AN AUTHORING TOOLKIT FOR SIMULATION ENTITIES

Daniel Fu, Ryan Houlette, and Oscar Bascara
Stottler Henke Associates, Inc.
San Mateo, California

Abstract

Instructors and analysts within the military community have noted the possibility of using video games for training or analysis purposes. Game developers, however, often deviate from actual doctrine. Unfortunately, the instructor or analyst is often left with no means to modify the entity behavior to conform to doctrine. At the same time, there has been interest in the Artificial Intelligence research and game development communities as to whether it is possible to create an "AI toolkit." Developers could use such a kit to create the entity behavior in a game quickly without having to start from scratch. In the ideal case, a kit would consolidate the existing branches of useful work in the field, thus providing developers easy access to the fruits of mature research.

In this paper we describe some initial steps towards a solution for both communities. There are two parts to our work. First, we have created an authoring tool that enables a developer to create entity behavior using a graphical "drag and drop" interface to quickly build up complex behavior. Second, we have created a runtime engine that works in conjunction with a simulation to operationalize the behaviors defined in the editor. The authoring tool allows authors to rapidly build complex behavior, while the runtime engine is built on well-understood game technology.

Biographical Sketches

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

Ryan Houlette is a project manager and software engineer at SHAI. His primary interests lie in the areas of intelligent interfaces, autonomous agents, and interactive narrative. During his stay at SHAI, he has participated in the development of a wide range of AI systems. 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.

Oscar Bascara is a consulting software engineer at SHAI. His interests include behavior network 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.

AN AUTHORING TOOLKIT FOR SIMULATION ENTITIES

Daniel Fu, Ryan Houlette, and Oscar Bascara
Stottler Henke Associates, Inc.
San Mateo, California
fu@shai.com, houlette@shai.com, bascara@shai.com

INTRODUCTION

Recently, instructors and analysts within the military community have noted the possibility of using video games for training or analysis purposes. The reason is that video games have become close to real life: better graphics and better entity behavior. Game developers build games for entertainment purposes, however, and thus deviate from reality when it suits them, unfettered by any actual doctrine. The absence of realistic doctrine is most evident in real time strategy or turn based games where entities do not behave as they would in the real world. Unfortunately, the instructor or analyst is often left with no recourse to modify the existing entity behavior to conform to doctrine. At the same time, there has been interest in the Artificial Intelligence (AI) research and game development communities as to whether it is possible to create an "AI toolkit." Developers could use such a kit to create the entity behavior in a game quickly without having to start from scratch. In the ideal case, a kit would consolidate the existing branches of useful work in the field, thus providing developers easy access to the fruits of mature research.

In this paper we describe some initial steps towards a solution for both communities. There are two parts to our work. First, we have created an authoring tool—also known as the "behavior editor"—that enables a developer to create entity behavior using a graphical "drag and drop" interface to quickly build up complex behavior. An immediate benefit is that an end user could use the same editor to modify entity behavior after release. Second, we have created a runtime engine that works in conjunction with a simulation to operationalize the behaviors defined in the editor. The engine technology is based on finite state machines. State machines are the most common way to control entities within a simulation, and appear to be the most appropriate place to begin a toolkit.

The rest of the paper is organized as follows. First, we describe the overall system architecture, the various components, and their roles. Second, we discuss aspects of the two games we're adopting as testbeds. Next, we describe our representation for describing entity behavior. Fourth, we describe how the runtime engine works. We end with a summary and lessons learned.

SYSTEM OVERVIEW

There are two major components to the system: the behavior editor and the runtime engine. Figure 1 shows these two major components in two corresponding rectangles. The left is the authoring component which generates a behavior library for a simulation entity; the right is the runtime portion where the engine controls entities within the simulation.

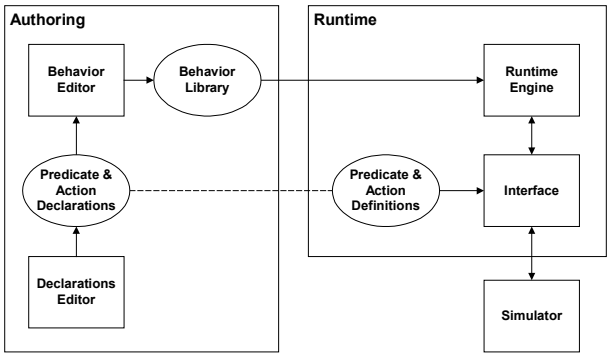

Figure 1: System Overview

Two editors constitute the authoring component: the declarations editor and the behavior editor. The declarations editor supplies a basic vocabulary of predicates and actions that serve as building blocks for the behavior editor. A developer uses these blocks to assemble complex behaviors for the library.

