Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

SPECIFYING THE BEHAVIOR OF COMPUTER-GENERATED FORCES WITHOUT PROGRAMMING

Daniel Fu
Randy Jensen

Stottler Henke Associates, Inc.
1660 S. Amphlett Blvd.
Suite 350
San Mateo, CA 94402, U.S.A.

Ryan Houlette

Stottler Henke Associates, Inc.
280 Broadway
Arlington, MA 02474, U.S.A.

ABSTRACT

The behavior of automated forces in military training simulations is frequently hard-coded by a software developer after conferring with subject matter experts. These experts do not directly participate in the development process, instead relying on the developer to correctly implement appropriate behavior. This dependency can result in increased simulation development time and cost.

We present a visual behavior representation and accompanying authoring tool that is meant to accelerate the development process by enabling experts to participate in development, while not hindering software developer productivity. An author using this tool constructs behaviors by assembling flowchart-like diagrams from a set of building blocks. The resulting behaviors can be directly executed in a simulation using a simplified yet powerful computational model, also described in this paper.

We also discuss the application of this visual behavior representation to the creation of automated players for the Counter-Strike computer game.

1 INTRODUCTION

Traditionally, the implementation of automated forces behavior has resided in the domain of the software developer. Recognizing the potential for subject matter experts to participate in the development process, we have investigated ways in which the representation of behavior could be accessible to non-technical personnel. The result is an alternate approach to specifying simulation behaviors: a visual behavior representation – embodied in a software editor and runtime engine – that opens the authoring process to non-programmers.

In this paper, we discuss our visual representation and the corresponding computational model that operationalizes behavior. The visual representation consists of visual building blocks that an author assembles using standard direct-manipulation idioms. These blocks are modular such that they (1) can be decomposed into simpler, more easily comprehensible subcomponents, and (2) can be reused repeatedly for easier authoring.

An authoring and runtime innovation we describe here takes the notion of functional polymorphism from object-oriented programming and applies it to simulation behavior. This form of "behavioral polymorphism" enables the author to define multiple versions of the same behavior, where each version is associated with particular attributes of a simulation entity. During a simulation run, the appropriate version of the behavior will dynamically be selected and invoked based on the current state of the entity. Polymorphism streamlines the authoring process when faced with different types of entities that share high-level behaviors.

We discuss the visual representation and computational model in the context of a team-level implementation of behavior for Counter-Strike, a multiplayer "first-person shooter" computer game.

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 behavior experienced or observed in the real world.

Behaviors for simulation entities are essentially created by "drawing" them as flowchart-like diagrams in an editing window. This intuitive visual approach allows subject matter experts to see a behavior's logic at a glance, and

quickly spot potential flaws, logical errors, 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 by a runtime engine. Figure 1 shows the high level architecture for this approach.

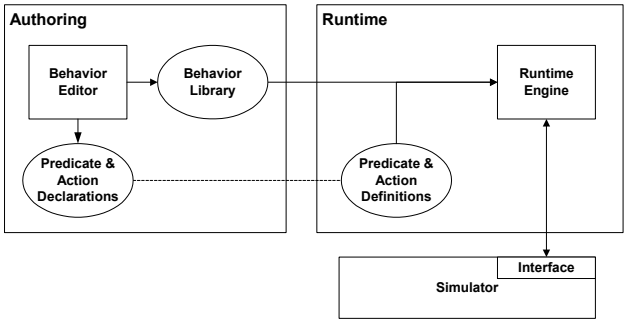

Figure 1: High-Level Architecture

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 BEHAVIOR REPRESENTATION

The visual behavior representation we have presented is based upon a generalization of finite state machines (FSMs) that we call behavior transition networks (BTNs). BTNs have current states and transitions like finite state machines, but also hierarchically decompose, can have variables, 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, behaviors, and connectors. 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.

