Adaptive Computer Games: Easing the Authorial Burden

Manish Mehta, Santi Ontañón, Ashwin Ram —Georgia Institute of Technology
{mehtama1, santi, ashwin}@cc.gatech.edu

Introduction

Game designers usually create AI behaviors by writing scripts that describe the reactions to all imaginable circumstances within the confines of the game world. The AI Programming Wisdom series [rabin02, rabin04] provides a good overview of current scripting techniques used in the game industry. Scripting is expensive and it's hard to plan. So, behaviors could be repetitive (resulting in breaking the atmosphere) or behaviors could fail to achieve their desired purpose. On one hand, creating AI with a rich behavior set requires a great deal of engineering effort on the part of game developers. On the other hand, the rich and dynamic nature of game worlds makes it hard to imagine and plan for all possible scenarios. When behaviors fail to achieve their desired purpose, the game AI is unable to identify such failure and will continue executing them. The techniques described in this article specifically deal with these issues.

Behavior (or script) creation for computer games typically involves two steps: a) generating a first version of behaviors using a programming language, b) debugging and adapting the behavior via experimentation. In this article we present techniques that aim at assisting the author from carrying out these two steps manually. Figure 1 depicts a global view of the architecture that we have developed for an adaptive game AI, showing the two components that automatically carry out those two steps and can be used as part of the behavior authoring workflow to assist the human author. These are labeled **behavior learning** and **behavior adaptation**. Let us present the general idea of these two modules.

In the behavior learning process, the game developers can specify the AI behavior by demonstrating it to the system instead of having to code the behavior using a programming language. The system extracts behaviors from these expert demonstrations and stores them. Then, at performance time, the system retrieves appropriate behaviors observed from the expert and revises them in response to the current situation it is dealing with (i.e., to the current game state)

In the behavior adaptation process, the system monitors the performance of these learned behaviors at runtime. The system keeps track of the status of the executing behaviors, infer from their execution trace what might be wrong, and perform appropriate adaptations to the behaviors once the game is over. This approach to behavior transformation enables the game AI to reflect on the issues in the learnt behaviors from expert demonstration and revises them after post analysis of things that went wrong

during the game. These set of techniques allow non-AI experts to define behaviors through demonstration that can then be adapted to different situations thereby reducing the development effort required to address all contingencies in a complex game.

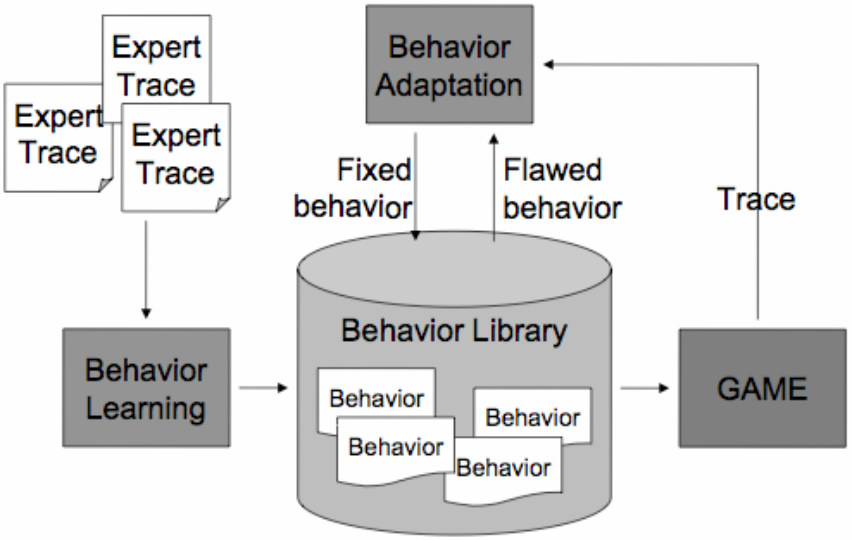

Figure 1: Overview of an automatic behavior learning and adaptation architecture.

The following sections describe these two processes in more detail, and present the design of a behavior representation language in order to support such automatic reasoning. We will also exemplify how the techniques can be applied in the domain of a real-time strategy game, Wargus (an open source version of Warcraft II).

Behavior Representation Language

The basic constituent piece of any behavior language is the *Behavior*. In order to support automatic reasoning over the language, a behavior must have two main parts: a *declarative* part (that tells the AI what a behavior does) and a *procedural* part (that contains the executable code of the behavior). Let us present a particular definition of a language, called BRL (Behavior Representation and Reasoning Language). BRL allows the developer to define three kinds of declarative information. Note that BRL does not require the author to define all this, but the more information is provided, the better the AI will understand the behaviors):

- A *goal* is a representation of the intended goal of the behavior. For each domain, an ontology of possible goals is defined (see below). For instance, a behavior might have the goal of ``building a tower''. The *"goal"* is the only declarative information that is mandatory if we want to use the techniques presented in this article.

