A Behavior Language for Story-based Believable Agents

Michael Mateas
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
michaelm@cs.cmu.edu
www.cs.cmu.edu/~michaelm

Andrew Stern
InteractiveStory.net
andrew@interactivestory.net
www.interactivestory.net

Abstract

ABL is a reactive planning language, based on the Oz Project language Hap, designed specifically for authoring believable agents - characters which express rich personality, and which, in our case, play roles in an interactive, dramatic story world. Here we give a brief overview of the language Hap and discuss the new features in ABL, focusing on ABL's support for multi-character coordination. We also describe the ABL idioms we are using to organize character behaviors in the context of an interactive drama.

Introduction

Façade is a serious attempt to move beyond traditional branching or hyper-linked narrative, to create a fully-realized interactive drama - a dramatically interesting virtual world inhabited by computer-controlled characters, within which the user (hereafter referred to as the player) experiences a story from a first person perspective (Mateas and Stern 2002, Mateas and Stern 2000). The complete, real-time 3D, one-act interactive drama will be available in a free public release at the end of 2002. In the story, Grace and Trip, a married couple in their early thirties, has invited the player over for drinks. Unbeknownst to the player, their marriage is in serious trouble, and in fact, tonight is the night that all their troubles are going to come to the surface. Whether and how their marriage falls apart, and the state of the player's relationship with Grace and Trip at the end of the story, depends on how the player interacts in the world. The player interacts by navigating in the world, manipulating objects, and, most significantly, through natural language dialog. This project raises a number of interesting AI research issues, including drama management for coordinating plot-level interactivity, broad but shallow support for natural language understanding and discourse management, and autonomous believable agents in the context of interactive story worlds. This paper focuses on the last issue, describing the custom believable agent language developed for this project, and the idioms developed within this language for organizing character behaviors.

ABL overview

ABL (**A B**ehavior **L**anguage, pronounced "able") is based on the Oz Project (Bates 1992) believable agent language Hap developed by A. B. Loyall (Loyall 1997, Bates, Loyall and Reilly 1992, Loyall and Bates 1991). The ABL compiler is written in Java and targets Java; the generated Java code is supported by the ABL runtime system.

ABL modifies Hap in a number of ways, changing the syntax (making it more Java-like), generalizing the mechanisms by which an ABL agent connects to a sensory-motor system, and, most significantly, adding new constructs to the language, including language support for multi-agent coordination in the carrying out of dramatic action. This section provides an overview of the ABL language and discusses some of the ways in which ABL modifies or extends Hap. The discussion of joint behaviors, the mechanism for multi-agent coordination, occurs in its own section below.

Hap Semantics

Since ABL builds on top of Hap, here we briefly describe the organization and semantics of a Hap program by walking through a series of examples. All examples use the ABL syntax.

Hap/ABL programs are organized as collections of behaviors. In sequential behaviors, the steps of the behavior are accomplished serially. As each step is executed, it either succeeds or fails; step success makes the next step available for execution. If any step fails, it causes the enclosing behavior to fail. An example sequential behavior is shown below.

```
sequential behavior AnswerTheDoor() {
  WME w;
  with success_test { w = (KnockWME) } wait;
  act sigh();
  subgoal OpenDoor();
  subgoal GreetGuest();
  mental_act { deleteWME(w); }
}
```

In this sequential behavior, an agent waits for someone to knock on a door, sighs, then opens the door and greets the guest. This behavior demonstrates the four basic step types, namely **`wait`**, **`act`**, **`subgoal`**, and

Appeared in: Ken Forbus and Magy El-Nasr Seif (Eds.), *Working notes of Artificial Intelligence and Interactive Entertainment*. AAAI Spring Symposium Series. Menlo Park, CA: AAAI Press. 2002.

mental_act. Wait steps are never chosen for execution; a naked wait step in a sequential behavior would block the behavior from executing past the wait. However, when combined with a success test, a wait step can be used to make a demon which waits for a condition to become true. Success tests are continuously monitored conditions which, when they become true, cause their associated step to immediately succeed. Though in this example the success test is associated with a wait step to make a demon, it can be associated with any step type.

