Towards Runtime Behavior Adaptation for Embodied Characters

Peng Zang[1], Manish Mehta[1], Michael Mateas[2], Ashwin Ram[1]

1. College of Computing
   Georgia Institute of Technology
   Atlanta, GA

2. Computer Science Department
   University of Santa Cruz
   Santa Cruz, CA

Abstract

Typically, autonomous believable agents are implemented using static, hand-authored reactive behaviors or scripts. This hand-authoring allows designers to craft expressive behavior for characters, but can lead to excessive authorial burden, as well as result in characters that are brittle to changing world dynamics. In this paper, we present an approach for the runtime adaptation of reactive behaviors for autonomous believable characters. Extending transformational planning, our system allows autonomous characters to monitor and reason about their behavior execution and to use this reasoning to dynamically rewrite their behaviors. In our evaluation, we transplant two characters in a sample tag game from the original world they were written for into a different one, resulting in behavior that violates the author intended personality. The reasoning layer successfully adapts the character's behaviors so as to bring its long-term behavior back into agreement with its personality.

1 Introduction

In interactive games, embodied characters typically have their own personalities, affecting the way they act in the game. Authors usually create such characters by writing behaviors or scripts that describes the character's reaction to all imaginable circumstances within the game world. This approach of authoring characters presents several difficulties. First, when authoring a character's behavior set, it is hard to imagine and plan for all possible scenarios it might encounter. Given the rich, dynamic nature of game worlds, this can require extensive programming effort [Mateas and Stern, 2003]. Second, over long game sessions, a character's static behavioral repertoire may result in repetitive behavior. Such repetition harms the believability of the characters. Third, when behaviors fail to achieve their desired purpose, characters are unable to identify such failure and will continue them. Ideally, we want a self-adapting behavior set for characters, allowing characters to autonomously exhibit their author-specified personalities in new and unforeseen circumstances, and relieving authors of the burden of writing behaviors for every possible situation.

In the field of embodied characters, there has been little work on characters that are introspectively aware of their internal state, let alone characters that can rewrite themselves based on deliberating over their internal state. In this paper, we introduce an approach to runtime rewriting of character behaviors. Agents keep track of the status of their executing behaviors, infer from their execution trace what might be wrong, and perform appropriate revisions to their behaviors. This approach to runtime behavior transformation enables characters to autonomously adapt during execution to changing game situations, taking a first step towards automatic generation of behavior that maintains desired personality characteristics.

The rest of this paper is organized as follows. We first discuss various approaches to this problem and introduce our specific approach. We then present our system in Section 3. In Section 4, we discuss our empirical evaluation. Finally, we situate our work in the literature and conclude.

2 Approaches to behavior transformation

A character's behavior set can be considered a reactive plan dictating what it should do under various conditions. Runtime behavior modifications 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 domain, all these assumptions are violated. Characters are constantly interacting with the user, actions are non-deterministic and their effects are often difficult to quantify. Furthermore, as we are interested in believable, embodied characters, additional challenges are imposed. For instance, parallel actions and the ability for characters to change and express emotion are key for characters to maintain their believability [Loyall, 1997]. Finally, our 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 [Peot and Smith, 1992] and Sensory Graph Plan [Weld et. al, 1998] 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 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. They provide poor affordances for authorial-specified, expressive control of behavior. In game worlds, it is imperative that game designers retain control of the overall flavor of character behavior.

Transformational planning (TP) is an approach that can potentially deal with the complexity and nondeterminism of our problem 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 such as ABL (see Section 3). Thus we developed novel behavior transformations and techniques for blame assignment, extending TP such as to enable us to leverage this approach in our system.

3. Behavior Transformation System

Before detailing our approach and system, we first present our game scenario which will help frame our discussion. Our current 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 1) is composed of a reactive layer which handles the real-time interactions, and a reasoning layer responsible for monitoring the character's state and making repairs as needed.

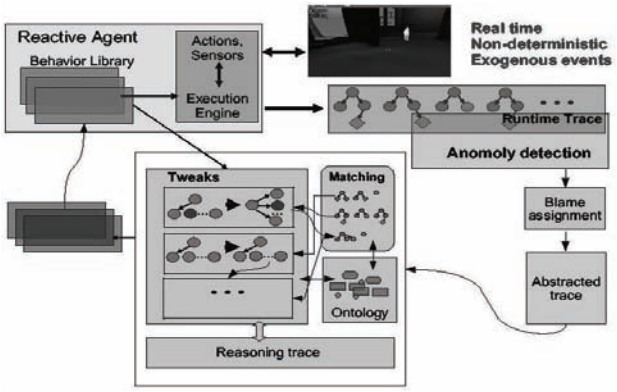

Figure 1: The figure shows the architectural diagram for our behavior transformation system.

