Runtime Behavior Adaptation for Real Time Interactive Games

Manish Mehta and Ashwin Ram

***Abstract*—Intelligent agents working in real-time domains need to adapt to changing circumstance so that they can improve their performance and avoid their mistakes. AI agents designed for interactive games, however, typically lack this ability. Game agents are traditionally implemented using static, hand-authored behaviors or scripts that are brittle to changing world dynamics and cause a break in player experience when they repeatedly fail. Furthermore, their static nature causes a lot of effort for the game designers as they have to think of all imaginable circumstances that can be encountered by the agent. The problem is exacerbated as state-of-the-art computer games have huge decision spaces, interactive user and real time performance that make the problem of creating AI approaches for these domains harder. In this paper we address the issue of non-adaptivity of game playing agents in complex real-time domains. The agents carry out runtime adaptation of their behavior sets by monitoring and reasoning about their behavior execution to dynamically carry out revisions on the behaviors. The behavior adaptation approaches has been instantiated in two real-time interactive game domains. The evaluation results shows that the agents in the two domains successfully adapt themselves by revising their behavior sets appropriately.**

***Index Terms*—Failure-driven learning, Computer Game AI**

I. Introduction

Failures in problem solving provides both humans and artificial agents with strong cues on what needs to be learned [1], [2]. Thus, in order to create intelligent agents, it is important to provide agents with the ability to learn from their own experiences and their failures. The need for agents that can learn from their experience and thus adapt themselves has been emphasized for real time games as well [3], [4]. The traditional approach for creating AI agents for games is to create hand authored behaviors that remain static and thus are non-adaptive to changing scenarios[1]. The static nature of the AI agents results in excessive authorial burden as it is hard to come up with a behavior set that can handle all possible situations within the confines of the game world. Over long game sessions, static behavioral repertoire further harms the believability of the game when they fail repeatedly. Incorporating knowledge representation and planning techniques can enable an agent's behavior to become more flexible in novel situations but this too can require extensive programming effort [6] and it does not guarantee success: when an agent's behavior fails to achieve their desired purpose, most agents are unable to identify such failure and will continue executing the ineffective behavior. Self-adapting AI agents that can learn from their own experience and adapt themselves could provide the ability to resolve these issues. In this paper, we address this problem of lack of adaptivity in game playing agents to new and unforeseen circumstances.

M. Mehta and A. Ram are with the College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332 USA e-mail: mehtama1@cc.gatech.edu, ashwin@cc.gatech.edu

[1]For an overview of current AI behavior authoring approaches in the game industry see [5]

The problem of the static nature of the AI agent's behavior set is exacerbated by the complexity of the state of the art computer games. Most state of the art computer games pose challenges for traditional AI approaches, as a result making it harder to develop AI approaches for them, as we see next.

A. The Problem: Challenges for AI approaches Imposed by Interactive Game Domains

Interactive game domains introduce considerable challenges for artificial intelligent approaches situated within them. These challenges are also applicable for our behavior adaptation approach. The list below provides an overview on the kind of problems that computer games pose to the AI community.

- *Real-time Nature*: interactive games operate in real-time. Their real-time nature imposes constraints in terms of processing time that could be taken by AI approaches situated within these domains.
- *Human in the loop*: state of the art games involve an interactive user that can change the game state instantaneously. Any considerable delay in AI decision making results in a decision impertinent to current game situation.
- *Large Decision Spaces*: most state of the art computer games have huge decision spaces[7], [8], and thus traditional search based AI techniques cannot be applied. Learning techniques or higher level representation is required to deal with such complex games.
- *Behavior Authoring Effort*: traditionally, computer games use handcrafted behaviors authored by game designers. Authoring the behavior sets in a game requires a huge human engineering effort.
- *Unanticipated Scenarios*: it is not feasible to anticipate all possible situations that can arise in a real game. As a result, it is hard to design behaviors that can handle all the possible situations and respond appropriately to all possible player actions.
- *Support Tools*: human-authored behaviors are, ultimately, software code in a complex programming language and as such they are prone to human errors. Sometimes there are ”bugs”, sometimes the behaviors may simply not have the intended result. Support tools are required that can help the author detect these issues.
- *Behavior Replayability and variability*: static behavior sets results in behaviors being repeated again and again. A player might become bored seeing repeated behaviors.

In recent years, interest in applying AI techniques to computer games has seen a notable increase (e.g., see workshops dedicated to game AI in conferences such as ICCBR 2005, IJCAI 2005, AIIDE and more recently at ECAI 2008). The vast majority of this work, however, focuses on small sub problems within a computer game (small tactical-level problems, coordination, path planning, etc.) or is not situated within a real game. Although this research provides interesting solutions and ideas, it cannot be directly applied by computer game companies. As computer games are being developed by increasingly large project teams with increasingly tight timelines, game developers do not have the necessary cycles needed to try and transition these techniques to their own games. One of the long-term goals of our work is to reduce the transition effort needed in applying academic AI techniques in real games. Our approach presented in this paper is a step in that direction.

Our approach has been implemented in two domains. The first domain is a real-time game domain with embodied characters that must behave in a human-like in a believable way. The second domain is a real-time game domain with a complex strategic behaviors. The two domains provide the ideal game domains for developing adaptive agents as they present the problems that we listed above for any AI approach situated within these domains. They share the characteristic of having huge decision spaces, require real-time performance by the AI approaches situated within it, have an interactive user and require extensive behavior libraries that can play at a reasonable level.

B. The Solution

Our approach to deal with the issue of lack of adaptivity in game playing agents is to provide agents with the ability to be introspectively aware of their internal state, and further revise themselves based on deliberating over their internal state. The agents monitor and reason about their behavior execution; when the agent fails, they use feedback from the world, and the trace of the failed process, to identify the cause(s) of its failure and perform appropriate revisions to their behaviors. The behavior adaptation approach consists of three parts: *Trace recording*, *Failure detection* and *Behavior revision*.

In the first part, a trace holding important events happening during the game is recorded. This trace holds the status of the executing behaviors and other information like external state of the world at various intervals during the game. The second part involves analyzing the execution trace to find possible issues with the executing behavior set by using a set of *failure patterns*. These failure patterns represent a set of pre-compiled patterns that can identify the cause of each particular failure by identifying instances of these patterns in the trace. Once a set of failure patterns has been identified, the failed condition can be resolved by appropriately revising the behaviors using a set of *behavior modification routines*. These behavior modification routines are created through a combination of basic operators (called modops). The modops change the behavior in different ways like modifying the original elements, introduction of new elements or reorganization of the elements inside the behavior. The solution addresses the need for AI agents to be adaptive in complex real-time game domains. The results from evaluation of the behavior adaptation approach in the two domains indicate that the agents successfully adapt themselves to changing game situations.

The rest of the paper is organized as follows. We first provide the background with theoretical foundations of our work. We then focus on our behavior adaptation architecture in Section III. Section IV presents concrete implementation and evaluation of the behavior adaptation work in the first game domain, inhabited with embodied characters. Section V presents the details on the behavior adaptation work in the area of real-time strategy game and the results of the evaluation. We discuss our approach and some lessons learned based on our implementation of the behavior adaptation approach in the two domains in Section VI. We present the related work in Section VII. Finally, we conclude with some future directions we plan to undertake.

