A Visual Environment for Rapid Behavior Definition

Daniel Fu
Ryan Houlette
Randy Jensen
Stottler Henke Associates, Inc.
1660 S. Amphlett Blvd., Suite 350
San Mateo, CA 94402
650-655-7242
fu@stottlerhenke.com, houlette@stottlerhenke.com, jensen@stottlerhenke.com

Keywords:
Behavior visualization, polymorphism

ABSTRACT: We describe a visual framework that simplifies authoring of simulated behavior. This framework consists of a canvas depicting behavior as a finite state machine (FSM) graph, a palette of geometric objects and glyphs, and a dictionary of actions and predicates. The user defines a basic vocabulary of actions and predicates which appear as textual and geometric shapes on the canvas. The actions correspond to states in an FSM. Predicates are used to determine valid transitions between states. The basic model is extended in two major ways. First, behaviors are hierarchical in that they may invoke each other. Second, each behavior may have a number of specializations indexed through a descriptor hierarchy. These two extensions serve to encapsulate functionality, and to selectively specialize behavior whenever necessary without arduous re-modification of existing behavior.

1. Introduction

In this paper we describe a visual approach for simplifying the formalization and implementation process involved in developing an executable behavior model for a given domain. Even in domains where modeled entities require little or no demonstrable cognitive capabilities, there is still a significant challenge in the task of behavior codification. A subject matter expert may have a strong understanding of a given domain and a clear notion of what the behaviors should be and how they should operate, but still face a bottleneck at the implementation step. This bottleneck is often a result of the gaps in domain familiarity and functional vocabulary between subject matter experts and programmers. As a result, relatively simple behaviors can be difficult to implement and test in the targeted simulation or control environment.

Complex behavior in domains with complex cognitive requirements may be best implemented with tools that provide native support for many of the artifacts of human reasoning, decision-making, knowledge representation, and behavior in general. However, simple behaviors should be simple to implement, and this paper describes an approach for accomplishing this with a methodology supported by software. Still, an important design consideration with this approach is the objective of supporting a scalable hierarchical development model. Scalability is important so that even behaviors that are initially implemented as simple models can be arbitrarily enhanced and extended as the subject matter expert may see fit, without requiring complete code overhauls.

We will give an overview of the approach and the main functional mechanisms involved. This paper assumes a basic familiarity with the traditional finite state machine (FSM) model which has historically been applied in many instances for the control of autonomous entities. Therefore we will limit the discussion of FSM concepts to the context of how our visual authoring paradigm and execution model represents extensions to the basic FSM. Although there is considerable precedent for an FSM-based behavior definition approach, this is often in the context of coded implementation as opposed to visual construction in an environment that can be intuitive for different kinds of users. This is the key innovation that this paper seeks to highlight, along with an architecture that allows for easy integration of behaviors constructed in this way. We will provide a simple example from a specific simulation domain to illustrate the differences between implementation using traditional programming and implementation using the visual methods prescribed by our approach.

This will be followed with a discussion of the support for hierarchical behaviors and polymorphism in this methodology, to facilitate easy authoring of more complex and adaptive behavior. Although the approach we describe incorporates these various extensions to the basic FSM model in order to increase the power and applicability of the visual toolset, it was an intentional design decision to *not* frontload the authoring environment with features targeted toward specific tasks such as cognitive modeling or knowledge representation. As mentioned earlier, there are other existing tools that already provide extensive feature-sets for constructing complex behaviors based on more complex human modeling. Our visual framework targets a spectrum of users, including programmers and non-programmers as well as subject matter experts, who may be familiar with a given domain but not with the methods of cognitive psychology.

2. Overview

In our methodology, behaviors for simulation entities are articulated in terms of four basic constructs: *actions*, which define all the different actions an entity can perform; *other behaviors* previously created (any behavior can invoke any other behavior, which allows for the construction of a library of behaviors that can lead to increased efficiency in the form of easy reuse); *conditions*, which specify the circumstances under which each action and behavior will happen; and *connectors*, which control the order in which conditions are evaluated and actions and behaviors take place. These four constructs allow subject matter experts to create AI behavior that mimics the behavior experienced or observed in the real world.