3.1 The Reactive Layer

Our game environment presents a certain set of challenges for the reactive layer. First, a real-time game domain requires the reactive layer to have a fast runtime processing component with a short sense-decide-act loop. Second, the game world's interactive nature entails that the reactive layer handles conditional execution appropriately and provide the ability to support varying behaviors under different situations at runtime. Finally, for game worlds containing embodied, believable characters, the reactive layer must provide support for the execution of multiple, simultaneous behaviors, allowing characters to gaze, speak, walk around, gesture with their hands and convey facial expressions, all at the same time.

To meet these requirements for our domain we use A Behavior Language (ABL) as the reactive layer. ABL is explicitly designed to support programming idioms for the creation of reactive, believable agents [Mateas and Stern, 2004]. 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 [Mateas and Stern, 2003]. To facilitate our discussion of the reasoning layer, we first describe ABL.

3.1.1 ABL as a programming Language

A character authored in ABL is composed of a library of behaviors, capturing the various activities the character can perform in the world. Behaviors are dynamically selected to accomplish goals - different behaviors are appropriate for accomplishing the same goal in different contexts. For ex-

ample, the goal of expressing anger can be accomplished through either a behavior that screams or a behavior that punches a hole in the wall. Behaviors themselves consist of a collection of sequential or parallel steps. Steps can be subgoals, mental acts (bits of computation, often used to update character memory), or primitive acts (actions, such as performing an arm gesture, that are native to the game world). The currently active goals and behaviors are captured in an intention structure called the active behavior tree. During execution, steps may fail (e.g. no behavior can be found to accomplish a subgoal, or a physical act fails in the game world), potentially causing the enclosing behavior to fail. ABL provides numerous step and behavior annotations that modify the cascading effects of success and failure. When a behavior fails, ABL typically attempts to find an alternate behavior to accomplish the goal; if no appropriate alternative behavior is found, the goal fails. Behavior preconditions are used to find appropriate behaviors for accomplishing a goal in the current context. Continuously monitored conditions, such as context conditions and success tests, provide immediate, reactive response. The various kinds of conditions test against working memory, which contains various working memory elements (WMEs) that encode both currently sensed information and agent-specific internal state (e.g. emotional state).

3.2.2 ABL as a runtime execution architecture

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 active behavior tree. 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.

3.2 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 behavior modification operators so as to repair the offending behaviors identified during blame assignment.

3.2.1 Anomaly detection and blame assignment

One of the essential requirements of a reasoning system responsible for runtime behavior modification is to detect when modification should be carried out. We need a way for authors to specify 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 model of emotion [Reilly, 1996]. Emotion values serve as compact representations of long-term behavior. The author specifies personality-specific constraints on behavior by specifying nominal bounds for emotion values. When an emotion value exceeds the bounds specified 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 specifying the character personality.

A second requirement on the reasoning module is 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.

3.2.2 Reasoning about traces

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 (also called *tweaks*) that can 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 1 for a subset of the annotations used in our current system) and an ontological description of the behaviors, their types, their relationships, and what they accomplish (see Figure 2 for a subset of the ontology).

Annotation	*Meaning*
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 1: Some example annotations.

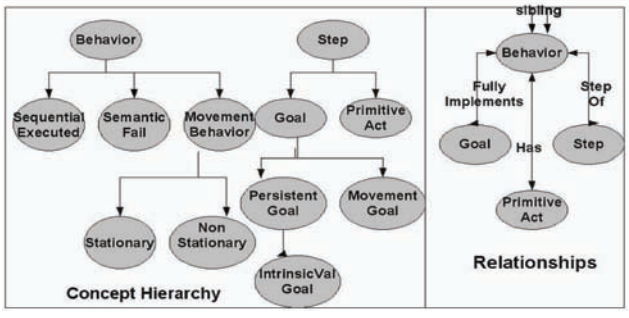

Figure 2: The figure shows the concepts hierarchy and relationships used during the tweaking process

Failure Pattern	*Behavior Modification Operator (modops)*
Insufficiently instantiated goal. Goal must be SMF	Loosen preconditions of behavior closest to matching (or a clone thereof)
Failing goal, all behaviors fail. Goal must have the annotation SMF. Modops only applicable to MBIG behaviors.	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 behavior. Behavior must be SMF	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
Failing parallel behavior. Behavior must be SMF	Modify failing step annotations or change its parameters
	Make behavior sequential.
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.
	Make the stopping condition succeed

Table 2: Some example failure patterns and their associated behavior modification operators.

Our system contains a collection of modification operators based on the currently defined ontological categories. Given that blame assignment has provided a behavior to modify, the applicability of a modification operator 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 operators are 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. On an implementation level, these failure patterns are encoded loosely as finite state machines that look for patterns in the execution trace. Figure 3 shows an example failure pattern that recognizes when a problematic behavior is repeatedly failing. Table 2 shows the association between modification operators and failure patterns.

