An AI Modeling Tool for Designers and Developers

Dan Fu, Ryan Houlette, and Jeremy Ludwig
Stottler Henke Associates, Inc.
951 Mariners Island Blvd, Suite 360
San Mateo, CA 94404
{fu, houlette, ludwig}@stottlerhenke.com

Abstract—We describe an AI modeling tool meant to be used by both designers and developers. The method for authoring is visual and meant to convey decision logic in a more intuitive manner while retaining expressiveness. This data-driven approach features an AI runtime engine which incorporates several augmentations which make it suitable for use across a wide array of deployed systems. [1] [2]

TABLE OF CONTENTS

1. INTRODUCTION

The creation of artificial intelligence (AI) is a common construction task, usually handled by technical developers. In recent years we have been working on techniques to open the process of writing AI behavior to a wider audience. One important aspect of our effort is to make an authoring tool that makes entity behavior accessible not only to developers, but to the rest of the team as well. These include designers and analysts who also possess most of the domain knowledge. They work with developers to ensure the entities behave appropriately. Unfortunately, a less efficient interaction occurs on the team when the designer communicates the desired behavior in a written document, waits while a developer implements the behavior, and then tests and revises the design based on the outcome. There are two reasons why this bottleneck happens. First, the implementation details are often hidden from the rest of the team. There are usually few developers who are responsible for the implementation, and while they ensure the AI works with the simulation engine, it is often not necessary for their code portion to be accessible to any other members on the team. The second reason is that even if the implementation details were available, their portrayal is often a body of programming code or AI representation bearing little resemblance to the design documents that specify their intended characteristics. As a result, improvements to the AI must go through developers [10].

In this paper, we discuss our ongoing efforts to build a tool called SimBionic that bridges the gap between team members, empowering them to participate in varying degrees in both design and implementation. The central challenge of this work is to arrive at a tool that is easily comprehensible, yet powerfully expressive.

SimBionic features a graphical editor which uses finite-state machines (FSM's) as a basic way of depicting behavior [1, 2]. FSM's look similar to flowchart diagrams where states and transitions are drawn as rectangles and arrows. This method of representation is ubiquitous enough to be one of the nine diagrams employed by the Unified Modeling Language (UML) for software engineering design and communication [3].

FSM's are most well known as a construct from computer science theory, defined as a set of states, an input vocabulary, and a transition function which maps a state and an input to another state. A single initial state is the designated start state. There are zero or more "accepting" states. After the FSM processes all input, whether the ending state is an accepting state dictates the machine accepting the input or not. More practically as a control model, we can view the FSM as a way for an AI entity to change its state (which corresponds to behavior) over time. The transition from theoretical to practical use happens in three ways. First, it's appealing to consider each state as representing some desired behavior or action. Each state has corresponding code so that as the entity's state changes, its behavior changes accordingly. Second, each state's set of transitions are conditions under which the entity behavior will change. Third, the notion of an accepting state is generally interpreted as the end of execution for the FSM; that is, the FSM has achieved some desired task or goal.

Adopting the practical use of an FSM model for our tool has two appealing properties; namely simple visual depiction and efficiency. FSM's have the most straightforward "what you see is what you get" kind of behavior representation, oftentimes better communicated as pictures. They have the positive computational properties of being simple, compact, and efficient. For these reasons we adopted FSM's as our basic depiction method. Still, for what is gained in comprehension, visual parsimony, and computational efficiency, finite state machines can grow to be visually cluttered while remaining computationally weak as they are composed of only states and transitions. To address these two unappealing properties of visual clutter and computational weakness, we created new augmentations

1

[1] 1-4244-0525-4/07/$20.00 ©2007 IEEE.
[2] IEEEAC paper #1001, Version 6, Updated July 1, 2006

that seek to support more computational power while maintaining a simple, intuitive interface.

The remainder of this paper is organized as follows. First, we provide an overview of SimBionic. This is followed by a discussion of key augmentations. Second, we describe the interface to a simulator or robot. Third, we show the SimBionic authoring application that supports the computational augmentations while striving to be as easy to use as possible. We then discuss software and robotics work related to SimBionic.