The runtime component references the behavior library to direct entities in the simulator. It does so indirectly through communications with an interface module residing between the engine and simulator. This interface was originally a basic framework in which the developer deposited computer code to operationalize the predicates and actions. During development, as the types of information available to entities, and their

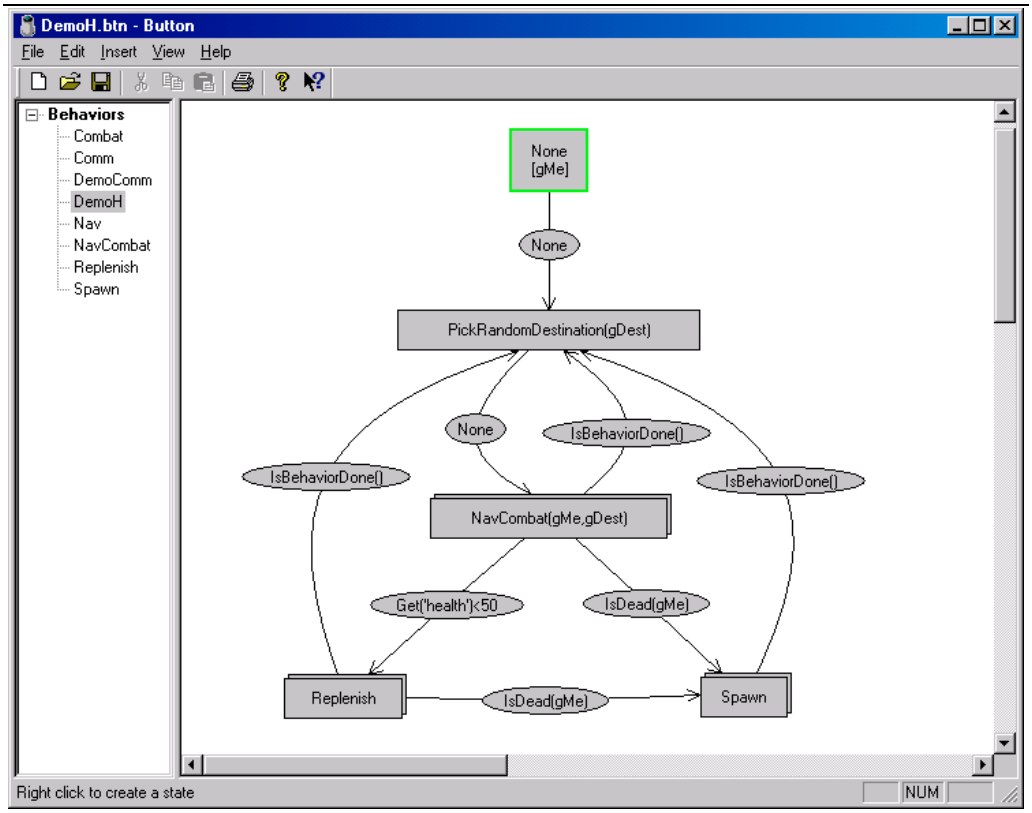

Figure 2. A screen snapshot of the behavior editor.

capabilities, become better known, the respective predicates and actions will be updated both in the declarations editor and the interface.

TESTBEDS

Our initial work started with a video game called "Half-Life," which is a multiplayer first-person shooter. Because parts of the source code were released to the public, programmers are able to make their own modifications, or "mods," to the game, thereby creating altogether different environments and games. We used "Team Fortress"—a mod emphasizing teamwork in "capture the flag" scenarios. Intelligent entities were created to work at tactical, operational, and strategic levels. For example, we created a defensive team where scouts patrolled routes as well as areas where threats were detected. 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 another mod "Counter-Strike," which features hostage, bomb, or escort situations with realistic weapon models. For example, counterterrorists must rescue hostages within an allotted time while terrorists strive to hold the hostages. A round ends when either 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. We are currently implementing better movement techniques.

Our second testbed is "Civilization"—a popular turn-based game. This game is markedly different from Half-Life, emphasizing strategy without real-time pressures. The objective for each player is to build a civilization starting from the beginning of 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, and determine the level of taxation. Work in this domain is in the initial stages. We are using an open source version of the game called "FreeCiv."