- *Context* information that encodes the circumstances under which the behavior makes sense. In BRL, two kind of context information can be defined: *preconditions* and *game state*. Preconditions are a set of conditions that must be true in order to execute the behavior (e.g., an "attack" behavior might have as its preconditions the existence of an army and an enemy). Game state is a more general definition of context, that specifies a particular game state in which the behavior makes sense. This informs the system that the behavior author had a particular game state in mind when writing/demonstrating the behavior, and that the further the current game state is from the one that is defined; the less applicable the behavior is to the current game state.

- A set of *alive-conditions* that represent the conditions that must be satisfied during the execution of the behavior for it to succeed. If at some moment during the execution, these alive-conditions are not met, the behavior is stopped, as it will not achieve its intended goal. For instance, the peasant in charge of building a farm must remain alive for the entire time it takes for the farm to develop. If he is killed, then the farm would not be built.

The procedural part of the behavior consists of executable code. As with any scripting language for computer games, it is necessary to define two extra elements: *sensors* and *actions*. Sensors are the basic variables and tests that the behaviors can use to consult the game state. For instance, we might define a sensor called "path(x,y)" that allows a script to verify if there is a path between the x and y locations. Other sensors might involve getting information about terrain, units and buildings (in the Wargus domain). Actions are the basic actions that our scripts can execute in the world. In our implementation we use all the possible actions available to a human player when playing Wargus, so we have actions such as `"move(unit,x,y)"`, `"build(unit,x,y,building-type)"`, etc.

Our current implementation uses a common way to define sensors, preconditions and alive-conditions as all our sensors are Boolean (although our approach can be easily extended to non-Boolean sensors,). At the implementation level, a global class called `Condition` or `Sensor` is defined that contains a function `test` that checks whether the condition is satisfied or not based on the given game state. By extending the Condition class for each different condition(s) or sensor(s), multiple sensors can be defined. For Wargus, we defined the subclasses `BuildingComplete(unitID)`, `Gold(minGold)`, `Oil(minOil)`, `Wood(minWood)`, `UnitExists(UnitID)`, `And(condition1, condition2)`, `Or(condition1,condition2)`, `not(condition)`, among others.

To enable the AI to properly understand the meaning of a particular behavior, we define an additional construct: a *subgoal*. A goal captures the intention of the author, i.e., represents *what* does he want to do, and a subgoal represents a child goal inside a behavior (that would be satisfied by executing another behavior). There might be different behaviors for achieving the same goal. When the author wants to invoke another

behavior, he inserts a subgoal in the behavior, and the system will decide regarding the most appropriate behavior for that subgoal (based on the goal and context information of the other behaviors). Let us explain in detail the idea of a "goal" to properly understand the process of defining goal ontologies and how subgoals work.

A goal is a particular task that can be achieved in a given domain. For instance, in the Wargus domain, possible goals are "win the game", "destroy a particular player", "build a base", "gather resources", "build a tower", etc. In order to represent such goals, we need to define a *goal ontology*, that is nothing more than a collection of possible goals. If there're relations among goals, the ontology might also contain them so that the system can further reason about them, but to keep it simple in this article we will consider the goal ontology to be a plain list of goals names. Each goal might have parameters, for instance, we can define the goal "BuildBuilding(type)" that represents the goal to build a new building of the specified type. In particular, for our Wargus implementation, we have used the following goal ontology:

- WinWargus(playerToWin)
- DefeatPlayer(playerToDefeat)
- BuildUnits(player,unitType,number,x,y)
- GetResources(player,gold,wood,oil)
- KillUnit(player,unitID)
- KillUnitType(player,enemyPlayer.unitType)
- Research(player,researchType)
- ResourceInfrastructure(player.nFarms,nPeasants)

In addition to the name and parameters of each goal, the AI system needs a way to verify if a particular goal has been achieved or not (as we will see in later sections). Thus, a simple way to implement goals is by by defining a `Goal` class, extend that class for each different goal type we want to define, and implement a function called `generateTest` that returns a `Condition` (built using the `Condition` or `Sensor` classes defined above) capable of testing whether the goal has been achieved. The source code on the companion CD-ROM includes a demo and the definition of our goal ontology.

A final consideration about the behavior representation language is that it should be able to support automatic behavior **modification** and **learning** routines. Thus, the executable code of the behaviors must be represented in structures that are easy to manipulate by the AI. The current implementation represents the behaviors in the form of classes such as *Sequence*, *BasicAction*, *Parallel*, etc., but any other representation of behaviors that the AI can easily manipulate is suitable. The demonstration code accompanying this article uses a simple behavior definition language based on XML syntax. However, any other text syntax that allows the system to load behaviors and store them in the appropriate classes (i.e., the goals, conditions and behavior classes) may be used.

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 requires lots of training examples to learn an appropriate behavior. The key idea to learn behaviors from a single demonstration is *annotation*. The annotation process involves the human expert providing the AI information regarding the goal being pursued at every action executed during the demonstration. An annotated demonstration contains much more information than a plain demonstration and, as discussed later, provides knowledge on how to automatically extract behaviors.

