Creating an AI modeling application for designers and developers

Ryan Houlette, Daniel Fu, Randy Jensen
Stottler Henke[*]

ABSTRACT

Simulation developers often realize an entity's AI by writing a program that exhibits the intended behavior. These behaviors are often the product of design documents written by designers. These individuals, while possessing a vast knowledge of the subject matter, might not have any programming knowledge whatsoever. To address this disconnect between design and subsequent development, we have created an AI application whereby a designer or developer sketches an entity's AI using a graphical "drag and drop" interface to quickly articulate behavior using a UML-like representation of state charts. Aside from the design-level benefits, the application also features a runtime engine that takes the application's data as input along with a simulation or game interface, and makes the AI operational. We discuss our experience in creating such an application for both designer and developer.

Keywords: Behavior modeling, AI authoring, finite state machines, behavior polymorphism

1 INTRODUCTION

The creation of the AI for a simulation is a common construction task, usually handled by technical developers. For the past three years we have been working on techniques to open the process of writing artificial intelligence behavior for simulations and videogames. One important aspect of our effort is to make an authoring tool that makes entity behavior accessible not only to developers, but to the rest of the team as well, such as simulation designers and analysts. Most of the domain knowledge resides with these individuals: They work with developers to ensure the entities behave appropriately. Unfortunately, a less efficient interaction occurs on the team when the designer communicates the desired behavior in a written document, waits while a developer implements the behavior, and then tests and revises the design based on the outcome. There are two reasons why this bottleneck happens. First, the implementation details are often hidden from the rest of the team. There are usually one or two developers who are responsible for the implementation, and while they ensure the AI works with the simulation engine, it is often not necessary for their code portion to be accessible to any other members on the team. The second reason is that even if the implementation details were available, their portrayal is often a body of programming code bearing little resemblance to the design documents that specify their intended characteristics. As a result, improvements to the AI must go through one or two developers.

In this paper, we discuss our efforts to build an authoring tool that bridges the gap between team members, empowering them to participate in varying degrees to both design and implementation. In the next section (Sec. 2) we discuss the origins of this work. Sec. 3 presents an overview of the system, describing the two major components of the software, an editing tool and runtime engine. Sec. 4 discusses extensions that we've made, including an authoring innovation that we refer to as "polymorphic behavior indexing" which simplifies behavior logic. Sec. 5 discusses user feedback, and Sec. 6 ends with a summary.

2 STEPS TO A SOLUTION

In 1999 we initiated work to construct reusable portions of our simulation middleware tools, starting with an AI tool. We knew most designers and end users did not know how to program a computer in a strict sense, but we did feel that if they could understand a visual representation of behavior, they could at least modify it, if not string together partially assembled behaviors later. We started with a graphical authoring tool which had its origins in a tactical action officer simulation and training system built in 1997 for the Navy's Surface Warfare Officers School. The system featured a

[*] email {houlette,fu,jensen}@stottlerhenke.com, phone 650-655-7242, fax 650-655-7243, web www.stottlerhenke.com, 1660 S Amphlett Blvd, Suite 350, San Mateo, CA 94402

suite of tools, one of which was a graphical finite state machine (FSM) editor. FSM's are a common type of language to describe behavior; indeed, almost all simulations and videogames use them in some capacity. In this case, they were used to describe entity behavior within the simulation, and to keep track of a student's situational context in order to evaluate whether doctrine was being correctly followed. FSM's have two favorable properties. First, they have the most straightforward "what you see is what you get" kind of behavior representation, oftentimes better communicated as pictures. Second, they have the positive computational properties of being simple, compact, and efficient. For these reasons we adopted FSM's as our basic computational model.

Still, for what we gained in computational efficiency, finite state machines are not computationally powerful. In 2001, we created a major extension that made the model hierarchical.[1] That is, an FSM's state can be an FSM itself. When writing FSM's, oftentimes a certain pattern of states and transitions will emerge. The author can decide to continually repeat the pattern across several FSM's, or to modularize the pattern as an FSM that can be invoked when needed. Had we adhered to the same FSM computational model, however, we would have seen the number of states increase exponentially according to the cross products of potential states and transitions. To reduce the number of necessary states, we used a hierarchical finite state machine (HFSM) model. This computational model is stack-based, where each time an HFSM is invoked, a reference to it is pushed onto the stack. We refer to each HFSM as a "behavior." This extension was significant in that it allowed us to approach Turing machine capability, modularize behavior, and not complicate the space and runtime efficiency.