BEHAVIOR EDITOR

The behavior editor is a standard Windows application featuring a "drag and drop" interface, enabling developers to quickly construct complex behavior visually. Because we use a visual representation, we believe the AI is exposed enough so that non-programmers could potentially use it to program their own behaviors, or at least understand how existing behaviors work.

Figure 2 shows a sample screenshot of the application showing a behavior for Team Fortress. The left pane holds the list of defined behaviors. Whenever the user selects a behavior, its definition appears in the right pane. In this instance, the user has selected "DEMOH." It appears as a set of rectangles connected by directed arcs with ovals. These are finite state machines (FSMs). As stated earlier, our authoring tool uses the notion of FSMs to describe behavior. Because we depart from the technical definition of a basic FSM, we refer to our FSMs as "behavioral transition networks" or "BTNs" for short. BTNs are hierarchical in nature and consist of two types of units: nodes and transitions. A node in a BTN represents an action or behavior that the entity will perform. A transition shifts the runtime focus from one node to another, but only if its set of predicates are satisfied.

The DEMOH behavior is a high-level behavior that has the entity pick a random destination and then move towards it using the NAVCOMBAT behavior. If the entity dies, it should SPAWN in another location. If the entity is low on health, it should heal itself by going to a supply room to Replenish. In either case, when done, it should resume its original activity by picking a new destination, and so forth. Note that parameters appear in parentheses after the behavior/action name.

REPRESENTATION

In this section we describe the vital elements that make up the behavior editor and power the runtime engine. Recall that behaviors are composed of nodes and transitions. Three nodes in a BTN are of special significance. The **current node** of a BTN denotes the action or behavior currently being carried out by the associated entity; a given BTN may have exactly one current node at a time. The **initial node** of a BTN is simply the node with which the BTN begins execution (i.e., the BTN's initial current node). There can be only one. In Figure 2, the "None" action is the initial node. If the current node is a **final node**, the behavior is considered complete.

Note that the action contained in a node may be either **primitive**—for example, RELOAD(current_weapon) to reload the weapon in hand—or **complex**—for example, MOVETO(x,y) that will navigate the entity to a known location. Primitive actions interact with the game engine through the interface module; e.g., a human player may just press the "R" key to reload while the interface would mimic pressing the same key for the artificial entity. A primitive 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. Complex actions are handled completely within the engine. Ultimately, they will boil down to primitive actions.

As an example, suppose a PREPAREATTACK behavior incorporates a transition to a SELECTBESTWEAPON BTN as part of the preparation for an attack. The latter will reach a final node when it determines the best weapon in the arsenal. At that point, the PREPAREATTACK behavior will proceed, perhaps next invoking a behavior to switch to the best weapon. Know that complex actions always include an initial node and usually include a final node.

The current node changes according to a **transition**, which is a directed arc connecting two nodes X and Y (or looping from one node back to itself) and indicating a potential direction of control flow. Each transition has a set of logical conditions that, when satisfied, make the transition **active** or **satisfied**. A transition is said to be active or satisfied if its conditions return a "true" result; an active transition indicates that the BTN may change its current node from node X to node Y. We refer to the logical conditions as a **trigger**. Most triggers ultimately rely on **predicates** to return useful values. Here, a predicate refers to an information-extracting operation from the interface to the game. For example, the condition

DISTANCE(x,y) > 70

is "true" when the DISTANCE predicate—the distance between two coordinates *x* and *y*—is greater than 70.

Figure 3 shows a hypothetical MOVETO action that contains nodes and transitions. The initial node is the upper left rectangle; the final, the lower right. See a cycle of determining the next step to take, and taking the step.

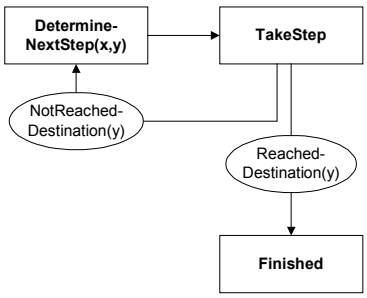

Figure 3. Simplified MOVETO(x,y) complex action.

The "x" and "y" associated with the DISTANCE predicate and MOVETO action are **variable** parameters. A variable confers a BTN the ability to store data. Typically, variables keep state information, or are parameters to a predicate or action (either primitive or complex). Each BTN comes with its own set of user-defined weakly typed variables. In Figure 2, there is only one variable called "gMe," which merely refers to itself. Other pieces of information, such as the amount of health "Get('health')" could be assigned to a variable, if so desired.

