In Artificial Intelligence and Interactive Digital Entertainment (AIIDE-2008), Stanford University, California

An Intelligent IDE for Behavior Authoring in Real-Time Strategy Games

Suhas Virmani and **Yatin Kanetkar** and **Manish Mehta** and **Santiago Ontañón** and **Ashwin Ram**
Cognitive Computing Lab (CCL)
College of Computing, Georgia Institute of Technology
Atlanta, Georgia, USA
{svirmani3,yatin.kanetkar}@mail.gatech.edu {mehtama1, santi, ashwin}@cc.gatech.edu

Abstract

Behavior authoring for computer games involves writing behaviors in a programming language and then iteratively refining them by detecting issues with them. The main bottlenecks are a) the effort required to author the behaviors and b) the revision cycle as, for most games, it is practically impossible to write a behavior for the computer game AI in a single attempt. The main problem is that the current development environments (IDE) are typically mere text editors that can only help the author by pointing out syntactical errors. In this paper we present an *intelligent IDE* (iIDE) that has the following capabilities: it allows the author to program initial versions of the behaviors through demonstration, presents visualizations of behavior execution for revision, lets the author define failure conditions on the existing behavior set, and select appropriate fixes for the failure conditions to correct the behaviors. We describe the underlying techniques that support these capabilities inside our implemented iIDE and the future steps that need to be carried out to improve the iIDE.We also provide details on a preliminary user study showing how the new features inside the iIDE can help authors in behavior authoring and debugging in a real time strategy game.

Introduction

Behavior authoring for computer games consists of first writing the behaviors in a programming language, iteratively refining these behaviors, testing the revisions by executing them, identifying new problems and then refining the behaviors again. One of the bottlenecks in creating a game AI is the great deal of engineering effort on the part of game developers, as the rich nature of game worlds makes it hard to plan for all possible scenarios. Another bottleneck in behavior authoring is the revision cycle, as it is practically impossible to write a behavior for the computer game AI in a single attempt. Traditional programming environments offer debuggers that can help with identifying the cause of some of the issues of an observed behavior failure. However, debuggers do not completely solve the problem, and the authoring bottleneck remains there. The main problem is that the development environment (IDE) is typically a mere editor that can only help the author by pointing out syntactical errors. In order to define next generation authoring environments, it is necessary to define intelligent IDEs that have some understanding of the task the author is attempting, in order to assist him beyond mere syntactical checking by allowing easy authoring of the behaviors and offering revision support.

In this paper we will present our ideas to achieve these objectives through an *intelligent IDEs* (iIDE), and describe a prototype iIDE where we have implemented techniques to achieve these objectives. Our iIDE can assist behavior authoring in a much deeper way than current state-of-the-art IDEs. The next generation of IDE's should be able to understand what task the author is trying to perform, and should be able to assist him better than current generation IDEs. In particular, in the iIDE reported in this paper, we have incorporated several key functionalities into a standard IDE to try and achieve this objective. These functionalities are namely: defining behaviors by demonstration, visualization of the behavior execution, failure detection and definition, and proposing behavior fixes for behavior failures. The system supports these activities by understanding the author's goals in defining the behaviors and assists him by automatically recognizing when a behavior is not achieving the desired goals. In particular, the iIDE allows the game developer to specify initial versions of the required AI behaviors by demonstrating them instead of having to explicitly code them. The iIDE observes these demonstrations and automatically learns behaviors from them. Then, at runtime, the system monitors the performance of these learned behaviors that are executed. The system allows the author to define new failure patterns on the executed behavior set, checks for pre-defined failure patterns and suggests appropriate revisions to correct failed behaviors. This approach to allow definition of possible failures with the behaviors, detecting them at run-time and proposing and allowing a fix selection for the failed conditions, enables the author to define potential failures within the learnt behaviors and revise them in response to things that went wrong during execution.

