A Visual, Object-Oriented Approach to Simulation Behavior Authoring

Daniel Fu, Ryan Houlette, Randy Jensen, Oscar Bascara
Stottler Henke Associates, Inc.
San Mateo, CA
{fu,houlette,jensen}@stottlerhenke.com, ocb1@earthlink.net

ABSTRACT

Realistic behaviors for computer-generated forces (CGF) are as crucial as realistic graphics and terrain to the creation of high-quality military simulations. While a variety of simulation-building and 3D-modeling tools exist to help with the construction of the latter, the development of CGF behaviors still typically requires programming code to be written, either in a standard language such as C++ or Java or in a custom scripting language. As a result, the subject matter experts (SMEs) with the tactical or operational knowledge about how CGF should behave are seldom able to directly specify that behavior for a simulation, because they lack the necessary programming skill. The researchers therefore set out to develop an approach to simulation behavior authoring that minimizes the amount of programming required while still allowing the creation of sophisticated behaviors. Two key observations guided this effort. First, there exist already a variety of largely visual "languages" for describing complex sequences of actions and conditions – such as flowcharts, finite-state machines, and decision trees – that are either familiar to or quickly understandable by non-programmers. Second, CGF behaviors, particularly at a tactical or operational level, can often be adequately specified using such lightweight procedural representations. The end result was a behavior authoring methodology that is founded on a lightweight, visual, procedural approach to modeling CGF behaviors. This methodology, which is embodied in a graphical editor and runtime engine, is intended to allow non-programmers to participate more directly in the behavior authoring process. It is also designed to encourage good development practices such as reuse and top-down design, to which end it borrows several elements of object-oriented programming, including the notion of behavioral polymorphism. This paper describes the basic authoring methodology and underlying behavior representation. Examples are drawn from the Counter-Strike simulation testbed constructed by the researchers.

ABOUT THE AUTHORS

Daniel Fu has been a Project Manager for Stottler Henke Associates, Inc. since September 1998. He has worked on a variety of AI-related projects including AI middleware tools for simulation, tutoring systems for air tactics and counter-terrorist intelligence, and link discovery methods for plan detection. Daniel received a Ph.D. in Computer Science from the University of Chicago.

Ryan Houlette is a project manager and lead software engineer at Stottler Henke Associates. He holds an M.S. in Computer Science (Artificial Intelligence) from Stanford University. He has participated in the development of a wide range of AI systems, with a particular focus on autonomous agents and intelligent interfaces. He is currently lead software engineer on an intelligent track identification and analysis system for the Navy.

Randy Jensen is a project manager and software engineer at Stottler Henke. He has developed numerous intelligent tutoring systems for Stottler Henke, as well as authoring tools, simulation controls, and assessment logic routines. He holds a B.S. in symbolic systems from Stanford University.

Oscar Bascara is a contracted software engineer for Stottler Henke Associates. His interests include simulation behavior modeling and user interface design. He holds an M.Eng in electrical engineering from Cornell University and an M.A. in mathematics from the University of California at Berkeley.

A Visual, Object-Oriented Approach to Simulation Behavior Authoring

Daniel Fu, Ryan Houlette, Randy Jensen, Oscar Bascara
Stottler Henke Associates, Inc.
San Mateo, CA
{fu,houlette,jensen}@stottlerhenke.com, ocb1@earthlink.net

INTRODUCTION

Realistic behaviors for computer-generated forces (CGF) are as crucial as realistic graphics and terrain to the creation of high-quality military simulations. While a variety of simulation-building and 3D-modeling tools exist to help with the construction of the latter, the development of CGF behaviors still typically requires programming code to be written, either in a standard language such as C++ or Java or in a custom scripting language. As a result, the subject matter experts (SMEs) with the tactical or operational knowledge about how CGF should behave are seldom able to directly specify behaviors for a simulation, because they lack the necessary programming skills. Instead, they must provide detailed specifications of the desired behavior to simulation programmers, who then translate it into code. This multi-stage development process leads to slower simulation development and a greater chance for errors in the translation process, which lead in turn to higher development costs. Giving SMEs the capacity to directly author CGF behaviors therefore has the potential to streamline the simulation development process and improve CGF quality as well.