When a variable is assigned a value, we refer to this action as a **binding**. The engine processes bindings when invoking a new BTN, when traversing a transition, when performing a node's action, and when finishing a BTN. Values can be any one of four types: *number*, *string*, *entity*, *vector*, and *data*.

RUNTIME ENGINE

In this section we describe how the engine processes BTNs. Recall that BTNs may be hierarchical—that is, any node can embody an arbitrary BTN. When such a node becomes current, execution passes to the BTN's initial node. Each time this occurs, a new **execution frame** is put on the **execution stack**. The execution frame holds the behavioral state at a level within the BTN hierarchy. This includes a BTN, current node, and variable values. The execution stack maintains the set of frames, thus encapsulating the entire hierarchy. Each entity has its own stack.

The run engine's basic execution cycle for an entity is as follows:

```
function Execute() {
    // Do the current action
    CurrExFrm = current (topmost) execution frame;
    CurrNode = CurrExFrm->currentBTN->currentNode;
    CurrNode->DoAction();

    // Determine the next current node
    ExFrm = bottommost frame on B's execution stack;
    while (ExFrm) {
        // Find an active transition
        CurrNode = current node in ExFrm;
        Transitions = CurrNode's transitions (possibly ordered)
        for each transition T of Transitions {
            if (T->Evaluate() == true) {
                // Pop all execution frames above ExFrm
                B->Pop(ExFrm);
                // Follow transition by changing CurrNode
                T->Traverse();
                // Done with this entity
                return from function;
            }
        }
        // Try the next execution frame up
        ExFrm = ExFrm->next;
    }
}

Node's function DoAction() {
    if (action is in a final node) {
        sync BTN parameters with passed-in variables;
        pop current execution frame from stack;
    }
    else if (action is a primitive action) {
        do the primitive action;
        sync parameters with passed-in variables;
    }
    else if (action is a complex action) {
        NewBTN = new BTN for complex action;
        NewBTN->currentNode = NewBTN->initialNode;
        NewFrame = new execution frame;
        NewFrame->currentBTN = NewBTN;
        push NewFrame on top of the execution stack;
    }
}

Transition's function Traverse() {
    apply transition's variable bindings to BTN;
    // Update the current node at this level
    NewCurr = terminating node of this transition;
    Set current node of current execution frame to NewCurr;
}
```

Note that only the basic execution stack traversal is hard-coded. All major decision points in the architecture—for example, selecting which transition to traverse out of the several that are active—are provided with "hooks" so that default decision algorithms can be easily modified and extended, even at runtime.

Example of Execution Flow

To give a rough idea of how the pseudocode given above works, here is a simple example. It refers to Figure 4 that shows the current execution state for a single entity. The rectangles denote nodes, and the

arrows between them represent transitions. Nested nodes indicate different levels of the hierarchy, and nodes with dashed borders are current nodes (in their particular execution frame).

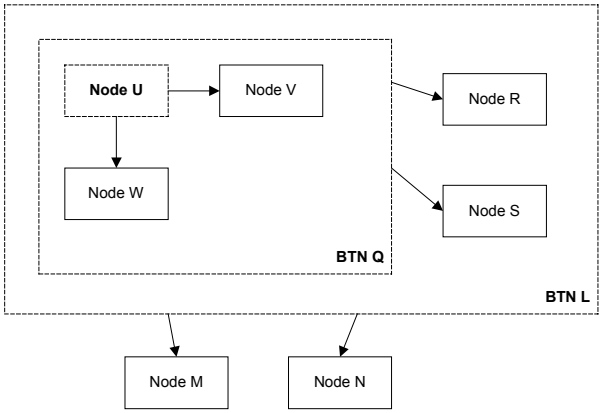

Figure 4: Current execution state for a single entity.

At the current moment, the execution stack for this entity looks like this:

Execution Frame #	Current Node	Possible Transitions
3	U	U→V, U→W
2	Q	Q→R, Q→S
1	L	L→M, L→N

Thus, on the next iteration when the entity starts the execution cycle, the first thing that will happen is that the engine calls U→DoAction(), since U is the current node in the topmost execution frame.