2. SIMBIONIC OVERVIEW

SimBionic is composed of two major components: authoring tool and runtime engine. The authoring tool component is responsible for having users describe the behavior for an entity. This specification is fed to the second component—the runtime engine—which creates behavior in the interfaced simulation based on the specification. Figure 1 shows a high level view of SimBionic. In the authoring portion, the user defines a vocabulary of actions and conditions which correspond to FSM notions of states and transitions, respectively. We refer to a constructed FSM as a "behavior." Users construct behaviors by linking actions with conditions.

The authoring portion does not actually implement behavior; rather, the product is a description or specification for the runtime engine. On the right side of Figure 1, see that the vocabulary has an analog. This is where the actions and conditions take real form. One can think of an action as a primitive effector that operates in the world, be it simulated or actual, while a condition can be thought of as depending on sensor information predicates. Since behaviors are synthesized from conditions and actions, the runtime engine will reference them and invoke the appropriate effector or sensor. This happens in the interface with the world.

The runtime engine functions by first invoking the action at the current state, and then determining a valid transition to the next state. Valid transitions are calculated by evaluating conditions. This fundamental two step process then repeats.

Key Augmentations

There are four augmentations made to SimBionic's computational model: expressions, hierarchical behaviors, interruptible behaviors, and polymorphism. The first improvement is arbitrary expression evaluation. The authoring tool is able to define the condition vocabulary, such as "FrontSonarReading()" or "RightSonarReading()" which return the number of feet an obstacle is detected in front or right of the robot, or "WaypointReached()" which returns whether the traveling entity has arrived at a waypoint yet. These vocabulary elements can be combined to form true or false statements for purposes of evaluating transitions. For example, there may be a behavior to travel forward until there is an obstacle blocking forward movement. The transition away from moving forward to stopping might be "FrontSonarReading() < 2" which would translate to "keep moving forward until the number of feet in front of the robot is less than two". In this way, the ability to synthesize arbitrary expressions and to evaluate them comprises the first major augmentation. As well, the results of expressions can be stored as variables for later use, or passed as parameters.

The second augmentation is a conversion to hierarchical states where a single state could refer another FSM. In SimBionic terms, the action could either remain as an action, or refer to another behavior. There are two reasons for this augmentation. First, when authoring there

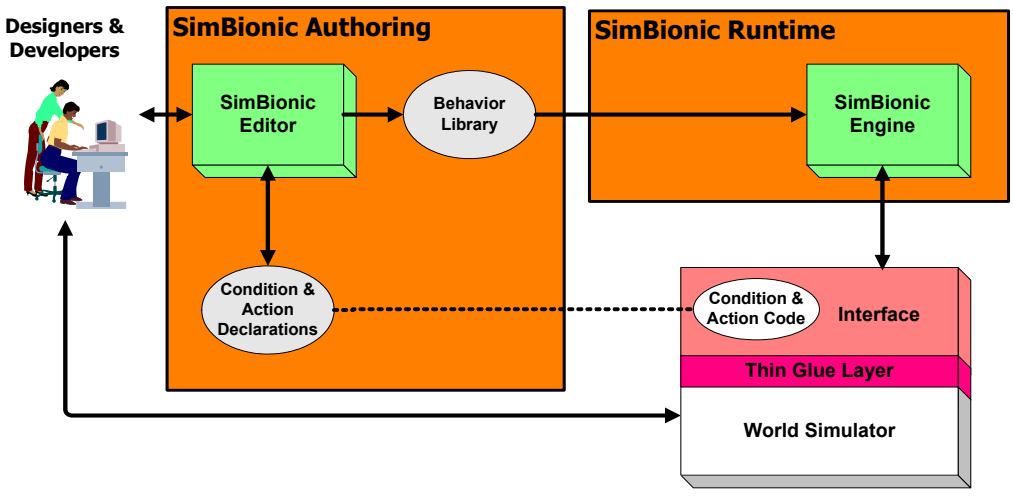

Figure 1- SimBionic Authoring Tool and Engine

frequently arises a need to invoke a recurring pattern of actions and conditions. The author can decide to continually repeat the pattern across several behaviors, or to modularize the pattern as a behavior that can be invoked when needed. This hierarchical computational model is stack-based, where each time the engine invokes a behavior, a reference to it is pushed onto the stack. The engine then pays attention to the behavior at the top of the stack, popping off behaviors that reach an "accepting" action.

