Submitted for Publication in *Journal of Game Development, vanlent@icts.usc.edu and niles@charlesriver.com*

An architecture for integrating plan-based behavior generation with interactive game environments

R. Michael Young
Department of Computer Science, Box 7535, North Carolina State University, Raleigh, NC, 27695, young@csc.ncsu.edu

Mark O. Riedl
Department of Computer Science, Box 7535, North Carolina State University, Raleigh, NC, 27695, young@csc.ncsu.edu

Mark Branly
Department of Computer Science, Box 7535, North Carolina State University, Raleigh, NC, 27695, young@csc.ncsu.edu

Arnav Jhala
Department of Computer Science, Box 7535, North Carolina State University, Raleigh, NC, 27695, young@csc.ncsu.edu

R. J. Martin
Knowwonder Digital Mediaworks, 12421 Willows Road NE #200
Kirkland, WA 98034, rockdeity@mindspring.com

C. J. Saretto
Microsoft Corporation, Box 1 Microsoft Way Redmond, WA 98052, cjsar@microsoft.com

ABSTRACT

One central challenge for game developers is the need to create compelling behavior for a game's characters and objects. Most approaches to behavior generation have either used scripting or finite-state approaches. Both of these approaches are restricted the requirement to anticipate game state at design time. Research in artificial intelligence has developed a number of techniques for automatic plan generation that create novel action sequences that are highly sensitive to run-time context.

In this paper, *An architecture for integrating plan-based behavior generation with interactive game environments*, we describe an architecture called Mimesis, designed to integrate a range of intelligent components with conventional game engines. The architecture is designed to bridge the gap between game engine design and development and much of the work in artificial intelligence that focuses on the automatic creation of novel and effective action sequences. Users of the system construct two parallel models of the game world, one using extensions to existing game engine code, the other using techniques for explicit modeling of actions in terms of their requirements for execution and their effects on the game world.

When integrated with a game engine, Mimesis acts as a run-time behavior generator, responsible for both generating *plans* – coherent action sequences that achieve a specific set of in-game goals – and maintaining the coherence of those plans as they execute in the face of unanticipated user activity. In this paper, we describe the architecture, its main components and the APIs available for integrating it into existing or new game engines.

1. INTRODUCTION

One central challenge for game developers is the need to create compelling behavior for a game's characters and objects. For instance, in simulation games, this challenge involves the accurate mathematical modeling of the rules of the real world being modeled (e.g., the creation of a physics engine). For many genres, however, engaging behavioral control is less prescribed, requiring the developer to build code that, hopefully, creates highly context-sensitive, dynamic behaviors for its system controlled characters. Most approaches to behavior generation have either involved the use of pre-compiled scripts or the development of control software that implements a finite state machine. While scripting approaches may require less complicated design than a that needed to construct a finite state behavioral model, a scripting language's run-

time requirements typically limit the behaviors of the characters that it controls to be less sensitive to context (e.g., all actions in a script's sequence are fixed at design time and cannot readily be adapted should the game context change or the script be run more than once). In contrast, finite-state approaches can provide characters with a wider range of behaviors, but require the programmer to anticipate the full range of expected states and the transitions between them [19]. These states and transitions, enumerated at design time, cannot be readily modified once a game is built.

Research in AI has developed a number of techniques for automatic plan generation that create novel action sequences that are highly sensitive to run-time context. Recent work [4,9] has sought to integrate this work with game environments in order to create intelligent, reactive characters, especially in the context of interactive narrative. In this paper, we describe an architecture called Mimesis, designed to integrate a range of special-purpose intelligent components with conventional game engines. We describe the architecture, several of its main components and the APIs available for integrating it into existing or new game engines.