Next, starting with the bottom of the stack and working up, the engine evaluates transitions until one is satisfied. Evaluating transitions in this order allows more general, ongoing actions (i.e., higher-level plans) to take precedence over low-level, short-term actions.

Thus, the possible transitions for execution frame #1 are checked first:

T_{LM}→Evaluate() == false

T_{LN}→Evaluate() == false

Suppose that neither of these is satisfied. Evaluation now moves to execution frame #2:

T_{QR}→Evaluate() == true

Note that since T_{QR} is now active, the engine never checks T_{QS}. The engine always short-circuits by following a satisfied transition immediately.

The engine calls T_{QR}→Traverse(), which pops off the now-irrelevant execution frame #3, sets the current execution frame to be #2, and sets the current node in that frame to be R.

At the end of the execution cycle for this entity, the execution stack looks like:

Execution Frame #	Current Node	Possible Transitions
2	R	not shown
1	L	L→M, L→N

In the case where the engine finds no active transitions, the current node does not change. Thus, U→DoAction() would be called again and again until a transition becomes active.

Transitions in Execution Flow

From the current node, the engine evaluates transitions until one is active, then follows that transition. Because one may wish to prefer one transition to another, each transition can have an assigned priority relative to other transitions. This priority compels the engine to evaluate transitions in a pre-specified order. Thus, the engine always traverses the highest-priority active transition. For efficiency reasons, the runtime engine will not evaluate all transitions; instead it sequentially evaluates until one becomes active. The ordering on transitions is a strict ordering. How this ordering is determined is up to the author. Currently, each transition is automatically assigned a priority number. Before evaluating transitions, the engine organizes them according to priority.

The runtime engine determines whether a given transition is active by calling its `Evaluate()` method. This method invokes the decision procedure for the transition and returns a true or false response.

There are two main stages to the evaluation of a transition. First, all variable bindings for the transition are applied one at a time *in the order specified in the behavior editor*. When a binding is applied, the expression on the right-hand side is evaluated, making all variable substitutions and function calls as necessary. The resulting value of the expression is then assigned to the left-hand variable.

In the second stage of evaluation, the transition conditions are evaluated one-by-one, in the order specified by the author, until either:

- A condition evaluates to false, in which case `Evaluate()` will return false, but not before rescinding the bindings.
- All conditions have been evaluated and have returned true, in which case `Evaluate()` returns true.

Actions in Execution Flow

The execution flow for actions is very similar to that for transitions except there is an additional stage to handle action parameters. In the first stage, the engine evaluates all variable bindings for the action in order, just as for transitions. In the second stage, the engine invokes the action along with its parameters bound to values. In the third stage, the resulting values of the action parameters are passed back to the variables to which they are bound. If a parameter is not bound to a variable, its value is not returned.

Example

MyLoc and *TargetLoc* are BTN variables of type vector and currently have the values [10, 392, -47] and [70, 200, -10], respectively. *Path* is a variable of type data, and its current value does not matter since it is being used solely to hold the return value of the action. There are three variable bindings for this node:

- src = MyLoc
- dst = TargetLoc
- path = Path

The action associated with this node is:

GeneratePath(src, dst, path)

After the first stage of execution, the action parameters will have the values

- src = [10, 392, -47]
- dst = [70, 200, -10]
- path = <don't care>

The action is then invoked with these parameters:

GeneratePath([10,392,-47], [70,200,-10], <don't care>)

It deliberates and produces a path, stored internally in the *path* variable. After the action has completed its execution, the internal value of path is passed back to the BTN variable *Path* in the third stage, at which point it becomes accessible by the rest of the BTN.

Note: In the case where no active transitions exist from the current node of the top-most execution frame, the node's action will repeat. If *GeneratePath* is a primitive action, *Path* will be modified continually. If the action is complex—i.e., a BTN—then *Path* will only be modified when the BTN reaches a final node.

As in the example, sometimes one or more parameters of an action are used only to capture return values from the action. At the same time, other types of actions may not modify their parameters at all. This distinction will be explicit in future releases of the behavior editor. Currently, primitive action parameters are "in," "out," or "in/out" to reflect input, output, and input & output parameters. In C++ terminology, one can think of these as passing by value, reference, and reference. The editor will make the distinction for primitive actions and predicates, but the engine does not enforce correct usage.

Variables

As we have seen, variables play an important role in defining behavior. Thus far, it's been necessary to explain BTN variables as we described other aspects of the representation and engine. In this subsection we underscore the salient points.