II. Background

A. Theoretical Foundation of Behavior Adaptation

The agent's behavior set can be considered a reactive plan dictating what it should do under various conditions. Run-time behavior adaptation can be considered a problem of runtime reactive-plan revision. One approach to runtime plan revision is to simply apply classical planning techniques to replan upon encountering failure. Such techniques, however, are ill-suited to the unique requirements of our domain. They typically assume the agent is the sole source of change, actions are deterministic and their effects well defined, that actions are sequential and take unit time, and that the world is fully observable. In an interactive, real-time game domain, all these assumptions are violated. Actions are non-deterministic and their effects are often difficult to quantify. Game domains are typically not fully observable. There are often occlusions blocking sensors from reaching the entire world.

Some of the more recent work in planning has focused on relaxing these assumptions. Conditional planners such as Conditional Non Linear Planner [9] and Sensory Graph Plan [10] support sensing actions so that during execution, changing environmental influences can be ascertained and the appropriate conditional branch of the plan taken based on the sensor values. Unfortunately, as the number of sensing actions and conditional branches increase, the size of the plan will grow exponentially. These techniques are mostly suited to deterministic domains with occasional exogenous or non-deterministic effects, not to continuously changing interactive domains.

Approaches to planning that deal best with exogenous events and non-determinism are decision-theoretic planners. These planners share much with reinforcement learning, commonly modeling the problem as a Markov decision process (MDP) and focusing on learning a policy. Partially observable MDPs can be used when the world is not fully observable. These approaches, however, require a large number of iterations to converge and only do so if certain conditions are met. In complex game domains, these techniques are intractable. Physical states alone are complex, upon adding game state information and the status, level and internal states of various characters, the state space quickly grows untenable. Further, these approaches generalize poorly. An interactive player can significantly change the virtual world; a learned static policy cannot be re-trained online during actual game play to accommodate such changes. Finally, these approaches invariably require significant engineering of for example, the state space and reward signal to make its application feasible.

Transformational planning (TP) is an approach that can potentially deal with the complexity and nondeterminism of our problem game domain. This technique isolates itself from the difficulties in the problem domain by focusing on reasoning about the plan itself. In TP, the goal is not to reason about the domain to generate a plan but to reason about a failing plan and transform it so as to fix the failing case without breaking the rest. This insight is key, but we cannot directly apply such a technique. TP is generally applied to plans consisting of STRIPS operators (or plan languages that provide relatively minor extensions of STRIPS); it is unsuitable for rich reactive planning languages required to represent behaviors for our game domains. Thus we developed novel behavior transformations and techniques for failure detection and revision of the failed behaviors, extending TP such as to enable us to leverage this approach in our system.

B. Theoretical Foundation of Learning from failures

Learning is a multi-faceted competence, fundamental to intelligent behavior. Intelligent Agents can learn by various means, namely a) learning by being told, b) by observing some expert in a particular domain and c) from their own experiences in problem solving. Failures in problem solving provide the cues on what the agent needs to learn. When an agent fails, it needs to learn to recover from the particular failure and eliminate the causes of this failure so that it does not repeat the same mistake in the future. This paper investigates this last type of learning, namely, learning from failures for complex game playing agents.

The problem of learning from failures can further be divided into three subtasks.

- Given a problem solving agent working in a domain, the first task is to recognize that a failure has happened.
- Once a failure is identified, the next task involves determining the cause of the failures, perhaps from symptoms of the failures or from some feedback from the environment. This is the well known problem of blame assignment in AI [11].
- Once the cause of the failure has been identified, the third task involves carrying out appropriate repairs so that the failure is not repeated in similar circumstances.

Failure driven learning (FDL) approach has been used in various AI systems. For example, Meta-Aqua system by Cox and Ram [12] operates in a story understanding domain. The system uses a library of pre-defined patterns of erroneous interactions among reasoning steps to recognize failures in the story understanding task. Autognostic system by Stroulia and Goel [13] uses a model of the system to localize the error in the system's element and uses the model to construct a possible alternative trace which could lead to the desired but unaccomplished solution. For an overview of AI systems using FDL see [13].

FDL forms the basis of our approach to create adaptive agents. FDL has traditionally been employed in scenarios where the agent is not interacting with the user in a complex real-time domain (for example all the systems described above share this characteristic). Furthermore, as with Transformational Planning, FDL when applied to the problem of plan revision, is generally applied to detect failures in plans consisting of STRIPS operators or minor extensions of STRIPS; using it for revising complex behavior sets for our game domains requires novel failure detection techniques, a vocabulary of failure patterns pertinent to game domains and behavior modification strategies that can revise rich constructs of our behavior representation language (described later in Section IV and V) used to author behaviors. Thus we have extended FDL with real-time failure detection and novel behavior modification routines to extend it for our real-time interactive game domains.

III. Real-Time Behavior Adaptation Architecture

Our Behavior adaptation architecture (Figure 1) can be divided into two main layers: *Behavior Representation and Execution layer* and *Reasoning layer*. The first layer involves representing the agent's behavior set in a representation language, a set of behaviors constituting a behavior library to play the game for which the agent has been developed and a behavior execution layer that deals with the real-time nature of the game and executes the authored behavior set in real-time. The second part monitors the performance of the executing behaviors at runtime in an execution trace, infer from their execution trace what might be wrong using a set of author defined patterns of common failures and perform appropriate modifications to the failed behaviors. We describe these two parts in detail next.

A. Behavior Representation and Execution

1) Behavior Library in Representation Language: The representation language for authoring behaviors provides a means for representing basic behaviors using various constituents namely, preconditions, alive-conditions, sensors and basic actions. Preconditions are a set of conditions that must be true in order to execute the behavior (e.g., an "attack" behavior for a strategy game might have as its preconditions the existence of an army and an enemy). A set of alive-conditions associated with a behavior 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. Sensors are the basic variables and tests that the behaviors can use to consult the current world state. Actions are the basic actions that can be sent to the game world to perform some low level things (e.g, moving to a particular part of the world, carrying out gestures etc). In order to adapt the behaviors, the language could further provide inherent

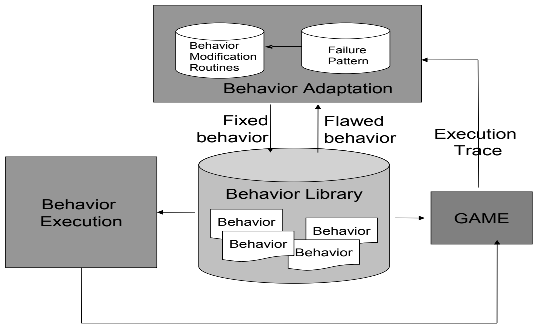

Fig. 1. The figure shows the architectural diagram for the behavior adaptation system.

support to modify the internal of the authored behaviors. These modification operators help in easily revising the failed behaviors once a modification is detected.

The behavior library holds the set of behaviors that would allow the agent to play in a given game domain. These behavior set could be hand authored by the game designer using the representation language or learned through some other mechanism (for example, the behavior set could be learned through designer's demonstration for our second case study).

2) Behavior Execution Layer: The execution layer provides the ability to execute behaviors in real-time. The execution layers provide the means to look at the current state of the world and select appropriate behavior for execution. The execution layer further needs to maintain currently active goals and behaviors. During execution, steps of the behavior may fail (e.g. no behavior can be found to accomplish a subgoal, or a basic action fails in the game world), potentially causing the enclosing behavior to fail. When a behavior fails, the execution layer needs to find an alternate behavior to accomplish the goal; if no appropriate alternative behavior is found, the goal would fail. Behavior preconditions are used to find appropriate behaviors for accomplishing a goal in the current context. When a behavior needs to execute a basic action, it needs to be sent to the game layer.