Success tests, as well as other tests which will be described shortly, perform their test against the agent's working memory. A working memory contains a number of working memory elements (WMEs) which hold information. WMEs are like classes in an object oriented language; every WME has a type plus some number of typed fields which can take on values. As described later on in the paper, WMEs are also the mechanism by which an agent becomes aware of sensed information. In this example, the success test is looking for WMEs of type KnockWME, which presumably is placed in the agent's working memory when someone knocks on a door. Since there are no field constraints in the test, the test succeeds as soon as a KnockWME appears.

An act step tells the agent's body (sensory-motor system) to perform an action. For graphical environments such as Façade, physical acts will ultimately be translated into calls to the animation engine, though the details of this translation are hidden from the Hap/ABL program. In this example, the act makes the body sigh. Note that physical acts can fail - if the sensory-motor system determines that it is unable to carry out the action, the corresponding act step fails, causing the enclosing behavior to fail.

Subgoal steps establish goals that must be accomplished in order to accomplish the behavior. The pursuit of a subgoal within a behavior recursively results in the selection of a behavior to accomplish the subgoal.

Mental acts are used to perform bits of pure computation, such as mathematical computations or modifications to working memory. In the final step of the example, the mental_act deletes the KnockWME (making a call to a method defined on ABL agent), since the knocking has now been dealt with. In ABL, mental acts are written in Java.

The next example demonstrates how Hap/ABL selects a behavior to accomplish a subgoal through signature matching and precondition satisfaction.

```
sequential behavior OpenDoor() {
  precondition {
    (KnockWME doorID :: door)
    (PosWME spriteID == door pos :: doorPos)
    (PosWME spriteID == me  pos :: myPos)
    (Util.computeDistance(doorPos, myPos) > 100)
  }
  specificity 2;
  // Too far to walk, yell for knocker to come in
  subgoal YellAndWaitForGuestToEnter(doorID);
}

sequential behavior OpenDoor() {
  precondition { (KnockWME doorID :: door) }
  specificity 1;
  // Default behavior - walk to door and open
  . . .
}
```

In this example there are two sequential behaviors **OpenDoor()**, either of which could potentially be used to satisfy the goal **OpenDoor()**. The first behavior opens the door by yelling for the guest to come in and waiting for them to open the door. The second behavior (details elided) opens the door by walking to the door and opening it. When **AnswerTheDoor()** pursues the subgoal **OpenDoor()**, Hap/ABL determines, based on signature matching, that there are two behaviors which could possibly open the door. The precondition of both behaviors is executed. In the event that only one of the preconditions is satisfied, that behavior is chosen as the method to use to accomplish the subgoal. In the event that both preconditions are satisfied, the behavior with the highest specificity is chosen. If there are multiple satisfied behaviors with highest specificity, one is chosen at random. In this example, the first **OpenDoor()** behavior is chosen if the lazy agent is too far from the door to walk there ("too far" is arbitrarily represented as a distance > "100").

The precondition demonstrates the testing of the fields of a WME. The **::** operator assigns the value of the named WME field on the left of the operator to the variable on the right.[1] This can be used both to grab values from working memory which are then used in the body of the behavior, and to chain constraints through the WME test.

The last example demonstrates parallel behaviors and context conditions.

```
parallel behavior
YellAndWaitForGuestToEnter(int doorID) {
  precondition { (CurrentTimeWME t :: startT) }
  context_condition {
    (CurrentTimeWME t <= startT + 10000) }
  number_needed_for_success 1;

  with success_test {
    (DoorOpenWME door == doorID) } wait;
  with (persistent) subgoal YellForGuest(doorID);
}
```

In a parallel behavior, the steps are pursued simultaneously. **YellAndWaitForGuestToEnter(int)** simultaneously yells "come in" towards the door (the door specified by the integer parameter) and waits to actually see the door open. The persistent modifier on the **YellForGuest(int)** subgoal makes the subgoal be repeatedly pursued, regardless of whether the subgoal succeeds or fails (one would imagine that the behavior that does the yelling always succeeds). The **number_needed_for_success** annotation (only usable on parallel behaviors) specifies that only one step