The rest of the paper is organized as follows. We start by first enumerating the key goals of our research and the iIDE, and the approach we have taken to achieve those goals.We then introduce a particular real time strategy game, WAR-

GUS (which we used in the user evaluation) together with the behavior execution environment, Darmok, used in our experiments. Next, we explain the key functionalities of the iIDE with emphasis on the AI techniques required by the iIDE to achieve its goals. Finally, we present a preliminary user evaluation of the iIDE that tests the iIDE in relation to its goals and suggests some future improvements and direction for the iIDE.

Goals of the iIDE

The iIDE aims at providing the user with full control over the behavior authoring and revision process, while assisting as much as possible. We have set the folowing goals for the iIDE and have used the following approaches in order to achieve them.

- *Easy Authoring of Initial Behavior set* Behavior authoring is ultimately a programming task, and as such is non-trivial when the set of behaviors that need to be authored is complex. Although several approaches have attempted to ease the task by defining graphical tools and other easy to use languages, their success in easing the task has been limited. The iIDE should allow the author to easily define these behaviors and reduce the authoring time.

  Approach: The iIDE uses a programming by demonstration approach where instead of editing behaviors, the author can simply demonstrate them and the system will automatically generate code that when executed, exhibits the demonstrated behavior. These techniques were previously developed by us in the context of real-time strategy games (Ontanon *et al.* 2007b), (Ashwin Ram & Mehta 2007). This is similar to GoCap (Alexander), which uses Machine Learning to learn behavior rules.

- *Behavior Execution Visualization and Debugging*: the iIDE should allow the author to visually see the result of executing behaviors, modify the internals of the behaviors based on their execution performance and propagate the modified change to the behavior library. The iIDE should further allow the user to iteratively perform revisions on the behaviors.

  Approach: The iIDE presents the results of the executing behaviors in a graphical format, where the author can view their progress and change them. The author can also pause and fast-forward the game to whichever point he chooses while running the behaviors, make a change in the behaviors if required and start it up again with the new behaviors to see the performance of the revised behaviors. The capability of the iIDE to fast forward and start from a particular point, further allows the author to easily replicate a possible bug late in the game execution and debug it.

- *Failure Detection and Definition and Proposing Fixes*: The iIDE system should allow the author to define new failure conditions based on monitoring the performance of the existing behavior set, allow detection of author defined failure conditions and provide appropriate suggestions for fixing the failed conditions.

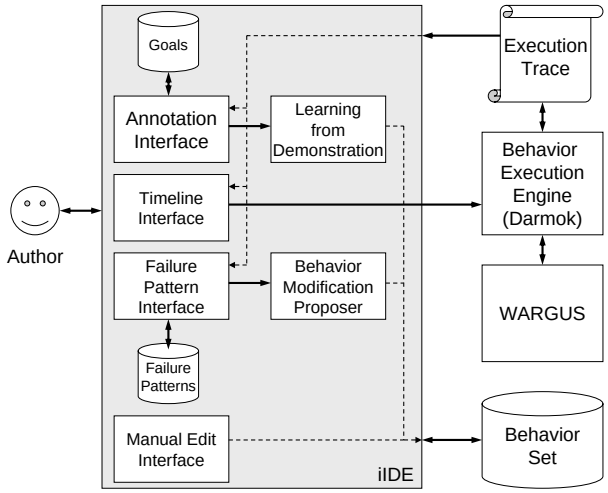

Figure 1: Overview of how the iIDE interacts with the author and the game.

Approach: The iIDE authoring tool allows the author to visualize relevant chronological events from a game execution trace. The data allows the author define new *failure patterns* by defining combinations of these basic events and pre-existing failure conditions. Each failure pattern is associated with a possible fix. A fix is basically a proposed modification for a behavior that fixes the error detected by the failure pattern. When a failure pattern is detected, the iIDE suggests a list of possible fixes, from which the author can select an appropriate one to correct the failed behavior. These techniques were also previously developed by us in the context of believable characters (Zang *et al.* 2007).