B. Reasoning Layer

The reasoning layer consists of three components: *Trace recording*, *Failure detection* and *Behavior Revision*. Figure **??** shows the behavior adaptation algorithm.

1) Trace Recording: The execution trace is generated by storing each game state recorded at specific time intervals. An abstracted version of the execution trace is used to record important events happening during the execution. From each game state , various events are extracted and recorded in the abstracted trace . These events are in the form of information regarding the behaviors that were executing: their start and end times, and their final execution status (i.e. whether the behavior started, failed or succeeded). The abstracted trace also contains external state of the world recorded at various intervals so that different information can be extracted from it during reasoning about the failures happening at execution time. The abstracted trace provides a considerable advantage in performing adaptation of behavior as search in the trace can help localize portions that could possibly have been responsible for the failure.

2) Failure Detection: Failure detection involves localizing the fault points. Although the abstracted execution trace defines the space of possible causes for the failure, it does not provide any help in localizing the cause of the failure. Traces can be extremely large, especially in the case of complex games on which the system may spend a lot of effort attempting to achieve a particular goal. In order to avoid this potentially very expensive search, a set of pre-compiled patterns of failures can be used that can help identify the cause of each particular failure by identifying instances of these patterns in the trace [1], [14]. The failure patterns essentially provide an abstraction mechanism to look for typical patterns of failure conditions over the execution trace. The failure patterns simplify the blame-assignment process into a search for instances of the particular problematic patterns. Once a set of failures is identified, they need to be appropriately revised.

3) Behavior Revision: Once the cause of the failure is identified, each failure needs to be addressed through appropriate modification. The collection of modification routines provide the necessary set of modifications to be applied . Each modification consists of basic operators (called modops). The modops change the behavior in different ways like modifying the original elements, introduction of new elements or reorganization of the elements inside the behavior. Table I shows some of the modops. Figure 2 shows an example of a behavior modification routine created using the combination of modops. Each of the modop for a modification are applied to produce a modified behavior. Once all the modifications are carried out, a modified behavior set is produced.

Our behavior adaptation architecture has been concretely implemented in two domains. Next we present these two case studies and the evaluation of the behavior adaptation approach

ModOps	**Original Behavior**	**Modified Behavior**
Add a step (B2', B2)	Behavior B{ step B1, step B2 step B3 }	Behavior B'{ step B1, step B2' step B2, step B3}
Reorder a step (B2, B3)	Behavior B{ step B1, step B2 step B3 }	Behavior B'{ step B1, step B3 step B2}
Change the Parameters (B, x, x')	Behavior B{ step B1 (x, y) step B2, step B3 }	Behavior B'{ step B1 (x', y) step B2, step B3 }

TABLE I
THE TABLE PROVIDES SOME OF THE MODOPS USED FOR CREATING BEHAVIOR MODIFICATION ROUTINES.

```
BehaviorModificationRoutine Bmod() {
    add step(B1, step s1, step s2);
    change parameter (B1, x, x');}
```

Fig. 2. The figure shows an example behavior modification routine created using modops.

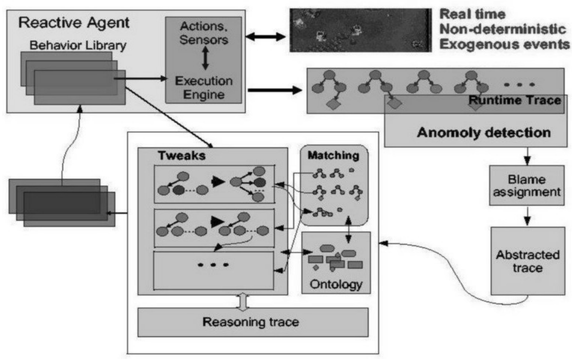

Fig. 3. The figure shows the architectural diagram for our behavior transformation system.

SMF/S	Semantically Meaningful Failure/Success: failure/success of this behavior implies something that the goal is important in the game world wrt. author intent (AvoiditPerson shown in Fig 6 is SMF because it is important to avoid the person who is ”it ” in the game)
FI	Fully Implementing: the behavior in and of itself is a complete and independent method of achieving goal.

TABLE II
SOME EXAMPLE ANNOTATIONS

in these two domains.

IV. CASE STUDY: BEHAVIOR MODIFICATION BASED ON TAG

In the first case study, our game scenario consists of two embodied characters named Jack and Jill. They are involved in a game of Tag where they chase the character who is ”It” around the game area. Each character has its own personality that affects the way they approach play. Jack for example, likes to daydream and is not particularly interested in the game. If he has to play he would prefer to hide somewhere where he can relax. Jill on the other hand, likes to be the center of attention. She is bored if she is not being chased or chasing someone. The behaviors authored for each character reflect their personalities. Each character’s behavior library currently consists of about 50 behaviors and contains approximately 1200 lines of ABL code (see below). Our system (see Figure 3) is composed of an execution layer which handles the real-time interactions, and a reasoning layer responsible for monitoring the character’s state and making repairs as needed.

A. Behavior Execution Layer

We have used A Behavior Language (ABL) as the behavior execution layer. ABL is explicitly designed to support programming idioms for the creation of reactive, believable agents [15]. Its fast runtime execution module makes it suitable for real-time scenarios. ABL is a proven language for believable characters, having been successfully used to author the central characters Trip and Grace for the interactive drama Facade [6]. A character authored in ABL is composed of a library of behaviors, capturing the various activities the character can perform in the world..

ABL’s runtime execution module acts as the front end for communication with the game environment. It constantly senses the world, keeps track of the current game state, updates the active behavior tree and initiates and monitors primitive actions in the game world. Furthermore, the runtime system provides support for meta behaviors that can monitor (and potentially change) the behaviors at run time. For our reasoning module, we have utilized this meta-reasoning capability of ABL to trace agent execution. We also modified ABL’s runtime system and compiler so that behaviors generated by the reasoning layer can be reloaded.

B. The Reasoning Layer

The reasoning layer consists of two components. The first component tracks long term patterns in the character’s behavior execution and detects violations of the author specified behavior contract (see below). When a contract violation is detected, it uses the execution trace to perform blame assignment, identifying one or more behaviors that should be changed. The second component applies modification operators so as to repair the offending behaviors identified during blame assignment.

1) Failure Detection: The reasoning system first needs to detect when a modification to a behavior should be carried out. We need a way for authors to define contracts about long term character behavior; when the contract is violated, the reasoning layer should modify the behavior library. To accomplish this, we use a simple emotion model based on Em, an OCC[2] model of emotion [16]. Emotion values serve as compact representations of long-term behavior. The author defines personality contract through constraints on behavior. The constraints are defined by providing nominal bounds for emotion values. When an emotion value exceeds the bounds provided by the author, this tells the reasoning layer that the current behavior library is creating inappropriate long-term behavior and that it should seek to assign blame and change its behavior. At runtime, a character’s emotional state is incremented when specific behaviors, annotated by the author, succeed or fail. The emotion increment value per behavior is defined by the author as part of creating the character personality. The reasoning module also needs to determine the behavior(s) that should be revised in response to a violation of the personality contract (in our case, an emotion value exceeding a bound). This process involves analyzing the past execution trace and identifying the behavior with the maximal contribution to the out-of-bound emotion value, amortized over time, as the responsible behavior.