[1] In ABL, a locally-scoped appropriately typed variable is automatically declared if it is assigned to in a WME test and has not been previously explicitly declared.

has to succeed in order for the behavior to succeed. In this case, that one step would be the demon step waiting for the door to actually open. The context condition is a continuously monitored condition which must remain true during the execution of a behavior. If the context condition fails during execution, then the behavior immediately fails. In this example, the context condition tests the current time, measured in milliseconds, against the time at which the behavior started. If after 10 seconds the door hasn't yet opened (the guest isn't coming in), then the context condition will cause the behavior to fail.

As failure propagates upwards through the subgoal chain, it will cause the first `OpenDoor()` behavior to fail, and eventually reach the `OpenDoor()` subgoal in `AnswerTheDoor()`. The subgoal will then note that there is another `OpenDoor()` behavior which has not been tried yet and whose precondition is satisfied; this behavior will be chosen in an attempt to satisfy the subgoal. So if the guest doesn't enter when the agent yells for awhile, the agent will then walk over to the door and open it.

Finally, note that parallel behaviors introduce multiple lines of expansion into a Hap/ABL program. Consequently, the current execution state of the program is represented by a tree, the active behavior tree (ABT), where the leaves of the tree constitute the current set of executable steps.

These examples give a sense for the Hap semantics which ABL reimplements and extends. There are many other features of Hap (also implemented in ABL) which it is not possible to re-describe here, including how multiple lines of expansion mix (based on priority, blocking on physical acts, and a preference for pursing the current line of expansion), declaration of behavior and step conflicts (and the resulting concept of suspended steps and behaviors), and numerous annotations which modify the default semantics of failure and success propagation. The definitive reference on Hap is of course Loyall's dissertation (Loyall 1997).

ABL Extensions

ABL extends Hap in a number of ways, including:

- Generalizing the mechanisms for connecting to the sensory-motor system. The ABL runtime provides abstract superclasses for sensors and actions. To connect an ABL program to a new sensory-motor system (e.g. animation engine, robot), the author merely defines specific sensors and actions as concrete subclasses of the abstract sensor and action classes. ABL also includes additional language constructs for binding sensors to WMEs. ABL then takes responsibility for calling the sensors appropriately when bound WMEs are referenced in working memory tests.
- Atomic behaviors. Atomic behaviors prevent other active behaviors from mixing in. Atomic behaviors are useful for atomically updating state (e.g. updating multiple WMEs atomically), though they should be used sparingly, as a time-consuming atomic behavior could impair reactivity.
- Reflection. ABL gives behaviors reflective access to the current state of the ABT, supporting the authoring of meta-behaviors which match on patterns in the ABT and dynamically modify other running behaviors. Supported ABT modifications include succeeding, failing or suspending a goal or behavior, and modifying the annotations of a subgoal step, such as changing the persistence or priority. Safe reflection is provided by wrapping all ABT nodes in special WMEs. Pattern matching on ABT state is then accomplished through normal WME tests. A behavior can only touch the ABT through the reflection API provided on these wrapper WMEs.
- Multiple named memories. Working memories can be given a public name, which then, through the name, are available to all ABL agents. Any WME test can simultaneously reference multiple memories (the default memory is the agent's private memory). Named memories are used by the joint behavior mechanisms (see below) for the construction of team memories. In Façade, named memories are also useful for giving agents access to a global story memory.
- Goal spawning. In addition to subgoaling, which roots the selected behavior at the subgoal step of the parent behavior, a behavior can spawn a goal, which roots the subgoal elsewhere in the tree (the default is the root collection behavior). Unlike a normal subgoal, the success or failure of a spawned goal does not effect the success or failure of the behavior which spawned the goal (though it will effect the success or failure of the behavior where it is rooted). The spawned goal continues to be held after the behavior which spawned the goal goes away (succeeds or fails). Goal spawning is useful for starting a behavior which should continue past the end (and during the suspension) of the spawning parent.