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. 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 environ-

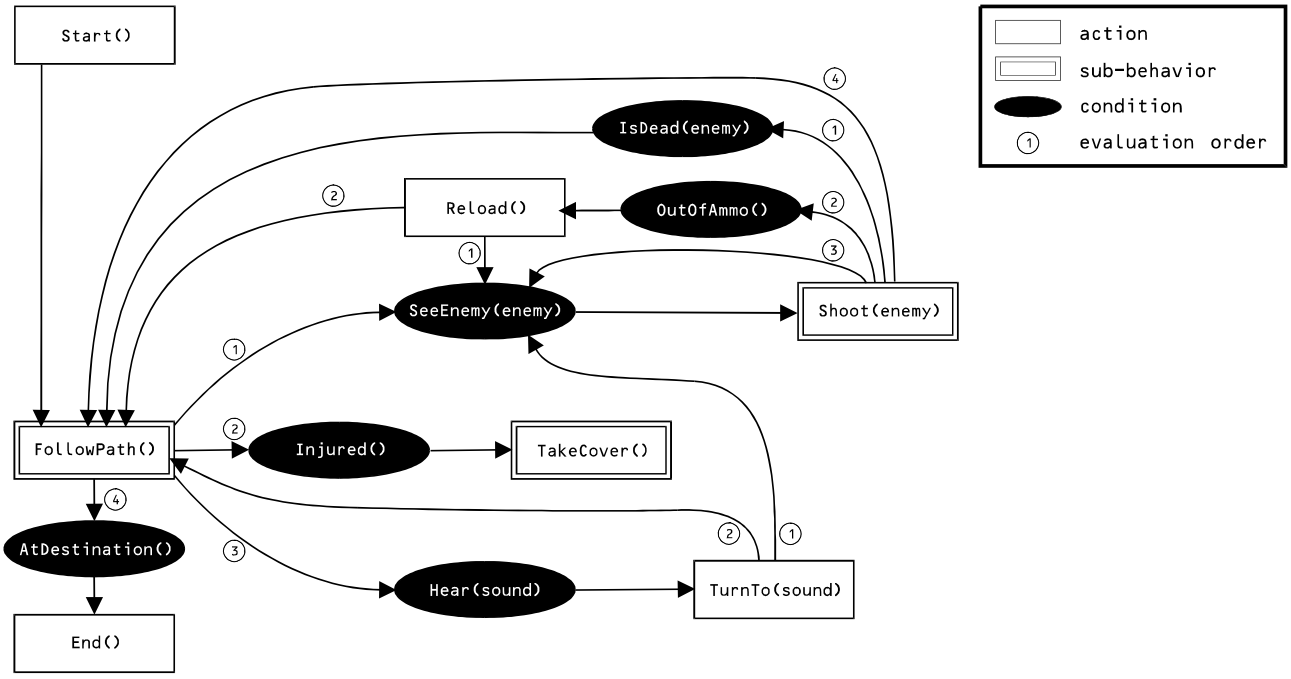

Figure 2: Sample BTN

ment 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.

3.1 Hierarchical Behaviors

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 stack-based hierarchical behavior model wherein each behavior is free to invoke other behaviors in the library just as it would invoke an action. When this occurs, the invoked BTN is instantiated and run as a fully independent behavior, with its own data separate from the invoking BTN. Each simulation entity stores its chain of currently-instantiated BTNs in a stack, where only the topmost BTN is "active" (that is, controlling the entity's actions). When the topmost BTN reaches a final state, it is popped from the stack, and the new topmost BTN becomes active.

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, hierarchical decomposition 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).

3.2 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 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 Figure 3.

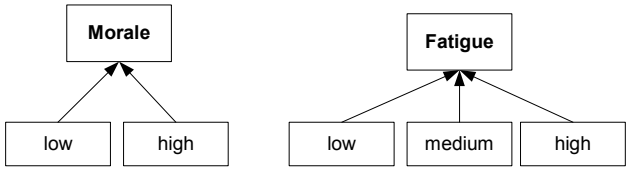

Figure 3: Polymorphic Descriptor Hierarchies

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 (including the roots of the trees).

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.

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.

3.3 Inter-Entity Communication

Most non-trivial simulations model the interaction of multiple entities, and coordination between those entities is often an important component of the model. Facilities are thus provided in our behavior representation to allow communication between entities. Two types of communication – *message-passing* and *blackboards* – are supported via a set of built-in actions and predicates.

The message-passing system uses a straightforward subscription-based addressing scheme. An entity may subscribe to any number of pre-existing groups and may also create and destroy groups at will. Messages, which consist of a message type identifier and an optional data value, can be sent to any group. Entities maintain individual message queues, which they can poll on demand. Message-passing is intended as a lightweight mechanism for synchronizing behavior between multiple entities as well as for representing hierarchical command structures among entities.