The Mimesis system is currently being used by the Liquid Narrative Group at North Carolina State University to construct interactive narrative-oriented games. The architecture is specifically designed to bridge the gap between game engine design/development and much of the work in artificial intelligence that focuses on the automatic creation of novel and effective action sequences. Users of the system construct two parallel models of the game world, one using extensions to existing game engine code, the other using techniques for explicit modeling of actions in terms of their requirements for execution and their effects on the game world. When integrated with a game engine, Mimesis acts as a run-time behavior generator, responsible for both generating *plans* – coherent action sequences that achieve a specific set of in-game goals

– and maintaining the coherence of those plans as they execute in the face of unanticipated user activity.

The process of constructing a plan involves a number of specialized functions, including reasoning about the actions of individual characters, generating any character dialog or narration to be provided by the system, creating cinematic camera control directives to convey the action that will unfold in the story, etc. To facilitate the integration of corresponding special-purpose reasoning components, the Mimesis architecture is highly modular. Individual components within the Mimesis run as distinct processes (typically on distinct processors, though this is not a requirement); components communicate with one another via a well-defined socket-based message-passing protocol; developers extending Mimesis to provide new functionality wrap their code within a message-passing shell that requires only a minimal amount of customization.

While game engines are well-suited for building compelling interactive game worlds, the representation that they use to model game worlds does not match well with models typically used by AI researchers. The internal representation of most game engines is *procedural* – it does not utilize any formal model of the characters, setting or the actions of the stories that take place within it. In contrast, most intelligent interfaces provide explicit *declarative* models of action, change and the knowledge used to reason about them. Consequently, direct integration of intelligent software components with a game engine is not straightforward. To facilitate the integration of approaches that use these disparate representations, Mimesis augments a game engine's default mechanisms for controlling its virtual environment, using instead a client/server architecture in which low-level control of the game environment is performed by a customized version of the game engine (called the MWorld) and high-level reasoning about plan structure and user interaction is performed externally by a number of intelligent control elements.

The remainder of this paper is organized as follows. In the next section, we summarize work integrating artificial intelligence research and development with computer game engines. We describe the Mimesis architecture further in Section 3. In Section 4, we discuss the techniques used in Mimesis to integrate the procedural representation of game engines with the declarative representations used in AI systems. We also describe the techniques used by the planning components of Mimesis to create plans for controlling action within a game engine and how we monitor those plans to effectively maintain their coherence during execution. In Section 5, we give a high-level discussion of the methodology for developing games that use the Mimesis architecture. In Section 6, we summarize the work described here and discuss our future work extending and employing the Mimesis architecture.

2. RELATED WORK

Work that integrates intelligent reasoning capabilities with existing game engines can be roughly categorized into three groups based on the degree to which specific AI and game-engine elements are linked in their design. The most prevalent approach is to develop systems in which the AI and game engine are *mutually specific*. In these systems, the focus has been on creating new functionality within a specific game engine using a specific set of intelligent reasoning tools. For example, several researchers have explored the use of planning algorithms similar to the ones employed within Mimesis in order to generate novel action sequences for characters inside existing game engines. For instance, Cavazza and his collaborators [4,5], have integrated both hierarchical task planning and heuristic search planning with Unreal Tournament to explore the creation of interactive narrative within games. The work on the Mission Rehearsal Exercise by Hill, et al [9] integrates an intelligent system based on the Soar cognitive architecture with a sophisticated virtual reality training environment.

A second category of systems integrating AI techniques with game engines can be defined as *AI specific*. In these approaches, a specific collection of AI tools have been designed or adapted for use across more than one game environment. For example, work by Laird and his students [11,12] has integrated the Soar architecture mentioned above into games such as Quake and Decent 3. The third category of systems can be defined as *game-specific*, that is, systems whose design goals include the ability for users to provide customized AI elements for integration into a specific game engine. For example, the Gamebots project, developed jointly at ISI and CMU [1], is a game-specific architecture that defines a socket-based interface allowing intelligent agents to control bots within Unreal Tournament.