In Execution Flow

The engine strictly controls when BTN variable values may change. They change at four points during the execution flow:

1. **Invoking a BTN**: The BTN's parameters will receive initial values from the calling BTN prior to instantiation of the initial node.
2. **Processing a current node**: When calling an action, the engine calculates values for bindings before and possibly after.
3. **Evaluating a transition**: The engine calculates values for bindings before evaluating the conditions. Bindings take effect only when all conditions are true, otherwise the bindings are undone.
4. **Leaving a BTN**: The BTN's parameters will retain modified values when the engine reaches a final node. If it doesn't, the BTN's parameters (if any) remain unaltered.

Types

Each BTN maintains a set of typed variables that can be accessed by all transitions and nodes in that BTN. Valid variable types comprise *number*, *string*, *entity*, *vector* (for vectors or 3D coordinates), and *data* (for arbitrary data types).

Bindings

Each transition has associated with it a decision process that, when executed, determines whether the transition is currently active (and thus able to be traversed). Typically, this decision process consists of the evaluation of a list of logical conditions, which in turn are defined over a set of BTN variables. These variables acquire their values through explicit variable bindings.

Each node or transition in a BTN can have associated with it one or more variable bindings. Each variable

binding creates a mapping between a specific variable and an author-specified expression, which may include references to other variables or to functions that compute further information. The following are all valid bindings:

- `x = 30.5` *[number]*
- `nearest_threat = NearestThreat()` *[entity]*
- `range = RangeToEntity(nearest_threat)` *[number]*
- `foo = (x * range) - ComputeInterestingValue()` *[number]*
- `utterance = 'hey'` *[string]*

Note that a binding must specify the arguments to a function if that function requires arguments (as in the third binding above). Those arguments may be literal values or variables, and they are passed by value to the function so that they cannot be changed.

There are two reasons to have explicit variable bindings. First, a transition or node typically involves one or more routines that gather, process, and act upon information. Frequently, these routines have information relevant to other transitions and actions. So instead of gathering the information again the next time it is needed, the transition or node caches the information. The second reason for variable bindings is that they make information visible, which in turn simplifies the authoring process by making explicit the data relationships between the nodes and transitions in a BTN.

Variables as Parameters

Each predicate and action is hard-coded with a set number (which may be zero) of typed **parameters**. These parameters represent the data transactions between the predicate or action and the rest of the world that are exposed to the author. Any internal manipulations of the predicate or action that do not affect its parameters are hidden and inaccessible. The engine treats parameters exactly like normal BTN variables except that they are not global; that is, they exist only for the action or predicate.

When one adds a new node to a BTN and specifies the associated action, one must also specify a binding for each parameter of that action. These bindings behave just like normal variable bindings with one exception: they are evaluated twice, once before execution (like normal bindings) and once after. This allows an action to "return" one or more values, which is necessary to support perceptual or deliberative actions whose sole effect is the generation of useful information. Predicates, by definition, return a value that can be bound to a variable.

Diverting Flow of Control: Interrupts

Occasionally an entity needs to interrupt its normal execution flow to handle an ancillary task. The task, however, does not demand the entity to refocus all computational resources towards completion of the task. Rather, the task can either be small enough to be executed via a temporary frame push on the execution stack—keeping the stack intact. This type of complex behavior occurs as a result of a special transition, referred to as an "interrupt" transition. We describe its use in this section.

Sometimes it's necessary for an entity to perform some small, but frequent task. Consider an entity that needs to reload its weapon, or parse an incoming message from a teammate. These occurrences are sure to happen; yet they are not necessarily germane to any singular task. We introduce the notion of an "interrupt" to handle such occasions. An **interrupt transition** causes the engine to deviate from the normal flow of pushing/popping execution frames by pushing a brand new execution frame and continuing from there. This type of transition, when complete, will leave the previous execution stack intact, allowing the entity to continue as before. This is important because if we wanted a regular transition to handle the interruption, it might clear the execution stack, or force the author to handle the task at a lower level in the behavior hierarchy. For example, suppose a Half-Life bot is following a path to some destination. If its weapon has small amount of ammunition, it should reload the weapon in preparation for the next appearing threat. This type of task should be the consequence of an interrupt transition. There are two reasons. First, suppose we used a regular transition from the abstract "navigation" behavior to the "reload" behavior. When followed, the portion of execution stack concerning navigation would be thrown out, supplanted by the behavior to reload the weapon. But after reloading, all the information would have to be restored by re-invoking the navigation behavior. This may, in general, be a costly overhead. Second, suppose we created regular transitions within the lower levels of the navigation behavior thereby pushing additional frames onto the stack instead of popping. Thus, after (say) the entity takes a step forward, we check and deviate to reload the weapon. This works, but would be arduous indeed. We'd have to construct transitions within every conceivable task. Further, it's counterintuitive to bother about reloading a weapon within the definition

of a navigation task, never mind creating confusing spaghetti networks.