In order to achieve these goals, the iIDE has to interact with the author, the game (WARGUS) and with the behavior execution engine (Darmok). Figure 1 shows an overview of how all those components fit together to allow the author to edit a proper behavior set for the game. The iIDE controls Darmok by sending the behaviors that the author is creating. Darmok then, runs the behaviors in the game, and generates a trace of what happened during execution. This trace is sent back to the iIDE so that proper information can be shown to the author. The remainder of this paper presents first WARGUS, then Darmok and finally we focus on how the iIDE works.

WARGUS

WARGUS is a real-time strategy game where each player's goal is to remain alive after defeating the other players (WARGUS is an open source implementation of WARCRAFT II). Each player has to maintain a colony of peasants, buildings and attack units, which he needs to protect from the enemy. He needs to continuously collect resources, which include wood, gold and oil, to build or train units.Buildings are required to upgrade attack units in order to fight the enemy more effectively. Players can also build defensive buildings such as walls and towers to protect his

colony. Therefore, it is easy to see that WARGUS will require complex reasoning to formulate an effective strategy.

Traditionally, games such as WARGUS used handcrafted behaviors for the built-in AI. Creating such behaviors requires a lot of effort, and even after that the result is that the built-in AI is static and has flaws which can be exploited, making it relatively easy to defeat.

Darmok

The Darmok system (Ontañón *et al.* 2007a) is a case-based planning system designed to play the game of WARGUS. Darmok learns plans (behavios) by observing a human playing the game, and then reuses such plans combining and adapting them to play new games using case-based planning methods. Darmok's architecture can be broken down into two parts: behavior acquisition and execution. The behavior acquisition process is performed in the following way. Each time someone plays a game, a trace is generated (containing a list of actions performed by the player) then a human annotates that trace stating which goals he was pursuing with each action. This annotated trace is processed by a plan learning module that extracts behaviors in the form of cases from the trace.

Thus, the case base of Darmok is composed of behaviors, and each behavior has an associated situation(episode) in which this behavior succeeded in the past. Behaviors are learnt from traces, and episodes can be learnt either from traces or from experience. During execution the system maintains a current plan that is executing. Each plan is composed of a collection of behaviors retrieved from the case base, combined together through plan expansion, plan adaptation and behavior retrieval modules. Each time the plan expansion wants to expand a goal, it asks the behavior retrieval module for a behavior.The behavior retrieval module selects the best behavior for the goal at hand in the current game state. This behavior then goes through the plan adaptation module folowing which it is inserted into the current plan for execution.

In this paper we will use Darmok as our behavior execution engine. The iIDE has been fully integrated with Darmok and allows the author to perform some Darmok specific operations, such as annotating traces in order to extract behaviors automatically. The iIDE can be used to create behaviors by hand, or to automatically modify the behavior set that Darmok has learnt by analyzing human traces. As we will show in the empirical evaluation section, even if Darmok is a learning system that can learn behavior on its own, we have shown that using an iIDE together with Darmok greatly improves the quality of the behaviors since the author is given full control over the behavior set that Darmok executes.

An Intelligent Integrated Development Enviroment for Darmok

In this section we will present how the iIDE works internally. We will describe the architecture of the iIDE and explain how it performs the various tasks which we mentioned earlier. When WARGUS is played by Darmok, Darmok records an execution trace (as shown in Figure 1). This trace contains all the information needed by the iIDE to perform its functions. A trace consists of a series of entries, where each entry is composed of the followong elements: a) A **time stamp**, b)The **game state** corresponding to that time, which is an xml description of the full game state, c) a **screenshot** of the game at that time (to be shown to the user as a graphical representation of the game state), d) a snapshot of the **goal tree** of Darmok at the time stamp. This contains a list of which behaviors were selected to execute each goal that Darmok was trying to pursue, and their execution status (ie. executing, succeeded or failed), and e)**basic event information** which consists of a recording of basic events that occured during the game. This event information consists of whether a particular behavior has started, and if so, whether it succeeded or failed. This information is useful for detecting failures during execution.