3. THE MIMESIS ARCHITECTURE

3.1 Design Overview

In this paper, we describe two specific functions that Mimesis provides when integrated with a conventional game engine:

- The generation of intelligent, plan-based character/system behavior at run-time.
- The automatic execution-monitoring and response generation within the context of the plans that it creates.

This paper focuses on the former rather than the later, though execution monitoring and re-planning are discussed in Section 4.2.2.

To provide this functionality, Mimesis addresses three main design challenges. First, it provides a well-defined bridge between the disparate representations used by game engines and AI software. The architecture specifies an action representation for u by AI components with well-understood syntax and semantics and a methodology for creating game-side code that preserves that semantics. Second, it provides an API for game developers that can be readily

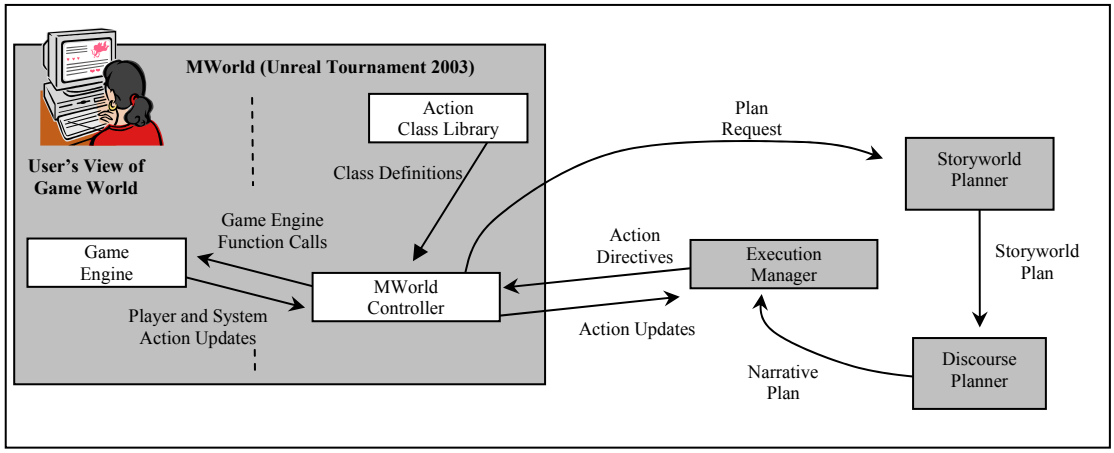


Figure 1. The Mimesis system architecture shown with an MWorld built using Unreal Tournament 2003 as a sample game engine. Individual gray boxes indicate components described in Section 3.2. Within the MWorld component, the vertical dashed line represents the boundary between code created by the Mimesis developer (to the right of the line) and that created by the game engine developer (to the left of the line).

integrated in a typical game engine design. This API has been used to construct a range of applications using both custom-built and commercial game engines. Finally, its architecture facilitates the integration of new intelligent modules, allowing researchers to extend the functionality that Mimesis can provide game developers. The architecture is component-based; individual components register themselves at system start-up and communicate via socket-based message passing, using a set of pre-defined XML message types. This approach facilitates the use of a collection of special-purpose processes that can be extended easily.

Work to date on the Mimesis architecture has been performed in the context of research integrating theories of interactive narrative with computer game development [6,20,27,30]. Consequently, much of the terminology used in this paper is related to narrative structure. Those terms containing references to story refer primarily to action taking place within the game world. This includes both the action of the game's characters as well as the behavior of inanimate objects such as doors, weapons, vehicles, etc. Those terms referring to *discourse* refer primarily to the media resources available within the game engine to tell the story. This includes items

such as a 3D camera, narration and background music. While the terminology is focused on narrative elements, the architecture itself can be applied to a range of game types.

