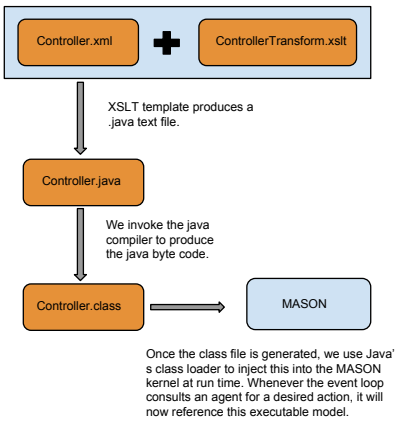

Figure 5: The process that transforms an XML description of behavior into an executable, which is injected at run-time into a simulation kernel.

```
<!-- BEHAVIORS -->
<behaviors>

        <behavior
        name="MOVE_TO_HOMEBASE_1"
        type="linear_attraction"
        target="HOMEBASE_1"
        controlled_zone="1.1"
        dead_zone="0"
        />
        <behavior
        name="MOVE_TO_HOMEBASE_2"
        type="linear_attraction"
        target="HOMEBASE_2"
        controlled_zone="1.1"
        dead_zone="0"
        />
        <behavior
        name="MOVE_TO_HOMEBASE_3"
        type="linear_attraction"
        target="HOMEBASE_3"
        controlled_zone="1.1"
        dead_zone="0"
        />
        <behavior name="NOISE" type="noise" timeout="2"/>
        <behavior name="AVOID_OBSTACLES" type="avoid" p1="2.0" p2="2.0"
target="OBSTACLE"/>

        <behavior name="SPIRAL" type="spiral" reset-when="TIMEOUT_5SEC" />

        <behavior name="STOP" type="stop" />
</behaviors>
```

Figure 6: The part of the XML description that describes the available motor schemas, or behavior primitives of the agent controller.

Language (XML) as an intermediate language. XML has several advantages for such a purpose. First, it is language-neutral. While our proof-of-concept generates Java code, suitable code should be producible in any number of traditional programming languages using template-driven techniques. Furthermore, XML is a human readable and writable specification. Users of the framework can directly observe and edit the XML representations of their agent controllers. Finally, and most interestingly, XML is machine readable and writable. This allows us several opportunities. First, it should be possible for a GUI interface to generate the XML models directly. Second, it should allow a software program to generate such a model from scratch. We intend to take advantage of this opportunity in future work by using these controllers as a gene in a genetic programming approach. Finally, it will allow agents to modify their own controllers while they are still running, by using Java's reflection capabilities.

To generate Java behavioral code, we use a template-based approach, in which Extensible Stylesheet Language Templates (XSLT) are used to match patterns in the controller XML document, and output plain text based on the pattern matched.

Once the Java code has been generated, it is compiled using ANT (Another Neat Tool), an XML-based build engine. ANT uses an XML build file, where each element specifies a target (such as compiling Java code, or transforming an XML document using a given XSL Template) and its dependencies. We generate this build file dynamically with the correct names of our controller and configuration classes based on a simple configuration file for the simulation. We use ANT's Java API for generating the build file and running during the simulation's run time. Finally, to load the freshly-built controllers, we use Java's ClassLoader capability to dynamically instantiate the class and inject it into the Java Runtime Environment. The process that allows us to go from a hierarchical specification of agent behavior to a fully-executable model is illustrated in Fig. 7.

Excerpts from an XML-based specification are shown in Figs. 6, 7 and 8. The structure of this XML document is derived from the Societal Theory and Schema terminology outlined previously. Fig. 6 describes Motor schemas, or primitive behaviors, which will be transformed into instances of Clay behaviors. These behaviors are then aggregated into Behavioral Assemblages or Agent Schemas as shown in Fig. 7. Finally, Fig. 8 shows the temporal coordinator, or Finite State Machine, that controls the transitions between these states. In this example, an agent (specifically an ant) wanders randomly between three different homebases, or nests, while avoiding obstacles in its path and interacting with other ants in a spiraling motion. Not shown here are the Perceptual Schemas, which specify those objects or agents in the environment of which this agent is aware, and the description of triggers between the states of the state machine.

Our executable Java code is built around Clay, a library of behavior-based primitives and coordination operators, with support for a controlling finite state machine. While XSL Templates could be written that target any such suitable behavior-based library, we found Clay to be a good candidate due to its origins as an implementation of the Societal Agents Theory. Furthermore, our generated code can be run in the MASON simulation kernel, a popular and feature-rich simulation engine.

As an alternative to the template-based approach described above, we considered the use of the Java Architecture for XML binding (JAXB), an architecture commonly used to generate web services. The JAXB framework uses Java annotations (preceded by the '@' symbol on class declarations) to bind Java classes to XML elements. So including such a bound XML element in an XML document produces an instance of the bound class. But we felt that this approach was too closely tied to the implementation details of the Java code, and did not sufficiently abstract away the semantic meaning of the controllers.