2) Failure Patterns and Behavior Modification Routines: Once the reasoning module has detected the behavior(s) that need to be modified, the next step is to identify the appropriate set of behavior modification operators that can

[2]OCC model derives its abbreviated name from it original inventors, Ortony, Clore and Collins

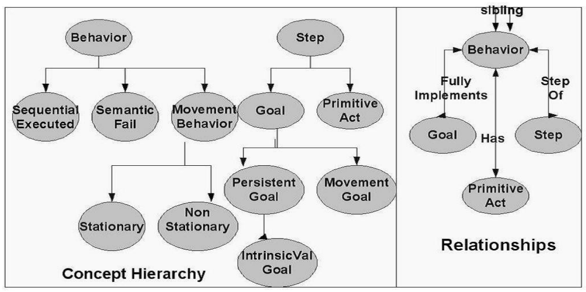

Fig. 4. The figure shows the concepts hierarchy and relationships used during the tweaking process

be applied to the offending behavior(s). We would like our behavior modification operators to be as domain independent as possible. However, domain specific knowledge about particular game worlds is necessary in order to reason about which operators to apply to a given behavior. Rather than rolling such knowledge into the operators, we factor it into author provided declarative knowledge about the character's behavior library. This declarative knowledge consists of two parts: annotations on the behaviors themselves (see Table II for a subset of the annotations used in our current system) and an ontological description of the behaviors, their types, relationships, and what they accomplish (see Figure 4 for a subset of the ontology). Each of the behavior (about 50 total behaviors for each character) written as part of the character's behavior library needs to be classified according to the ontological categories. The ontological classification provides the necessary declarative knowledge about the behavior library. In the ontology, among other relationships parent-child, part-subpart relationship among the different behaviors are specified. For example behavior "Avoid_It_Person" (shown later in figure 8) has a parent-child relationship with "Hide" and "TurnAroundEnsureEscape". This knowledge is utilized by few of the modification routines (as shown in Table III) to find appropriate similar behaviors to replace the failing ones.

Our system contains a collection of modification routines based on the currently defined ontological categories. Given that blame assignment has provided a behavior to modify, the applicability of a modification routine depends on the role the problematic behavior plays in the execution trace, that is, an explanation of how the problematic behavior contributed to a contract violation. Thus, modification routine is categorized according to failure patterns. The failure patterns provide an abstraction mechanism over the execution trace to detect the type of failure that is taking place. Failure patterns are encoded loosely as finite state machines that look for patterns in the execution trace. Figure 5 shows an example failure pattern that recognizes when a problematic behavior is repeatedly failing. Table III shows the association between modification routine and failure patterns.

Now that the major components of the reasoning layer have been described, we can provide a brief summary of the behavior modification process. At runtime, the system detects when the author provided behavior contract has been violated. Once

Failure Pattern	**Behavior Modification Routines**
Continuously repeating behavior	If behavior is part of a persistent goal, halt persistent goal. Use alternate behavior or alternate parent behavior. Recursively fix the behavior itself.
Failing SMF type goal, all behaviors fail.	Recursively fix one of the failing behaviors (or a clone thereof) If some behaviors are never run, try loosening preconditions of those behaviors (or a clone). Modify goal annotations (eg. priority of the goal)
Failing sequential, SMF type behavior	Replace failing step with a sibling (closest equivalent behavior from the ontology) Modify step annotations or change its parameters Remove failing step or reorder steps

TABLE III
SOME EXAMPLE FAILURE PATTERNS AND THEIR ASSOCIATED BEHAVIOR MODIFICATION ROUTINES

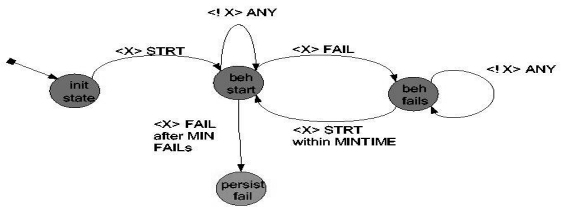

Fig. 5. The figure shows a matcher for the failure pattern: persistent failing behavior. In this diagram, <X>denotes the behavior in question, STRT (start), FAIL and ANY (start, fail, succeed) represents the status of the behavior and "after MINFAILs" or "within MINTIME" are conditions

blame assignment has determined the offending behavior, the system uses the failure patterns to explain the behavior's role in the contract violation. This involves matching each of the finite state machines associated with failure pattern against the execution trace. The set of matching failure patterns provide an associated set of applicable behavior modification routines to try on the offending behavior. The order in which the routines are tried is defined through annotated priority specifications. Modification routines are tried one at a time until one succeeds (routines can fail if the behavior they are tweaking lacks the structural prerequisites for the application of the routine). If the routines fail, the conducted changes are undone and behavior is tweaked with the next modification routine. The modified behavior is compiled and reloaded into the agent.

C. Illustrated Example

To better understand the inner workings of the reasoning module, let's look at an illustrative example. In our tag game, when Jack is chasing Jill, he will use behavior *RunTowardsPlayer_1* to run towards Jill and tag her when he sees her (see Figure 6). Unfortunately, this behavior fails if Jill is standing on an elevated surface. Although Jack is able to see Jill, he cannot reach her without jumping. The behavior author

```
sequential behavior RunTowardsPlayer(){
  precondition{ (ItWME itPlayerName :: itAgent)
    !(AgentPositionWME x::x y::y z::z
      objectID == itAgent)}
      act Walkto(x,y,z)}

sequential behavior GoToElevatedPos(double x,
    double y, double z){
    mental_act{ pathplan_closest(x,y,z);}
    subgoal walkpath();
    act jump();}
```

Fig. 6. Example behaviors defined in ABL

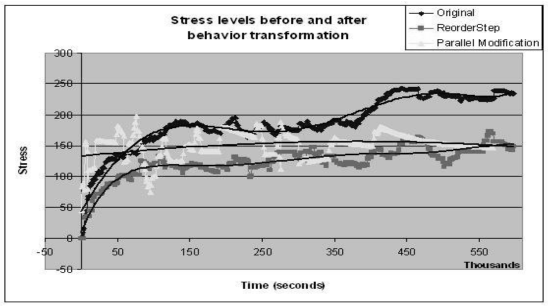

Fig. 7. The figure shows the results for average stress level from the evaluation experiment

forgot to handle this case, being either unaware that there were elevated surfaces in the world or perhaps because the world has changed since the characters were authored. Due to this deficiency, behavior *RunTowardsPlayer_1* will persistently fail. Since it has been marked with emotion annotations (not shown), Jack's stress level will rise as the behavior persistently fails, eventually going beyond his nominal bounds for stress, triggering the behavior adaptation reasoning layer.

The reasoning module first analyzes the execution trace. The blame assignment module identifies the responsible behavior by calculating a temporally normalized emotional contribution for each behavior. In the current example, it detects that *RunTowardsPlayer_1* is the offending behavior. Analyzing the trace based on this behavior, the matcher identifies the failure pattern as continuously repeating behavior (Table III). The set of associated behavior modification routines are then tried. Because the system is unable to find a stopping condition for the parent persistent goal to halt it, nor to find an alternative behavior, the first applicable routine instructs the reasoning module to recursively modify the steps of the behavior itself. Sending this back to the matcher, leads us to the failure pattern Failing Sequential Behavior. The first associated routine is applicable and involves replacing the failing step in the behavior with the closest ontological match that achieves the same purpose. The failing step in our case is walkto, which is of type Movement. Querying the ontology, we see that *GoToElevatedPosition_1*, one of Jack's behaviors defined for him to locate and hide on an elevated platform, is also of type Movement. Thus, we can apply the tweak, replacing the walkto step with *GoToElevatedPosition_1*. Finally the modified behavior library is reloaded into the character. It should be noted that finding the closest match from the ontological classification is one of the several ways to modify the behavior. The modification routines involves changing the internals of the behavior in different ways as shown in Table III. Other ways of modifying the behavior include reordering the steps of the behaviors, changing the parameters of the behaviors, preconditions etc.