When integrated with a game engine, activity within Mimesis is initiated by a *plan request* made by the game engine itself. To make a plan request, the game engine sends a message to the component called the *story world planner*. The content of a plan request identifies a specific story problem that needs to be solved (e.g., what goals the story actions must achieve, what library of actions are available to build the action sequence from). In response, the system a) creates a plan to control action within the game designed to achieve the story's goals, b) executes the plan's actions, and c) monitors their execution, adapting the plan should one or more of the actions fail to execute correctly.

To create and execute a plan, the Mimesis components follow the process outlined in Figure 1. When the story world planner receives a plan request, it creates a *story world plan*, a data structure that defines an action sequence for the relevant characters and other system-controlled objects within the game world. The story world planner passes this plan to the *discourse planner*, a component responsible for building a *discourse* plan, a structure that controls the camera, backgroung music and other media resources within the game world during the execution of the story world plan.

The discourse planner synchronizes the communicative actions of the discourse plan with the character and environmental actions of the story world plan, creating an integrated *narrative plan* describing all system and user activity that will execute in response to the game engine's plan request. The narrative plan is sent to the *execution manager*, which builds a directed acyclic graph (or a DAG) representing the temporal dependencies between the execution of each action within the narrative plan. The execution manager then acts as a process scheduler, iterating through a cycle of a) selecting the minimal elements in the DAG (that is, those actions that are

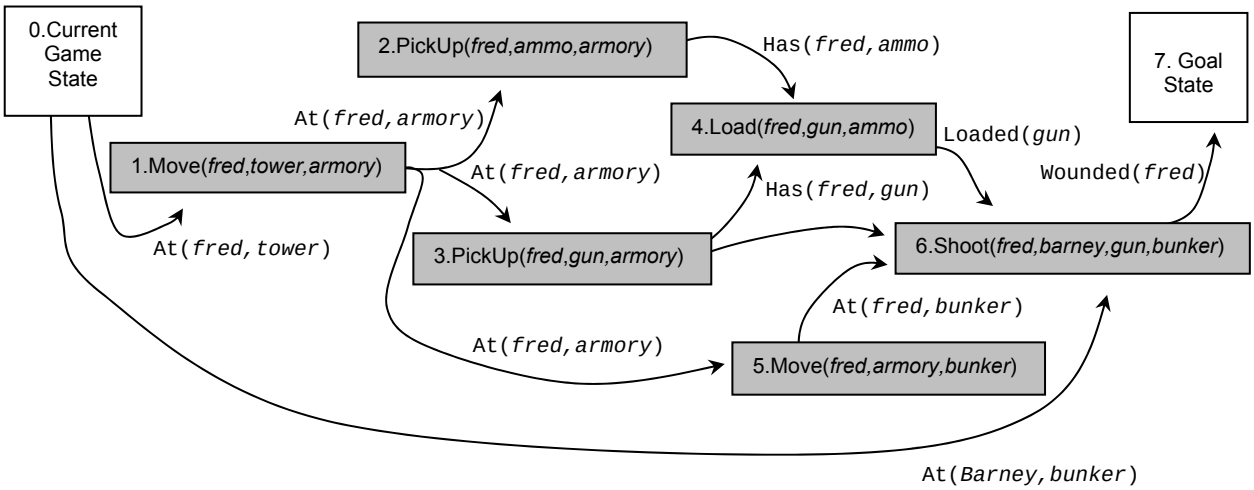


Figure 2. A Mimesis storyworld plan (for simplicity, the plan's hierarchical structure has been elided). Gray rectangles represent character actions and are labeled with an integer reference number, the actions' names and a specification of the actions' arguments. Arrows indicate causal links connecting two steps when an effect of one step establishes a condition in the game world needed by one of the preconditions of a subsequent step. Each causal link is labeled with the relevant world state condition. Temporal ordering is roughly indicated by left-to-right spatial ordering. The white box in the upper left indicates the game's current state description, and the box in the uppr right indicates the current planning problem's goal description.

