Generating Ambient Behaviors in Computer Role-Playing Games

Maria Cutumisu[1], Duane Szafron[1], Jonathan Schaeffer[1], Matthew McNaughton[1], Thomas Roy[1], Curtis Onuczko[1], and Mike Carbonaro[2]

[1] Department of Computing Science, University of Alberta, Canada
{meric, duane, jonathan, mcnaught, troy, onuczko}@cs.ualberta.ca
[2] Department of Educational Psychology, University of Alberta, Canada
{mike.carbonaro}@ualberta.ca

Abstract. Many computer games use custom scripts to control the ambient behaviors of non-player characters (NPCs). Therefore, a story writer must write fragments of computer code for the hundreds or thousands of NPCs in the game world. The challenge is to create entertaining and non-repetitive behaviors for the NPCs without investing substantial programming effort to write custom non-trivial scripts for each NPC. Current computer games have simplistic ambient behaviors for NPCs; it is rare for NPCs to interact with each other. In this paper, we describe how generative behavior patterns can be used to quickly and reliably generate ambient behavior scripts that are more realistic, entertaining and non-repetitive, even for the more difficult case of interacting NPCs. We demonstrate this approach using BioWare Corp.'s Neverwinter Nights game.

Keywords: Ambient behavior, non-player character, intelligent agents, scripting language, generative pattern, collaborative behavior, computer games.

1 Introduction

A computer role-playing game (CRPG) is an interactive story where the game player controls an avatar called a player character (PC). Quickly and reliably creating engaging game stories is essential in today's market. Game companies must create intricate and interesting storylines cost-effectively and realism that goes beyond graphics has become a major product differentiator. Using AI to create non-player characters (NPCs) that exhibit near-realistic ambient behaviors is essential, since a richer background "tapestry" makes the game more entertaining. However, this requirement must be put in context: the storyline comes first. NPCs that are not critical to the plot are often added at the end of the game development cycle, only if development resources are available. Consider the state-of-the-art for ambient behaviors in recent CRPGs. In Fable (Lionhead Studios), the NPCs wake at dawn, walk to work, run errands, go home at night, and make random comments about the disposition and appearance of the PC. However, the behaviors and comments are "canned" and repetitive and NPCs never interact with each other. The Elder Scrolls 3: Morrowind (Bethesda Softworks) has a huge immersive world. However, NPCs either wander around areas on predefined paths or stand still, performing a simple animation, never interacting with each other and ignoring the simulated day. In The Sims 2 (Electronic Arts), players control the NPCs (Sims) by choosing their behaviors. Each Sim chooses its own behaviors using a motivational system if it is not told what to do. The ambient behaviors are impressive, but they hinge on a game model (simulation) that is integral to this game and not easily transferable to other game genres, including CRPGs. Halo 2 (Bungie) is a first person shooter with about 50 behaviors, including support for "joint behaviors" [1]. The Halo 2's general AI model is described, but no model for joint behaviors is given. Façade [2] has an excellent collaborative behavior model for NPCs, but there are only a few NPCs, so it is not clear if it will scale to thousands of ambient NPCs. They also comment about the amount of manual work that must be done by a writer when using their framework. Other research includes planning, PaTNets, sensor-control-action loops [3], and automata controlled by a universal stack-based control system for both low-level and high-level animation control, but not in the domain of commercial-scale computer games. However, planning is starting to be used in commercial computer games in the context of Unreal Tournament [4]. Crowd control research involves low-level behaviors such as flocking and collisions and has recently been extended to a higher-level behavioral engine [5]. Group behaviors provide a formal way to reason about joint plans, intentions and beliefs. Our approach is dictated by the practical requirements of commercial computer games. Our model is robust, flexible, extendable, and scalable [6] to thousands of ambient NPCs, while requiring minimal CPU resources. Moreover, our generative pattern abstraction shields story designers from manual scripting and the synchronization issues of collaborative behaviors, and allows them to concentrate on story construction.

In most games, scripts control NPC behaviors. A game engine renders the story world objects, generates events on the objects, dispatches events to scripts and executes the scripts. Different stories can be "played" with the same game engine using story-specific objects and scripts. Programmers create game engines using programming languages such as C or C++. Writers and artists, who are not usually programmers [7], write game stories by creating objects and scripts for each story. The goal of our research is to improve the way game stories, not game engines, are created.

A writer may create thousands of game objects for each story. If a game object must interact with the PC or another game object, a script must be written. For example, BioWare Corp.'s popular Neverwinter Nights (NWN) (http://nwn.bioware.com) campaign story contains 54,300 game objects of which 29,510 are scripted, including 8,992 objects with custom scripts, while the others share a set of predefined scripts. The scripts consist of 141,267 lines of code in 7,857 script files. Many games have a toolset that allows a writer to create game objects and attach scripts to them. Examples are BioWare's Aurora toolset that uses NWScript and Epic Game's UnrealEd that uses UnrealScript.

The difficulties of writing manual scripts are well documented [8]. Writers want to create custom scripts without adapting predefined scripts or relying on a programmer to write custom scripts. However, story creation should be more like writing than programming.