This augmentation results in a method similar to a push down automata or Von Neumann architecture, except that all transitions on the stack are evaluated, starting at the bottom. The reason is that the behaviors on the lower stack frames are thought of as taking precedence. For example, the world could change such that whatever task the robot is currently engaged in, for which it is doing a primitive action of turning, is no longer worth pursuing. So while the turning action is in a behavior at the top of the stack, the particular behavior lower down in the stack where the task is deemed pointless needs to pursue something else. This conclusion would entail popping off all behaviors above it as a result of following a transition to a different behavior.

The third augmentation is the ability to interrupt behaviors. Frequently in robotic three layer architectures [4], there can exist several tasks or goals that the robot tries to achieve, or there may be unexpected or improvised events for which the robot should act immediately and appropriately. Though there are hierarchical behaviors in SimBionic that could handle impromptu events, they would require the current task to be abandoned by popping off behaviors from the stack. Thus, there exists the need for an "interrupt transition" which is a special transition for cases where the original behavior is not abandoned, but just temporarily suspended while the new behavior executes. When the new behavior finishes, control returns to the original behavior, which picks up where it left off. For example, a radio communication event arrives from a teammate. There may be a behavior to process incoming messages, but we would not want the current task to be abandoned (and eventually reinstantiated) as processing the message should only be a brief interruption. The "radio handling behavior" would be pushed onto the stack. Transitions on stack frames below this radio behavior would not be evaluated until the behavior has concluded. Typically, interrupts are meant to be handled quickly. Understanding error handling

Another type of interrupt handles errors which occur in the underlying code that implements an action or predicate. SimBionic provides a built-in facility, modeled after Java's exception-handling mechanism, for handling these types of run-time errors in a graceful and consistent fashion within behaviors. To use this facility, particular exceptions must be thrown within the action and predicate code. SimBionic provides two error-handling constructs and three core actions devoted to error recovery:

A "catch" action is the core error-handling construct that catches exceptions, much like the catch block of a try-catch statement in Java. When an exception is thrown during the execution of a behavior containing the catch action, the catch becomes the current action or behavior for that behavior, and the execution mode for the behavior is set temporarily to non-interruptible because error recovery should be treated as a critical section. Execution continues normally from that point, executing the catch and any subsequent actions.

The "always" action ensures that certain operations are always executed at a behavior's termination. It is very similar to the "finally" block of a try-catch statement in Java. Whenever a behavior is about to be popped from the stack for any reason, the always action becomes the current action or behavior of the behavior, and the execution mode for the behavior is set temporarily to non-interruptible. Execution continues normally from that point, executing the always rectangle and any subsequent actions until completion.

The "retry, resume, and rethrow" core actions help recover from an error. Common uses might be to select an alternate execution path, clean up any "loose ends" left behind by the error (e.g., open files), or simply to log the error. The "retry" action causes behavior execution to jump to where the exception was thrown in this behavior, executing that action or behavior exactly as if it had just become the current one in normal fashion. Retry is generally invoked after attempting to diagnose and fix whatever error condition caused the exception to be thrown, in the hopes that the second attempt will be more successful. The "resume" action causes execution to resume at the point in the behavior where the exception was thrown without attempting to retry execution. Resume is typically invoked when it is not possible to fix the error condition, but it is still possible to clean up after the error and continue with the behavior's normal course of execution. The "rethrow" action throws the current exception down the stack to the current behavior's invoking behavior and then terminates the current behavior just as if the behavior normally ended. The invoking behavior then becomes the new current behavior and must attempt to handle the exception exactly as if it had been thrown in that behavior. Rethrow is called when a behavior is unable to recover from an error and needs to pass responsibility for error recovery on to another behavior.

The fourth and final augmentation is polymorphism whose origin comes out of object oriented software development. As behaviors are created, the author often encounters situations where it is desirable to create behaviors that differ only slightly from already-existing behaviors. For example, an author may decide the expertise of an opposing force is a descriptor that affects behavior. When the force is in conflict with friendly forces, the "Combat()" behavior

would dispatch a specialized version of a behavior based on the entity's descriptor, say, low expertise.