This plan involves a character, Fred, moving from a tower to the armory (Step 1), picking up some ammo (Step 2) and a gun (Step 3), loading the gun (Step 4) and moving to the bunker (Step 5). There, Fred uses the gun to shoot another character, Barney, wounding him (Step 6).

currently ready to begin execution), b) sending commands to the game engine to execute those actions and c) receiving updates from the game indicating that actions have successfully (or unsuccessfully) completed their execution.

3.2 Components

Mimesis uses five core components, which we describe in more detail in this section. The **mimesis controller** (or MC) acts as the system's central registry; upon initialization, each component connects to the MC via a socket connection and identifies the set of messages that it can accept. Once all components have registered, the MC serves as a message router. Components create messages with specific message types and send them via a socket connection to the MC. The MC forwards each message, again via socket connection, to the appropriate handler component(s) based on the registration information each component has provided.

The **story planner** is responsible for handling plan requests initiated by the game engine. A plan request contains three elements. First, it contains an encoding of all relevant aspects of the current game state. Second, it names one of a set of pre-defined libraries of actions that can be used by the story planner to compose action sequences. Finally, it contains a set of goals for the plan, that is a listing of conditions in the game that must be true at the time that the plan ends its execution. The story planner responds to the plan request by creating a *story world plan*, a data structure that specifies the actions of the characters in the game and the system-controlled objects that will execute over time to achieve the plan request's goals. Section 4 discusses our approach to plan creation in more detail; however, a complete review of the type of planning used in Mimesis is beyond the scope of this paper. Readers should see [26] for a more thorough introduction.

To create the plan, the planner composes sequences of actions drawn from the action library specified in the plan request, searching for an action sequence that will lead from the current game state to one in which all of the plan's goals are met. By creating behavioral control on-demand, a planning approach has several advantages over those techniques that pre-script behavior. First, a planning approach can create action sequences that will successfully execute starting in complex game states that may not be anticipated by developers at design time. Second, because planners search through a space of possible plans while constructing the story plan, heuristics can be used to guide this search so that the nature of the final plan can be tailored to individual users' preferences. For instance, should a player prefer stealthy opponents, plans that use covert movement and stealth attacks can be selected over ones that employ frontal attacks. Third, as we discuss briefly in Section 4.2, planners use techniques to create their plans that add explicit models of the causal and temporal relationships between their plans' actions. Analysis of these structures facilitates re-planning in the face of action failure as well as the

generation of effective explanations for story action, similar to the techniques used by intelligent tutoring systems to explain their own instructional plans [23].

Once the story planner has created the story world plan, it sends the plan to the **discourse planner,** along with a library of communicative actions that can be used by the game engine to convey the unfolding action of the story. These actions might include directives for a 3D camera controller, narrative voice-overs or the use of background music. The discourse planner creates an action sequence containing actions to be carried out not by characters in the story world but by the game engine's media resources. Because these actions must be executed concurrently with the story plan itself, the discourse planner integrates the two plans, creating a *narrative plan* that contains explicit ordering relationships between actions from the two.

Depending on the plan request issued by the game engine, the discourse plan may be as transparent as a sequence of player-controlled first-person camera shots or as complex as a sequence of pans, tracks and other cinematic techniques (for cut scenes or fly-throughs). Like the story planner, the discourse planner uses special-purpose knowledge to construct its plans. For instance, knowledge encoding the use of film idioms and contextual information about the current level's scene geometry can be used to construct cinematic camera plans tailored to the layout of a given game map and the current location of players and characters involved in the action. Because of space limitations, the remainder of the paper will focus on the use of story world plans rather than discourse plans. Young, *et al* [28] provides a discussion of Longbow, the discourse planner used in Mimesis. Bares [2] and Christiansen, et al [7] provide further discussion on automatic camera control in 3D worlds.