Programming code is essentially created by "drawing" it as flow-chart-like diagrams in an editing window. Specifically, actions are represented as rectangles, behaviors as boldfaced rectangles, conditions as ovals, and connectors as lines. Each element accommodates as many variable assignments, complex expressions, and explanatory comments as desired.

This intuitive visual approach allows subject matter experts to see a behavior's logic at a glance, and quickly spot potential flaws, bugs or other difficulties. A visual flowcharting paradigm is consistent with design and specification methods naturally used in many domains, and helps to enforce a structured approach to developing the preliminary formalization that experts normally sketch out in the behavior modeling process.

Behaviors created visually are then interfaced with a simulation or control software by a runtime engine. *Figure 1* shows the high level architecture for this approach.

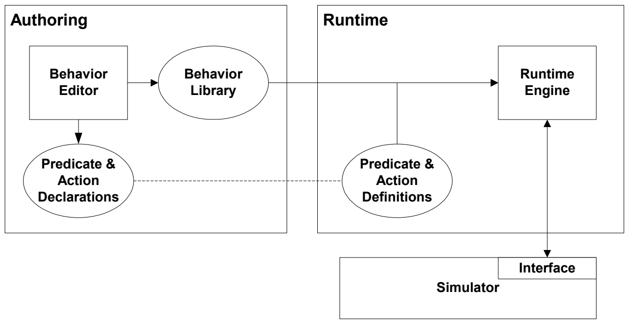


Figure 1: High level architecture.

The authoring component generates a behavior library for a simulation entity; the runtime component contains the engine which controls entities within the simulation. The *Behavior Editor* provides the visual environment for building behaviors from a basic vocabulary of predicates and actions. These may include primitives that are available or easily implemented from the simulation API. The runtime component references the *Behavior Library* to direct entities in the *Simulation* via an *Interface* module that provides a common communication layer with the *Simulation* API.

3. Example Problem

For the purposes of an example, we will refer to behaviors developed for a simulation of individual combatants in a first-person 3D environment, which is intended to be used as a trainer for coordination among team members in an urban combat setting. As such, the combat behaviors for the simulated opponents need not exhibit advanced cognitive modeling, but they need to meet a minimum baseline level of realism.

3.1 Visual behavior representation

The methodology we present for visual behavior representation utilizes the concept of behavior transition networks (BTNs), which are generalizations of finite state machines (FSMs). BTNs have current states and transitions like finite state machines, but also hierarchically decompose, can have variables, communicate to each other through a blackboard, and can execute arbitrary perceptual or action-oriented code. A large number can run in parallel.

Figure 2 below shows a sample BTN containing actions, conditions, and connections. The visual artifacts (rectangles, ovals, and connectors) are the same as those used in standard flowcharting, and also map directly to the visual elements provided to subject matter experts making use of our approach.