If we were to create variations without polymorphic specializations, they would be named something along the lines of "Combat_LowExpertise()," prefaced with a "dispatch" behavior that would examine the entity's descriptors and choose the right version of behavior. This works fine, but results in some visual clutter before arriving at the new specialization for Combat. To simplify the construction of specialized behaviors, polymorphism was introduced. In this extended representation, a single behavior can now possess multiple versions. Exactly which version of a behavior gets invoked depends on a set of entity descriptors defined by the author. In this case, "expertise" is the only descriptor introduced. It serves as the root of its own descriptor hierarchy. An author specializes, or indexes, a behavior graph by associating it with exactly one node per descriptor tree. We could have a tree with "expertise" as the root and three children "low", "medium", and "high" as children. Selection of any one of these four nodes serves to index a behavior.

The engine, given, a set of descriptors for an entity, will always pick the most specific behavior version according to the degree of match between the entity and behavior indices. For example, if the entity has a descriptor "low" for experience, then the behavior version of Combat() indexed with "low" experience would be the best match. A behavior with the root "expertise" chosen would be the second best match. Behaviors indexed using roots of descriptor hierarchies are "default" behaviors because they are the most general. Any behaviors for "medium" or "high" expertise could never be chosen unless the entity's descriptor changed accordingly. This method of indexing affords the author the flexibility to specialize any behavior depending on particular attributes of the entity being controlled.

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 expertise from low to high would select a different version of the Combat () 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.

Augmentation Discussion

All augmentations but the first seek to simplify the visual representation of behavior. The first consists of expressions which can be fairly complex. Syntax for the expressions is similar to the programming language C where variables, functions (conditions), and numbers can be used.

The second augmentation encapsulates a commonly used pattern of actions and conditions into a behavior. For authoring, there are three benefits. First, the visual depiction is simplified by being able to refer to a behavior as an invokable abstract "action." Whatever the behavior accomplishes can be viewed as a separate behavior graph. This recursive simplification of behavior allows one to author or design behavior at a high level, but eventually ground behavior into real action. Second, since the author is not forced to spread identical sub-graphs across several behaviors, there is less visual clutter. Third, the author can make a single change in a behavior and have it affect all behaviors that make use of it.

The third augmentation, interrupt transitions, provides a way for entities to attend to other tasks before resuming their suspended activity. This engine feature is meant for transient tasks—most often for bookkeeping—where the additional processing minimally affects the overall behavior of the entity. For other tasks of primary importance, the evaluation of transitions based on stack frames provides a way for an entity to shift completely away from a current task to attend to one more urgent.

The last augmentation is polymorphism. This provides a way to specialize a particular task according to entity attributes. The alternate way of combining all specializations into a single behavior graph would require extra dispatch conditions, ultimately leading to a more complex graph for authoring. Polymorphism allows an author to selectively specialize behavior, and focus on that particular version. Together with the hierarchical references to and invocation of behaviors, the visual representation is vastly simplified. The only tradeoff made is the implicit dispatch: matching the entity's descriptors to the most specific behavior. This is accomplished in the runtime engine, but controlled indirectly through the author's attachment of descriptors to entity.

3. INTERFACE AND RUNTIME ENGINE

Between the runtime engine and the world there exists an interface which connects the actions and conditions to their respective effectors and sensors. Figure 1 shows a "thin glue layer" separating the interface from the "world simulator." The glue layer is simply mapping the action and condition calls to actual action and perception as defined in an API for the simulation or robot controller. Figure 2 shows the relationship between the runtime engine and the world as illustrated through a UML sequence diagram. There are two types of interactions illustrated. The top most portion shows the straightforward execution of an action. The "DoAction" function essentially takes the action identification number—assigned by the authoring tool when the action was declared—and invokes the corresponding entity effector in the world. This is essentially a simple mapping from the action declared to execution of the action