Hierarchical behaviors have other advantages as well. By having authors break behaviors down into their logical functional components, modularity promotes reuse across the development team rather than reinvention. Once a behavior has been added to the behavior library, it is henceforth available as a ready-made building blocks for other, future behaviors. There are two important implications. First, a designer can reuse these building blocks to synthesize behavior at a more abstract level—perhaps nearing the level at which the designer would have otherwise communicated a written design. Instead, the designer can now use the behavior representation directly in an application that scales from abstract specifications of behavior down to the lowest-level behavior. The second implication is that each particular bit of functionality need only be implemented once in the library. After which sweeping modifications to entity behavior can be made by editing a single low-level behavior, effectively propagating to all higher-level behaviors that invoke it.

A number of other extensions were made, such as variables and arbitrary C-expression evaluation. We defer further discussion of extensions to Sec. 4.

2.1 Testbeds

For testbeds we chose two very different games. Our initial work started with a video game called "Half-Life." Parts of the source code were released by the game developer, Valve, to the public, allowing us to hook our interface and runtime engine to the game. The game is a first-person shooter that entails real-time action with 3-D perspective. Further, we used multiplayer versions of the game, allowing up to thirty-two people to play simultaneously. Our second testbed is "Civilization"—a popular turn-based game. This game is markedly different from Half-Life, emphasizing strategy without real-time pressures. We used an open source version of the game called "FreeCiv".

During development, we used two modifications on Half-Life, referred to as "mods." We started with one called "Team Fortress"—a mod emphasizing teamwork in "capture the flag" scenarios. We created behaviors to control entities at tactical and operational levels. For example, we created a defensive team where scouts patrolled routes and broadcasted enemy locations to the team. As threats were detected, reserves were dispatched to handle them. Depending on the outcome, more reserves were committed, or allowed to resupply themselves. One entity per team was appointed the leader who gave orders to teammates. All coordination happened through text communications within the game, thus a human player could issue commands to artificial teammates. We later generated offensive teams; e.g., a simple bounding overwatch involving two entities.

We later switched to a more realistic mod called "Counter-Strike," which features hostage, bomb, and escort situations with better weapon models. For example, counterterrorists must rescue hostages within an allotted time while terrorists strive to hold the hostages. A round ends when time runs out, all hostages have been rescued, or all members of a team are eliminated. The winning team is rewarded with resources to purchase better equipment for the next round. Because teams are rewarded, there is a much greater emphasis on teamwork.

The second testbed is "Civilization." The objective for each player is to build a civilization starting at some point in history, and either conquer all enemies, or become extremely advanced in technology. Players can decide whether to attack or ally with opponents, research newer technologies to further their economic and military capabilities, change the form of government, or determine the level of taxation.

3 SYSTEM OVERVIEW

Fig. 1 shows a system overview. There are two major components outlined: the authoring component and the runtime engine. On the left is the authoring component. First, the simulation designer or developer uses the authoring tool to declare a basic vocabulary of actions and predicates. A primitive action could be, say, to jump up, or to compute a path from one location to another. A predicate could be: Is there a threat nearby? Note that this editor only declares the capabilities of an entity for the behavior editor. When it comes time for an entity to compute a path, or to detect whether a threat is nearby, the functionality has to be realized over to the right in the interface.

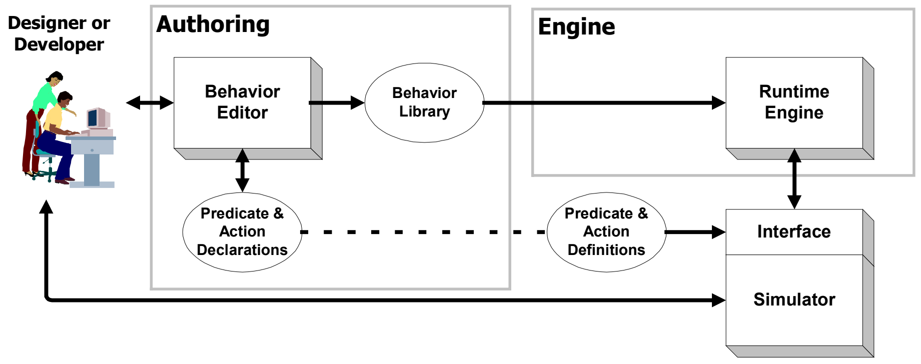

Figure 1: System Overview

The behavior editor uses the action and predicate vocabulary as building blocks to construct behaviors. Each behavior consists of actions, predicates, and other behaviors. Together, the behaviors constitute the behavior library that is used by the runtime engine. The runtime component then directs entities in the simulator. It does so indirectly through communications with an interface module residing between the runtime engine and simulator. A developer writes simulator-specific computer code in this interface to make the predicates and actions do something in the simulated world.