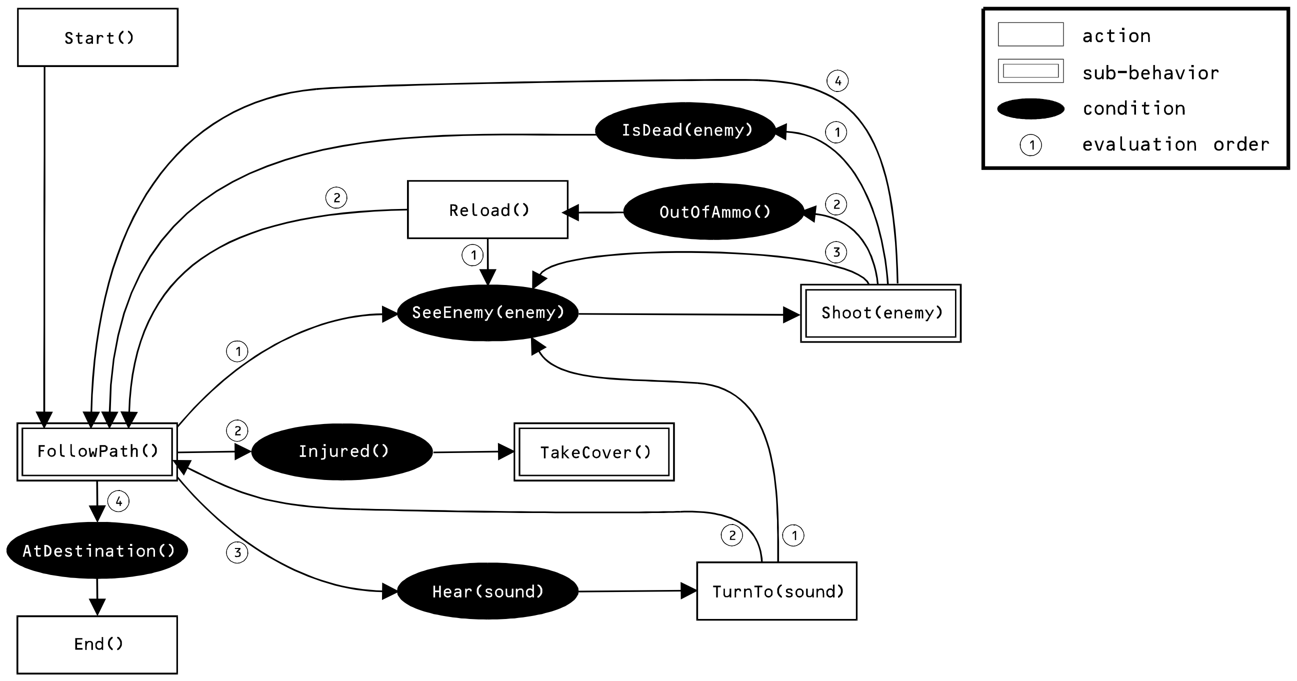


Figure 2: Visual representation of CombatPatrol behavior.

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.

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 seeks 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 BTN 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 involves an assessment of the source of the threat which just caused an injury to the current entity, as well as an analysis of the surrounding physical environment in the simulation in order to find useful cover. But these are independent activities that lend themselves well to abstraction, so that they may be applied elsewhere 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 for a new reader to understand.

3.2 Traditional programming implementation

The C code below shows an example of implementing the same behavior in a standard programming language.

```
void CombatPatrol()
{
   do
   {
      FollowPath();
      Entity enemy;
      SoundEvent sound;
      if ((enemy=SeeEnemy()) != NULL)
         KillEnemy(enemy);
      else if (Injured())
         TakeCover();
      else if (Hear(sound))
      {
         TurnTo(sound);
         if ((enemy=SeeEnemy()) != NULL)
            KillEnemy(enemy);
      }
   }
   while (!AtDestination());
}

void KillEnemy(Entity enemy)
{
   do
   {
      Shoot(enemy);
      if (IsDead(enemy))
         break;
      if (OutOfAmmo())
         Reload();
   }
   while ((enemy=SeeEnemy()) != NULL);
}
```

Examining this C-code implementation of the behavior, we see that its interpretation demands a very different set of mental skills than does its visual counterpart – skills which are typically gained through programming experience. This is partly due to the fact that while code is by its nature essentially linear, proceeding from statement to statement one step at a time, behaviors often are not. In particular, the linear quality of code obfuscates the branching structures that are characteristic of making a decision between several possible actions (consider the *FollowPath()* rectangle in *Figure 2*).

Another difficulty with code-based behavior representations is that they make no ready distinction between the control elements that direct execution flow and the commands that invoke entity actions in the simulated world. Both are represented by textual keywords. As a result, the reader is forced to parse each line of code to determine whether it directly affects the entity's behavior in the simulation (by invoking a hypothetical *FireWeapon()* action, for example) or is involved in controlling execution flow. The visual representation in Figure 2, on the other hand, denotes entity actions clearly with rectangles, which are instantly graphically distinguishable from the ovals and lines that make up the control framework.

Of course, on top of these semantic difficulties, code also poses a syntactic challenge to the nonprogrammer. Comprehending a hard-coded behavior requires knowledge of the keywords, symbology, and grammar that make up the particular implementation language; without at least minimal linguistic familiarity, interpreting behaviors written in code can be challenging. For a subject matter expert whose expertise lies in the complexities of a domain and the human behaviors common to that domain, as opposed to the constructs and syntax of computer science methods, this challenge often represents a frustrating barrier to effective behavior modeling.