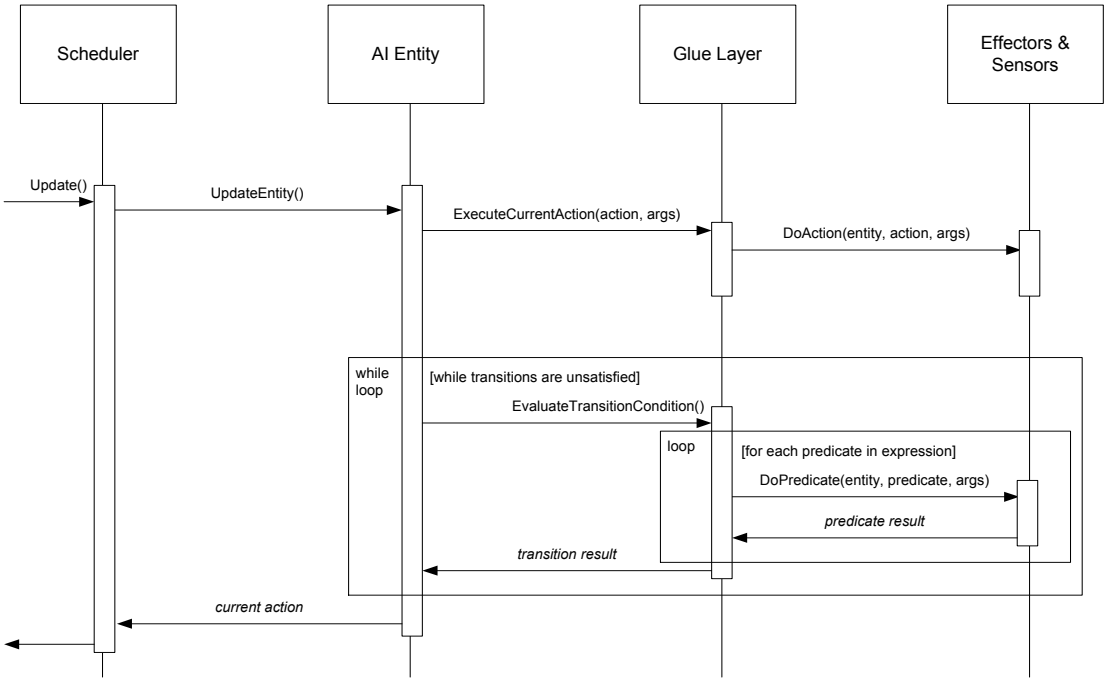

Figure 2- Sequence Diagram of Entity and World Interface

in the world. This function must be defined by the developer.

The lower portion of Figure 2 shows the interaction to calculate the next action via expression evaluation. This is considerably more complicated as there are transitions in each stack frame, interrupt transitions, variable settings, potentially several expressions to evaluate, and behaviors to push onto the stack. The engine searches for the next valid transition by examining interrupt transitions, transitions from the current behavior on the lower stack frames, and transitions from the current action on the top most frame. For each condition in a transition, there are likely to be calls to entity sensors, comparison operators, references to variable values, or numbers. For entity sensors, there must be a "DoPredicate" function defined which is an analog of the function for action by mapping from the sensor declared to execution of the perception in the world. Incidentally, for Java-based entities or simulations, it is easy to connect effectors and sensors to the runtime engine by assigning the appropriate class methods. This affords the developer the ability to make a direct mapping in the authoring tool instead of writing the two dispatch functions separately.

4. AUTHORING APPLICATION

The authoring tool's job is to enable users to define the vocabulary of actions and conditions. Using those as primitive building blocks, behavior graphs can be constructed. These behavior graphs can be indexed via a descriptor hierarchy for polymorphic selection of behavior. Behaviors themselves can invoke each other. Figure 3 is a screenshot of the authoring tool. The left most project pane shows the vocabulary of conditions and actions as well as the behaviors. Clicking on a behavior shows its corresponding definition in the canvas pane to the right. The lower output pane shows status information.

The project pane has two tabs: catalog and descriptors. The catalog hosts a palette of condition and action declarations, the behaviors built, and any variables for the entity. The author uses this pane to declare, modify, or use items. The descriptor tab, whose contents are not shown here, enables the author to create several tree hierarchies. For each behavior created, its initial index will be the root of each tree. The author can create specializations by creating or re-indexing a behavior according to the descriptors in the trees.

The canvas pane shows the definition of a behavior which consists of rectangles, directed lines, and ovals. Computationally, rectangles correspond to states in an FSM, while ovals correspond to conditions for transitions. Each rectangle contains a reference to an action or behavior. References to behaviors appear as bold, outlined rectangles. Anything appearing in parentheses is a parameter. Conditions are logical formulas that evaluate to true or false. Numbers on directed lines determine the order of evaluation of conditions. Dashed lines denote an interrupt.