ScriptEase (http://www.cs.ualberta.ca/~script) is a publicly available tool for creating game stories using a high-level menu-driven "programming" model. ScriptEase solves the non-programmer problem by letting the writer create scenes at the level of "patterns" [9]. A writer begins by using BioWare's NWN Aurora toolset to create the physical layout of a story, without attaching any scripts to objects. The writer then selects appropriate behavior patterns that generate scripting code for NPCs in the story. For example, in a tavern scene, behavior patterns for customers, servers and the owner would be used to generate all the scripting code to make the tavern come alive. Figure 1 shows a NWN tavern scene in which the ambient behaviors for several customers ("This place is getting better and better", "A walk is nice"), two servers ("Good crowd tonight", "I'm on my way, Traiani") and an owner (not shown) have been generated. The PC is at the front of the scene and determines the camera location. The ScriptEase generative pattern approach is much easier for non-programmers than manually scripting events, even if a library of behaviors such as the Memetic AI toolkit (http://www.memeticai.org) is used.

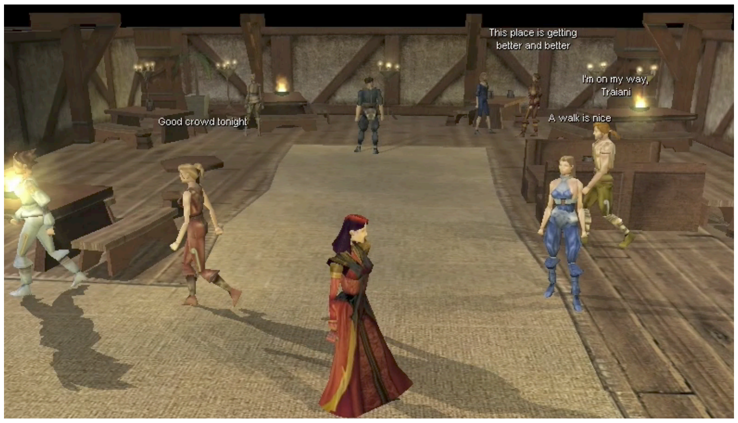

Figure 1. An NWN tavern scene with ScriptEase ambient behaviors.

We already showed that ScriptEase is usable by non-programmers, by integrating it into an interactive short story creation exercise in the Grade 10 high school English curriculum [10]. The version of ScriptEase that was used had a rich set of patterns for supporting interactions between the PC and inanimate objects such as doors, props and triggers. It also had limited support for plot and dialogue patterns (the subject of on-going work). We have now extended the generative pattern approach of ScriptEase to support the *ambient behaviors* of NPCs [11].

NPC interactions require concurrency control to ensure that neither deadlock nor indefinite postponement can occur, and to ensure that interactions are realistic. We constructed an NPC interaction concurrency model and built generative behavior patterns for it. We used these patterns to generate all of the scripting code for a tavern scene to illustrate how easy it is to use behavior patterns to create complex NPC interactions. The ambient background includes customers, servers and an owner going about their business but, most importantly, interacting with each other in a natural way, based on our novel approach to NPC ambient behaviors. It is the first time patterns have been used to generate behavior scripts for computer games. The research makes three key contributions: 1) rich backgrounds populated with interacting NPCs with realistic ambient behaviors are easy to create with the right model, 2) pattern-based programming is a powerful tool and 3) our model and patterns can be used to generate code for a real game (NWN).

2 Ambient Behavior Patterns

A CRPG tavern scene demonstrates ambient behavior patterns. We define three ambient behavior patterns for this scene: owner, server, and customer. Each behavior generates more complex interactions than most NPCs display in most CRPGs.

A behavior pattern is defined by a set of behaviors and two control models that select the most appropriate behavior at any given time. A behavior can be used *proactively* (P) in a spontaneous manner or *reactively* (R) in response to another behavior. Table 1 lists the behaviors used in the tavern. Some are used independently by a single NPC. For example, *posing* and *returning* to the original scene location are *independent behaviors*. This article addresses only high-level behaviors, since the NWN game engine solves low-level problems. For example, if the original location is occupied by another creature when an NPC tries to return, the game engine moves the NPC as close as possible and subsequent return behaviors may succeed. Behaviors that involve more than one NPC are *collaborative* (joint) *behaviors*. For example, an *offer* involves two NPCs, one to make the offer and one to accept/reject it.

The first column of Table 1 indicates whether a proactive behavior is independent or collaborative. Note that interactions with the PC are not considered ambient so they are not supported by ambient behavior patterns. The most novel and challenging ambient behaviors are the ones that use behaviors collaboratively (interacting NPCs). The second column lists the proactive behaviors. The letters in parentheses indicate which kind of NPC can initiate the proactive behavior. For a collaborative behavior, the kind of collaborator is given as part of the behavior name, e.g., the *approach random C* behavior can be initiated by a server or customer (S, C) and the collaborator is a random customer (C).

Table 1. Behaviors in the Server (S), Customer (C), and Owner (O) Patterns.