Let us examine the annotation and behavior learning processes in more detail. The output of a human demonstration is an *execution trace* (or *trace* for short). A trace is a list of each action that the human performed in the demonstration. The annotation process simply involves labeling each action in the trace with the goal(s) that the human was pursuing within the game. Annotation process might seem tedious, but can be easily automated in several ways. One way of automating the process is to develop a tool that loads the demonstration and allows the human to associate groups of actions with goals. This is the approach we have currently implemented. A second, and better, way is to develop a small tool that runs alongside the game that allows the human expert to select the goals he is pursuing in real time, thereby providing an annotated trace at the end of the game. More advanced ways might involve AI plan recognition techniques to infer plans and goals automatically from observed actions, but that is out of the scope of this article.

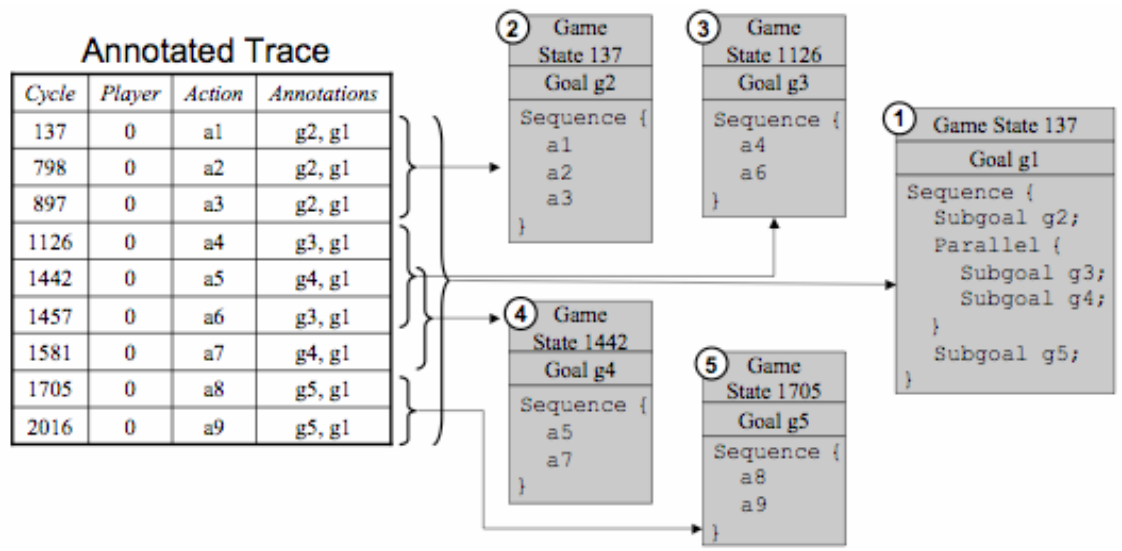

Figure 2: Illustration of 5 behaviors being extracted from a sample annotated trace.

In order to learn behavior, the annotated trace is analyzed to determine the temporal relations among the individual goals appearing in the trace. The left hand side of Figure 2 shows a sample annotated trace, indicating the relationships among the goals, 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 [Allen83],

where 13 basic different temporal relations among events were identified. For our purposes, the temporal reasoning helps to 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 that *g* happens *during g'*, then *g* is a subgoal of *g'*. For instance, in Figure 2, *g2*, *g3*, *g4*, and *g5* happen *during g1*; thus they are considered subgoals of *g1*. In this example, we could imagine that *g1* means "WinWargus", and *g2* to *g5* are individual steps to win the game, such as "build a base", etc. Notice that each action can be annotated with more than one goal and that, in principle, all the actions should be annotated at least with the top-level goal ("WinWargus" in the Wargus domain).

From temporal analysis, procedural descriptions of the behavior of the expert can be extracted. Figure 2 shows the behaviors that are extracted from a sample annotated trace. In particular, 5 behaviors are extracted from the given example trace. Each behavior in Figure 2 consists of three parts: the *game state* a *goal*, and the behavior itself. Notice that preconditions or alive conditions cannot be extracted trivially from the annotated trace, so we simply leave them blank (they could be inferred from a definition of the preconditions and alive conditions of the basic actions of our domain, but this is beyond the scope of this article). For instance, behavior number 1 (shown in Figure 2) can be extracted, specifying that to achieve goal *g1* in the game state at game cycle 137, the expert first tried to achieve goal *g2*, then attempted *g3* and *g4* in parallel, and pursued *g5* after that. Then, a similar analysis is performed for each one of the subgoals, leading to four more behaviors. For example, behavior 3 states that to achieve goal *g2* in that particular game state, basic actions *a4* and *a6* should be executed sequentially.

Note that in our system we don't attempt any kind of generalization of the expert actions. If a particular expert action in the trace is "Build(5,"farm",4,22)", it is stored in exactly the same form without any modifications. When the learned behavior is used to play a new scenario in Wargus, it is likely that the particular values of the parameters in the action are not the most appropriate for the new scenario (for instance, it might be the case that in the new map the coordinates 4, 22 correspond to a water location, and thus a farm cannot be built there). There are two possible ways to solve that issue: the first one is to generalize behaviors beforehand, and simply store that a "farm" has to be built (then we could hand-code some code that selects the best location for it). The second solution, the one we followed, is to use a simple *revision* process based on the Case-Based Reasoning (CBR) paradigm. CBR is an artificial intelligence technique [Aamodt94] based on reusing previously found solutions to solve new problems. Following that approach, the system stores the behaviors as they are, without any modifications and at run time, when the behaviors have to be executed, they are revised appropriately as explained above. The next section explains that process in detail.