There are three types of states when interpreting a behavior at runtime. The current state denotes the action or behavior the entity is currently carrying out; a behavior can have exactly one current state at a time. The initial state is simply the rectangle in which the behavior starts. There can be only one initial state per behavior. When a final state is reached,

we consider the behavior to have finished. An action appearing in a rectangle will interact with the game engine through the interface layer.

Figure 3 shows a simple behavior definition for MoveToLocationTimed which moves the controlled entity towards a location. The behavior finishes when the entity either reaches the location or times out. The green rectangle is the starting rectangle. The text in brackets are variables being assigned as a result of expression evaluations. There is no action. The main action of the behavior is MoveToLocation. The first transition evaluated checks whether the entity's current location is within an acceptable distance to the location. If so, there will be a transition to the StopMovement action. If not, there is an alternate path where the engine evaluates the second transition to check whether the current time exceeds the allotted time. If neither of the two transitions are satisfied, then the variable holding the distance to the location is updated, and the MoveToLocation action will remain the current action; i.e., the next action to be executed. After the StopMovement action the next action is a final or accepting state which indicates the end of the behavior.

At the bottom of the canvas are two tabs. Each tab shows the indices chosen from the descriptor hierarchy. The tab on the left is the "default" behavior as its indices are the respective roots. The other tab, whose contents we discussed, shows an index to a "Relaxed" combat state and an "Inexperienced" experience level.

The output pane shows status information such as: results from compilation, interactive debugger information, and results from searching.

6. RELATED WORK

SimBionic has enjoyed continuous development since 1996 and has been fielded in a wide range of software systems, including military simulation and training systems. Its origin has two roots: a Navy training system and robotics research. In 1996, a Navy project [5] to develop a training system for tactical action officers began. This project included requirements not only on the behavior of friendly, hostile, and neutral force, but also on the evaluation criteria of student actions within the simulation. The resulting software to satisfy these requirements was a simple state machine editor which featured variable assignment, but lacked the other three augmentations mentioned in this paper. Entity behavior was authored for all forces, while the evaluations were the result of creating a state machine to process event information from a simulation run. Training principles would be tested against student performance.

In 2000, SimBionic's editor and runtime engine received numerous augmentations, such as stack-based execution. The work was informed by robotics research of the era which included three tier sequencers such as RAPs [6] and Hap [7]. RAPs employs task networks where each network strives to reach a success criterion. It is assumed that a planner has resolved any thorny interactions among goals. The focus, then, is on robust execution because the environment dynamically changes and the entire world state is not always known. Robust execution must be adaptive to those changes as well as new sensor information. Hap shares several architectural similarities, but was used for interactive fiction and virtual reality. SimBionic's focus has been on simulation and decision logic for training.

There are several similarities between each system's architectural organization. RAPs uses "tasks" and "methods" as its main organization while Hap uses "goals" and "plans" instead. Here, tasks and goals are used as references for invocation while methods and plans describe what should be done to accomplish the task or goal. SimBionic uses behaviors for its main organization with the corresponding visual definition as its method or plan. The visual definition makes fewer constraints on the organization of a behavior. For example, a cycle can easily be constructed in a behavior, while both RAPs and Hap specify partially ordered plans (directed acyclic graphs) instead. The partial ordering allows for arbitrary execution of some plan steps. For a robotic system where several pieces of hardware are operating concurrently, parallel execution is a nice benefit. In SimBionic there is no direct support for parallel execution though in practice one can create multiple execution stacks and rely on built-in communications actions to coordinate activity. For example, the ability to move is often best handled as a dedicated behavior while the decision logic of where to move or what to do while moving is handled in one or more separate behaviors.

The notion of behavior polymorphism can be loosely compared to the context conditions which indicate the method or plan's applicability to a given situation. However, SimBionic is agnostic as this is just one potential way of deciding a behavior's applicability.

Although the origin of RAPs started with a simulated "Truck World", it has enjoyed deployment in actual robots [8] as well as Deep Space 1 [9]. Hap was created for the Oz project. SimBionic, though it started in a Navy training system for both entity control and student evaluation, has evolved into more of a generic design and development tool, as exemplified by several deployed systems ranging from teaching students how to interpret NASA satellite data, to research testbeds for examining the military effectiveness of a given command and control structure.