A behavior doesn't need a computer code representation. Ultimately, it decomposes into simple actions and predicates. We've found during development that as the types of information available to entities, and their capabilities, become better known and mature, the respective predicates and actions are updated both in the editor and the interface.

3.1 Behavior Editor

The behavior editor is a standard Windows application, enabling developers to quickly construct behavior using a visual syntax. Fig. 2 shows a screen snapshot of the editor. It shows a behavior built for Counter-Strike. The left pane holds a palette of actions, predicates, and defined behaviors. Whenever the user selects a behavior, its definition appears in the right pane. Here, a behavior appears as a set of rectangles connected by directed lines with ovals. The lower pane is an output pane for behavior compilation and debugging purposes. Before discussing the screenshot further, let us first turn attention to the representation.

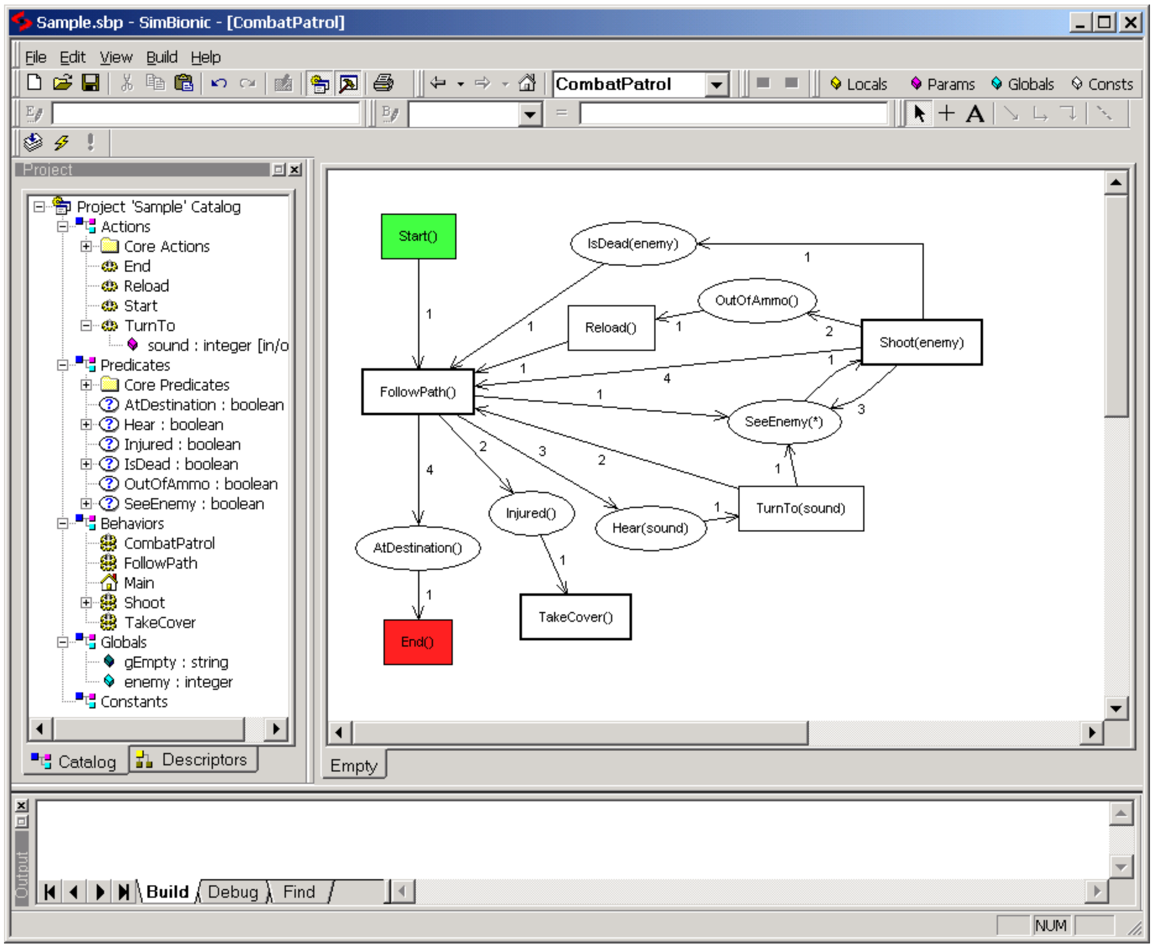

Figure 2: Authoring tool screenshot.