Learned Behavior Execution

As explained earlier, a behavior representation language that supports automatic behavior learning and adaptation is richer than a standard behavior language. Subroutine calls are replaced with subgoals, and a behavior revision process is used at runtime. Let us look at these two factors that change the execution of behaviors. Initially the system is started by defining an initial goal (in the Wargus domain, it is "WinWargus"). The system tries to retrieve a behavior for this initial goal from the behavior library. The behavior library provides a behavior that has a similar goal and a similar context to the current game state. The selected behavior is associated with the initial goal. If the selected behavior has any subgoal(s), system recursively performs the same process of retrieving appropriate behaviors for them only when the subgoal needs to executed. The collection of current goals and the associated behaviors is called *the current plan*. Each time a basic action is ready to be executed, the action is sent to the revision subsystem to make it suitable for the current game state.

As discussed above, in order to execute behaviors the system needs three components: behavior retrieval, behavior revision, and plan expansion. We discuss each of them next.

Behavior Retrieval

When the system needs to find a behavior for a particular goal in the current game state, it performs a two step process. First, the system selects all the behaviors whose preconditions are satisfied, and then looks for the behavior with most similar goal and game state. In order to carry this process, similarity metrics among goals and among game states needs to be defined. To assess similarity between game states, a set of features for the map and game state should be computed. These features should capture the key elements in the game, that a human player would consider to decide which actions to perform next. For instance in the Wargus domain, we use features such as: map size, number of wood in the map, amount of gold of the player, number of peasants, etc. In particular, we have defined 35 features that represent the terrain, each player's troops, buildings and the available resources (gold, wood and oil). To compute the similarity between two game states, we compute the 35 features for each of them and compare the values of these features using a simple Euclidean distance.

Assessing similarity between goals is a bit trickier. A simple approach would be to assess the similarity between two goals of different types to be 0, and calculate similarity between goals of the same type based on Euclidean distance of the parameters of the goals. This simple goal similarity computation can be enhanced by making use of the ontology. We can define a base similarity between goal types in the ontology and combine that with the parameter comparison. For instance, we might define that the goal "DefeatPlayer" is more similar to "WinWargus" than to "GatherResources".

Once these two distances have been computed, they are aggregated through a simple average of the two and the behavior with the highest average similarity is retrieved. The underlying assumption behind the similarity calculation is that retrieved behavior should the one with the most similar goal that was learned in the most similar game state to the current goal and game state.

Plan Expansion

The plan expansion module is in charge of maintaining the current plan. Our current implementation is based on the execution module of the A Behavior Language (ABL) [Mateas2002], with which BRL shares some ideas. The current plan is represented as a *partial goal/behavior tree* (that we simply refer to as the *plan*). The plan is a tree composed of two types of nodes: *goals* and *behaviors*. Initially, the plan consists of a single goal: “WinWargus”. Then, the plan expansion module asks for a behavior for that goal. That behavior might have several subgoals, for which the plan expansion module will recursively ask for behaviors at runtime. For instance, the right side of Figure 3 shows a sample plan, where the top goal is to “WinWargus”. The behavior assigned to the “WinWargus” goal has three subgoals, namely “build base”, “build army” and ``attack". The “build base” goal has a behavior assigned to it and contain no further subgoals. The rest of the subgoals still don't have a retrieved behavior. When a goal still doesn't have an assigned behavior, the goal is termed as being *open*.

Open goals can be either *ready* or *waiting*. An open goal is ready when all the behaviors that had to be executed before this goal have succeeded, otherwise it is waiting. For instance, in Figure 3, “behavior 0” is a sequential behavior and therefore the goal “build army” is ready since the “build base” goal has already succeeded and thus “build army” can be started. However, the goal “attack” is waiting, since “attack" has to be executed after “build army" succeeds.