D. Experimental results

We evaluated our behavior adaptation system on two hand-authored embodied characters, Jack and Jill, designed to play a game of Tag. Jack and Jill were initially authored by people on a different research project. This provided a great opportunity for us to evaluate our system. Their fixed behavior set must invariably make assumptions about world dynamics and thus will be ineffective at maintaining personality invariants in the face of change. If our system can help maintain those invariants then it is an effective means of behavior adaptation.

Specifically, we provide emotion annotations by associating a stress emotion with being chased and placing nominal bounds on stress, specifying a contract on Jack's intended personality. We then tested whether our system is able to successfully modify the behavior library to changing environments. In our experiment, we simulated a changing world by moving the tag agent whose behaviors had been built for a specific map into a larger and sparser version.

Our experimental procedure involves first running the game scenario without the adaptation mechanisms and continuously observing Jack's stress level. We then run Jack with the adaptation mechanisms. Figure 7 shows Jack's stress levels averaged over five 10 minute games before adaptation, and with two different behavior libraries modified by our system. Blame assignment found that the behavior *Run_Away_1* is responsible for stress exceeding bounds. In the ideal case, Jack would run away for a while, until he was able to escape out of sight, at which point, he would head for a hiding place. Trace analysis however shows that Jack turning around to ensure he is not being followed always fails. Jack is never able to run away and escape out of sight long enough to risk going to a hiding place. This situation tends to occur on our test maps because they are sparse; with fewer obstacles it is more difficult for Jack to ever escape out of sight. As a result, Jack is continuously under immediate pursuit and his stress level quickly exceeds bounds.

In our runs, the behavior adaptation system found two different modifications that brought stress back in bounds. In the first case, the system changed the *AvoidItPerson_3* behavior (see Figure 8) from a sequential behavior to a parallel behavior. Originally the authors had expected Jack to first ensure no one is following before hiding, but the system's change is actually quite reasonable. When pressed, it makes sense to keep running while turning around. If it turns out some one is following you, you can always change course and not go to the secret hiding place. Visually, this change was quite appealing. Jack, when running away, would start strafing towards his hiding place, allowing him to move towards

```
sequential behavior AvoidItPerson() {
precondition {(ItWME itPlayerName :: itAgent)
        !(AgentPositionWME objectID == itAgent)}
    with(post) subgoal Hide();
    with(post) subgoal TurnAroundEnsureEscape();}
```

Fig. 8. The figure shows the modified behavior

his destination while keeping a look out. Unfortunately, this change was unstable. Due to how Jack navigates, if he cannot see his next navigation point, he will stall (a defect in his navigation behaviors). Surprisingly, even with this defect, Jack with this change is able to stay within his normal stress bounds. We initially assumed this was because the defect happened rarely, but in fact it was the opposite. While running away, Jack was always getting stuck, allowing Jill to tag him. This decreases stress because Jack is not as stressed when he is the pursuer; he can take his time and is not pressed. This change is nevertheless undesirable. Jack is violating an implicit behavior contract that Jack should try to escape when he is "It" and not allow himself to be tagged. The adaptation system essentially found a clever way to take advantage of the under specification of the author's intent. After amending the specifications, our behavior adaptation system found an alternate change: to reorder the steps inside *AvoidItPerson_3*. In the new behavior set, *AvoidItPerson_3* first hides and then turns around to ensure no one is following instead of the other way around. This results in behavior that is as good as the parallel version if not better than it.

V. Case Study: Behavior Adaptation based on Real-Time Strategy Game Wargus

Realtime strategy games have been recognized as domain rich in interesting problems for artificial intelligence researchers [8], [7]. Our second case study uses Wargus a realtime strategy game as an application domain. The goal of each player in Wargus is to survive and destroy the other players. Each player has a number of troops, buildings, and workers who gather resources (gold, wood and oil) in order to produce more units. Buildings are required to produce more advanced troops, and troops are required to attack the enemy. In addition, players can also build defensive buildings such as walls and towers. Therefore, Wargus involves complex reasoning to determine where, when and which buildings and troops to build.

A. Behavior Library and Execution Layer

We have used Darmok as the real-time planning and execution system. Darmok has been designed to play games such as WARGUS [17]. The behavior adaptation approach for Wargus uses Darmok as the execution layer to carry out the behaviors that play the game. Darmok learns behaviors from expert demonstrations and uses case-based planning techniques to reuse the behaviors for new situations. Darmok's execution can be divided in two main stages:

- *Behavior acquisition*: During this first stage, an expert plays a game of Wargus and the trace of that game is stored. Then, the expert annotates the trace explaining the goals he was pursuing with the actions he took while playing. Using those annotations, a set of behaviors are extracted from the trace and stored as a set of cases. Figure 9 shows an example of a behavior representing using the representation language.
- *Execution*: The execution engine maintains a current plan to win the game. It is in charge of executing the current plan and updating its state (marking which actions succeeded or failed). It also identifies open goals in the current plan and expands them.

```
seqbeh SetupResourceInfrastructure(){
  precondition{(resource gold > gold1,
                         oil > oil1)}
      action Build(2,pig-farm,26,20);
      action Train(4,"peon);
      action Build(2,troll-lumber-mill,22,20)
      action Train(4,peon)
  alivecondition{peasantcount > 2}}
```

Fig. 9. The figure shows an example behavior in the representational language

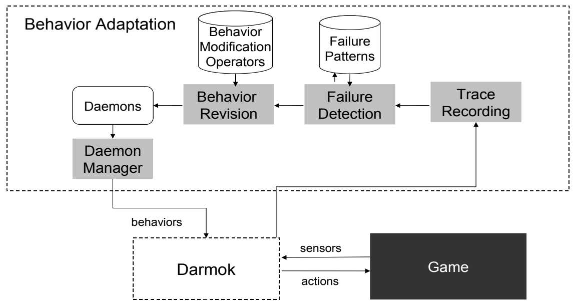

Fig. 10. The figure shows the behavior modification architecture.

B. Reasoning Layer

The reasoning layer for Wargus consists of four components: *Trace recording*, *Failure detection*, *Behavior Revision*, and *Daemon Manager*, as shown in Figure 10. The three first components are executed off-line, after a game playing episode has finished. The last component executes on-line, while the system is playing a new game. We describe each of the different components of the meta-level behavior adaptation in detail next.