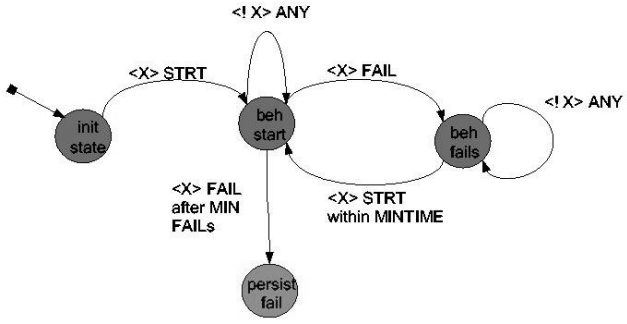

Figure 3 : The figure shows a matcher for the failure pattern: persistent failing behavior. In this diagram, <X> denotes the behavior in question, STRT (start), FAIL and ANY (start, fail, succeed) represents the status of the behavior and "after MINFAILs" or "within MINTIME" are conditions

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 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 operators to try on the offending behavior. The order in which the operators are tried is defined through annotated priority specifications. Operators are tried one at a time until one succeeds (operators can fail if the behavior they are tweaking lacks the structural prerequisites for the application of the operator). The modified behavior is compiled and reloaded into the agent.

3.5. An illustrative 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 4). 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 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 modification 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 2). The set of associated behavior modification operators 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 operator 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 operator 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.

```
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();}
```

Figure 4: Example behaviors defined in ABL

4. Experimental Evaluation

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 provided 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 5 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.

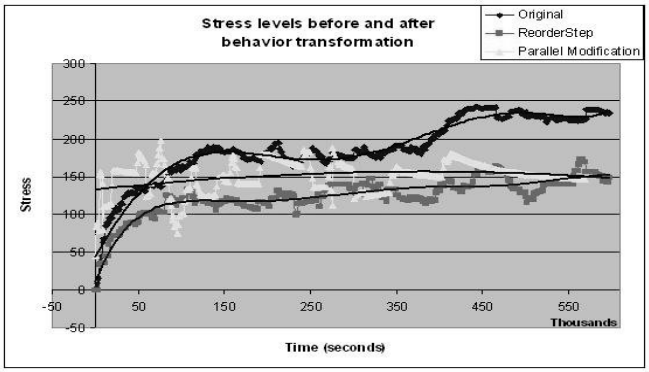

Figure 5: The figure shows the results for average stress level from the evaluation experiment

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 6) 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 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 en-

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

Figure 6: The figure shows the modified behavior

sure no one is following instead of the other way around. This results in behavior as good if not better than the parallel version.

5. Related Work

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 [Gat, 1992] and 3T [Bonasso et. al., 1997] 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. [McDermott, 1992] 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 [Beetz, 2000]. 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.

6. Conclusion

Our goal is to relieve the human author of the burden of programming behaviors for all possible situations, while still providing the author with expressive control over a character's behavior through both the hand-authoring of some behaviors plus the specification of a behavior contract. In this paper, we presented an approach for runtime behavior transformation for reactive, real-time agents (with a focus on believable, embodied characters) that achieves this. 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 in an effort to plan a course of action. This is exemplified by transformational planning, which we extended in order to apply to such a domain. In particular, we developed novel behavior transformations, specification of a behavior contract capturing personality invariants the author wishes the character to maintain, and a mechanism for blame assignment that uses the violated constraints plus behavior ontology to determine which behavior(s) must be repaired. Experiments showed that when Jack's emotion levels violated constraints due to unexpected world conditions, our system found transformations which generated a modified behavior set that was able to successfully satisfy the emotion constraints and maintain Jack's personality.

References

[McDermott, 1992] Drew McDermott. *Transformational Planing of Reactive Behavior.* Research report. YALEU/DCS/RR-941, Yale University, 1992.

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

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

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

[Mateas and Stern, 2004] M. Mateas, and A. Stern A Behavior Language: Joint action and Behavioral Idioms. In *Life-like Characters. Tools, Affective Functions and Applications*, Springer, 2004.

[Reilly, 1996] S. Reilly. *Believable Social and Emotional Agents.* Ph.D. Thesis. Technical Report CMU-CS-96-138, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. May 1996.

[Weld et. al, 1998] D. S. Weld, C. R. Anderson, D. Smith. Extending Graphplan to Handle Uncertainty & Sensing Actions. In *Proceedings of AAAI* 1998.

[Peot and Smith, 1992] M. Peot and D. Smith. Conditional Nonlinear Planning, *First International Conference on AI Planning Systems*, 1992.

[Mateas and Stern, 2003] M. Mateas, and A. Stern. Facade: An Experiment in Building a Fully-Realized Interactive Drama. In *Game Developer's Conference: Game Design Track*, San Jose, California, March 2003.

[Loyall, 1997] *Believable Agents: Building Interactive Personalities* A. Bryan Loyall. Ph.D. Thesis. Technical Report CMU-CS-97-123, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. May 1997.