Another deep vein of related work exists in the construction of embedded controllers (e.g., [11]). Frequently, FSM's are routinely used to articulate the control logic. Two example languages are Argos [12] and Esterel [13]. They provide a

combination of graphical and textual FSM representations. The emphasis is on the construction, simplification, and formal verification of controllers which can provably work.

7. CONCLUSION

We described an AI authoring tool SimBionic: its visual depiction of behavior, features of behaviors and how they're implemented in an AI runtime engine, and the interface to a simulator or world. Finite-state machines were the original basis for describing behavior, but four major augmentations were created to provide both visual expressiveness and computational efficacy. The augmentations were: expression evaluation, hierarchical or stack-based execution, behavior interrupt handling, and polymorphism. Their subsequent visual depiction was described.

REFERENCES

[1] Ryan Houlette, Dan Fu, and David Ross, "Towards an AI Behavior Toolkit for Games," AAAI 2001 Spring Symposium on AI and Interactive Entertainment, 2001.

[2] Dan Fu and Ryan Houlette, "Putting AI in Entertainment: An AI Authoring Tool for Simulation and Games," IEEE Intelligent Systems, pp 81-84, July-August 2002.

[3] Martin Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, Addison-Wesley, 2000.

[4] Erann Gat, "On Three-Layer Architectures," in Artificial Intelligence and Mobile Robotics, D. Kortenkamp, R. P. Bonnasso and R. Murphy (eds.), AAAI Press, 195—210, 1998.

[5] Richard H. Stottler and LCDR Michael Vinkavich, "Tactical Action Officer Intelligent Tutoring System," Proceedings of the Industry/Interservice, Training, Simulation & Education Conference, 2000.

[6] R. James Firby, "An Investigation into Reactive Planning in Complex Domains," AAAI-87, pp. 202-206, Seattle, WA, 1987.

[7] A. Bryan Loyall and Joseph Bates, "Hap: A Reactive, Adaptive Architecture for Agents," Technical Report CMU-CS-91-147, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1991.

[8] R. James Firby, "Task networks for controlling continuous processes," Proceedings of the Second International Conference on AI Planning Systems, AAAI Press, 1994.

[9] Brian C. Williams and P. Pandurang Nayak, "A Model-based Approach to Reactive Self-Configuring Systems," Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14-16, 1999.

[10] Douglas J. Pearson and John E. Laird, "Redux: Example-driven diagrammatic tools for rapid knowledge acquisition," Behavior Representation in Modeling and Simulation, Washington, D.C., 2004.

[11] Nicholas Halbwachs, Synchronous Programming of Reactive Systems, Springer, 2001.

[12] F. Maraninchi, "The Argos language: Graphical representation of automata and description of reactive systems," IEEE Workshop on Visual Languages, October 1991.

[13] Gérard Berry, "Esterel v7: From Verified Formal Specification to Efficient Industrial Designs," in 8th International Conference, FASE 2005, Held as Part of the Joint Conferences on Theory and Practice of Software, ETAPS 2005, Maura Cerioli (ed.), 2005.

BIOGRAPHY

Dan Fu joined Stottler Henke in 1998 and has worked on several AI systems including authoring tools, wargaming toolsets, immersive training systems, and AI for simulations. Dan was principal investigator on the intelligent agents project to create AI middleware for simulations and videogames. The result was SimBionic, which enables users to graphically author entity behavior for a simulation or videogame. Dan holds a B.S. from Cornell University and a Ph.D. from the University of Chicago, both in computer science.

Ryan Houlette holds a Master's Degree in Computer Science with a concentration in Artificial Intelligence from Stanford University and a Bachelor of Arts Degree in Computer Science from DePauw University. He has been with Stottler Henke since 1998 and is the lead architect of the SimBionic product line. He currently manages a project develop a next generation adversary modeling engine for the U.S. Air Force.