3.2 Representation

Each behavior as seen on the canvas consists of rectangles, directed connector lines, and ovals. Computationally, rectangles correspond to states in a finite state machine, while ovals correspond to conditions placed on state-to-state transitions. Within each rectangle, there is a reference to an action or behavior. References to behaviors appear as a bold outlined rectangle. Anything appearing in parentheses is a parameter. Conditions are logical formulas that evaluate to true or false. Numbers on transitions determine the order of evaluation of conditions.

Three states are of special significance when interpreting a behavior at runtime. The *current state* denotes the action or behavior currently being carried out by the entity; a behavior may have exactly one current state at a time. The *initial state* is simply the rectangle with which the behavior starts. There can be only one initial state per behavior, and it must appear with a green background. When a *final state*, marked with a red background, is reached, we consider the behavior to have finished (in FSM parlance, it has reached an accepting state).

An action appearing in a rectangle will interact with the game engine through the interface module; for example, in Counter-Strike, a human player may just press the "R" key to reload while the interface would mimic pressing the same key for the artificial entity. An action may also represent a deliberative or perceptual activity that has no direct physical effect on the game world, such as invoking a path planning algorithm. References to behaviors in rectangles are handled completely within the engine. Ultimately though, they boil down to primitive actions.

The current state in a behavior changes according to transitions. A transition is a chain of one or more connectors between two states. Between connectors are conditions that, if all evaluate to true, will change the current state from the source of the line to the destination.

3.3 Example

Fig. 2 shows a sample HFSM called "CombatPatrol" containing actions, transitions, and connections. The behavior on the canvas describes a fairly simple combat patrol behavior that causes a simulated soldier to move toward a specified destination, keeping an eye out for enemy soldiers. If an enemy is seen or heard, the entity will engage; if injured, the entity will take cover.

With a very simple visual paradigm, the user can quickly assemble behaviors with varying degrees of complexity for different levels of operations. Often the author wants to define a specific ordering for the evaluation of different conditions, and this is the motivation for the simple visual specification of numbers for evaluation ordering. The behavior in this example contains simple primitive actions like "TurnTo(sound)", as well as references to other behaviors defined elsewhere, such as "TakeCover."

The "TakeCover" sub-behavior is non-trivial in itself because it figures out the source of the threat which just caused an injury to the current entity, and performs a scan of its surroundings in order to find useful cover. These are independent activities that lend themselves well to abstraction, so that they can be used elsewhere (and by other development team members) as components of other behaviors. By doing so, the combat behavior can use the abstracted "TakeCover" sub-behavior, resulting in a simpler visual representation which is easier to understand.

4 EXTENDED FUNCTIONALITY

We've made a number of improvements to the basic HFSM model. There are three major augmentations: variables, interrupt transitions, and polymorphic indexing.

4.1 Variables

Variables are used for storage of data. There are two types of variables: global and local. Global variables can be used by any HFSM and typically carry information about the entity that's invoking the top-level HFSM. Local variables are used in one single HFSM and are not visible to other HFSM's unless passed as input. Because variables can be assigned the results of arbitrary expressions, our computational model has the power of a Turing machine.

4.2 Interrupt Transitions

Interrupt transitions temporarily push a special HFSM onto the stack which will then take precedence over HFSM's below it. Frequently there are instances where the entity will need to do some actions not associated with its primary task. For example, a Counter-Strike entity needs to notify its teammates of its location every ten seconds. This need exists no matter what its current task—be it combat, navigation, or support. However, sending a message to teammates shouldn't derail the current task by forcing a high-level transition to a broadcasting behavior. A better solution is to create an interrupt transition that will temporarily divert the flow of control towards an essential behavior. When that behavior is finished, the flow reverts back.

4.3 Polymorphic Indexing

The last major extension we made was polymorphism, an object-oriented programming term which means that a single object can be interpreted differently depending on the situation at hand. This translates into a single behavior having more than one definition. Which definition is actually invoked depends on the entity characteristics. For example, supposed we wanted to create a behavior to attack an opposing force. One simulation entity might be dismounted, while the other one is armored. We could define two behaviors "AttackDismounted" and "AttackArmored". After some use we may wish to create several more versions of an attacking behavior depending on the variety of simulation entities.