Behavior Type	Proactive Behavior	Reactive Chains
Independent	pose (S, C, O)	pose, done
	return (C, O)	return, done
	approach bar (S, C)	approach, done
	fetch (O)	fetch, done
Collaborative	approach random C (S, C)	approach, done
	talk to nearest C (C)	speak, speak, converse*
	converse with nearest C (C)	(speak, speak)+ done
	ask-fetch nearest S (C, O)	speak, fetch, receive, speak, done
	ask-give O (C)	speak, give, receive, speak, done
	offer-give to nearest C (O)	speak, decide, ask-give*; *(accept)* speak, decide, speak, done *(reject)*
	offer-fetch to nearest C (S)	speak, decide, ask-fetch*; *(accept)* speak, decide, speak, done *(reject)*

The third column of Table 1 shows the reactive chains for each proactive behavior. For example, the *ask-fetch* proactive behavior has a reactive chain where the initiator *speaks* (selecting an appropriate one-liner randomly from a conversation file), the collaborator *fetches* (goes to the supply room while speaking), the initiator *receives* something, the collaborator *speaks* and the *done* behavior terminates the chain. Each reactive chain ends in a *done* behavior, unless another chain is reused (denoted by an asterisk such as *converse** in the *talk* behavior). Each behavior consists of several actions. For example, a *speak* behavior consists of facing a partner, pausing, performing a speech animation and uttering the text. An entry marked with *()+* indicates that the parenthesized behaviors are repeated one or more (random) times. For example, the *converse* proactive behavior starts a reactive chain with one or more *speak* behaviors alternating between two NPCs. The *talk* proactive behavior starts a reactive chain with a *speak* behavior (a greeting) for each interlocutor, followed by a *converse* behavior. The *offer-give* (owner offers a drink) and *offer-fetch* (server offers to fetch a drink) proactive behaviors each have two different reactive chains (shown in Table 1) depending on whether the collaborator *decides* to accept or reject the offer.

The writer uses a simple process to create these behaviors, with no programming (script writing) involved. Begin by using the Aurora toolset to construct the tavern area, populate it with customers, servers and an owner, and save the area in a module. Open the module in ScriptEase and perform three kinds of actions. First, create one instance of the server, customer and owner patterns respectively, by selecting the patterns from a menu. Second, set the options of each pattern instance to game objects and/or values using dialog boxes. For example, the server has three options that must be set – the `Actor`, the `Bar` and the `Customer`. Each is set to an object constructed using the Aurora toolset. Note that one pattern instance can generate code that is used by all game objects with the same tag (`Server`), created in the Aurora toolset. The third step is to select the "Save and

Compile" menu command to generate NWScript code (for the entire tavern scene) that could be edited in the Aurora toolset if desired.

3 Evaluating Ambient Behavior Patterns

The simplicity of the process hides the fact that a large amount of scripting code is generated to model complex collaborative interactive behaviors. In fact, 889 lines of NWScript code are generated for the server, while 1087 and 886 lines are generated for the customer and owner respectively. It takes about 30 minutes to use ScriptEase to generate this code, whereas it takes several days to write the code manually.

The generated code is efficient, producing ambient behaviors that are crisp and responsive, with no perceptible effect on response time for PC movement and actions. The NPCs interact with each other flawlessly with natural movements. A scene with 18 customers, 2 servers and one owner was left to play for hours without any deadlock, degradation in performance, repetition or indefinite postponement of behaviors for any actor. There is no limitation on the number of NPCs supported by our model. However, since NPC behaviors are only active if the NPC is close to the PC, a larger number of active NPCs would not be necessary in practice.

Since the effectiveness and performance of ambient behaviors is best evaluated visually, we illustrate our approach using a series of movies captured from actual game-play (http://www.cs.ualberta.ca/~script/movies/tavern/). These tavern scene patterns are general enough to generate scripts for other scenes. For example, in a house scene, the customer pattern can be used for the inhabitants, the server pattern for a butler, and the owner pattern for a cook. The butler interacts with the inhabitants, fetching for them by going to the kitchen. The inhabitants talk amongst themselves and the cook occasionally fetches supplies. Our approach handles group (crowd) behaviors in a natural way. The customers constitute an example of a crowd – a group of characters with the same behavior, but each selecting different behaviors based on local context.

To determine the range of CRPG ambient behaviors that can be accommodated by patterns, we conducted a case study for the Prelude chapter of the NWN official campaign story. The original code used ad-hoc scripts to simulate collaborative behaviors. We removed all of the manually scripted ambient NPC behaviors and replaced them with behaviors generated from patterns. Six new ambient behavior patterns were identified: *Poser*, *Bystander*, *Speaker*, *Expert*, *Striker*, and *Duet*, although *Duet* is actually a meta-pattern, as described later. The simplest ambient behavior is a *Poser* where the NPC performs a simple animation. The pattern provides default initial values for the specific animation and its duration. For example, one *Poser* in the Prelude is an injured man whose ambient behavior is an animation to beg for help. A door guard is another *Poser*. In the original Prelude module, this door guard character displays a default standing animation. We improved its behavior by allowing the character to also randomly utter a sentence from a conversation file.