Jeremy Ludwig joined Stottler Henke in the fall of 2000 after completing his Master's Degree in Computer Science, with a concentration in Intelligent Systems, at the University of Pittsburgh. Currently, Mr. Ludwig is the technical lead on a simulation construction toolkit called SimVentive. He is also the technical lead for a simulation and training project, currently deployed at NAS North Island, for the Navy's common cockpit helicopter. Other research includes building cognitive models with the ACT-R, EPIC, and Soar cognitive architectures as well as simple models in CogTool.

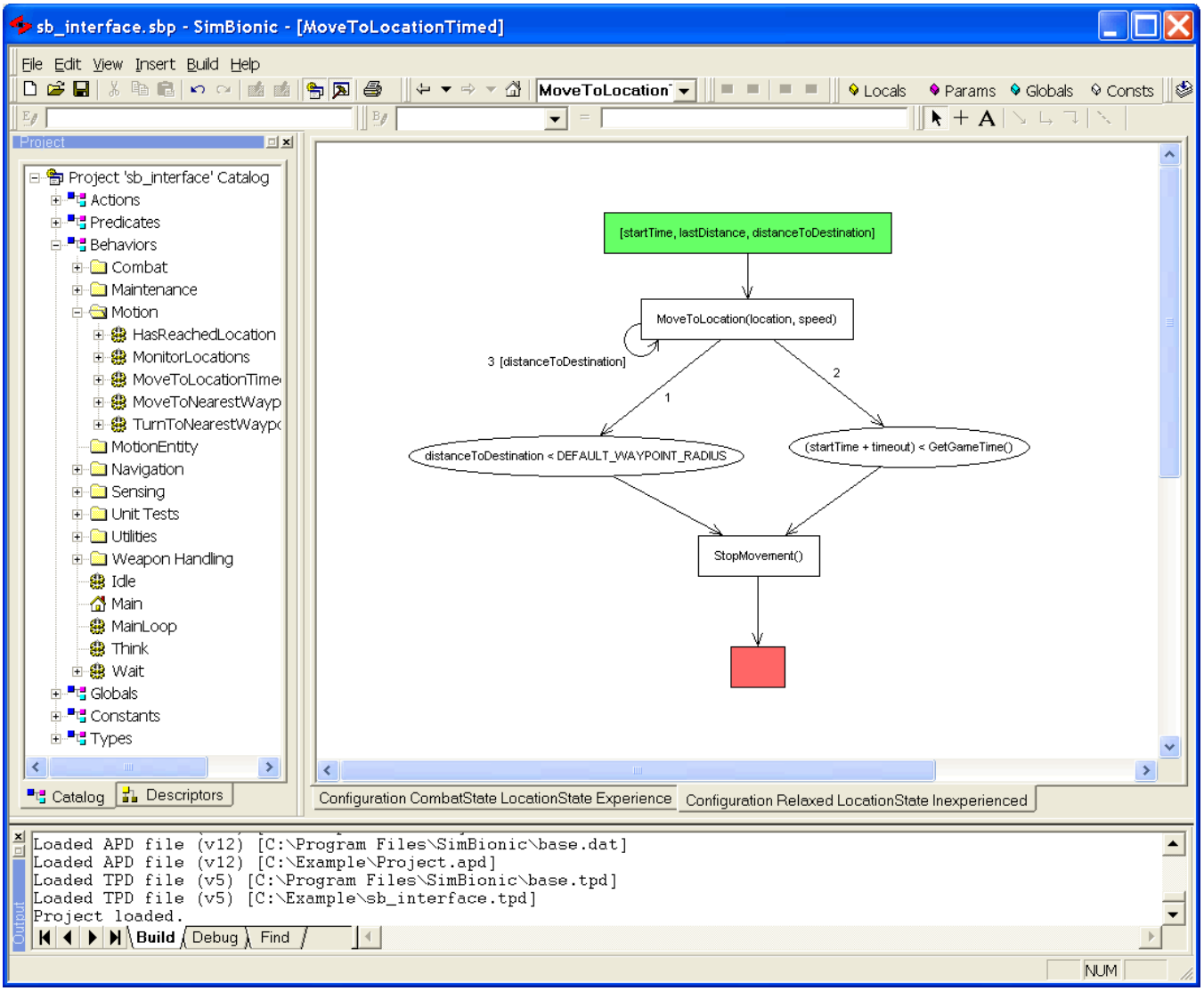

Figure 3- Screenshot of Authoring Tool