1) Trace Recording: The behavior execution system during execution records a trace that contains information related to basic events as explained earlier in Section III-B1. Once a game episode finishes, an abstracted version of the execution trace is created for processing by other components. The abstracted trace, consists of various relevant domain-dependent features: like information regarding units such as hit points, or location, information related to units that were idle, killed or attacked and the cycles at which this happens. The abstracted trace also contains basic behavior failure data that consists of the apparent reason for behavior failures, such as whether it was due to insufficient resources or not having a particular

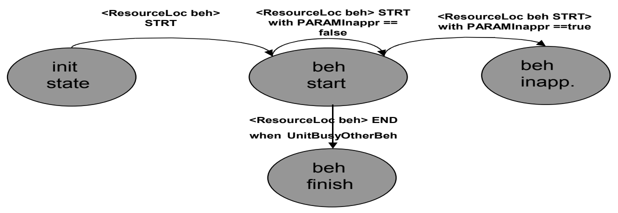
<ResourceLoc beh> STRT
<ResourceLoc beh> STRT with PARAMInappr == false
<ResourceLoc beh STRT> with PARAMInappr ==true
init state
beh start
beh inapp.
<ResourceLoc beh> END when UnitBusyOtherBeh
beh finish

Fig. 11. The figure shows the failure pattern VCRGLfail in WARGUS. STRT, END represents the status of the behavior. PARAMInappr. represents the routine that checks whether the parameters of ResourceLoc beh. i.e. the location where peasant is gathering resources is at a location that is too close to the enemy compared to other possible resource gathering locations.

Failure Pattern	Behavior Modification Routine
Resource Idle failure	Utilize the resource in a more productive manner (for example, send peasant to gather more resources or use the peasant to create a more appr. building)
Inappr. Enemy Attacked	Attack more dangerous enemy unit
Inappr. Attack Location	Change attack location to a more appropriate one
Basic Operator failure	Add a basic action that fixes failed condition

TABLE IV
SOME EXAMPLE FAILURE PATTERNS AND THEIR ASSOCIATED BEHAVIOR MODIFICATION ROUTINES IN WARGUS

unit available to carry out a behavior. The abstracted trace is used to find out various failure patterns as explained next.

2) Failure Patterns and Behavior Modification Routines: The failure patterns defined over the abstracted trace is used to detect different failures that occur at run time and find appropriate modifications for them. These patterns 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 and some of their modification routines are shown in table IV. Each failure pattern is associated with modification routines. When a failure pattern generates a match in the abstracted trace, an instantiation of the failure pattern is created. Each instantiation contains which were the particular events in the abstracted trace that matched with the pattern. Such instantiation is used to instantiate particular behavior modification routines that are targeted to the particular behaviors that were to blame for the failure. Figure 11 shown an example failure pattern defined for the Wargus domain.

Once the failure patterns are detected from the execution trace, the corresponding behavior modification operators and the failed conditions are inserted in the base reasoner as daemons. Each daemon has a set of conditions that represent when the failure pattern was detected. When the base reasoner plays another game, different daemons might trigger and prevent previous failures. The daemons act as meta-level reactive behaviors that operate over the executing behavior-set at runtime. The *daemon manager* uses the daemons to operate over the executing behavior set at runtime. The *daemon manager* monitors the behavior execution, detect whether a failure pattern is satisfied and repair the behavior according to the defined behavior modification operator.

In the current system, we have defined about 12 failure patterns and 20 behavior modification routines. 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. In order to understand better, let us look at an example.

C. Illustrated Example

Consider a game scenario where the system has two long range attacking units (named ballistae in Wargus) and issues attack command to the ballistae to go to the location of a peon (one of the enemy units) and start attacking it. As the ballistae reach an appropriate attacking point, they are attacked by other nearby enemy units (named axe-throwers). The attack inflicts considerable damage to the ballistae and they get killed as a result. These events are recorded as part of the execution trace. Once the game is over, the system takes the recorded execution trace and abstracts various data from it. The system applies the failure patterns over the abstracted trace and detects that two failure patterns are applicable to the portion of the trace explained above. These two failure patterns are *Inappropriate Enemy Attacked failure* and *Inappropriate Attack Location failure*. The failure pattern *Inappropriate Enemy Attacked failure* detects the fact that the enemy unit that was being attacked was not the most appropriate one to destroy first. This failure happened as the ballistae attacked a less harmful enemy unit peon rather than a more dangerous enemy unit axe-thrower. The second failure pattern *Inappropriate Attack Location failure* detects the fact that the location used by an attacking unit is inappropriate and there are better locations available for attack. This failure, happens as the ballistae attacks from a location that is very close to an axe thrower. A better location for the ballistae to attack would have been one where it was not very close to the axe-thrower, outside of its range but still at a distance where the ballistae itself can attack the axe thrower.

The behavior modification operators for the two failed plans are to a) issue the ballistae new behaviors which have the enemy unit set as axe-thrower and b) change the location for attack a little further away from axe-thrower by computing difference between the location of axe-thrower and ballistae attacking range. These failure patterns and their associated behavior modification operators are inserted as daemons that can be checked at runtime during game execution.

In the next game run, the daemon manager checks these daemons during execution. As the ballistae's starts moving forward for attack towards the peon and a nearby axe-thrower is present, the new daemon conditions are satisfied and it gets activated. The daemon detects the two failed condition of attacking inappropriate enemy unit and attacking it from an inappropriate location and perform appropriate revisions on it. The existing behaviors related to the current attack towards the peon and current location are removed and appropriate new behaviors are inserted. These new behaviors have the attack location and the attacked enemy set based on the following heuristic. The heuristic for selecting the appropriate enemy is to pick the one that is most strong among the ones that

```
Daemon InappropriateAttack() {
precondition
{(attacklocation == unsuitable)
(attackedenemy != most_dangerous)}
    action changeAttackParameters();
    action changeAttackedEnemy();}
```

Fig. 12. The figure shows the daemon that changes the attack behavior in response to unsuitable location and enemy for an attack behavior. Unsuitable location and enemy are detected using routines (not shown here) that perform operations over the location and enemy parameters of the original attack behavior.

are currently alive and is in attacking range. The heuristic for selecting the appropriate location is based on picking the location for attack such that it is currently not in the attacking range of the enemy.

D. Experimental Results

To evaluate our behavior adaptation approach, we conducted two sets of experiments by turning the behavior adaptation on and off respectively. Table V provide both these results. The experiments were conducted on 8 different variations of the well known map "Nowhere to run nowhere to hide" (NWTR). NWTR maps have a wall of trees separating the opponents that introduces a highly strategic component in the game (one can attempt ranged attacks over the wall of trees, or prevent the trees to be chopped by building towers, etc.). 6 different expert demonstration were used for evaluation from NWTR maps. The games were played against the built-in AI [3] that comes along with Wargus. Table V shows the results of the experiments with and without adaptation. NT indicates the number of traces. For each experiment 6 values are shown: W, D and L indicate the number of wins, draws and loses respectively. ADS and AOS indicate the average system score and the average opponent score (where the "score" is a number that Wargus itself calculates and assigns to each player at the end of each game). Finally, WP shows the win percentage. The right most row PI presents the improvement in win percentage comparing adaptation with respect to no adaptation. The bottom row shows a summary of the results. The results show that behavior adaptation leads to an improvement of the percentage of wins as well as the player score to opponent score ratio. An improvement occurs in all cases irrespective of the number of traces used. There were a few occasions when the behavior adaptation module introduced unwanted changes that degraded the system performance. The system lost the game with the adaptation turned on whereas it was winning without it. Overall system performance improved considerably (overall increase of 62.3% as shown in Table V). The issue of unwanted changes could be resolved with some future modifications (as discussed later in Section VI).