The iIDE uses this trace to represent the behavior tree, graphically, over a time-line. A user can zoom in to any portion of the goal tree to focus on a particular goal or zoom out to get a better understanding of the behavior as a whole. In the rest of this section we will first present an overview of the iIDE architecture, and then explain how we addressed each goal.

iIDE Architecture

As Figure 1 shows, the iIDE is composed of 4 main interfaces which the author interacts with: the *Annotation Interface*, the *Time-line Interface*, the *Failure Pattern Interface*, and the *Manual Edit Interface*.

The *Annotation Interface* allows the author to access the learning from demonstration capabilities of the iIDE. For doing so, the author runs the game and plays it. Then the interface loads an execution trace and will allow the author to annotate the actions in the trace with a set of goals (ie. associates particular actions to particular goals). Goals are things like "I wanted to destroy an enemy tower" or "I was building a base", etc. The annotated trace is passed on to the learning from demonstration module which generates behaviors to be stored in the current behavior set.

The *Time-line Interface* reads the execution trace generated after executing behaviors and displays it in a tree format as explained in Figure 2).

The *Failure Pattern Interface* shows the author where pre defined failure patterns occurred during the run of the game, it allows the author to create new failure patterns, and provides access to fixes proposed by the behavior modification section of the iIDE.

Finally the *Manual Edit Interface* is a classic text editor interface (the standard view in traditional IDEs) that allows the author to manually edit the behavior set.

Let us present each one of the functionalities of the iIDE in detail.

Behavior Learning from Human Demonstration

Automatic behavior learning can be performed via the analysis of human demonstrations. The techniques presented in this section differ from the classical numerical machine learning approach that require lots of training examples to learn an appropriate behavior. The key idea that makes it

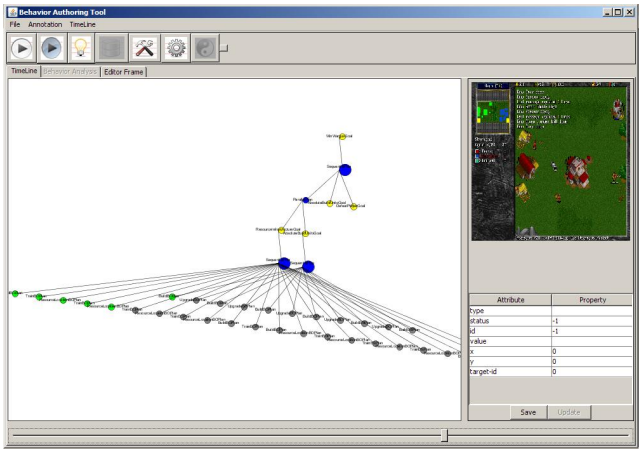

Figure 2: A screenshot of the behavior execution visualization. Showing the time-line in the bottom, the game state on the right, and the behavior execution tree on the left.

possible to learn behaviors from a single demonstration is *annotation*.

The annotation process involves a human expert providing the iIDE with information regarding the goal being pursued, for each action performed during the demonstration. An annotated demonstration contains much more information than a simple un-annotated one and provides the data that is required to automatically learn behaviors. During annotation the author simply labels each action in the trace with the goal or goals that he was pursuing at the time. More advanced approaches could be developed that might involve AI plan recognition techniques inferring the annotations automatically from observed actions, but these are still at the research stage.

In order to learn behaviors, an annotated trace is analyzed to determine the temporal relations between the individual goals that appear in the trace, such as "the goal g2 was attempted before the goal g3" or "the goal g3 was attempted in parallel with the goal g4". The analysis needed to detect the temporal ordering is based on a simplified version of the temporal reasoning framework presented by Allen (Allen 1983), where 13 basic different temporal relations among events were identified. For our purposes, the temporal reasoning helps to figure out if two goals are being pursued in sequence, in parallel, or if one is a subgoal of the other. We assume that if the temporal relation between a particular goal g and another goal g' is such that g happens during g', then g is a subgoal of g'. In our implementation of the iIDE we incorporated the behavior learning by demonstration capabilities of Darmok, see (Ontañón *et al.* 2007a) for a more detailed explanation of how this process works.