Now that we have defined an interrupt transition and how it affects the execution stack, one may wonder whether interruptions are recursive. The answer is a qualified "yes." There are certain restrictions on how interrupts are handled. First, when the engine places an interrupt transition's destination behavior (a complex action) in an execution frame, transitions in some frames must be ignored. These include the frames starting from the originating frame (from where the interrupt happened) up to, but not including, the interrupt's frame. To see why, consider this execution stack:

Execution Frame	Current Node	Possible Transitions
I	I_1	$I_1 \rightarrow I_2$
C	C_1	$C_1 \rightarrow C_2$
B	B_1	$B_1 \rightarrow C$, $B_1 \Rightarrow I$
A	A_1	$A_1 \rightarrow B$, $A_1 \rightarrow D$, $A_1 \Rightarrow J$

This stack is the result of behavior "C" being overridden by an interrupt transition taking effect in behavior "B" ($B_1 \Rightarrow I$). Frame "I" is the new interrupt frame. Execution proceeds by checking the transitions in frames "A" and "I" only. If the engine followed a transition in "B" or "C," a counterintuitive situation develops. Namely, for "B," infinite interrupts would be placed on the stack unless evaluation of B_1 has side effects; for "C," its next transition would have the engine place a new frame above frame "I," thereby undermining the interrupt.

Note that satisfied transitions appearing in frames "A" or "I"—either of regular or interrupt—would be followed. A regular transition originating in "A" ($A_1 \rightarrow D$) would clear the stack above "A" and place a new frame as so:

Execution Frame	Current Node	Possible Transitions
D	D_1	
A	A_1	$A_1 \rightarrow B$, $A_1 \rightarrow D$, $A_1 \Rightarrow J$

An interrupt transition originating in "A" ($A_1 \Rightarrow J$) would be placed on the stack like so:

Execution Frame	Current Node	Possible Transitions
J	J_1	

Execution Frame	Current Node	Possible Transitions
I	I_1	$I_1 \rightarrow I_2$
C	C_1	$C_1 \rightarrow C_2$
B	B_1	$B_1 \rightarrow C$, $B_1 \Rightarrow I$
A	A_1	$A_1 \rightarrow B$, $A_1 \rightarrow D$, $A_1 \Rightarrow J$

After the "J" interrupt completes, "I" resumes.

LESSONS LEARNED

Hierarchical FSMs are a vast improvement over flat ones. Initially we started using legacy code which implemented finite state machine technology. After authoring some sample behaviors, it became arduous to improve the behavior. Attempts to do so resulted in spaghetti-like graphs. A transition to a hierarchical representation greatly simplified the authoring process, while making the runtime engine more complex.

Defining an expressive vocabulary is nontrivial, especially for games or simulations that are multiplayer in nature. Because we implemented an AI for an otherwise non-AI game, we had to declare all useful predicates and actions. Because the game was not designed with artificial players in mind, information extraction for predicates and actions in the game were not straightforward to implement. The open source code for our next testbed, FreeCiv, is much more accommodating as it features AI already, making it much easier to improve.

Defining a separate runtime engine and interface encourages a clean separation from the simulation itself. In creating the interface we often found the necessary pieces of data and functions residing within the server side of the simulation. Development of a game happens in months, not years. We believe a developer starting with an existing runtime engine will end with a cleaner separation of code modules, making them easier to reuse.

Visual authoring works well, but better visual metaphors are needed. So far, the authoring process is familiar to people who have some programming experience. However, we lack a good metaphor for visual programming that can also convey the complexities of the engine.

SUMMARY

This paper has described two important contributions to the field of simulation development. The first is an authoring toolkit that, if used in development, will allow end users to customize entity behavior in a

simulation. Moreover, end users can reuse existing work, thus speeding the development of other behavior.

The second contribution is an initial step towards a body of work encapsulating common AI needs for simulation and game development.

ACKNOWLEDGEMENTS

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