While behaviors do provide modular building blocks upon which we can construct behavior, their long-term use eventually introduces a proliferation of similar behaviors, usually with very minor changes introduced for new types of entities. Because of the references made in a behavior to other behaviors as part of a "behavior hierarchy," these minor changes introduced at an abstract level often entail propagating changes to successively lower-level behaviors. For example, a user may decide to model the morale and fatigue of a friendly force and have those attributes affect behavior. Thus, when the force is in conflict with opposing forces, the "AttackArmored" behavior would then dispatch a specialized version of a behavior based on, say, low morale and high fatigue. The invoked behavior, then, would be named "AttackArmored_LowMorale_HighFatigue." Likely, the lower-level behaviors will also need specialized versions as well. The unfortunate result is a bigger behavior library with no particular way for the user to simplify it through refactoring.

To cut the growth of the behavior library while at the same time maintaining specialized behavior, we created a polymorphic extension so that a single "CombatPatrol" behavior could entertain multiple versions. Exactly which version gets invoked depends on a set of hierarchical entity descriptors defined by the author. In this case, "Morale" and "Fatigue" descriptors are introduced, each with leaf values shown in two trees in Fig. 3.

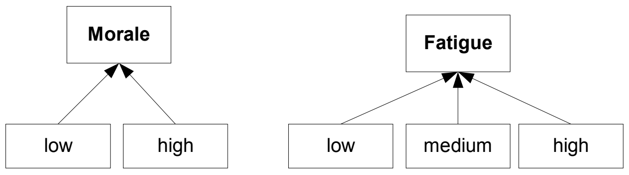

Figure 3: Hierarchical entity descriptors for polymorphic indexing.

A user specializes, or indexes, a behavior by associating it with exactly one node per tree. Here, there are six obvious combinations involving low/high and low/medium/high for each of the two descriptors, but in addition, inner nodes such as the roots may be selected, resulting in twelve possible specializations.

Each entity possesses a set of descriptors as well. In the case of the friendly force, that entity has "low" morale and "high" fatigue. Behavior selection for an entity proceeds by always picking the most specific version according to the degree of match between the entity and behavior indexes. If there is a behavior version of "CombatPatrol" indexed with low morale and high fatigue, then that version will be selected for the force.

Although, here, twelve behavior versions may be defined, in practice there are far fewer. The descriptor tree affords the ability to selectively customize behavior through the structured tree hierarchies. If a user wants only to define one version of a behavior, it would be indexed using the two roots. The friendly force uses this version of the behavior because a more specific version cannot be found. If the user wants to define a special case concerned only when morale is low, then he indexes the behavior by picking "low" from the first tree, and the root for the second. The friendly force would then use this version instead.

Note that these trees may be of arbitrary height and mirror the familiar notion of "multiple class inheritance" in object-oriented programming. Indexing behavior under this scheme allows us to condense the behavior library while at the same time freeing us to selectively specialize behavior.

Entities may change their indexes at any time. This change affects behavior selection from that point on, but does not require existing behaviors on the stack to be popped. For example, a friendly force that switches its morale from low to high and its fatigue from high to medium we would expect to exhibit different behavior with perhaps different adjudication results. Fig. 4 shows a sample specialization of the "CombatPatrol" behavior for this example. See the two selectable tabs below the canvas which indicate which descriptors on the left are its indexes.

By constructing behaviors using polymorphic indexing, users can easily change entity indexes to effect consistent behavior. If an operator wanted to "turn up the aggression" in a simulation, only a simple change in indexes is required.