Behavior Execution Visualization and Debugging

The iIDE tries to make the process of behavior authoring as simple as possible by allowing the author to easily understand the behaviors by displaying them in a hierarchical goal tree (see Figure 2). The user can see the effect of a particular plan or change thereof in the game using the hierarchical goal tree representation. The user can see points where the plan starts and succeeds/fails. This display is dynamic in nature and presents the currently executing plan along with whether any of the previous plans had failed. The user is provided with a timeline representation to depict the extent of time that has passed to give him an idea of how far into the game he wants to move forward. The slider can be controlled at will (located in the bottom of the screen, as Figure 2 shows). Further, screenshots at regular intervals give the user the approximate state at that corresponding instant of time (right hand side of Figure 2). This helps the user easily recognize the game state. The users can modify the internals of the behaviors based on their execution performance and propagate the modified change to the behavior library.

The time line representation inside the iIDE further allows the user to iteratively perform revisions on the behaviors. The user can pause and forward the game to whichever point he chooses, make a change in the behaviors if required and start it up again with the new behaviors loaded. The ability to forward the game to any point also allows the user to easily replicate a possible bug late in the game execution and debug it. For example, if a bug occurs at some point late in the game (say cycle 15000), it would be very difficult to replicate it by restarting the game from the beginning. The fast forwarding feature of the iIDE allows the user to forward the game easily to the point (in this case to a point near cycle 15000) where the bug occured and replicate it.

Failure Detection and Behavior Modification

In order to support the goals of detecting existing failure patterns, definition of new ones and suggesting possible fixes within the iIDE, a reasoning layer carries out appropriate processing in the background. The reasoning layer inside the iIDE consists of two components. The first component identifies basic events and failures in behavior execution, and detects whether the existing behaviors have achieved their specified goals. These goals are specified in terms of author defined desired effects of completing a behavior and constraint conditions that should be satisfied during the game execution (for example, a peasant not being idle beyond a certain amount of time). When a goal that should have been satisfied by the execution of a set of behaviors has not been achieved, the system uses the execution trace to perform blame assignment, which aims to identify one or more behaviors that should be changed. The second component inside the reasoning layer involves suggesting fixing operators (called modops) so as to allow the user to select a possible repair for the offending behaviors identified during blame assignment. The applicability of a modop depends on the type of failure that was encountered and modops are therefore categorized according to failure patterns. These modops are in the form of inserting a new appropriate behavior at the correct position in the failed behavior, or removing some steps. Once these modops have been applied to produce modified behaviors, the modified behaviors are added to the behavior library.

The iIDE exposes the capabilities of the reasoning layer to the user by allowing the user to create new failure pat-

terns (by combinations of basic events and existing failure patterns) and assign corresponding fixes for them (by suggesting a set of plans). In order to understand the whole process better, let us look at each of the steps in more detail.

Recording the trace of executing behaviors The Darmok system during its run records an execution trace (as shown in Figure 1). Once a game is finished, an abstracted version of the execution trace is used by the iIDE to present basic events to the user who can then use this data to find interesting patterns of failures in the system. The abstracted version of trace, consists of, a) Unit Data: information regarding units such as hit points, status (whether a unit is idle, for example), location, etc., b) Idle Data: which units were idle and the cycle intervals for which they were idle. c) Kill Data: the cycles at which particular units were attacked and killed d) Resource Data: for each entry in the trace, the corresponding resources that were available, such as the amount of food, gold, wood and oil e) Attack Data: the units that were involved in an attack. For example, the enemy units that were attacking and the AI units that were under attack and f) Basic Behavior Failure Data: the reason for behavior failures, such as whether it was due to insufficient resources or not having a particular unit available to carry out a behavior.