The discourse planner sends the narrative plan to the **execution manager**, the component responsible for driving the story's action. The execution manager builds a DAG whose nodes represent individual actions in the plan and whose arcs define temporal constraints between

actions' orderings. The execution manager iteratively removes an action node from the front of the DAG, sends a message to the game engine that initiates the action's execution, and updates the DAG to reflect the state of all actions that are currently running or have recently completed execution.

The **MWorld** includes three elements: the game engine, the code controlling communication between the MWorld and the other Mimesis components and the ac*tion classes,* class definitions that specify the behaviors of each action operator represented within the story and discourse planners (Mimesis' dual approach to action representation is discussed further in Section 4). When the MWorld receives a message from the execution manager directing an individual action to execute within the game world, it extracts the string name of the action from the message and maps the name onto a specific action class. This mapping is done via conventions used in the naming of the actions in both the operator library used by the planners and in the action class hierarchy used by the MWorld. In a similar manner, the Mworld extracts the string identifiers of each of the action's arguments and maps them onto game world objects. This mapping is done, however, by accessing a look-up table created and maintained by the MWorld. Game code registers object references in this table, along with their unique string identifiers, and the MWorld maps objects into an action instance's actual arguments based on positional ordering in the message arriving from the execution manager. From this mapping, an instance of the action class is created, the action's arguments are passed to it and the action's default execution method is called. When each action halts its execution, it notifies the execution manager of its successful (or unsuccessful) completion.

In addition to the five core components described above, Mimesis can be configured with additional components in order to provide extended functionality. For instance, components that provide SQL database access, natural language generation capabilities [8,28] and HTTP services

```
Operator Shoot (?shooter ?target ?weapon ?room)

      Constraints:
            (health-level ?target ?t_health)
            (damage-level ?weapon ?damage_amount)
      Preconditions:
            (has-weapon ?shooter ?weapon)
            (has-ammo ?weapon)
            (in-room ?shooter ?room)
            (in-room ?target ?room)
      Effects:
            (health-level ?target
                  (- ?t_health ?damage_amount))
```

Figure 3. A DPOCL plan operator for the Shoot action used in the plan in Figure 2. In this operator, the Constraints section is used to provide bindings between the relevant game world objects and the opeartor's local variables. The Preconditions ensure that a) the character doing the shooting has the weapon being used to shoot, b) the weapon is loaded, c-d) the shooter and the target character are in the same room. The Effects of the action specify that the health level of the target character is decremented by the damage inflicted by the weapon being used.

have been used in our prototype applications. The component architecture facilitates the integration of these and other new modules into the framework. APIs for constructing new components are available for a range of environments, including Java 2SE, C++ and Allegro Common Lisp.

4. MIMESIS CONTROL MODEL

4.1 The Mimesis Dual Action Representation

Mimesis brings together two representations for action: the procedural representations used by game engines and the declarative representations used by AI systems such as planners. Each of these representations has individual strengths – the efficient management of game state by game engine code and the ability to reason explicitly about action and state change by planning systems. In the following sections, we describe how Mimesis attempts to link the two in a way that preserves the advantages of both.

4.1.1 Model-Based Action Generation

Declarative representations of actions typically used by AI systems characterize the properties of actions – for instance, under what circumstances an action can be executed and how the action alters the world state – without explicitly stating *how* the action performs its tasks. The planners in Mimesis use a declarative representation in which an action is represented using two main elements: its *preconditions* and its *effects*. An action's preconditions are a set of predicates describing those conditions of the game world that must hold in order for the action to execute correctly. An action's effects are a set of predicates capturing all changes to the world state made by the action once it successfully executes. Figure 3 shows an example plan operator for the action of one character shooting another with a weapon.[1]

Mimesis uses DPOCL [29] as the planning algorithm for story planning. A DPOCL plan contains elements composed from five central types. First, they contain *steps* representing the plan's actions. *Ordering constraints* define a partial temporal order over the steps in a DPOCL plan, indicating the order the steps must be executed in. Hierarchical structure in a DPOCL plan is represented by *decomposition links*: a decomposition link connects an abstract step to each of the steps in its immediate sub-plan.[2] Finally, DPOCL plans contain *causal links* between pairs of steps. A causal link connects one step to another just when the first step has an effect that is used in the plan to establish a precondition of the second step.