With this goal in mind, the researchers set out to develop a behavior authoring approach that minimizes the amount of programming required while still allowing the creation of sophisticated behaviors. Two key observations guided this effort. First, there exist already a variety of largely visual "languages" for describing complex sequences of actions and conditions, such as flowcharts, finite-state machines, and decision trees, that are either familiar to or quickly understandable by non-programmers. Using such a language as the basis for a behavior representation would have the advantage of flattening the learning curve for behavior authors and also of avoiding unnecessary proliferation of terminology and formalisms.

Second, although cognitive architectures like SOAR and ACT-R provide powerful and flexible frameworks for modeling general human behavior, only a fraction of their capability is needed to capture many reasonably complex real-world behaviors. CGF behaviors, particularly at a tactical or operational level, can often be adequately specified using lightweight procedural representations such as those mentioned above. Because developing behaviors in a cognitive architecture requires a level of technical skill that the average SME is unlikely to possess (at least without special training), it is therefore preferable to make use of these lightweight representations whenever possible to maximize the SME's ability to view and manipulate behaviors.

This research culminated in a behavior authoring methodology that is founded on a lightweight, visual, procedural approach to modeling CGF behaviors. This methodology, which is embodied in a graphical editor and runtime engine, is intended to allow non-programmers to participate more directly in the behavior authoring process. It is also designed to encourage good development practices such as reuse and top-down design, to which end it borrows several elements of object-oriented programming. The methodology has been used to create behaviors for a number of simulation domains, ranging from the game Pac-Man to a full-fledged tactical simulation for training naval Tactical Action Officers.

This paper describes the basic authoring methodology and its underlying behavior representation. Examples are drawn from the Counter-Strike domain that was used as a primary testbed (described below). The results of several informal evaluations of the methodology are also presented.

COUNTER-STRIKE TESTBED

To serve as a testbed for the behavior representation methodology, the researchers implemented a set of automated players for the popular multiplayer game Counter-Strike, which is a freely-available add-on, or "mod", for the commercial first-person shooter Half-Life (Counter-Strike, 2003). Counter-Strike depicts a urban hostage-rescue scenario in which one team of soldiers attempts to infiltrate an enemy base and rescue the hostages held within, who are guarded by a team of opposing soldiers. These soldiers are typically controlled by human players, but it is also possible to construct computer-controlled players, known as "bots," that can play against humans or other bots.