Failure Patterns and Fixes This abstracted version of the trace can be used to detect different failures that occurred at run time and suggest appropriate fixes for them. These failures are defined using a combination of basic elements of the abstracted trace. Some examples of such failure patterns that have been already defined in the system are:

a) *Peasant Idle failure*: detects if peasants have been idle beyond a certain number of cycles. If a peasant has been idle for a sufficiently long time, then he is not being utilized properly by the existing behavior set in the behavior library. A possible fix for this failure is to utilize the peasant to gather more resources or use the peasant to create a building that could be needed later on.

b) *Building Idle failure*: detects whether a particular building type has been free for certain number of cycles even though there were sufficient resources available to make use of it. A possible fix for this failure is to utilize the free building. For example, "barracks" can be used for developing "footmen" if sufficient resources are available.

c) *Peasant attacked military units idle failure*: detects whether the player's military units were idle while a peasant or a building was under attack. One of the fixes for this failure type can be in the form of inserting a basic action that issues an attack command to the attacking units to attack.

d) *Basic Operator failures* detect when behaviors are failing due to the "preconditions", "alive conditions" or "success conditions" not being met at run time. For example, the preconditions for a behavior for building a farm or barracks could fail due to the lack of availability of resources or a peasant being unavailable to carry out the plan. These types of failures differ from the above two because they are direct failures caused by conditions specified as part of the behavior itself. The failures mentioned in a), b) and c) are based on detecting patterns of events from the execution trace. Basic Operator failures are fixed by adding a basic action that fixes the failed condition. For example, if the condition failed due to limited availability of resources, a basic action is added to gather the corresponding resource. Similarly if no peasant was available to build a farm, then another basic action to train the peasant is added before the failed behavior.

In the current system, we have defined 4 failure patterns and 9 fixes. The user can easily extend the modification system by writing other patterns of failure (using the failure pattern interface inside the iIDE) based on the basic event data from the trace that are used to define existing failure patterns. The user can further select the appropriate system suggested fixes to the corresponding behaviors. These fixes need to be carried out in order to correct the failed situation. Once the user has selected a fix, it is applied to the behavior and the modified behavior is loaded into the behavior library.

Preliminary Evaluation

In order to get some initial insights on the usefulness of the iIDE, we performed a preliminary user study where 3 users used the iIDE to author behaviors for the WARGUS game. These users has been programming behaviors for wargus for about a year. In order to evaluate and cover the core functionalities of the iIDE, we made the users execute a series of use cases, as listed below:

Use Case 1: Demonstration of Behaviors. In this use case, the users think of a strategy that they want to encode in form of behaviors and play a game of WARGUS using that strategy. Then they use the iIDE to annotate the resulting trace and save the resulting behaviors to disk.

Use Case 2: Run the Behaviors. In this use case, the users took the behaviors generated in use case 1, ran them, and they browsed the execution trace in the graphical visualization view of the iIDE.

Use Case 3: Replicate Problems. The users load a behavior set that has problems (unfortunately, they lose the game because of the problematic behavior). They can execute the behaviors, identify the problem during the run time of the game, and then use the timeline to fast forward to the point to view the problem again.

Use Case 4: Edit Behaviors By Hand. The users open a set of behaviors and edit them by hand as they would do using any standard IDE.

Use Case 5: Edit through Failure Patterns. The user uses the iIDE to match an execution trace with the set of failure patterns that iIDE has in its data base. The iIDE identifies problems in the current behavior set and proposes fixes.

During the user interaction, a researcher logged his observations of user actions and any unusual reactions. The

users were asked a series of open ended questions regarding their experiences after each use case and were asked for their feedback on the system as a whole at the end of all the use cases. Their feedback provided us with useful inputs regarding future improvements to the iIDE and its different functionalities. We present the results from our analysis and interviews categorized according to iIDE's different capabilities.