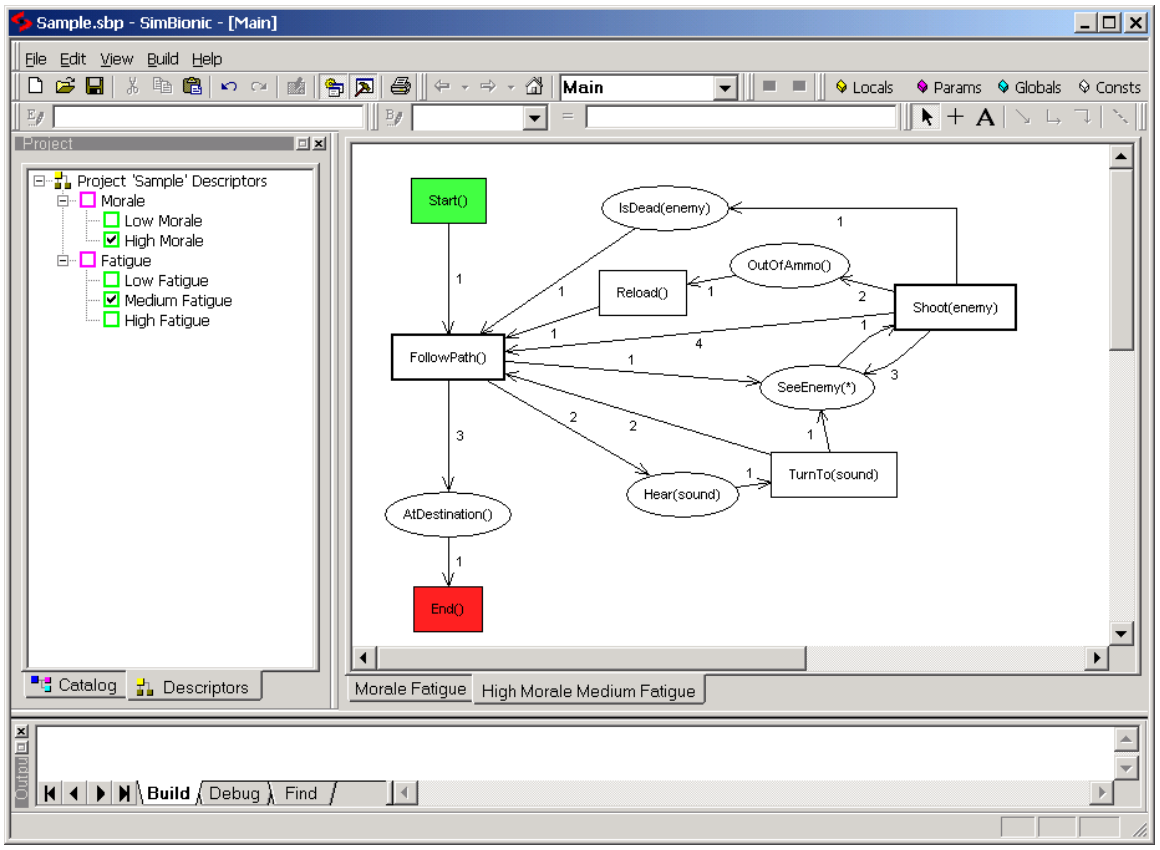

Figure 4: Authoring tool screenshot showing descriptor hierarchy. Tabs below the canvas show specialized indexes.

5 USER FEEDBACK

This approach has been validated with usability studies we have conducted in previous work. In a project conducted for the Navy[2], we adapted the technology to provide Navy instructors with a tool for creating intelligent agent based behaviors for use in a simulation trainer. Subject matter experts used the visual behavior definition environment provided by the tool to specify software agents to control enemy platforms as well as simulated team members within the simulation. A usability study was conducted with the end users, who reported quick authoring times and overall satisfaction as a result of the ability to author and modify simulation behaviors without relying on programmers. Another common response was that without this option, they simply could not have devoted the time to learn to use a more complex tool, and would therefore have been forced to rely on a collaborative implementation process with programmers.

6 SUMMARY

This paper has described an AI middleware tool that can be used by the entire simulation development team, including analysts and designers. Because of the hierarchical visual behavior representation, much of the implementation detail need not be seen in order to create new behavior. The computational model, while at its core a very simple mechanism, was augmented in ways to make it as powerful as a Turing machine.

An authoring innovation, behavior polymorphism, was introduced. It simplifies the authoring process by selectively specializing behavior dependent on the entity's description. Polymorphism also reduces complexity of dispatching logic as well as reducing the "name space" of behaviors.

ACKNOWLEDGEMENTS

This research was supported in part by Air Force Research Laboratory grant F30602-00-C-0036.

REFERENCES

1. R. Houlette, D. Fu, and D. Ross, "Towards an AI Behavior Toolkit for Games," AAAI Symposium on AI and Interactive Entertainment, 2001.
2. R. Stottler and M. Vinkavich, "Tactical Action Officer Intelligent Tutoring System," in the proceedings of the Industry/Interservice, Training, Simulation & Education Conference (I/ITSEC), 2000.