A limitation of this work is that, while it is possible to dynamically generate code at runtime in many languages, that code may not be loadable at runtime. We rely on Java's Reflection API to make this possible, but some languages, notably C and C++ do not natively allow for dynamic class

```
<!-- AGENT SCHEMA -->
  <!-- Behaviors co-ordinated by a FSM -->
  <states>

        <state name="MAKE_DECISION">
        <coordinator type="weighted_sum"/>
        <behavior weight="1.0" embedded="STOP"/>
        </state>

        <state name="GO_HOME_1">
        <coordinator type="weighted_sum"/>
        <behavior weight="1.0" embedded="MOVE_TO_HOMEBASE_1"/>
        <behavior weight="5.0" embedded="AVOID_OBSTACLES"/>
        <behavior weight="1.5" embedded="NOISE"/>
        </state>

        <state name="GO_HOME_2">
        <coordinator type="weighted_sum"/>
        <behavior weight="1.0" embedded="MOVE_TO_HOMEBASE_2"/>
        <behavior weight="5.0" embedded="AVOID_OBSTACLES"/>
        <behavior weight="1.5" embedded="NOISE"/>
        </state>

        <state name="GO_HOME_3">
        <coordinator type="weighted_sum"/>
        <behavior weight="1.0" embedded="MOVE_TO_HOMEBASE_3"/>
        <behavior weight="5.0" embedded="AVOID_OBSTACLES"/>
        <behavior weight="1.5" embedded="NOISE"/>
        </state>


        <state name="SPIRAL_CONT">
        <coordinator type="weighted_sum"/>
        <behavior weight="1.0" embedded="SPIRAL" />
        </state>

        <state name="DEFAULT">
        <coordinator type="weighted_sum"/>
        <behavior weight="1.0" embedded="STOP"/>
        </state>

  </states>
```

Figure 7: The part of the XML description that describes how the Motor Schemas are combined to form Behavioral Assemblages, or Agent Schemas.

```
<!-- Finite State Machine CONFIG -->
  <configuration>
        <start_state>MAKE_DECISION</start_state>

        <transition from="GO_HOME_3" to="MAKE_DECISION" trigger="CLOSE_TO_HOME_3"/>
        <transition from="GO_HOME_2" to="MAKE_DECISION" trigger="CLOSE_TO_HOME_2"/>
        <transition from="GO_HOME_1"  to="MAKE_DECISION" trigger="CLOSE_TO_HOME_1"/>
        <transition from="GO_HOME_3" to="SPIRAL_CONT" trigger="ANT_BUMP"/>
        <transition from="GO_HOME_2" to="SPIRAL_CONT" trigger="ANT_BUMP"/>
        <transition from="GO_HOME_1"  to="SPIRAL_CONT" trigger="ANT_BUMP"/>
        <transition from="SPIRAL_CONT"  to="MAKE_DECISION" trigger="TIMEOUT_10SEC"/>
        <transition from="MAKE_DECISION"  to="GO_HOME_1" trigger="CHOOSE_1"/>
        <transition from="MAKE_DECISION"  to="GO_HOME_2" trigger="CHOOSE_2"/>
        <transition from="MAKE_DECISION"  to="GO_HOME_3" trigger="CHOOSE_3"/>

  </configuration>
```

Figure 8: This section of the XML controller describes the transition table for the temporal coordination operator or Finite State Machine. Each state is a Behavioral Assemblage. Triggers, or edges, are described elsewhere.

loading (although there are some workarounds in the Linux environment).

Furthermore, our framework is really tailored for behavior-based agent controllers. A user who requires the use of sequential controllers with branching and looping logic will probably be frustrated by trying to adapt our framework to those techniques.

Finally, behavior-based controllers are really more like heuristics than algorithms. While they are robust in unpredictable environments, their behavior is not guaranteed, and thus may not be suitable for traditional AOSD systems such as web security systems, or e-commerce sites, where correctness is paramount.

8. CONCLUSION

We have presented a language-neutral technique to generate behavior-based agent controllers from hierarchical, text-based descriptions of those controllers. We believe this to be a compelling, new approach that opens up a variety of exciting opportunities for future research.

For the ABMS community, it means that agents can be prototyped quickly, or even automatically. For the AOSD community, it means that long-running agents can be modified on the fly to take advantage of new algorithms and libraries.

One goal of further research is to use these machine-writable controllers as genes in a genetic programming system to evolve agent controllers that mimic the behavior of animals. While there has been similar work on evolving robot controllers and ant-like agents by Koza, the controllers generated were sequential controllers, without the robustness of the behavior-based paradigm. Furthermore, we intend to use the technique to generate models, rather than optimized behavior, which is a novel approach.

We also intend to incorporate these controllers into a GUI environment, to allow non-programmers such as ethologists or sociologists to quickly and easily prototype agent controllers for their simulations.

Finally, a fully-featured solution should not be tied too closely to a particular behavior library or simulation engine. We plan to add support for several engines and behavior libraries in the future.

9. REFERENCES

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

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

[3] Michael Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

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

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

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

[7] Marvin Minsky. *Society of mind*. SimonandSchuster. com, 1988.

[8] Allen Newell. The knowledge level. *Artificial Intelligence*, 18(1):82–127, 1982.

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