Beat Idioms

Developing a believable agent language such as ABL involves simultaneously defining and implementing language constructs which support the authoring of expressive behavior, and the exploration of idioms for expressive behavior *using* the language. In Façade, character behavior is organized around the dramatic beat, in the theory of dramatic writing the smallest unit of dramatic action (see for example McKee 1997). This section describes the ABL idioms used in authoring beat behaviors.

Beat behaviors are divided into three categories: *beat goals*, *handlers*, and *cross-beat behaviors*. A greeting beat, in which Trip greets the player at the door, will

provide examples of these three behaviors categories and the relationships between the categories.[1]

In the greeting beat, Trip wants to initially greet the player ("Hey! So glad you could make it. Thanks for coming over man."), yell for Grace ("Grace, come on out! Our guest is here."), and invite the player in ("Come on in, don't be shy"). These are the three *beat goals* of the greeting beat and should be accomplished sequentially.

Of course, during this greeting, the player will engage in various actions which should be handled in the context of the greeting. These interactions take the form of physical movement, object manipulation, and natural language text typed by the player. At the beat behavior level, player text is captured by WMEs representing the meaning of the text as a discourse act.[2] *Handlers* are demons responsible for handling player interaction. For the purposes of this example, assume that the greeting beat wants to handle the cases of the player greeting Trip, the player referring to Grace, and the player preemptively walking into the apartment before she has been invited in.[3] The code below starts the handlers and begins the sequence of beat goals.

```
parallel behavior StartTheBeat() {
  with (priority 1)
    subgoal StartTheHandlers();
  subgoal BeatGoals();
}

parallel behavior StartTheHandlers() {
  with (persistent, priority 20)
    subgoal handlerDAGreet();
  with (persistent, priority 15)
    subgoal handlerDAReferTo_grace();
  with (priority 10, ignore_failure)
    subgoal handlerPreInviteAptMove();
}

sequential behavior BeatGoals() {
  with (persistent when_fails)
    bgOpenDoorAndGreetPlayer();
  with (persistent when_fails) bgYellForGrace();
  with (persistent when_fails) bgInviteIntoApt();
}
```

Handlers are started in various priority tiers corresponding to the relative importance of handling that interaction. Priorities are used to resolve cases where another player interaction happens in the middle of handling the previous player interaction, or when simultaneous player interactions occur. A higher priority handler can interrupt a lower priority handler, while same or lower priority handlers must wait for a higher priority handler to finish before handling the nested interaction. Generally handlers are persistent; when a handler finishes responding to an interaction, it should "reset" and be ready to deal with another interaction in the same category. In general handlers are higher priority than beat goals so that if an interaction occurs in the middle of the beat goal, the handler will "wake up" and interrupt it.

In general, handlers are meta-behaviors, that is, they make use of reflection to directly modify the ABT state. When a handler triggers, it fails the current beat goal, potentially succeeds other beat goals, possibly pursues a beat goal within the handler (effectively reordering beat goals), and engages in its own bit of handler specific behavior. In some cases the handler specific behavior may entail mapping the recognized action to a different recognized action, which will then trigger a different corresponding handler. Below is a simplified version of **`handlerDAReferTo_grace()`**.

```
sequential behavior handlerDAReferTo_grace() {
  with (success_test { (DAReferToWME topicID ==
                        eTopic_grace) } ) wait;
  with (ignore_failure) subgoal
    handlerDAReferTo_grace_Body();
  subgoal DAReferTo_grace_Cleanup();
}

// by mentioning Grace, we will say "Grace? uh
// yeah" and then yell for Grace but only if we
// aren't currently doing bgYellForGrace!
sequential behavior handlerDAReferTo_grace_Body()
{
  precondition {
    (GoalStepWME signature == "bgYellForGrace()"
                 isExecuting == false) }
  subgoal handlerDA_InterruptWith(
    eTripScript_graceuhyeah,
    eFullExpression_blank);
  subgoal handlerDAReferTo_grace_Body2();
}

// we aren't currently doing yellForGrace, and if
// we haven't completed yellForGrace, then do it
sequential behavior
  handlerDAReferTo_grace_Body2() {
  // Goal still exists in the ABT so it hasn't
  // been completed
  precondition {
   (GoalStepWME signature == "bgYellForGrace()")
  }
  specificity 2;
  subgoal SetBeatGoalSatisfied(
    "bgYellForGrace()", true);
  with (persistent when_fails)
    subgoal bgYellForGrace();
}

// otherwise we must have already completed
// yellForGrace, so say "She's coming, I don't
// know where she's hiding"
sequential behavior
  handlerDAReferTo_grace_Body2() {
  specificity 1;
  subgoal handlerDA_InterruptWith(
    etripScript_shescomingidontknow,
    eFullExpression_smallSmile);
}
```