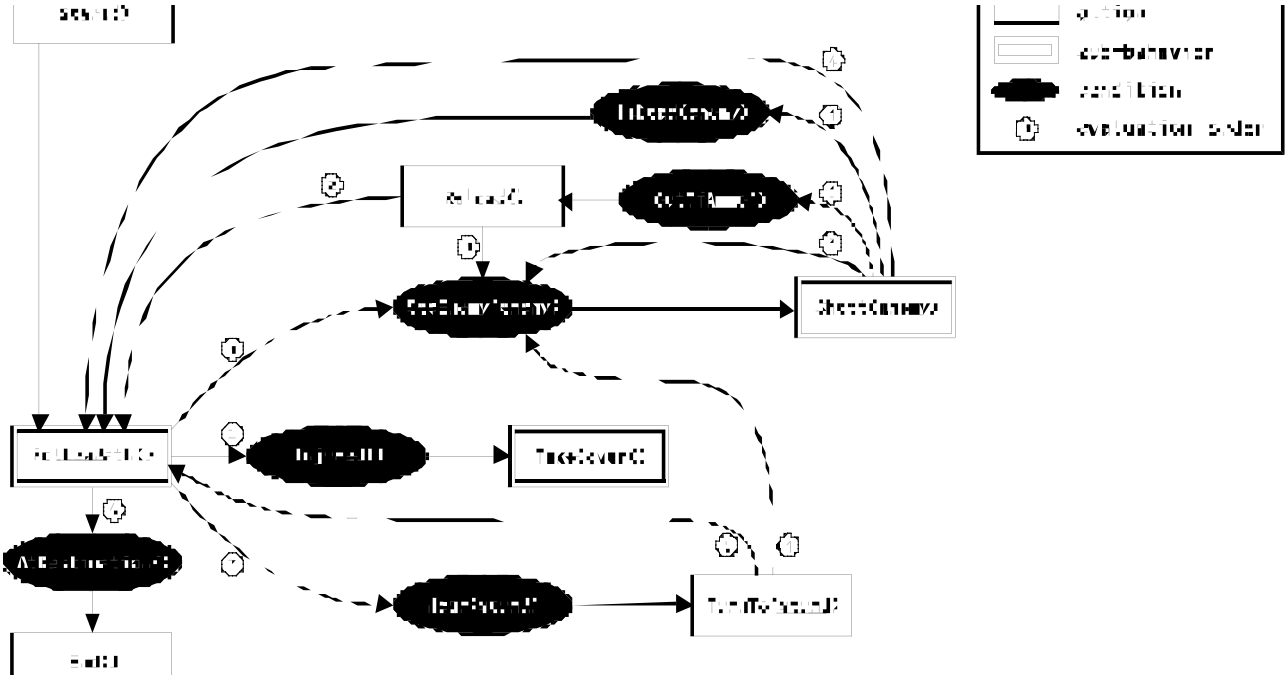

Figure 1. The *CombatPatrol* Behavior Transition Network.

Counter-Strike was chosen as the testbed for several reasons. First, it offered a 3D world that was continuous in both time and space, which provided a rich and challenging environment for automated players to act in and respond to. Second, it presented an interesting domain with opportunities for complex tactics and team coordination between entities. Finally, the Counter-Strike source code has been made available to the public, which greatly simplified the task of interfacing the runtime engine to the game.

The researchers developed a custom C++ interface to the game that allowed the bots to interact with the game engine in exactly the same manner as a human player. This permitted them to focus on authoring realistic behaviors rather than on low-level implementation details. Once the interface was complete, it was possible to construct behaviors for two opposing teams of three bots each in approximately four person-weeks. The resulting automated teams were capable of successfully completing their objectives of either rescuing the hostages or preventing them from being rescued. In addition, they performed competently when pitted against moderately skillful human players.

BEHAVIOR REPRESENTATION

The underlying representation for behaviors in this methodology is an augmented version of the basic finite-state machine called a *behavior transition network*, or BTN. Like a finite-state machine, a BTN consists of a collection of nodes connected by conditional transitions. Each node describes an action to be performed by the simulated entity running that behavior. The entity executes only a single node in the BTN at a time; this node is designated the *current node*. The current node changes when the conditions attached to one of its outgoing transitions become satisfied. Hence, a BTN essentially describes a sequence of actions and decisions that define how an entity will act in the simulation.

The set of actions available for use by a behavior author is determined by the nature of the simulation for which the behavior is being created. For example, a tactical MOUT simulation for individual soldiers is not likely to have actions related to controlling radar systems or firing torpedoes. In addition, actions will not be available for real-world capabilities that are simply not modeled in the simulation (e.g., fatigue or illness). Note that actions need not always be physical: an action can also perform a perceptual or mental task.