A *Duet* is a more complex ambient behavior pattern that expresses collaboration between two NPCs. A *Duet* is a meta-pattern that allows game designers to create a series of collaborative patterns by combining other patterns. In the Prelude, there are three types of collaborative behaviors that we abstracted using the *Duet* meta-pattern. Although the original Prelude does not have collaborative behaviors per se, the story designers attempted to simulate collaborative behaviors in many scenes. For example, in the original Prelude, six NPCs grouped in pairs mimic a conversation by facing each other and performing independent speaking gestures. We replaced the manual scripting code that controls these six NPCs by code automatically generated from instances of the *Duet-converser-converser* ambient behavior pattern. This pattern constitutes a true collaborative behavior involving two *converse* behaviors that alternate for the two NPCs so that the conversation between them seems natural. An NPC waits for its collaborator's reply before it provides a response or initiates a new collaboration. The left pane of Figure 2 shows the generated *Duet-converser-converser* NPCs.

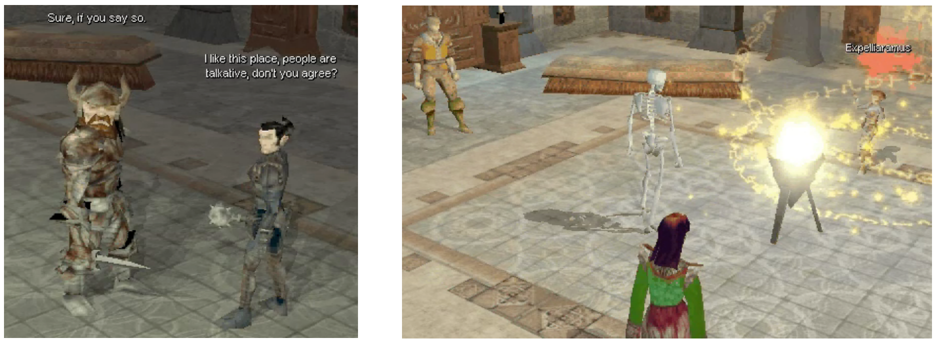

Figure 2. Generated ambient behaviors in the Prelude: Duet-converser-converser and Duet-spawner-destroyer.

Another example of simulated collaboration in the original Prelude involves a pair of spellcaster NPCs. Two NPCs successively cast spells on a combat dummy by applying a delay to one of the NPCs, so that their actions appear to alternate. A third example has a pair of NPCs performing another type of training: one NPC spawns a skeleton and the other destroys the spawned skeleton. The right pane of Figure 2 shows the replacement scene as generated from a *Duet-spawner-destroyer* behavior pattern. In the original code, a different ad-hoc technique was used to compensate for not having true collaboration support: one NPC spawns skeletons and the other destroys any perceived skeletons – no delay was used. This does not constitute true collaboration and the various ad-hoc techniques used to compensate for a lack of support for true collaboration make the scripts hard to understand and maintain. Figure 3 shows the manually written NWScript code for the spawner and destroyer NPCs.

NWScript code for Ansel (the spawner):
OnPerception:
```
void main() {
  if(GetIsPC(GetLastPerceived()) && GetLastPerceptionSeen())
    {
      SignalEvent(OBJECT_SELF,EventUserDefined(0));
    }
}
```
OnUserDefined:
```
void main() {
  if(IsInConversation(OBJECT_SELF) == FALSE &&
    GetIsDead(OBJECT_SELF) == FALSE)
    {
      location lLoc = GetLocation(GetNearestObjectByTag(
        "WP_Skeleton"));
      ActionCastFakeSpellAtLocation(SPELL_ANIMATE_DEAD,lLoc);
      CreateObject(OBJECT_TYPE_CREATURE,"M1Q0BSUM_SK",lLoc);
    }
  DelayCommand(30.0,SignalEvent(OBJECT_SELF,
    EventUserDefined(0)));
}
```

NWScript code for Tabitha (the destroyer):
OnPerception:
```
void main() {
  object oPerceived = GetLastPerceived();
  if(GetRacialType(oPerceived) == RACIAL_TYPE_UNDEAD &&
    GetLastPerceptionSeen() &&
    GetTag(oPerceived) != "M0Q0_SKELETON" &&
    IsInConversation(OBJECT_SELF) == FALSE)
    {
      ActionCastSpellAtObject(SPELLABILITY_TURN_UNDEAD,
        oPerceived,METAMAGIC_ANY,TRUE);
    }
}
```

Figure 3. Manually written NWScript code for the spawner and destroyer NPCs.

When the spawner NPC (`Ansel`) perceives the PC, it executes the script code attached to the *OnPerception* event, firing a user defined event on itself. This causes the code attached to its *OnUserDefined* event to be executed. As a result, `Ansel` spawns a skeleton with tag "`M1Q0BSUM_SK`" at a waypoint "`WP_Skeleton`" and casts a fake spell at this waypoint. Then, `Ansel` fires the same user defined event with a 30 seconds delay, so that a new skeleton is spawned. The destroyer NPC (`Tabitha`) casts a spell that destroys any creature with the racial type undead perceived that is different from a skeleton with tag “M0Q0_SKELETON” already located in the room. The intent of the designer is to simulate a collaborative spellcaster training of these two NPCs. However, if the destroyer NPC takes more time to destroy the spawned skeleton, then the spawner can create another skeleton, since the spawner generates skeletons every 30 seconds, regardless of the destroyer’s actions. Their collaboration is achieved not by communication between the NPCs, but through the common object of their training: the skeleton. This does not reflect a true NPC collaboration.