The blackboard system, by contrast, provides a shared-memory framework that entities can use to store and exchange team knowledge. A blackboard is essentially a named collection of key-value pairs that has global visibility. Entities can post values to and read values from any key on any blackboard. Entities can also create and destroy blackboards as desired. Typically, blackboards would be useful in simulations where groups of entities need to share a common situational picture, or where entities need to publish information without knowing the identity of the recipients.

4 COUNTER-STRIKE

As an empirical trial of our behavior representation methodology, we 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 game Half-Life (Counter-Strike 2003). Counter-Strike depicts a hostage-rescue scenario where 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.

We chose Counter-Strike for our trial 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 our 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 our runtime engine to the game.

For our Counter-Strike bots, we 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 us to focus on authoring realistic behaviors rather than on low-level implementation details. Once the interface was complete, we were able 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.

4.1 Navigation and Pathfinding

Navigation is a fundamental challenge for any computer-controlled entity in a complex world, and it is particularly difficult in a continuous 3D environment such as that found in Counter-Strike. Although the low-level sensory perception and movement control necessary for navigation pose interesting problems for the behavior author, these issues were not the focus of our effort, which was targeted at higher-level, more tactically significant entity behaviors. We therefore implemented a navigation layer in C++ that handles the low-level control of entity movement. A set of special actions and predicates allow the high-level visual behaviors to give high-level directions to the navigation layer (for example, "go to destination waypoint X33").

The inclusion of the navigation layer enabled us to further abstract our behaviors away from the specific implementation details of the game environment, leading to behaviors that were more concise and easier to author. In addition, the navigation layer allowed us to avoid embedding a pervasive parallelism into the entity behaviors, which would have been otherwise necessary to deal with the fact that in Counter-Strike, players often simultaneously move while taking other actions. While such parallelism is possible in our current representation, it is considerably more difficult to author and debug than normal sequential behaviors.

4.2 Reacting to Possible Events

The behaviors developed for Counter-Strike include a variety of reactive mechanisms that make use of the stack-based nature of the behavior representation. With the visual paradigm, this often results in a "flower" behavior structure, by which a bot has a central state or activity which is interrupted based on a variety of situational inputs in the virtual environment. For example, in a navigation

behavior for a "rescue" bot, the central node is a path-following action. While performing this task, if the bot receives an enemy-inflicted bullet wound, the path-following action is interrupted to follow one of the flower "petals" branching out from the central node in the BTN. Petal segments contain the predefined behaviors for reactions, such as seeking cover or returning fire when injured. When it is safe, the bot then returns to the path following action by returning to the central node in the BTN. Similarly, a "sniper" bot may have a sentry action as the central node, which is interrupted when a target comes into view, and resumed after the target is handled.

The "flower" BTN structure is helpful for the behavior author, because it affords a simple visual representation for the kinds of events that are significant in a given behavior. If there are five kinds of key events that a bot should react to while performing a task such as navigation, then it's very easy to see all five depicted with relative priorities in one BTN. It is non-trivial to implement the same reactive functionality in a traditional procedural programming environment, and such an approach also cannot provide the same level of readability. So the visual BTNs are not only more readable, they are immediately executable given the fact that the runtime engine supports the underlying interrupt structures that are represented in the authoring environment.

4.3 Decomposing Complex Behaviors

The hierarchical structure of BTNs provides a means to break down large or complex behaviors into more manageable, readable, and even re-usable components. In many ways the hierarchical breakdown corresponds to levels of decision-making. For our Counter-Strike bots, a high level decision is reached about whether to navigate to a sniping position, navigate to a strategic position such as the hostage area, or patrol the map, and so on. Once this decision is made, a second-level behavior for each of these kinds of activities is initiated. Consider a "rescue" bot engaged in performing the behavior for navigating to the hostage area. At this point, the execution stack essentially has two levels, the main top-level behavior and the second level navigational behavior. Within the navigational behavior, it is reasonable to have sub-behaviors for handling enemies encountered while en route. At the lowest level under one of those behaviors is a simple "shoot enemy" behavior, which involves targeting the entity and firing the weapon. At this point, the execution stack may be five or six levels deep.

By assembling behaviors in a hierarchy such as this, each BTN can be relatively simple and easy to read, but the amalgamation can represent very complex behaviors. Furthermore, BTNs designed for modularity lend themselves to reuse in different behaviors. For example, the same weapon firing behavior that may appear at the lowest level under a navigation behavior may also be used under a sniping behavior or a patrolling behavior.