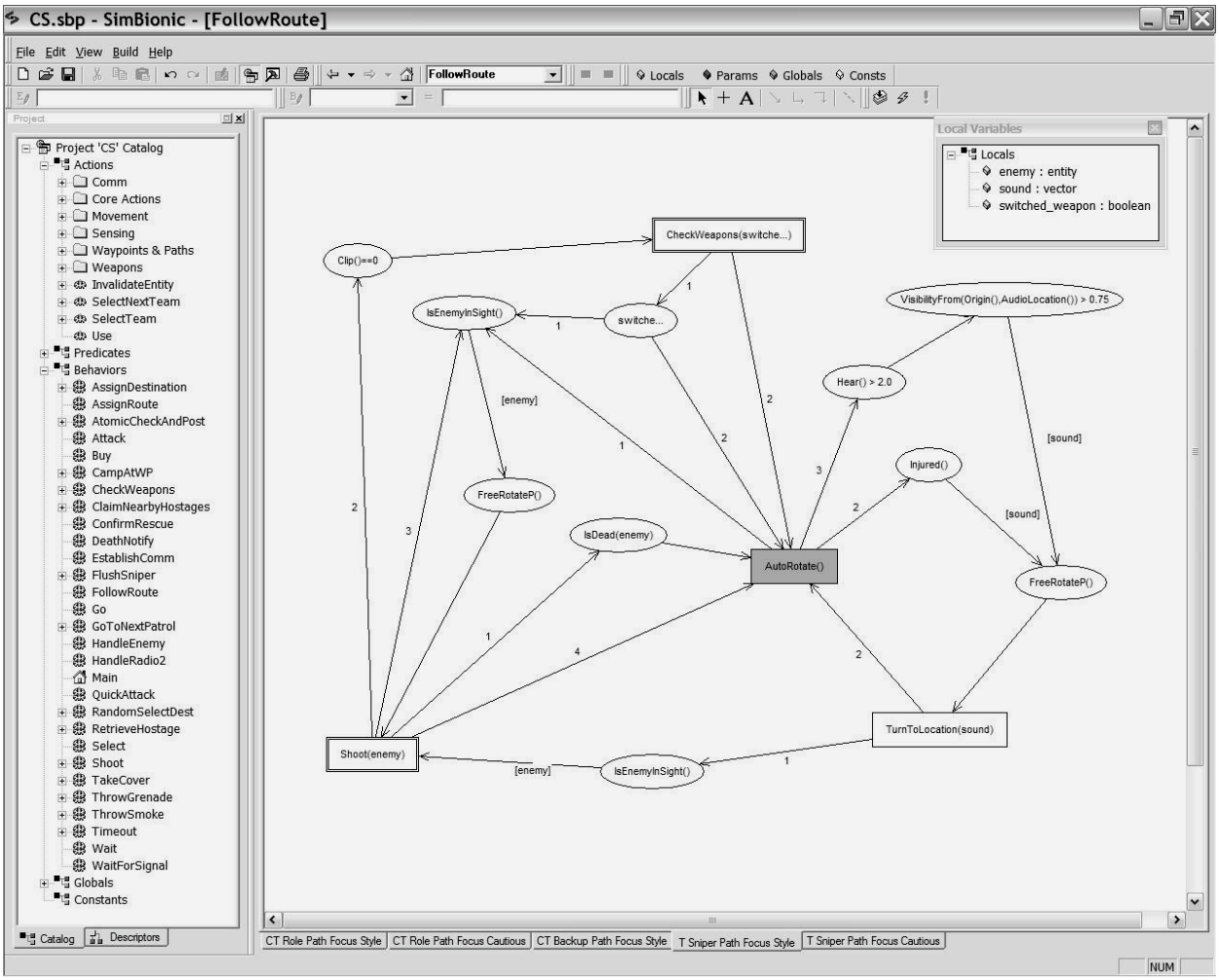

Figure 2. Behavior Editor Screenshot.

Figure 1 shows an example of a behavior created for Counter-Strike. Nodes are shown as rectangles, and transitions are depicted as chains of ovals and arrows. The text labels indicate the action or condition associated with each node or transition, respectively. The numbers in circles next to transitions indicate the order in which the outgoing transitions from a given node are checked. This BTN 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 and attempt to kill him; if injured, the entity will take cover.

While the basic structure of behavior transition networks is similar to that of finite-state machines, BTNs include a number of extensions to the standard finite-state machine model. For instance, BTNs can store information in local variables, enabling transition conditions to refer both to the current and previous state of an entity. The representation also provides mechanisms for inter-entity communication between BTNs, allowing for coordinated team behaviors. In addition, BTNs can be hierarchical and polymorphic; this aspect will be described in more detail in the section on object-oriented authoring.

VISUAL AUTHORING

Constructing behaviors using the BTN representation is done entirely via a graphical editing tool (see Figure 2). There is no scripting or programming language involved at any level. The objective is to provide a "canvas" on which a SME can intuitively and rapidly sketch out a behavior. Once an initial behavior has been roughed out, the author can then iteratively refine it until it matches his mental model of how a given simulated entity should act. The graphical representation allows SMEs to see a behavior's logic at a glance, and quickly spot obvious flaws, bugs or other difficulties that might be more difficult to find in a textually- or code-defined behavior.

This process is facilitated by the consistent and pervasive use of standard direct-manipulation user interface idioms such as drag-and-drop and right-click context menus. For instance, the user can assemble a behavior by simply dragging the desired actions and conditions from the left-hand catalog pane onto the right-hand "behavior canvas" and then drawing the necessary connections between them.

The Runtime Engine

While the graphical editor permits users to create and manipulate behaviors, the resulting BTNs are merely static specifications. The accompanying runtime engine, however, can use these behavior specifications to control CGF within a simulation (see Figure 3). A thin C++ interface must be implemented to integrate the runtime engine with the simulation.