When the player refers to Grace (perhaps saying, "I'm looking forward to meeting Grace", or "Where is Grace", or "Hi Grace") this handler is triggered. The handler body

[1] To simplify the discussion, the example will focus on a beat in which only a single character interacts with the player. See the next section for a discussion of ABL's support for multi-agent coordination.

[2] For translating surface text into formally represented discourse acts, Façade employs a custom rule language for specifying templates and discourse chaining rules. The discourse rule compiler targets Jess, a CLIPS-like forward-chaining rule language (available at http://herzberg.ca.sandia.gov/jess/).

[3] The real greeting beat employs ~50 handlers.

behavior uses reflection to test if the beat goal to yell for Grace is currently not executing. If it is executing (e.g. Trip was in the middle of yelling for Grace when the player said "Where's Grace"), the body precondition fails, causing the handler to fail, which then restarts because of the persistence annotation, leaving Trip ready to handle another reference to Grace. Effectively Trip ignores references to Grace if he's in the middle of yelling for Grace. Otherwise, Trip interrupts whatever he is saying with "Oh, yeah…". **`handlerDAInterruptWith`** uses reflection to fail the currently executing beat goal, thus, as the name implies, interrupting the beat goal. When the handler is finished, the **`persistent_when_fails`** annotation will causes any handler-failed beat goals to restart. After saying "Oh yeah…" Trip either performs the yell for Grace beat goal within the handler (and succeeds it out of the **`BeatGoals`** behavior) or, if yell for Grace has already happened, says "She's coming. I don't know where she's hiding." This handler demonstrates how player interaction can cause beat goals to be interrupted, effectively reordered, and responded to in a way dependent on what has happened in the beat so far.

The final category of beat behaviors are the *cross-beat behaviors*. These are behaviors that cross beat goal and handler boundaries. An example beat goal behavior is the staging behavior which an agent uses to move to certain dramatically significant positions (e.g. close or far conversation position with the player or another agent, into position to pickup or manipulate another object, etc.). A staging request to move to close conversation position with the player might be initiated by the first beat goal in a beat. The staging goal is spawned to another part of the ABT. After the first beat goal completes its behavior, other beat goals and handlers can happen as the agent continues to walk towards the requested staging point. Of course at any time during a cross-beat behavior, beat goals and handlers can use reflection to find out what cross-beat behaviors are currently happening and succeed or fail them if the cross-beat behaviors are inappropriate for the current beat goal's or handler's situation.

Support for Joint Action

In (Mateas and Stern 2000) we argued that much work in believable agents is organized around the principle of strong autonomy, and that, for story-based believable agents, this assumption of strong autonomy is problematic. An agent organized around the notion of strong autonomy chooses its next action based on local perception of its environment plus internal state corresponding to the goals and possibly the emotional state of the agent. All decision making is organized around the accomplishment of the individual, private goals of the agent. But believable agents in a story must also participate in story events, which requires making decisions based on global story state (the entire past history of interaction considered *as* a story) and tightly coordinating the activity of multiple agents so as to accomplish joint story goals. In order to resolve the tension between local and global control of characters, we proposed organizing behaviors around the dramatic beat. In order to facilitate the coordination of multiple characters, we proposed extending the semantics of Hap, in a manner analogous to the STEAM multi-agent coordination framework (Tambe 1997). This section describes *joint behaviors*, ABL's support for multi-agent coordination.

The driving design goal of joint behaviors is to combine the rich semantics for individual expressive behavior offered by Hap with support for the automatic synchronization of behavior across multiple agents.