	No Behavior Adaptation						Behavior Adaptation						
NT	*W*	*D*	*L*	*ADS*	*AOS*	*WP(%)*	*W*	*D*	*L*	*ADS*	*AOS*	*WP(%)*	*PI(%)*
1	4	5	15	1253	1510	**16.67**	8	7	9	1834	483	**33.33**	100
2	5	6	13	1193	1008	**20.83**	7	8	9	1987	234	**29.17**	40.00
3	6	5	13	1092	1567	**25**	9	9	7	1990	794	**37.5**	50
4	5	7	12	1034	1243	**20.83**	8	6	10	1793	687	**33.33**	60
5	7	5	12	1343	1234	**29.17**	11	7	6	2190	1183	**45.83**	57.14
6	6	8	10	1432	1678	**25**	10	7	7	2245	1087	**41.67**	66.67
	33	36	75	7347	8240	22.92	53	42	49	120394468		36.8	62.3

TABLE V
EFFECT OF BEHAVIOR ADAPTATION ON GAME STATISTICS

VI. DISCUSSION

AI agents designed for real-time settings need to adjust themselves to changing circumstances as that allows them to improve their performance and remedy their faults. Agents typically designed for computer games, however, lack this ability. Real-time nature of the game, complex decision spaces and an interactive user make designing adaptive agents for games a challenging and hard problem. We have presented an approach for game agents situated in complex real-time interactive domains that handle these challenges. It is based on the idea that it is much more efficient to reason about plans and how to fix them than it is to reason directly about an interactive, real-time and non-deterministic domain to plan a course of action. This is exemplified by transformational planning, which we extended in order to apply to such a domain. Our approach is based on the idea that failed experiences in problem solving provide both humans and artificial systems with strong cues on what needs to be learned. When an agent fails, it needs to learn to recover from the particular failure and eliminate the causes of this failure so that it does not repeat the same mistake in the future. This is exemplified by FDL, which we extended in order to apply to our game domain. The game playing agents learn from their failed experiences, improve their performance and avoid their mistakes. The approach is based on revising complex behavior sets for our game domains using novel failure detection techniques, a vocabulary of failure patterns pertinent to game domains and behavior modification strategies that revise rich constructs of our behavior representation language. Failure detection from execution trace, can be very expensive and time consuming as these traces can be very long in these game domains. In order to deal with this expensive search and achieve real-time performance, we use a set of pre-compiled patterns of failures that can be used that can help identify the cause of each particular failure by identifying instances of these patterns in the trace [1], [14].

Adaptive approaches can be used to improve performance along various metrics namely:

- Believable Character Behavior: the goal of the adaptive AI here is to behave in accordance with their personality.
- Strong AI Performance: the goal of the resulting adaptive AI is to achieve a more winning percentage and thus provide a more challenging opponent to the player.
- Adaptation to Challenge Level: the goal of the adaptive game AI is to continuously scale a game's difficulty level to the point that the human player is challenged, but not completely overpowered.
- Better Player Experience: the goal of the resulting adap-

[3]On a higher level, the strategy for the built-in AI is to: a) train a wave of units, b) send it to attack, c) strengthen defenses and advance technology and d) repeat the steps a), b and c)

tive AI is to provide the player with a better playing experience.

The adaptive AI approach in our domains focuses on improving the performance metrics singularly i.e. it aims to improve on one aspect and doesn't claim on improving on the other dimensions. However it can be argued that improvement along one performance metric could possibly also result in improvement along other metrics as well.

Game development is an expensive pursuit with a lot of money riding on their success. Game developers, as a result, want to use approaches that they can fully test for at design time. Agents that can adapt their behavior sets at run-time is a relatively new approach towards developing AI agents for games. Even if today, the game companies are not employing run time adaptation, in the future we expect to see adaptive games that can adapt themselves to unforeseen situations. Our research would help build a foundation for techniques which can eventually impact game developers. Exploring these techniques from a scientific perspective motivates future type of games that can learn from their own experiences and change themselves to unforeseen circumstances.

As game complexity increases in the future, creating hand authored behavior sets would require huge engineering effort. As a result of increase in complexity, manually behavior sets are likely to contain design issues. Adaptive approaches can provide benefits by helping game developer identify these issues when these approaches are integrated as part of the debugging interface as we have shown in our earlier work [18].

Revisions to the failed behaviors, at times, can cause inconsistencies that lead to a degradation in performance. These issues need to be addressed through a mechanism where the system keeps track of the system performance with the introduction of modifications. If the system performance degrades, it could realize that the adaptation is causing unwanted changes in system performance. Another way to tackle the issue of introducing unwanted changes would be to learn the important failure pattern that should be addressed. The system could keep track of successful (where system wins) and unsuccessful (where system loses) traces and the corresponding failure patterns present in them. The system could address only the failure patterns present in unsuccessful traces based on the heuristic that the ones common across successful and unsuccessful might not be important to address.

VII. Related Work

A. *Behavior Adaptation*

A character's behavior set can be considered a reactive plan which dictates what they should do under different conditions. Runtime behavior modifications can thus be considered a problem of runtime reactive-plan revision. One approach to runtime plan revision is to combine deliberative or generative planning with a reactive layer such that the deliberative planner can regenerate and replace failing portions of the reactive plan. In the AI planning community, there has been previous work on techniques for combining deliberative and reactive planning. For example, Atlantis [19] and 3T [20] are all aimed at combining deliberative and reactive components. Unfortunately they all, to varying degrees, make classical planning assumptions and are thus not applicable to our domain of interest, real-time interactive games. These approaches, furthermore, treat reactive plans as black boxes; planning sequences of black-boxed reactive plans, but not modifying the internals of the reactive plans themselves. Our approach directly modifies the internal structure of existing reactive behaviors. Behavior-set rewriting can be cast as a transformational planning problem. In transformational planning, the goal is to improve an existing reactive plan by applying a set of plan transformations. [21] describes such an approach where the agent tries to improve the expected utility of its plan in a world where its job is to transport balls from one location to another through an obstacle-filled space. More recently, other transformational planning approaches have used temporal projection of a robot's plan to detect problems with the plans using a causal model of the world to represent the effects of their actions [22]. These approaches, although promising, are of limited usefulness for us. They require a detailed casual model of the world. In our domain, we have neither the time for extended projective reasoning nor can we perform accurate projection due to the interactive and stochastic nature of game domains.

More recent work has relaxed these assumptions and has been applied to more complex game domains. Introspect system, for example, observe its own behavior so as to detect its own inefficiencies and repair them. The system has been implemented for the game of Go, a deterministic, perfect information, zero-sum game of strategy between two players [23]. Ulam et. al. present a model based system for FreeCiv game [24] that uses a self-model to identify the appropriate Reinforcement Learning space for a specific task. Differently from their approach, our work explores repairing behaviors that fail and then learning to avoid future performance failures.

Guestrin et. al. [25] present a relational Markov decision process models for a limited real-time strategy game scenario as the MDP's complexity grew exponentially with the number of units. Their approach is a learned static policy that cannot be re-trained online during actual game play to accommodate such changes. In our approach the game agent learns from its own experiences and modifies its internal processing based on the issues in the executing behavior. Another body of related work applied to complex game domains is in Adaptive AI [4]. Dynamic Scripting is a technique based on reinforcement learning, that is able to generate "scripts" by drawing subsets of rules from a pre-authored large collection of rules. The main difference with our approach is that dynamic scripting creates new scripts only by drawing different subsets of rules, but not by modifying those rules. In the approach presented in this paper, we will use meta-reasoning to adapt the behaviors of the base system.