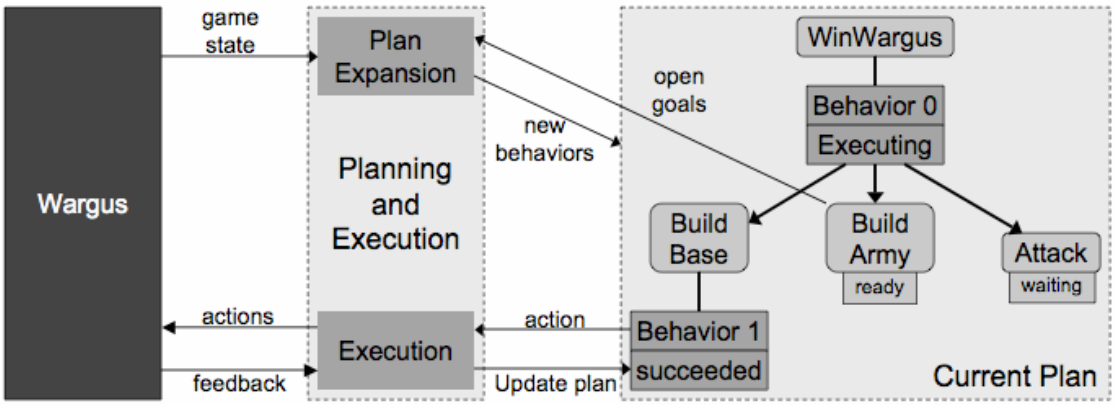

Figure 3: Illustration of how the plan expansion module maintains the current plan composed of goals and behaviors.

As mentioned earlier, behaviors might contain *basic actions* that represent the actions that can be performed in the game (move, attack, build, etc.). When a behavior needs to execute a basic action, the action is first sent to the *Revision Module* that will revise the parameters of the action to the current game state. Once the action is revised, it is sent to the game for execution.

The plan expansion module also monitors the execution of behaviors. If a behavior has *alive-conditions*, it checks for them periodically. The behavior is cancelled, if its

associated alive-conditions are not satisfied, as it has no further chance of succeeding. Moreover, since the plan expansion module knows the goal that a particular behavior is attempting, as soon as a behavior finishes execution, the goal is checked to verify if the behavior succeeded or failed. Notice that knowing the goal that a particular behavior is attempting allows the AI to properly understand what's happening. If we use a classical scripting language that does not allow defining the goals of behaviors, when a behavior finishes the AI will not know whether the behavior succeeded.

When a behavior fails, its associated subgoal is considered again as *open*, and the plan expansion module will attempt to find another behavior to satisfy it. To prevent infinite loops (where a goal is never satisfied, and the system keeps trying to execute different behaviors to achieve it), we define a limit in the number of retries (current set to 3). If the system has already tried three different behaviors to satisfy a goal unsuccessfully, then the goal is considered to have failed and the failure is propagated one level up.

An additional consideration when implementing such plan expansion systems is that if we allow parallel behaviors (i.e., behaviors that can spawn subgoals in parallel, or that can execute basic actions in parallel), we might have two parallel subtrees in the plan that can potentially interfere with each other. There are two possible solutions to this problem. The first one is to disregard the issue of behavior interference, assuming that the expert who generated the demonstrations used two behaviors in parallel (that achieve two different goals) because these two goals can be executed in parallel without interference at all times. The disadvantage of this solution is that it leaves all the responsibility to the author. The second solution is to make the plan expansion module responsible for maintaining the consistency of the plan. Each individual behavior uses a set of "resources" (in Wargus, resources are individual troops, map positions that the troops use, and also gold, wood and oil). The planner has to make sure that no two behaviors that are executing in parallel use the same resource. In our Wargus example source code, we have implemented a simple version of this that only considers troops as resources. We do not consider conflicts in the map where two behaviors try to use the same map space. To make the system more robust, the issue of conflicts in map space should be addressed; however, we found that in our implementation this was not necessary for the system to play at an acceptable level.

Behavior Revision

The revision process consists of a series of rules that are applied to each one of the basic actions of a behavior so that it can be applied in the current game state. Specifically, we have used two revision rules in our system:

- *Unit revision:* each basic action sends a particular command to a given unit. Note that when a behavior is retrieved and applied to a different map, the particular units that the behavior refers to might not correspond to the appropriate units in the new map or might not even exist (since actions refer to units using a simple identifier that is an integer). Thus, the unit revision rule tries to find the most similar unit to the one used in the case for this particular basic action. To perform

that search, each unit is characterized by a set of 5 features: owner, type, position (x,y), hit-points, and status (*idle*, *moving*, *attacking*, etc.). The most similar unit (based an Euclidean distance using those 5 features) in the current map to the one specified in the basic action is used.

- *Coordinate revision:* some basic actions make reference to some particular coordinates in the map (such as the *move* or *build* commands). To revise the coordinates, the revision module gets (from the context information of the behavior) the 5x5 map window surrounding the specified coordinates. Then, it looks in the current map for a spot in the map that is most similar to that 5x5 window, and uses those coordinates. Depending on the particular domain, other window sizes might be considered or even other features such as the relative position of the coordinates of the units.

Once a basic action has gone through these revisions, it is sent to the game for execution.

Automatic Behavior Adaptation

Once the system has learned a behavior library through demonstration by the human expert, it can play the Wargus game with the learned behavior library. However, if the behavior set remains static the system will suffer from some of the same issues of hand-crafted scripts. First, it is hard for the human expert to imagine and demonstrate all possible scenarios that the system might encounter. Given the rich, dynamic nature of game worlds, this can require extensive demonstration efforts in case of our learning by demonstration system or a large programming effort if behaviors are manually authored through scripts. Second, when an existing behavior set fails to achieve its desired purpose, the system is unable to identify such failures and will continue execution. Ideally, we want a self-adapting behavior set, allowing the system to autonomously adapt to new and unforeseen circumstances, thereby relieving authors of the burden of demonstrating or hand-authoring behaviors for every possible situation.