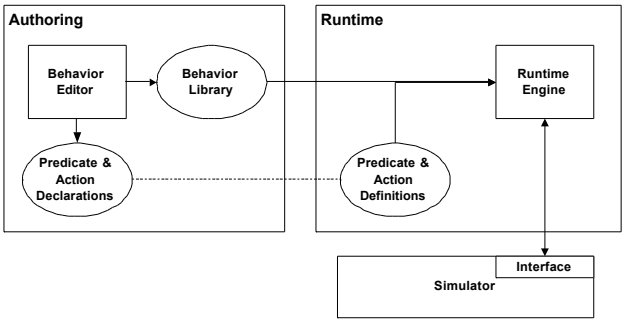

Figure 3. Components of Authoring System

The combination of the editor and runtime engine permit the SME to edit a behavior and then immediately see the effect of the change in the simulation. This can reduce the time required to fine-tune behaviors so that they perform exactly as desired. A built-in interactive debugging system provides additional assistance with finding and fixing problems.

OBJECT-ORIENTED BEHAVIORS

As the behaviors developed for a given simulation grow in number and increase in complexity, there is likely to be a substantial amount of functional duplication among them. Not only is this a waste of development effort, but it also generally produces hard-to-maintain behaviors, since a change to one behavior may need to be manually replicated across many others. To combat this problem, the researchers borrowed the principle of decomposability from object-oriented programming. The result was the notion of *hierarchical BTNs*, in which a node can refer to another BTN instead of a simple action. By allowing BTNs to be nested in this manner, a complex behavior can be broken up into an assembly of many smaller sub-behaviors.

This decomposition yields simpler and more readable BTNs. More importantly, it permits functional units to be re-used across multiple behaviors without duplication. When a sub-behavior needs modification, the author need only change it in a single location in order to have that change automatically affect all of the behaviors that invoke the sub-behavior. A third benefit is the ability to take a top-down authoring approach, starting with the highest-level, most abstract behavior and gradually adding more and more detailed sub-behaviors. Such an approach is particularly useful when the low-level details of a behavior have not yet been worked out, but the general outline is well-understood.

Hierarchical BTNs rely on a stack-based execution model, where an entity's initial behavior is at the bottom of the stack. Each time a behavior node invokes another behavior, a new level is pushed on top of the stack containing the newly-invoked BTN. When a hierarchical BTN finishes execution, it is popped from the stack. Every BTN on the stack maintains its own current node, but only the BTN at the topmost level is executed.

For example, in the Counter-Strike testbed, the hostage-rescue CGF have a high-level behavior *RescueHostages* that describes the entire plan for rescuing the hostages: finding the captors' headquarters, breaking into the headquarters, freeing the hostages, and escorting them to safety (see Figure 4). Each of these sub-goals has a corresponding sub-behavior, which may in turn have its own sub-sub-behaviors.

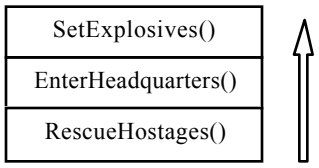

Figure 4. Example of Behavior Stack.

Note that although only the topmost BTN on the stack is actually executed, outgoing transition conditions are checked for every current node in every BTN on the stack, starting with the bottom. If a condition is satisfied in a BTN below the top of the stack (say *EnterHeadquarters*), all of the BTNs above it are discarded, and execution continues with the newly topmost BTN. This transition mechanism permits a kind of prioritization among behaviors whereby a high-level behavior such as *RescueHostages* can effectively override its sub-behaviors if an important situation arises (for example, a sudden enemy attack).

For cases where a high-level behavior needs to *temporarily* interrupt the performance of a lower-level sub-behavior, the author may designate a transition as

an *interrupt transition*. When such a transition is traversed, it does not cause the BTNs above it on the stack to be discarded but instead simply pushes the newly-invoked behavior on top of the stack, regardless of where on the stack the invoking node lies. Once the interrupting behavior has terminated, it is popped from the stack and execution resumes with the previous topmost behavior. In Counter-Strike, for example, a soldier may be following a route to the hostage location when he suddenly receives enemy fire. An interrupt transition allows him to take cover and return fire until the enemy is neutralized, at which point he can pick up where he left off along his route.