The *Duet-spawner-destroyer* pattern generates true collaborative scripts that ensure synchronization between the NPCs. The second NPC’s destroy behavior does not start until the first NPC finishes its spawn behavior. Contrast the complex hand-written code shown in Figure 3 with Figure 4 that shows how a ScriptEase *Duet-spawner-destroyer* pattern can be used to generate code by simply instantiating an instance of the pattern and selecting three options: the actor (`Ansel`), the partner (`Tabitha`) and the target creature to be spawned-destroyed (`Spawned Skeleton`).

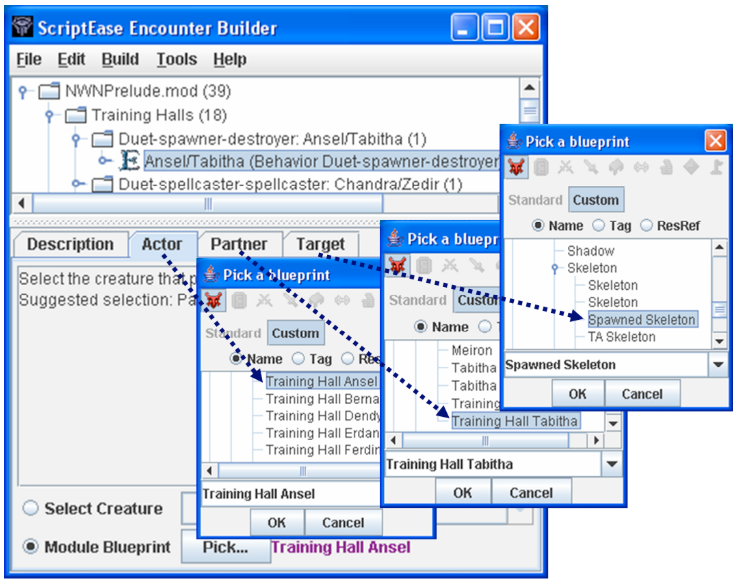

Figure 4. An instance of the Duet-spawner-destroyer pattern in the NWN Prelude.

Collaborative behaviors are rare in CRPGs because their existence complicates event synchronization. Each behavior pattern is composed of behaviors that are re-usable and easy to assemble together. For example, the *Duet* ambient behavior meta-pattern is used to generate the behaviors of all interacting pairs of NPCs that take turns, such as the *Duet-converser-converser*, *Duet-spellcaster-spellcaster*, and *Duet-spawner-destroyer* patterns.

The five simple patterns and one meta-pattern we identified were sufficient to generate all of the NPC ambient behavior scripts in the Prelude for 45 ambient NPCs. In the original Prelude, of these 45 ambient NPCs, only 39 had scripts attached to them. We replaced 265 lines of manual written scripting code in 25 files called 73 times for all the 39 ambient NPCs of the original scripted Prelude. For the other 6 NPCs, including the door guard mentioned previously, that were not attributed any behaviors in the original Prelude, we attached a *Poser* behavior to each of them. Figure 5 shows the number of instances of each kind of ambient behavior pattern that were used in each of the five areas in the Prelude chapter. Note that the 32 ambient behavior pattern instances in Figure 5 are applied to more than 45 NPCs. For example, each instance of the Duet meta-pattern involves two NPCs. Moreover, only one *Poser* behavior instance generates the ambient behaviors of 9 Goblin NPCs that share a common tag.

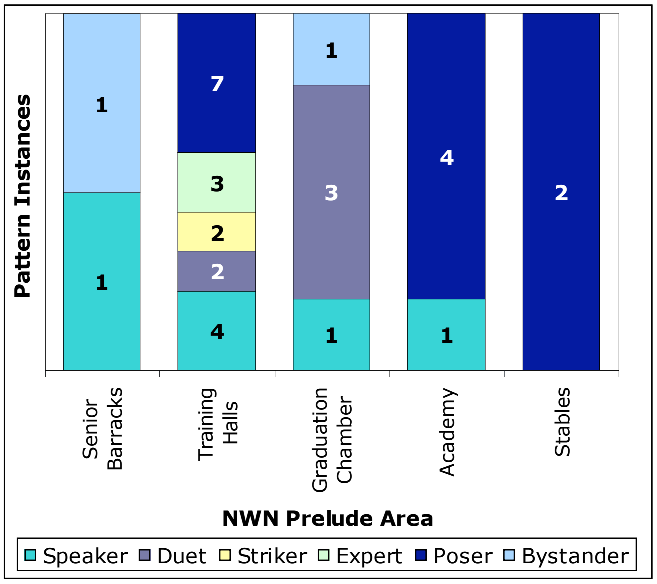

Figure 5. Using ScriptEase ambient behavior patterns to generate behavior scripts in the NWN Prelude.