The reasoning layer for the behavior adaptation system consists of two components. The first component detects basic failures and long-term patterns in behavior execution, and detects violations in whether the existing behaviors have achieved their specified goals. These goals are specified in terms of desired effects from the accomplishment of the behavior and certain conditions that should be satisfied during the game execution. When a goal that is not satisfied is detected, the system uses the execution trace to perform blame assignment, identifying one or more behaviors that should be changed. The second component applies fixing operators (*modops*) so as to repair the offending behaviors identified during blame assignment.

One of the essential requirements of a reasoning system responsible for behavior adaptation is to detect when an adaptation should be carried out. We need a way for authors to specify the intended effects of behaviors; when the effects are not achieved, the reasoning layer should modify the behavior library. To accomplish this, we specify effects that are achieved when the behavior finishes successfully. Behavior-specific

effects are encoded using goals in BRL. These goals are defined as desired states of the game when a behavior is successfully achieved. When the conditions specified in the goals are not achieved, this tells the reasoning layer that the current behavior library is not able to achieve the intended effects and that it should seek to change its behavior set. Further, the author may specify desired conditions during game execution that should hold at run time. For example, one of the conditions defined for Wargus is that the peasant should not be idle for a significant amount of time.

A second requirement on the reasoning module is to determine the behavior(s) that should be revised in response to a violation of the intended effects (in our case, conditions specified in the success test of a goal). The process involves analyzing the past execution trace and identifying the behavior with the contribution towards not achieving the desired effect, as the responsible behavior [Cox99]. Once the reasoning module has detected the behavior(s) that needs to be modified, the behaviors are modified using a set of modops. The applicability of a modop depends on the type of failure that is encountered. Thus, modops are 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 in the behavior library.

Let us illustrate how the complete process works. During a particular run of Wargus with the existing behavior library, the system records an execution trace of the events that happen during the game. Once a particular run of the Wargus game is over, the system abstracts data from the execution trace to prepare it for further analysis. The abstracted trace is further used to find out the failures that happened in the execution trace with the existing behavior library. These failures are in the form of a behavior not achieving the desired goals or a behavior not being able to complete execution (maybe some of the preconditions of the subgoals or basic actions were not satisfied). Once a set of failure(s) are detected, the system looks for a set of matching failure patterns that provide an associated set of applicable modops to try on the existing failed behavior Once fixed, the modified behavior is added to the behavior library. Depending on the application, we might want to *substitute* old behaviors by the modified ones, or we might just want to append the new ones to the behavior library. For example, if we are sure that the behaviors in the behavior library are all correct, and that the behavior modification system is only adapting them to new situations, we might want to preserve the old behaviors. In order to understand the process better, let’s look at each of the steps in more detail.

Recording the trace of executing behaviors

The first step involves keeping track of the different events that take place in the game. These events are in the form of whether a particular behavior started, succeeded or failed. Other information related to the event includes the name of the behavior that was involved in the event, the current game state, the time at which the behavior started, failed or succeeded, and the delay from the moment the behavior became ready for

execution and the time when it actually started executing is also recorded within the execution trace. The system keeps updating this execution trace with the events that occur at runtime. Once a Wargus game is finished, the execution trace is used by the behavior adaptation system to find interesting patterns and information that could be used to adapt the failed plans.

Abstracting the trace

In order to reason about the trace, the system abstracts certain data from the recorded execution trace. This abstracted data, for example, consists of:

- *Unit Data*: information regarding the units such as the current hit points, its status (i.e., whether it is idle or researching), its map location and other features.
- *Idle Data*: the idle units and the cycle intervals for which they are idle.
- *Kill Data*: the cycles at which a particular unit was attacked and killed.
- *Resource Data*: for each of the entries in the trace, the corresponding resources that were available. For example, amount of food, gold, wood and oil.
- *Attack Data*: the units that were involved in an attack. For example, the enemy units that are attacking and self units that were under attack.
- *Basic Behavior Failures Data*: the reason for the behavior failure, e.g., whether it was a failure due to insufficient resources or not having a particular unit available to carry out the behavior or some other type of failure due to which a behavior failed.

Detecting failures from abstracted trace

Once an abstracted version of the trace is extracted, the adaptation system detects different failures that occur at run time and tries to fix them appropriately. Some examples of these failures are:

a) Peasant Idle failure: The failure detects if the peasants have been idle for a certain number of cycles. If the peasant has been idle for sufficiently long, 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 build a unit (like farm or barracks) that could be needed later on. Figure 4 shows how this fix is applied after detecting a peasant idle failure.

b) Building Idle failure: The failure detects whether a particular building type has been free for certain number of cycles and there were sufficient resources during the idle time frame. A possible fix for this failure is to utilize the free building by using it for training a corresponding unit that the building is capable of training. For example, “barracks” can be used for developing “footmen” if sufficient resources are available.

c) Basic Operator failures: these types of failures are results of the existing behaviors 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 unavailability of sufficient resources at that time or a peasant being unavailable to carry out the plan. These types of failures differ from the above two that they are based on direct failures caused due to conditions specified as part of the behavior itself. The other failures mentioned in a), b) and d) are based on detecting patterns of things 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 not enough resources available, a basic action is added to gather the corresponding resource. Similarly if there is no peasant available at the time to build a farm, then another basic action to train the peasant is added before the failed behavior.

d) Peasant attacked enemy unit's idle failure: This failure detects whether enemy 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.