Joint Behaviors

In ABL, the basic unit of coordination is the joint behavior. When a behavior is marked as joint, ABL enforces synchronized entry and exit into the behavior. Part of the specification for an "offer the player a drink" behavior from Façade is shown below. To simplify the discussion, the example leaves out the specification of how player activity would modify the performance of this beat. This will be used as the guiding behavior specification in the joint behavior examples provided in this paper.

> (At the beginning of the behavior, Trip starts walking to the bar. If he gets to the bar before the end of the behavior, he stands behind it while delivering lines.)
> **Trip**: A beer? Glass of wine? (Grace smiles at player. Short pause)
> **Trip**: You know I make a mean martini. (Grace glances at Trip with slight frown partway into line. At the end of line, rolls her eyes at the ceiling.)
> **Grace**: (shaking her head, smiling) Tch, Trip just bought these fancy new cocktail shakers. He's always looking for a chance to show them off. (If Trip is still walking to the bar, he stops at "shakers". At "shakers" Trip looks at Grace and frowns slightly. At the end of the line he looks back at player and smiles. If he was still on the way to the bar, he resumes walking to the bar).

In order to perform this coordinated activity, the first thing that Grace and Trip must do is synchronize on offering a drink, so that they both know they are working together to offer the drink. Grace and Trip both have the following behavior definition in their respective behavior libraries.

```
joint sequential behavior OfferDrink() {
  team Grace, Trip;
  // The steps of Grace's and Trip's OfferDrink()
  // behaviors differ.
}
```

The declaration of a behavior as joint tells ABL that entry into and exit from the behavior must be coordinated with team members, in this case Grace and Trip. *Entry* into a behavior occurs when the behavior is chosen to satisfy a subgoal. *Exit* from the behavior occurs when the behavior

succeeds, fails, or is suspended. Synchronization is achieved by means of a three-phase commit protocol:

1. The initiating agent broadcasts an intention (to enter, succeed, fail or suspend) to the team.
2. All agents receiving an intention respond by, in the case of an entry intention, signaling their own intention to enter or a rejection of entry, or in the case of exit signaling their intention own intention to succeed, fail, or suspend.
3. When an agent receives intentions from all team members, it sends a ready message. When ready messages from all agents have been received, the agent performs the appropriate entry into or exit from the behavior.[1]

Imagine that Trip pursues an **`OfferDrink()`** subgoal and picks the joint **`OfferDrink()`** behavior to accomplish the subgoal. After the behavior has been chosen, but before it is added to the ABT, Trip negotiates entry with his teammate Grace. On receipt of the intention-to-enter **`OfferDrink()`**, Grace checks if she has a joint behavior **`OfferDrink()`** with a satisfied precondition. If she does, she signals her intention-to-enter. Trip and Grace then exchange ready-messages and enter the behavior. In Trip's case the behavior is rooted normally in the ABT at the subgoal which initiated behavior selection, while in Grace the spawned subgoal and corresponding joint behavior are rooted at the collection behavior at the root of the ABT.[2] If Grace didn't have a satisfied joint **`OfferDrink()`** behavior, she would send a reject message to Trip, which would cause Trip's **`OfferDrink()`** subgoal to fail, with all the normal effects of failure propagation (perhaps causing Trip to instead choose a lower specificity *individual* **`OfferDrink()`** behavior). Note that during the negotiation protocol, the agents continue to pursue other lines of expansion in their ABT's; if the protocol takes awhile to negotiate, behavior continues along these other lines.