ScriptEase represents each behavior pattern as a set of reactions to events. Each reaction is represented by a tree of definitions, conditions, and actions. ScriptEase generates NWScript code by traversing these tree representations and generating appropriate NWScript code at each tree node.

To determine whether our patterns were easy to use and to measure the effort required, we asked a Grade 11 high school student (who did not know how to program) to generate the Prelude ambient behavior code. She spent less than a day generating the ambient behaviors of all the NPCs in the Prelude module from these patterns after spending 6 hours learning to play the game, use the Aurora Toolset and using ScriptEase. Further tests on a large group of high school student authors are scheduled.

Further evidence of the generality of ambient behavior patterns will require a case study that replaces behaviors in other game genres as well. There is no reason why a soccer or hockey goaltender could not be provided with entertaining ambient behaviors to exhibit when the ball (puck) is in the other end of play, such as standing on one leg, stretching, leaning against a goal post, or trying to quiet the crowd with a gesture. For example, one of the criticisms for EA FIFA 04 was directed to the goalie's behavior and will be addressed in the announced EA FIFA 06.

4 Creating Ambient Behavior Patterns

A pattern designer can compose reusable behaviors to create a new behavior pattern or add behaviors to existing patterns, without writing any scripts. It is easy to mix/combine behaviors. Each behavior pattern includes a proactive model and a reactive model. The *proactive* model selects a *proactive behavior* based on probabilities. This simplest proactive model uses static probabilities assigned by the writer. For example, the server pattern consists of the proactive behaviors *approach* a random customer, *approach* the bar, *offer-fetch* a drink to the nearest customer and *pose*. In this case, a static probability distribution function [.10, .05, .03, .82] could be used to select one of these behaviors for each proactive event. The left pane of Figure 6 shows the proactive model for the server. The *reactive* model specifies a *reactive chain* for each proactive behavior. For example, the right pane of Figure 6 shows the reactive chain for the server's *offer-fetch* proactive behavior listed in Table 1. The completion of each reactive behavior triggers the next reactive behavior until a *done* behavior signals the end of the reactive chain. The circle identifies the actor that performs the behavior (S, server; C, customer). Other options, such as what is spoken, have been removed from the diagram for clarity. Each of the other three proactive behaviors for the server (*approach bar*, *approach customer,* and *pose*) has a reactive chain that consists of a single behavior followed by a *done* behavior, as listed in Table 1.

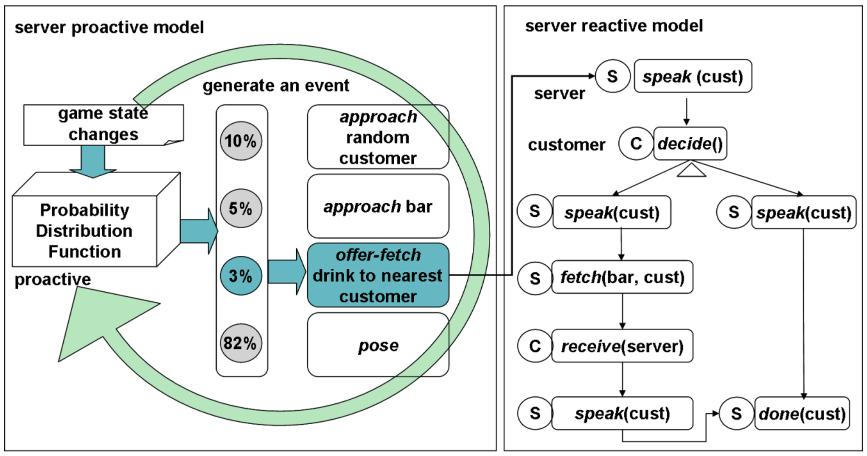

Figure 6. The proactive model and a reactive chain for the Server pattern's offer-fetch behavior.

A behavior can use selection to choose between multiple possible following behaviors. For example, the *decide* behavior can trigger one of two *speak* behaviors based on the customer's drink wishes. A loop can be formed when a behavior later in the chain triggers a behavior earlier in the chain. Loop termination can result from using selection to exit the loop. In general, the reactive model could be a cyclic graph, providing complete expressive power. For ambient behaviors, loops do not appear to be necessary – reactive chains (with decision points) are sufficient for all behaviors we have required so far. For non-ambient behaviors, these loops may be necessary. Each proactive behavior that has reactive components serves as an entry point into the reactive model. The simplicity of the reactive model hides a necessarily complex concurrency model underneath (described in Section 5).

The probabilities shown in Figure 6 are used to select from proactive behaviors. These probabilities can be static or can be dynamic, based on either the context (state of the world) or the current motivations (state of the NPC). We used static probabilities for the ambient behavior patterns used in the tavern scene and the Prelude. So far, our experience indicates that static probabilities are sufficient for ambient behaviors – most NPC "extras" do not need motivational models to control their ambient behaviors. However, our proactive model supports dynamic probabilities and we have started using motivational models for *PC-interactive* behaviors, where the NPC interacts with the PC. One example we have developed is a guard who is motivated by duty, tiredness and a sense of threat to the item being guarded. In this case, our proactive model dynamically generates a probability vector based on motivation levels, before "spinning" for a proactive behavior.