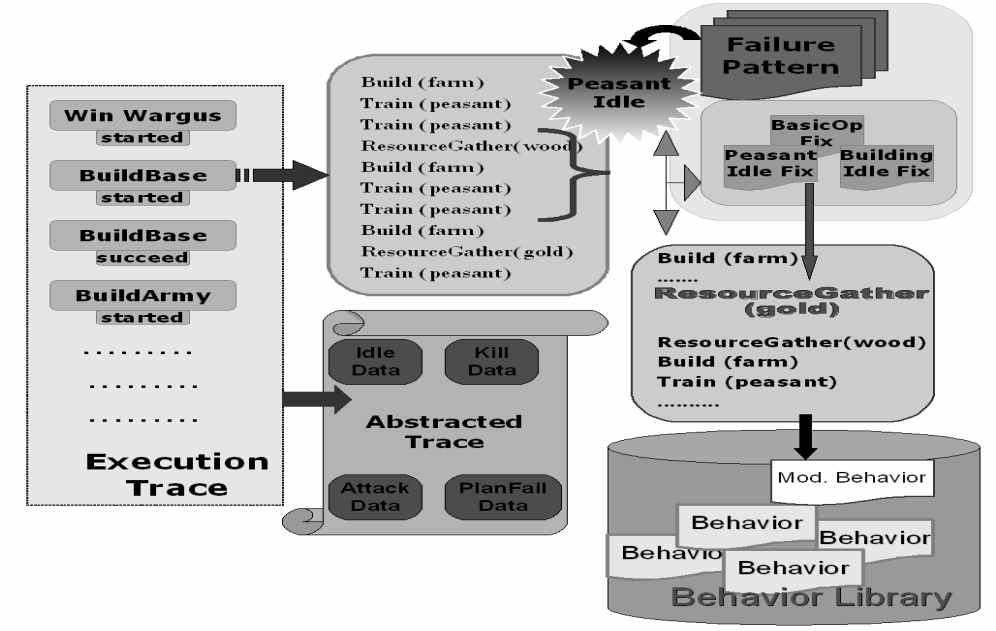

Figure 4: The figure shows adaptation to the "build base" behavior by inserting a Resourcegather (gold) basic operator. The adaptation is carried out in response to the failure "peasant idle"

The adaptation system can be easily extended by writing other patterns of failure that could be detected from the abstracted trace and the appropriate fixes to the corresponding behaviors that need to be carried out in order to correct the failed situation. Once a behavior is fixed, the modified behavior is loaded in the behavior library. The new behavior consists of: 1) the same goal as the original failed behavior, 2) the game state for which the behavior was adapted, and 3) the adapted behavior.

Discussion

The techniques presented in this article have several advantages and, of course, certain limitations. The main advantage is that they provide an easy way to define behaviors by demonstration, and a way in which behaviors can be automatically adapted to new game scenarios. The behavior adaptation system offers a clean way to specify domain knowledge; by defining failure patterns with associated modops, the system can automatically adapt any behavior, and just by adding new failure patterns, the system is automatically able to detect them and modify any behavior that might have those failures.

The techniques as presented in this article still have certain limitations since they are based on some assumptions. First, the features that are used to characterize the game states should reflect all the important details in the game state. If the features don't capture that, the system might retrieve wrong behaviors. For instance, if the maps are characterized only as "percentage of land and percentage of water", the system might not be able to distinguish between maps with the same percentage of land and water but with different configurations that make some strategies better than others. Second, the current approach has been used for a real time strategy game where a human can play the game and control all aspects that need to be modeled and learned through his traces. In a game where the non-playing characters (NPCs) have to conduct facial expressions and gestures, the expert will have to control different aspects (movement, gesture and facial movements) at the same time, which might not be easy. One solution to this problem could be to provide an abstracted action set to the expert that would provide a more controllable space of basic behaviors that the expert can demonstrate. For example, instead of providing individual controls for all the facial animations, gestures and body motions for being happy, the expert could be provided knobs for being happy and the underlying system takes care of the corresponding body and facial movements. Third, if behaviors are long, the annotation process could become tedious. In strategy games, annotation is easy but traces for some other kinds of games (such as the ones described above) might not be so easy to annotate.