Behavioral Polymorphism

As the behavior library grows, it often becomes desirable to create behaviors that differ only slightly from existing behaviors. 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 changes in lower-level behaviors. For example, a user may decide to model the morale and fatigue of an opposing force and have those attributes affect behavior. Thus, when the force is in conflict with friendly forces, the *CombatPatrol()* behavior would dispatch a specialized version of a behavior based on, say, low morale and high fatigue. The invoked behavior would be named something along the lines of "*Combat_LowMorale_HighFatigue()*." Most likely, this behavior's sub-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 handle the growth of the behavior library while at the same time simplifying the construction of specialized behaviors, the representation was extended with the concept of polymorphism from object-oriented programming. In this extended representation, a single behavior can now possess 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 the possible values shown in these two trees:

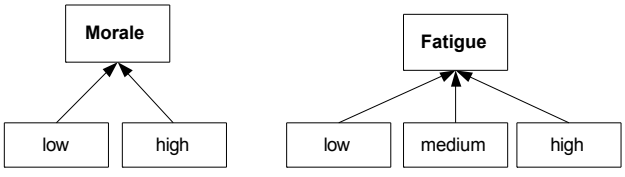

Figure 5. Polymorphic Descriptor Hierarchies

A user specializes, or indexes, a behavior graph by associating it with exactly one node per tree. In this example, there are twelve possible specializations.

Each entity possesses a set of descriptors as well. In the case of the opposing 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 indices. For example, if there is a behavior version of *CombatPatrol()* indexed with "low" morale and "high" fatigue, then that version will be selected for the opposing force. Note that if no more specific match can be found, the "default" behavior indexed by the root of the descriptor tree (e.g., "Morale") will be selected.

Although here a total of twelve behavior specializations may be defined (counting the roots), in practice not all of these will actually be used. The descriptor tree affords the ability to selectively customize behavior through the structured tree hierarchies. In the above example, if a user wants to define only one version of the *CombatPatrol()* behavior, it would be indexed using the two roots. The opposing force would use this version of the behavior because a more specific version cannot be found. If the user wants to define a special case relevant only when morale is low, then he indexes the behavior by picking "low" from the first tree, and the root for the second. The opposing force would then use this version instead.

Entities may change their descriptors at any time. This change affects all behavior invocations from that point on. For example, an opposing force that switches its morale from low to high and its fatigue from high to medium would select a different version of the *CombatPatrol()* behavior, and hence would perform differently in the simulation. Changes to an entity's descriptors do not, however, affect any behavior that that entity might already be executing.

AUTHORING EXAMPLE

While the visual behavior authoring methodology described in this paper had been applied to a variety of simulation applications, the set of behaviors developed for the Counter-Strike testbed was substantially larger and more complex than any the researchers had previously authored. At the same time, the domain was not well understood, which made it difficult to completely specify in advance the full range of behavioral capabilities that would be needed to create competent automated players. As a result, a highly iterative and incremental approach to authoring was taken.

The researchers began by sketching out a set of two or three very high-level behaviors that would serve as an outline for the entities' behavior. These behaviors contained no concrete actions themselves, but were instead composed of slightly lower-level behaviors whose details we had not yet defined. Once this top-level skeleton was roughly complete, the process was repeated at the next lower level, and this top-down decomposition was recursively continued until the behaviors were fleshed out to the level of concrete actions. At this point, it was possible to start testing the bots within the game environment and making refinements to the behaviors.

During the process of authoring a first draft of the behaviors, it was found that the initial vocabulary of actions that had been defined was insufficient. This vocabulary was based on the primitive interactions that were naturally suggested by the human player's interface to the game – jump, turn, shoot, reload, etc. – rather than any anticipation of the concrete actions required to implement the target behaviors. After a first pass through the authoring process, the list of actions was therefore revised and expanded considerably. In most cases, this was simply to add new capabilities to the bots, but sometimes actions were eliminated or even broken into several finer-grained actions.

The authoring process up to this point had essentially produced behaviors for two distinct entities, one rescuing soldier and one guarding soldier. To introduce more variation on the teams, the basic set of behaviors was extended polymorphically using "Team," "Role," and "Attack Style" descriptors (among others). This approach made it easy to add new varieties of bots simply by specializing one or two behaviors. This phase of the authoring process can be thought of as a lateral expansion or broadening of the behavior set, as contrasted with the top-down authoring phase, which is focused on completing the chain from abstract behaviors to concrete in-game actions.

EVALUATION

This approach has been validated with usability studies conducted in previous research. In a project conducted for the Navy (Stottler and Vinkavich, 2000), the researchers adapted the technology to provide Navy instructors with a tool for creating intelligent agent-based behaviors for use in a tactical 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.