The proactive (*approach creature*, *approach object*, *offer-fetch*, *pose*, etc.) and reactive behaviors (*speak*, *decide*, *receive* etc.) we created for the tavern scene provide sufficient reusable components to create other ambient behavior patterns. A non-programmer can construct a new behavior pattern out of existing behaviors in about an hour. For example, a high-school student was able to create several of her own new patterns from existing behaviors to use in a new story she wrote. Each new pattern took about an hour to create.

It is also easy to create new reusable behaviors as well. A new behavior does not require programming and also takes about an hour to complete – validated by our high-school student. A proactive behavior is a series of reactive behaviors. A reactive behavior is a series of simple ScriptEase actions, such as move to a location/object or face a direction. For example, the number of ScriptEase actions required for the reactive behaviors used to refactor the NWN Prelude varies from 5 (for *pose*, *return*, and *speak*) to 14 (for *strike*). Once made, behaviors can be reused in many behavior patterns and behavior patterns can be reused in many stories. ScriptEase already contained a pattern builder that allowed a pattern designer to create new encounter patterns. We added support to it for building behaviors and ambient behavior patterns.

5 The Concurrency Control Model

Concurrency models have been studied extensively for general-purpose computing. A description of the difficulties in building a concurrency model for interacting NPCs is beyond the scope of this article. However, we raise a few points to indicate the difficulty of this problem. First, synchronization between actors is essential so that an actor completes all of the actions for a behavior before the next behavior begins. For example, the server should not fetch a drink before the customer has decided whether to order a drink or not. Second, deadlock must be avoided so a pair of actors does not wait forever for each other to perform a behavior in a reactive chain. Third, indefinite postponement must be avoided or some behaviors will not be performed.

Our concurrency control mechanism is invisible to the story writer and is only partially visible to the pattern designer. It has *proactive* and *reactive* components that use proactive and reactive behaviors respectively. The *proactive* model has a proactive controller. When the PC enters an area, the controller triggers a *register proactive* script on each NPC within a range of the PC. There is no need to control ambient behaviors in areas not visible to the user, since doing so slows down game response. In games such as Fable, NPCs uphold their daily routine whether the user can see them or not. Computational shortcuts are needed to minimize the overhead. On each NPC, the registering proactive script triggers a spin behavior that, in turn, triggers a single proactive behavior (for instance, *offer-fetch*) as a result of a probabilistic choice among all the proactive behaviors that the actor could initiate. The selected proactive behavior (*offer-fetch*) triggers the first *reactive* behavior (*speak*) in the reactive chain. For each reactive behavior (except *decide* and *done*), ScriptEase generates code so that when the reactive behavior is completed, the next reactive behavior in the chain is triggered. The pattern designer creates a reactive chain by listing the appropriate reactive behaviors in the correct order. For example, to construct the *ask-fetch* chain from Table 1, the designer selects the reactive behaviors:

"speak", "fetch", "receive", "speak", "done". To use the *decide* behavior, the pattern designer lists two tail chains, one that gets selected if a spinner in the decide behavior spins "yes" and one if it "spins" no.

This reactive control model ensures synchronization in a single chain by preventing an actor from starting a behavior before the previous behavior is done. However, it does not prevent synchronization problems due to multiple chains. For example, suppose the server begins the reactive chain for the *offer-fetch* proactive behavior shown in Figure 6 by *speaking* a drink offer, and suppose the owner starts a proactive *ask-fetch* behavior to send the server to the supply room. Reactive behaviors from the server's own reactive *offer-fetch* chain and the owner's reactive *ask-fetch* chain may be triggered in an interleaved manner that violates synchronization.

To ensure synchronization, we introduced an *eye-contact protocol* that ensures both actors agree to participate in a collaborative reactive chain before the chain is started. Actor$_1$ suspends all proactive behaviors and tries to make eye-contact with actor$_2$. If actor$_2$ is involved in a reactive chain, actor$_2$ denies eye-contact and restarts actor$_1$'s proactive behaviors. If actor$_2$ is not involved in a reactive chain, actor$_2$ triggers a reactive behavior on actor$_1$ to start the appropriate reactive chain. This protocol cannot be implemented with behaviors alone, so we use state variables of the actors.

We use another mechanism to eliminate deadlock and indefinite postponement. Either of these situations can arise in the following way. First, an eye-contact is established with an actor, so that the proactive controller does not trigger another proactive behavior. Second, at the conclusion of the reactive chain started by the eye-contact, the actor is not re-registered to trigger a new proactive behavior. Not only will this actor wait forever, but the other actor in the collaborative reactive chain can wait forever as well. One way for this situation to occur is for a script to clear all of the actions in an actor's action queue, including an expected action to trigger a behavior in the reactive chain. In this case, the reactive chain is broken and the proactive controller will never trigger another proactive behavior for the NPC. For example an NPC's action queue is cleared if the user clicks on an NPC to start a conversation between the PC and the NPC. Our solution uses a heartbeat event to increment a counter for every NPC and to check whether the counter has reached a specific value. The game engine fires a heartbeat event every 6 seconds. If the counter reaches a threshold value, that NPC's ambient proactive controller is restarted. The counter is reset to zero every time a reactive behavior is performed by the NPC, so as long as the NPC is performing behaviors (not deadlocked) no restart will occur. Neither the story writer nor the pattern designer need be aware of these transparent concurrency control mechanisms.