The current work in this article can be taken in various different directions. First, in the current system the expert has to play the complete game in order to provide game traces from which the system can play the game later. In order to provide more authorial support, a possible extension could be to provide the author the ability to intervene and demonstrate only sub-portions of the game. The expert could see the performance of the learned behavior library and be able to tweak and modify the learned behaviors by intervening at any intermediate point. The expert can take control of the game at these intermediate moments and demonstrate the right set of behaviors from that point on. The sub-portions that need further tweaking could be suggested based on analysis of the execution traces by the behavior adaptation sub-system. The expert can then demonstrate the right set of behaviors for the sub-portion, thereby providing the appropriate fixes to the failed behavior.

Second the BRL language, still doesn't provide the capability to create daemons that are active at all times in the game. It is difficult to define behavior-learning techniques that can learn when a human demonstrated a daemon. One solution could be to define adaptation rules in the behavior adaptation system that writes daemons as part of fixing existing behaviors. For example, instead of fixing the plan library each time a peasant idle failure is detected, the behavior adaptation system could instead create a daemon that is always active at run time and detects the peasant idle condition and fixes the current behavior library by inserting a proper fix in the executing behavior set.

Conclusion

Artificial intelligence behaviors in games are typically implemented using static, hand-authored scripts. Hand-authoring results in two issues. First, it leads to excessive authorial burden where the author has to craft behaviors for all the possible circumstances that might occur in the dynamic world. Second, it results in games that are brittle to changing world dynamics. In this article, we have presented an approach to address these issues using techniques that a) reduce the burden of writing behaviors, and b) increasing the adaptivity of these behaviors to game world dynamics. We have discussed a behavior learning system that can learn behavior from human demonstrations and also automatically adapt behaviors on run-time when they are not achieving their intended purpose. We have suggested some future directions in which the current work can be taken.

References

[Aamodt94] Aamodt Agnar, and Plaza Enric. *Case-based Reasoning: Foundational Issues, Methodological Variations, and System Approaches.* Artificial Intelligence Communications, 7(1): pp. 39-59, 1994.

[Allen83] Allen James. *Maintaining Knowledge About Temporal Intervals.* Communications of the ACM 26(11): pp. 832-843, 1983.

[Cox99] Cox. Michael and Ram Ashwin. *Introspective Multistrategy Learning: On the Construction of Learning Strategies*, Artificial Intelligence, 112: pp. 1-55, 1999.

[Mateas2002] Michael Mateas and Andrew Stern, *A Behavior Language for Story-Based Believable Agents*. IEEE Intelligent Systems. 17(4): pp. 39-47, 2002.

[Rabin02] Rabin, Steve, *AI Game Programming Wisdom*. Charles River Media, 2002.

[Rabin04] Rabin, Steve, *AI Game Programming Wisdom II*. Charles River Media, 2004.

Bio

Manish Mehta
mehtama1@cc.gatech.edu

Manish Mehta is a PhD student at the College of Computing in Georgia Institute of Technology. He worked full time on a project aiming to demonstrate universal natural interactive access (in particular for children and adolescents) by developing natural, fun

and experientially rich communication between humans and embodied historical and literary characters from the fairy tale universe of Hans Christian Andersen. He has also been involved in developing augmented reality version of desktop based game called Façade. Façade is an artificial intelligence based art/research experiment. An attempt to create a fully-realized one-act interactive drama. Augmented Reality Façade moves this interactive narrative from the screen into the physical world. The player wears a video see through display allowing the virtual characters, Trip and Grace, to inhabit the physical room with them. More details about his work can be obtained at http://www.cc.gatech.edu/~mehtama1

Santi Ontanon
santi@cc.gatech.edu

Santi Ontanon is a Post Doc researcher at the College of Computing in the Georgia Institute of Technology. His Ph.D. thesis focused on case-based reasoning techniques applied to multi-agent systems. His main research goal is to enhance the case-based reasoning paradigm so that it can deal with real tasks, such as computer games. His current research involves the application of case-based reasoning techniques to computer games, in particular strategy games and interactive adventures, in order to provide computer games AI with adaptive capabilities. More details about his work can be obtained at http://www.cc.gatech.edu/~santi

Ashwin Ram
ashwin@cc.gatech.edu

Prof. Ashwin Ram is a recognized leader in introspective learning and case-based reasoning, two of the key aspects of this proposal. In his earlier work, he developed a novel approach to self-adaptation in which introspective analysis of reasoning traces was used to determine learning goals (similar to behavior modification goals in this proposal), and planning was used to carry out the modifications. This work was well received and published in major journals (including *Artificial Intelligence* and *Cognitive Science*) in addition to serving as a framework for an MIT Press book *Goal-Driven Learning*. More details about his publications can be obtained at http://www.cc.gatech.edu/faculty/ashwin.

Article Abstract

Artificial intelligence behaviors in games are typically implemented using static, hand-authored scripts. Hand-authoring results in two issues. First, it leads to excessive authorial burden where the author has to craft behaviors for all the possible circumstances that might occur in the dynamic world. Second, it results in games that are brittle to changing world dynamics. In this paper, we present our effort to address these two issues by presenting techniques that a) reduce the burden of writing behaviors, and b) increase the adaptivity of these behaviors to game world dynamics. We discuss a

behavior learning system that can learn behavior from human demonstrations and also automatically adapt behaviors when they are not achieving their intended purpose.