Annotation Tool and Possible Improvements Users felt that authoring behaviors by demonstration was much more convenient than writing out behaviors through coding in a programming language. Apart from these opinions of the user, we would also like to remark on the the low time required to generate behaviors with our system to play in a particular map (versus the time required to write a handcrafted behavior to play the same map) using the annotation capability. Specifically, to record a trace an expert has to play a complete game (that takes between 10 and 15 minutes in the maps we used) and then annotate it (to annotate our traces, the expert required about 10 minutes per trace). Therefore, in 20 to 25 minutes it is possible to generate a behaviors to play a set of WARGUS maps similar to the one where the trace was recorded. In contrast, one of our students required several weeks to hand code a strategy to play WARGUS at the same level of play.

During the annotation process, users felt that it was difficult to remember the actions they had performed, so at times, it was difficult to annotate actions and find the correlation of the action with the goals. The users suggested that it would be helpful to have annotation ability while playing the game or the ability to set markers for the actions while playing in order to help him correlate later. Another possibility would be to have a screenshot (as used in behavior visualization) to provide a cue to the user regarding the state of the game before the action. The users felt that the fixed nature of the goals for annotation was a hindrance and it would be useful to have a way of defining new goals.

Behavior Visualization and Debugging Users felt that using a time-line to fast forward to a particular point, visualizing the behavior tree at that point, making changes to it and then re-running the game was very effective in debugging a particular behavior. It was very helpful in saving time as the user was not required to play the game all over again in order to re-annotate, change behaviors or replicate a bug. Screenshots of the game provided cues to the user to discover the right point to fast forward. Users felt the need for dynamic node removal and addition capability in the hierarchical representation (currently unsupported). They further suggested that it would be useful to integrate the annotation process with graphical visualization of behaviors and the time-line as it would allow them to simply add nodes and edges to the goal tree, based on the actions performed instead of manually annotating the goals.

Failure Pattern and Fix Detection Users had a little difficulty in using this feature as it was still at an early stage of development. A visualization of failure patterns helped the user discover new failure locations inside the game which they might not have been able to recognize if they weren't given the graph display. Users felt that the requirement that a failure pattern should occur inside the game in order to be able to define it, was a setback. As, they could think of simple failure patterns which they would like to add without having to even run the game. The preliminary user evaluation has helped us think of some changes that could be incorporated into the iIDE. We aim to improve the iIDE with these features and carry out a more comprehensive user study in the future.

Conclusions and Future Work

The main bottlenecks in behavior authoring for a computer game are the effort required to write an initial version of behaviors and the revision cycle, as it is practically impossible to write a behavior for the computer game AI in a single attempt. In this paper, we have presented techniques for an *intelligent IDE* that addresses these issues and has the following capabilities: it allows the author to get initial versions of the behaviors through demonstration, presents visualization of behavior execution for revision, lets the author define new failure conditions on the existing behavior set, and select appropriate fixes for the failure conditions to correct the behaviors. A preliminary user evaluation has helped us gather some improvements that could be carried for the iIDE.

As future work, we plan to incorporate the improvements suggested by our participants into our iIDE. We also plan on conducting a comprehensive user study in the future comparing our iIDE with a standard IDE. Finally, we would like to plug our iIDE to other games, in order to investigate how the techniques included in it could be made domain independent.

References

Alexander, T. Gocap: Game observation capture. chapter 11.3.

Allen, J. 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM* 26(11):832–843.

Ashwin Ram, S. O., and Mehta, M. 2007. Artificial intelligence for adaptive computer games. In *FLAIRS Conference on Artificial Intelligence*.

Ontañón, S.; Mishra, K.; Sugandh, N.; and Ram, A. 2007a. Case-based planning and execution for real-time strategy games. In *Proceedings of ICCBR-2007*, 164–178.

Ontanon, S.; Mishra, K.; Sugandh, N.; and Ram, A. 2007b. Case-based planning and execution for real-time strategy games. In *International Conference on Case-Based Reasoning ICCBR-2007*.

Zang, P.; Mehta, M.; Mateas, M.; and Ram, A. 2007. Towards runtime behavior adaptation for embodied characters. In *International Joint Conference on Artificial Intelligence*.