4. Extended Functionality

4.1 Hierachical behavior structure

The visual behavior representation we have described permits the construction of arbitrarily intricate sequences of actions and decisions. As with most complex systems, however, it is generally good - for reasons of both maintainability and understandability - to break large behaviors down into smaller, more easily digestible subcomponents. For this reason, our representation supports a hierarchical behavior model wherein each behavior is free to invoke other behaviors in the library just as it would invoke an action.

By taking advantage of this capability, an author can decompose an overly-complex behavior into a few high-level behaviors, each of which encapsulates some distinct and functionally consistent portion of the original behavior. The result is a set of nested behaviors that is much easier to understand and modify.

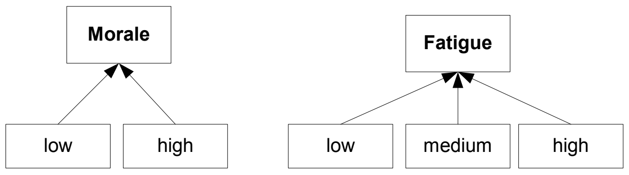
Hierarchical behaviors have other advantages as well. By permitting authors to break behaviors down into their logical functional components, hierarchicalism promotes reuse rather than reinvention. Once a behavior has been added to the behavior library, it is henceforth available as a ready-made building block for other, future behaviors. And since each particular bit of functionality need only be implemented once in the library, sweeping modifications to a simulated entity's behavior can be made by editing a single low-level behavior (effectively propagating to all higher-level behaviors that invoke it).

The hierarchical structure is also effective for embedding a form of goal representation into behavior models. During execution, the implicit priorities of a modeled entity are reflected in the level of the call stack at which a given execution condition or simulation state is evaluated. When a condition in a high level BTN is satisfied, the call stack containing lower level behaviors can be interrupted temporarily or permanently depending on how the author has defined the high level behavior [1]. This is a powerful mechanism which does not have a direct counterpart in most non-visual programming environments.

4.2 Polymorphism

While behaviors provide modular building blocks out of which which users can construct new, more complex behaviors, their long-term use introduces a proliferation of similar behaviors, often with 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 necessary 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 then dispatch a specialized version of a behavior based on, say, low morale and high fatigue. The invoked behavior, then, would be named "*Combat_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 handle the growth of the behavior library while at the same time simplifying the construction of 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 the possible values shown in these two trees:



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, 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.

Note that these trees may be of arbitrary height and mirror the 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 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.

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

5. User Feedback

This approach has been validated with usability studies we have conducted in previous research. 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.

We also recently performed an informal study in which we made available on the Web a free version of our authoring software customized for the popular computer game Neverwinter Nights™. Neverwinter Nights™ features a C-like scripting language that knowledgeable players can use to create their own game content. Our tool was intended to make scripting possible for players with little or no programming experience. We 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, our own use of the authoring tool on in-house simulation projects has enabled us 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.

6. Related Work

The notion of having a visual representation of AI behavior is not new. Von der Lippe et al. [3] 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.

MacKenzie et al. [4] 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.

UML state charts are a well-recognized standard for formally describing finite state machines. A number of commercially-available object-oriented analysis and design (OOA&D) tools, such as Rational Rose and Together, offer a visual interface for the creation of UML state chart diagrams. These tools do not, however, provide the capability to actually execute state charts created by the user, which limits their applicability to requirements and design specification.

7. References

[1] Fu, D., Houlette, R., and Bascara, O. "An Authoring Toolkit for Simulation Entities," in Proceedings of I/ITSEC 2001.

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

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

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

8. Acknowledgements

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

Author Biographies

DANIEL FU is a project manager and software engineer at Stottler Henke Associates, Inc. His research interests are in Artificial Intelligence (AI) autonomous agents and planning. While at Stottler Henke, he has applied AI techniques to a number of intelligent tutoring systems and autonomous agents projects. Dan holds a Ph.D. in computer science from the University of Chicago.

RYAN HOULETTE is a project manager and software engineer at Stottler Henke. His primary interests lie in the areas of intelligent interfaces, autonomous agents, and interactive narrative. Mr. Houlette is currently leading a project to develop a mixed-initiative scheduling system that will include as a core component a rich capacity for human interaction and collaboration. He holds an M.S. in computer science from Stanford University.

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. Mr. Jensen also participated in authoring autonomous "bot" behaviors for a multiplayer game environment. He holds a B.S. in symbolic systems from Stanford University.