B. *Behavior Authoring In Interactive Games*

Simple finite state automaton have been used to author game characters [5]. While finite state automata are easy to develop, they are predictable. Over long game sessions, a character's static behavioral repertoire may result in repetitive behavior which harms the believability of the characters [26]. Therefore, many developers of computer games and robotic toys have turned to hierarchical, probabilistic and fuzzy state machines [27]. The advantage of these systems is that layered control enables authoring sophisticated behaviors, and probabilistic transitions makes the actual behavior of the agent nondeterministic, less predictable, and more realistic. Other game developers have turned to scripting languages which allow arbitrarily sophisticated behaviors. Common scripting languages, such as Python [28], have been adopted by some companies for AI scripting [29], [30]. Some other companies have used Lua [31] for scripting AI behaviors [32]. These scripting languages allow arbitrarily sophisticated behaviors [33]. However, creating AI using scripting languages can be very labor intensive. Computer game manufacturers typically employ dozens of engineers to perfect very simple game AI(for examples, see the Postmortems in Game Developer Magazine, e.g. [34], [35]). The static nature of the scripted behaviors means that when the behaviors fail to achieve their desired purpose, the game AI is unable to identify such failure and will continue executing them.

VIII. Conclusions and Future Work

Our goal is to create adaptive agents for complex real-time interactive domains that learn from their experience, improving their performance and avoid their mistakes. In this paper, we have presented our approach to behavior adaptation for real-time game playing agents that achieves this. Our approach was implemented and tested in two domains, one a game world containing two embodied characters and the other a real-time strategy game Wargus.

As part of the future work, in our Tag work, to increase the transformational power of our system we are adding more behavior adaptation operators, which have several effects. First, as the number of operators increases, the time required to reason about them and finds the applicable set increases. Second, operators for more complex scenarios may have a lower success rate, requiring us to focus the search through behavior transformation space. It will become necessary for the reasoning layer to learn which operators are best applicable in which situations, such that fewer operators have to be tried. These characteristics of the problem make a case-based approach, as a form of speedup learning, very attractive.

There are number of research directions we plan to undertake for Behavior adaptation in Wargus. Currently the system applies the behavior modification routines without keeping track of whether and which of the changes introduced, improve in-game performance during execution over long term. As we see from our experimentation results, this is not an issue as overall performance of the system improves with the introduced behavior adaptations. However, a few times some of the introduced changes cause unwanted revisions of the behavior causing a degradation in system performance. We plan to address this issue in the future.

a) **Acknowledgements***:* The authors would like to thank Michael Mateas, author of the ABL language and collaborator on the Tag behavior modification project along with Peng Zang.

References

[1] M. T. Cox and A. Ram, "Failure-driven learning as input bias," in *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, 1994, pp. 231–236.

[2] K. J. Hammond, "Learning to anticipate and avoid planning problems through the explanation of failures," in *Proceedings of the Fifth National Conference on Artificial Intelligence*, 1986, pp. 556–560.

[3] A. Ram, S. Ontañón, and M. Mehta, "Artificial intelligence for adaptive computer games," in *FLAIRS 2007*, 2007.

[4] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, and E. Postma, "Adaptive game ai with dynamic scripting," *Mach. Learn.*, vol. 63, no. 3, pp. 217–248, 2006.

[5] S. Rabin, *AI Game Programming Wisdom*. Charles River Media, 2002.

[6] M. Mateas and A. Stern, "Facade: An experiment in building a fully-realized interactive drama." in *Game Developer's Conference: Game Design Track*, 2003.

[7] D. Aha, M. Molineaux, and M. Ponsen, "Learning to win: Case-based plan selection in a real-time strategy game," in *ICCBR'2005*, ser. LNCS, no. 3620. Springer-Verlag, 2005, pp. 5–20.

[8] M. Buro, "Real-time strategy games: A new AI research challenge," in *IJCAI 2003*. Morgan Kaufmann, 2003, pp. 1534–1535.

[9] M. Peot and D. Smith, "Conditional nonlinear planning," in *First International Conference on AI Planning Systems*, 1992.

[10] D. S. Weld, C. R. Anderson, and D. Smith, "Extending graphplan to handle uncertainty and sensing actions," in *Proceedings of AAAI*, 1998.

[11] M. Minsky, *Steps Towards Artificial Intelligence*. McGraw-Hill, 1963.

[12] M. T. Cox and A. Ram, "Introspective multistrategy learning: On the construction of learning strategies," Tech. Rep., 1996.

[13] E. Stroulia and A. K. Goel, "Functional representation and reasoning in reflective systems," *Journal of Applied Intelligence*, vol. 9, pp. 101–124, 1995.

[14] J. Sussman, *A Computational Model of Skill Acquisition*. American Elsevier, 1975.

[15] M. Mateas and A. Stern, "A behavior language for story-based believable agents," *IEEE intelligent systems and their applications*, vol. 17, no. 4, pp. 39–47, 2002.

[16] S. Reilly, "Believable social and emotional agents," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, 1996.

[17] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Case-based planning and execution for real-time strategy games," in *Proceedings of ICCBR-2007*, 2007, pp. 164–178.

[18] S. Virmani, Y. Kanetkar, M. Mehta, S. Ontañón, and A. Ram, "An intelligent ide for behavior authoring in real-time strategy games," in *AIIDE*, 2008.

[19] E. Gat, "Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots," in *Proceedings of the AAAI Conference*, 1992.

[20] P. Bonasso, J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack, "Experiences with an architecture for intelligent reactive agents," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, pp. 237–256, 1997.

[21] D. McDermott, "Transformational planing of reactive behavior." Yale University, Tech. Rep., 1992, research Report YALEU/DCS/RR-941.

[22] M. Beetz, "Structured reactive controllers a computational model of everyday activity," in *Proceedings of the Third International Conference on Autonomous Agents*, 2000.

[23] T. Cazenave, "Metarules to improve tactical go knowledge," *Inf. Sci. Inf. Comput. Sci.*, vol. 154, no. 3-4, pp. 173–188, 2003.

[24] P. Ulam, J. Jones, and A. K. Goel, "Combining model-based meta-reasoning and reinforcement learning for adapting game-playing agents." in *AIIDE*, 2008.

[25] C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia, "Generalizing plans to new environments in relational mdps," in *In International Joint Conference on Artificial Intelligence (IJCAI-03*, 2003, pp. 1003–1010.

[26] M. Saltzman, *Game Design: Secrets of the Sages*. Brady Publishing, 1999.

[27] B. Schwab, *AI Game Engine Programming*. Charles River Media, 2004.

[28] P. T. P. Language, "http://www.python.org," 2008.

[29] E. Arts, "http://www.ea.com/official/battlefield/battlefield2/us/," 2006.

[30] F. Games, "http://www.firaxis.com/games/game_detail.php?gameid=6," 2006.

[31] L. T. P. Language, "http://www.lua.org," 2008.

[32] B. Corp., "http://www.bioware.com/games/mdk2/," 2006.

[33] I. Millington, *Artificial Intelligence for Games*. Morgan Kaufmann, 2006.

[34] R. Huebner, "Postmortem: Nihilistic softwares vampire: The masquerade redemption." in *Game Developer Magazine*, July 2000, pp. 44–51.

[35] W. Spector, "Postmortem: Ion storms deus ex," in *Game Developer Magazine*, November 2000, pp. 50–58.