We have recently added a perceptive model that allows NPCs to be aware of the PC's presence and act accordingly. When the perceptive model triggers a perceptive behavior, the default is to clear all actions on the NPCs action queue, suspend the proactive model and trigger a reactive chain in response to the perceptive behavior. When the proactive behavior is complete, the proactive model is restarted. Consider a *Bystander* ambient behavior that performs two proactive behaviors, *pose* (run an animation) and *return* (to its original scene location). A perceptive behavior, called *challenge* can be added so that when the PC is perceived and within a certain distance of the NPC, the NPC walks to the PC and starts a conversation. When the conversation is done, the NPC returns to its ambient behaviors (*pose* and *return*). We used this *Challenger* pattern to completely replace all of the manual scripting code for two non-ambient NPC behaviors in the Prelude module. This smooth transition to perceptive behaviors illustrates the robustness and flexibility of our proactive and reactive models, and of the underlying concurrency control mechanism.

6 Conclusion

We described a model for representing NPC ambient behaviors using generative patterns that solves the difficult problem of interacting NPCs. We implemented this model in the NWN game using ScriptEase generative patterns. We are building a common library of rich ambient behavior patterns for use and reuse across CRPGs. Our next goals are to create PC-interactive behavior patterns and to develop patterns that support NPCs that are more central to the plot of the game, as well as NPCs that act as companions for the PC. Each of these goals involves escalating challenges, but we have constructed our ambient behavior model with these challenges in mind. For example, the model supports the non-deterministic selection of behavior actions based on game state. For ambient behaviors this approach can be used with a static probability function to eliminate repetitive behaviors that are boring to the player. For PC-interactive behaviors these probabilities can be dynamic and motivation-based for more challenging opponents and allies. We have constructed a synchronization model that is scalable to the more complex interactions that can take place between major NPCs and between these NPCs and the PC. We demonstrated our approach using a real commercial application, BioWare Corp.'s Neverwinter Nights game. However, our model could have a broader application domain that includes other kinds of computer games, synthetic performance, autonomous agents in virtual worlds, and animation of interactive objects.

Acknowledgements: This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Institute for Robotics and Intelligent Systems (IRIS), and Alberta’s Informatics Circle of Research Excellence (iCORE). We would like to thank former ScriptEase team members, James Redford and Dominique Parker who are now “working on the frontline” at BioWare and Electronic Arts. We also thank the University of Alberta Women in Scholarship, Engineering, Science and Technology (WISEST) program for their support and our WISEST summer student, Danielle Wiebe, for her help with the project. We are grateful to the anonymous reviewers who provided very valuable feedback on the INTETAIN 2005 conference paper that served as a preliminary version of this article.

References

1. D. Isla, “Handling Complexity in the Halo 2 AI”, *Game Developers Conf.* (GDC 05), 2005.
2. M. Mateas and A. Stern, “Façade: An Experiment in Building a Fully-Realized Interactive Drama”, *Game Developers Conf.* (GDC 03), Game Design Track, 2003.
3. K. Perlin and A. Goldberg, “Improv: A System for Scripting Interactive Actors in Virtual Worlds”, *Proc. SIGGRAPH 96,* vol. 29, no. 3, 1996, pp. 205-216.
4. R.M. Young et al., “An architecture for integrating plan-based behavior generation with interactive game environments”, *Journal of Game Development*, vol. 1, no. 1, 2004, pp. 51-70.
5. A. Caicedo and D. Thalmann, “Virtual Humanoids: Let Them Be Autonomous without Losing Control”, *Proc. 4th Conference on Computer Graphics and Artificial Intelligence*, 2000.
6. F. Charles and M. Cavazza, “Exploring the Scalability of Character-based Storytelling”, *Proc. ACM Joint Conf. on Autonomous Agents and Multi-Agent Systems*, 2004, pp. 872-879.
7. F. Poiker, “Creating Scripting Languages for Non-programmers”, *AI Game Programming Wisdom*, Charles River Media, 2002, pp. 520-529.
8. M. McNaughton et al., “ScriptEase: Generative Design Patterns for Computer Role-Playing Games”, *Proc. 19th IEEE Conf. on Automated Software Engineering* (ASE 04), pp. 88-99.
9. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, MA, Addison-Wesley, 1994.
10. M. Carbonaro et al., “Interactive Story Writing in the Classroom: Using Computer Games”, *Proc. Int’l Digital Games Research Conf.* (DiGRA 05). Vancouver, Canada, 2005, pp. 323-338.
11. M. Cutumisu et al., “Generating Ambient Behaviors in Computer Role-Playing Games”, *LNAI 3814, Springer-Verlag* (Intetain 05), 2005, pp. 34-43.