An informal study was also recently performed in which a version of the BTN graphical editor customized for the popular computer game Neverwinter Nights™ was made available on the Web (Neverwinter Nights, 2002). Neverwinter Nights™ features a C-like scripting language that knowledgeable players can use to create their own game content. The modified editor was intended to make scripting possible for players with little or no programming experience. The researchers collected feedback from over a dozen users, including samples of scripts developed using our tool. This feedback indicated that users with no knowledge of C programming were quickly able to learn to use the tool to create complicated scripts that would have otherwise been beyond their means.

In addition, use of the behavior editor on in-house simulation projects has enabled the researchers to reduce the time required to define complex finite state machine logic by as much as seventy percent compared to standard code-based implementations. More significantly, once the FSMs had been created in the visual tool, modifications to their logic required approximately ten percent of the time that would have been needed to make similar changes in code. This indicates that even for programmers, the use of visual authoring environments can result in substantial time savings.

RELATED WORK

The notion of having a visual representation or description of behavior is not new. One important issue to distinguish is the use of a visual representation for communication purposes versus implementation. The use of graphs to communicate some design or behavior is pervasive.

While much of our early runtime architecture work was based on AI robotic literature (Loyall and Bates, 1991; Firby, 1987; Georgeff and Lanksy, 1987), the visual representations AI researchers have employed in their articles are overwhelmingly graph-based (some examples are Tate, 1977 and Sacerdoti, 1977), which has influenced the way we portray behavior.

UML state charts are the most well-recognized standard for formally describing the states in which a software object can be (Fowler, 2000).

A number of commercially available object-oriented analysis and design tools, such as Rational Rose and Together, offer a visual interface for the creation of UML state chart diagrams. These tools, however, were never intended to execute the actual state charts created by the user. This confines their applicability to requirements and design specification.

For actual visual-to-implementation work, there has been past work in the military simulation field, perhaps starting with ModSAF (Calder et al, 1993). Von der Lippe et al. (2000) describe the CBT project which employs a similar visual representation, but focused on command and control for teams of entities. Thus, the behavior definition is of a composite behavior. Specialization of behavior happens through "behavior roles" so that a set of entities may be participating in the same mission, each with its own role in the simulation.

In the robotics field, MacKenzie et al. (1997) describe the MISSIONLAB system that allows an end user to specify the behavior of multiple robots. The user does this visually using hierarchical state and transition links.

CONCLUSION

This paper has presented a lightweight visual approach to authoring behaviors for computer-generated forces. This approach has the potential to put behavior authoring capability in the hands of subject matter experts who lack the programming skill necessary to use existing simulation behavior systems. Of course, very complex behaviors, especially those that are largely mental or abstract in nature, are not so easily captured by procedural representations, and in such cases the additional modeling infrastructure furnished by cognitive architectures is called for. Further work must be done to determine when each technique is most appropriate.

ACKNOWLEDGEMENTS

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

REFERENCES

Calder, R.B.; Smith, J.E.; Courtemanche, A.J.; Mar, J.M.F.; and Ceranowicz, A.Z. (1993). "ModSAF Behavior Simulation and Control." Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation.

Counter-Strike, (2003). Counter-Strike Mod Official Website. Retrieved May 15, 2003, from http://www.counter-strike.net/.

Firby, R.J. (1987). "An Investigation into Reactive Planning in Complex Domains." Proceedings of AAAI.

Fowler, M. and Scott, K. (2000). UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison Wesley.

Georgeff, M. and Lansky, A. (1987). "Reactive Reasoning and Planning." Proceedings of AAAI.

Loyall, B. and Bates, J. (1991). "Hap: A Reactive, Adaptive Architecture for Agents." CMU Tech Report CMU-CS-91-147.

MacKenzie, D., Arkin, R.C., and Cameron, J. (1997). "Multiagent Mission Specification and Execution." Autonomous Robots, 4(1), 29-57.

Neverwinter Nights, (2002). Neverwinter Nights Official Community Website. Retrieved June 5, 2003, from http://nwn.bioware.com/.

Sacerdoti, E.D. (1977). A Structure for Plans and Behavior. American Elsevier, New York.

Stottler, R. H. and Vinkavich M. (2000). "Tactical Action Officer Intelligent Tutoring System (TAO ITS)." Proceedings of I/ITSEC 2000.

Tate, A. (1977). "Generating Project Networks." IJCAI.

Von Der Lippe, S., McCormack, J. S., and Kalphat, M. (2000). "Embracing Temporal Relations and Command and Control in Composable Behavior Technologies." Proceedings of the Ninth Conference on Computer Generated Forces and Behavioral Representation.