The negotiation protocol may seem overly complex. In the case that all the team members are on the same machine (the case for Façade), one can assume that negotiation will be very fast and no messages will be lost. In this case agents only need to exchange a pair of messages for behavior entry, while the initiator only needs to send a single message for behavior exit. However, this simplified protocol would break in the distributed case where team member's messages may be lost, or in cases where an agent might disappear unexpectedly (e.g. a game where agents can be killed) in the middle of the negotiation. More interestingly, the more complex negotiation protocol provides authorial "hooks" for attaching transition behaviors to joint behavior entry and exit. Sengers, in her analysis of the Luxo Jr. short by Pixar, identified behavior transitions as a major means by which narrative flow is communicated (Sengers 1998a). Animators actively communicate changes in the behavior state of their characters (e.g. the change from playing to resting) by having the characters engage in short transitional behaviors that communicate why the behavior change is happening. Sengers' architectural extensions to Hap provided support for authoring individual transition behaviors (Sengers 1998a, Sengers 1998b). However, she also noted that animators make use of coordinated multi-character transitions to communicate changes in multi-character behavioral state, but did not provide architectural support for this in her system. By exposing the negotiation protocol to the agent programmer, ABL can support the authoring of behaviors which *communicate transitions in multi-agent behavior state*.

[1] Appropriate timeouts handle the case of non-responding agents who fail to send appropriate intention or ready messages.

[2] A collection behavior is a variety of parallel behavior in which every step need only be *attempted* for the behavior to succeed.

Posting Actions and Step Synchronization

In addition to synchronizing on behavior entry and exit, ABL provides other mechanisms for synchronizing agents, namely support for posting information to a team working memory, and the ability to synchronize the steps of sequential behaviors. Below are the two **`OfferDrink()`** behaviors for Trip and Grace.

Trip's behavior:

```
joint sequential behavior OfferDrink() {
  team Trip, Grace;

  with (post-to OfferDrinkMemory)
    // Individual behavior for initial offer
    subgoal iInitialDrinkOffer();
  subgoal iLookAtPlayerAndWait(0.5);
  with (synchronize) subgoal jSuggestMartini();

  // react to Grace's line about fancy shakers
  with (synchronize) subgoal
    jFancyCocktailShakers();
}
```

Grace's behavior:

```
joint sequential behavior OfferDrink() {
  team Trip, Grace;

  // wait for Trip to say first line
  with (success_test { OfferDrinkMemory
    (CompletedGoalWME name == iInitialDrinkOffer
                      status == SUCCEEDED)})
    wait;
  subgoal iLookAtPlayerAndWait(0.5);

  // react to Martini suggestion
  with (synchronize) subgoal jSuggestMartini();
  with (synchronize) subgoal
    jFancyCocktailShakers();
}
```

For readability, the subgoals have been named with an initial "i" if only an individual behavior is available to satisfy the subgoal, and named with an initial "j" if only a joint behavior is available.

Whenever a joint behavior is entered, the ABL runtime automatically creates a new named team working memory

which persists for the duration of the joint behavior.[1] This team memory, which can be written to and read from by any member of the team, can be used as a communication mechanism for coordinating team activity. The first subgoal of Trip's behavior is annotated with a **`post-to`** annotation; for any subgoal marked with **`post-to`**, a CompletedGoalWME is added to the named memory when the subgoal completes (with either success or failure). A CompletedGoalWME, the definition of which is provided by the ABL runtime, contains the name of the goal, its completion state (success or failure), the name of the agent who performed the goal, any goal arguments, and a timestamp. The **`post-to`** annotation automatically fills in the appropriate arguments. This facility, inspired by the sign management system in Senger's extension of Hap (Sengers 1998a, Sengers 1998b), can be used to provide an agent with a selective episodic memory. This facility is useful even in a single agent situation, as the future behavior of an agent may conditionally depend on past episodic sequences. Since the ABT no longer has state for *already completed* subgoals and actions, an ABL agent's reflective access to its own ABT doesn't provide access to past episodic sequences. However, in a team situation, access to episodic state can be used to coordinate team members. In the first line of Grace's behavior, a demon step monitors the team memory for the completion of **`iInitialDrinkOffer()`**. In the behavior spec above, Grace doesn't begin directly reacting to Trip until after Trip's first line. Keep in mind that an ABL agent pursues multiple lines of expansion, so while Grace is waiting for Trip to complete his first line, she will continue to behave, in this case engaging in small idle movements as she smiles at the player. When Trip completes his first subgoal, an appropriate CompletedGoalWME is posted to the team memory; Trip then moves onto his second subgoal, to look at the player and wait for about half a second. The posting of the CompletedGoalWME causes Grace's first line to succeed, and she also, independently, waits for about half a second. One of them will be first to finish waiting, and will move onto the next line, which, being a joint behavior, reestablishes synchronization.