4.4 Team Cooperation

Through the use of the native blackboard system described earlier, a number of cooperative behaviors were developed for the Counter-Strike bots. The two teams in Counter-Strike have different objectives. The rescue team must navigate to where hostages are held, rescue them, and escort them back to a rescue zone, all while under the threat of encountering defenders at any point. The defending team simply seeks to stay alive and prevent the rescuers from accomplishing their objectives.

Thus, the rescuers collaborate to provide support to each other while navigating and when under fire. As such, they divide into leader and follower roles, for which polymorphic behaviors were defined for the performance of their various tasks. For example, the leader and follower each perform their own version of a follow path behavior when maneuvering together. During navigation, the follower provides cover for the leader's movements, with the "stop and cover" actions and the "catch up" actions regulated by the communications the team members share via the blackboard. If a leader is killed and a follower is available, the follower simply changes roles to become a leader, and subsequently performs the version of any given behavior defined for that alternate role.

The defending team seeks to find strategic sniping positions and ambush positions, and in some cases to patrol for rescuers. For this team, collaboration is necessary primarily for deconfliction purposes; in other words, to make sure that no two snipers attempt to go to the same sniping point, and that no two patrolling defenders follow the same route at the same time. This deconfliction is carried out via a simple blackboard posting mechanism, where each team member simply consults the destinations or routes that have already been posted, decides on one that is absent from the list, and posts this destination or route for others to see. Again, this is easily implemented in the authoring environment, but a critical element of believable and effective team behaviors.

4.5 Authoring Methodology

While we had applied the visual behavior authoring methodology described in this paper to a variety of simulation applications, the set of behaviors that we developed for Counter-Strike was substantially larger and more complex than any we 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, we took a highly iterative and incremental approach to authoring.

We 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, we repeated the process at the next lower level, and recursively continued this top-down decomposition until the behaviors were fleshed out to the level of concrete actions. At this point, we were able 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, we found that the initial vocabulary of actions and predicates we had defined was insufficient for our needs. 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 on our part of the concrete actions required to implement our target behaviors. After we made a first pass through the authoring process, we therefore revised and expanded the list of actions and predicates 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. Deliberative or sensory actions were occasionally translated into predicates (or vice versa) as usage determined which version would be most understandable and convenient.

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, we extended the basic set of behaviors polymorphically using "Team," "Role," and "Attack Style" descriptors (among others). This approach allowed us to easily 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.

5 DISCUSSION

This paper has presented an overview of our visual behavior representation for simulated entities, and it has also described an instance of the application of this representation to the development of automated players within a commercial computer game.

The process of authoring the complex behaviors required for competent automated Counter-Strike players provoked a number of interesting observations regarding our behavior representation and authoring methodology. On the positive side, we found that the use of a visual representation (along with the associated visual authoring tools) did in fact reduce the time required to develop the behaviors from that which would have been required to write them in C++ (based on our experience with similar development efforts in the past). The visual representation was particularly helpful when it was necessary to share behaviors between developers, because it substantially reduced the amount of effort required to decipher a "foreign" behavior. The hierarchical and polymorphic facets of the representation also lent themselves very naturally to an iterative authoring style that worked well on an underspecified effort such as this.

The Counter-Strike effort also made it clear, however, that some areas of our representation could benefit from further thought. In particular, there is little support for authoring complex, highly-interdependent team behaviors aside from the basic set of actions and predicates supplied for inter-entity communication. While it is possible to create entities that exhibit teamwork (as we have demonstrated), it requires much more care, forethought, and attention than developing behaviors for a single independent entity. We plan to explore ways of extending our methodology to simplify the task of building coordinated entity behaviors.

Another weakness in our representation is the lack of explicit support for parallel action within a single entity. To model an entity that can perform multiple simultaneous actions, it is necessary to divide the entity into several cooperating sub-entities. This is not an unusual tactic for dealing with parallelism, but given the above-described challenges involved in coordinating multiple entities, it is currently non-trivial to author behaviors featuring true parallelism. We hope to alleviate this difficulty in future versions of our representation, possibly by adding explicit capabilities for parallel and team authoring to our visual authoring toolset.

ACKNOWLEDGMENTS

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

REFERENCES

Counter-Strike 2003. Counter-Strike Mod Official Website. <http://www.counter-strike.net/>

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, and his email address is <fu@stottlerhenke.com>.

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 au-

thoring 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 and his email address is <jensen@stottlerhenke.com>.

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 and his email address is <houlette@stottlerhenke.com>.