The last two subgoals of Grace's and Trip's behaviors are annotated with a **`synchronize`** annotation. To understand what this does, first imagine the case where the annotation is *absent*. Assume Grace is the first to finish the second subgoal (the goal to look at the player and wait). Grace will then attempt to satisfy the subgoal **`jSuggestMartini()`**, causing Trip to spawn this goal at the root of his ABT and enter his local version of **`jSuggestMartini()`**. As they jointly pursue the **`jSuggestMartini()`** line of expansion, Trip will continue to pursue the **`OfferDrink()`** line of expansion, eventually initiating **`jSuggestMartini()`** on his side, causing Grace to spawn the goal at her root and enter another copy of the behavior. At this point each is pursuing two copies of the joint behavior **`jSuggestMartini()`**, one copy rooted at the subgoal within **`OfferDrink()`**, and the other rooted at the root of the ABT. This is not what the behavior author intended; rather it was intended that when the characters synchronize on **`jSuggestMartini()`**, they would each begin pursing their local version of **`jSuggestMartini()`** rooted at the respective subgoals within their local versions of **`OfferDrink()`**. The **`synchronize`** annotation allows a behavior author to specify that a joint behavior should be rooted at a specific subgoal, rather than at the ABT root. **`Synchronize`** is only allowed within joint behaviors as an annotation on a goal that has at least one joint behavior with matching signature in the behavior library. In the case of sequential joint behaviors, synchronization on a **`synchronize`** subgoal forces the success of all steps between the current step counter position and the **`synchronize`** subgoal, and moves the step counter up to the **`synchronize`** subgoal.

The example used in this section did not take account of player interaction. Multi-agent beats use the same idioms as described above for coordinating beat goals, responding to player interaction, and pursing longer term goals; the various beat behaviors just become joint behaviors instead of individual behaviors.

Conclusion

ABL provides a rich programming framework for authoring story-based believable agents. Here we've described ABL's novel features and provided examples of how we're using these features to author characters for Façade, an interactive dramatic world.

References

Bates, J. 1992. Virtual Reality, Art, and Entertainment. Presence: *The Journal of Teleoperators and Virtual Environments* 1(1): 133-138.

Bates, J., Loyall, A. B., and Reilly, W. S. 1992. Integrating Reactivity, Goals, and Emotion in a Broad Agent. *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Bloomington, Indiana, July 1992.

Loyall, A. B. 1997. *Believable Agents*. Ph.D. thesis, Tech report CMU-CS-97-123, Carnegie Mellon University.

Loyall, A. B., and Bates, J. 1991. Hap: A Reactive, Adaptive Architecture for Agents. Technical Report CMU-CS-91-147. Department of Computer Science. Carnegie Mellon University.

Mateas, M. and Stern, A. 2000. Towards Integrating Plot and Character for Interactive Drama. In *Working notes of the Social Intelligent Agents: The Human in the Loop*

[1] By default the name of the team memory is the concatenation of the name of the behavior and the string "Memory".

Symposium. AAAI Fall Symposium Series. Menlo Park, CA: AAAI Press.

Mateas, M. and Stern, A. 2002 (forthcoming). Towards Integrating Plot and Character for Interactive Drama. In K. Dautenhahn (Ed.), *Socially Intelligent Agents: The Human in the Loop*. Kluwer.

McKee, R. 1997. *Story: Substance, Structure, Style, and the Principles of Screenwriting*. New York, NY: HarperCollins.

Sengers, P. 1998a. *Anti-Boxology: Agent Design in Cultural Context*. Ph.D. Thesis. School of Computer Science, Carnegie Mellon University.

Sengers, P. 1998b. Do the Thing Right: An Architecture for Action-Expression. In *Proceedings of the Second International Conference on Autonomous Agents*. pp. 24-31.

Tambe, M. 1997. Towards Flexible Teamwork. Journal of *Artificial Intelligence Research* (7) 83-124.