---

[1] Additional elements can also be added to plan representations. For instance, the operator in Figure 3 uses additional category of constraints, used to query the world state to obtain state variable values for use in the context of the operator.

[2] For expository purposes, the examples in this section have been structured to eliminate the need for hierarchical planning; only casual planning is performed. The DPOCL algortithm, however, along with the plan-based techniques that we present here, are applicable to planning problems using more expressive representations (i.e., causal as well as decompositional structure).

DPOCL uses *refinement search* [10] as a model for its plan reasoning process. Refinement search is a general characterization of the planning process as search through a space of plans. A refinement planning algorithm represents the space of plans that it searches using a directed graph; each node in the graph is a (possibly partial) plan. An arc from one node to the next indicates that the second node is a refinement of the first (that is, the plan associated with the second node is constructed by repairing some flaw present in the plan associated with the first node). In typical refinement search algorithms, the root node of the plan space graph is the empty plan containing just the initial state description and the list of goals that together specify A the planning problem. Nodes in the interior of the graph correspond to partial plans and leaf nodes in the graph are identified with completed plans (solutions to the planning problem) or plans that cannot be further refined due for instance, to inconsistencies within the plans that the algorithm cannot resolve. In Mimesis, the initial planning problem for DPOCL is created using the specifications of the current and goal states taken from the game engine's plan request. The approach to plan generation as search facilitates the creation of plans tailored not just to the particular state of the game world at planning time, but to preferences for certain types of action structure. Search control rules can be defined that direct search towards (or away from) plans that use certain objects, tools, routes, characters or types of action.

4.1.2 Procedural Representations for Action

In order to ensure that every step in a plan can be executed by the game engine, the game developer must create one action class for every action operator in the plan library. The implementation of each action class is responsible for preserving the semantics of the action operator defined in the planner's action library. To this end, the MWorld's abstract action class defines four functions, three which the game developer must provide definitions for in the action

```
class Shoot extends Action;

var MController Agent;
var Pawn MyTarget;
var Weapon MyWeapon
var int OriginalHealth;
var int precondResult;
var int effectsResult;

function int CheckPreconds(){
        if (Agent.Pawn.Weapon != MyWeapon)
                {return 0;}
        else if (!(Agent.Pawn.Weapon.HasAmmo()))
                {return 1;}
        else if (Agent.Room != MyTarget.Room)
                {return 2;}
        else
                {return -1;}
}

function int CheckEffects(){
        if( (MyTarget.Health >= OriginalHealth))
                {return 0;}
        else {return -1;}

function void Body(){
        local int fireMode;
        OriginalHealth = MyTarget.Health;
        Agent.Pawn.SetPhysics(PHYS_None);
        Agent.Pawn.SetViewRotation(rotator(MyTarget.Location -
                Agent.Pawn.Location));
        fireMode = Agent.Pawn.Weapon.BestMode();
        Agent.Pawn.Weapon.StartFire(fireMode);
}

state Executing {

Begin:
        precondResult = CheckPreconds();
        if (precondResult != -1) {
                reportPrecondFailure(precondResult);
                gotoState('Idle'); }
        Body();
        effectsResult = CheckEffects();
        if (effectsResult != -1) {
                reportEffectFailure(effectsResult);
                gotoState('Idle'); }
        reportActionSuccess();
}
```

CheckPreconds() checks each precondition from the corresponding operator in the order in which they are defined there. When a precondition does not hold in the current game state, an integer identifying the failed precondition is returned. If all preconditions hold, the function returns -1.

CheckEffects() runs after the body of the action completes. It verifies each of the operator's effects When an effect does not hold in the current game state, an integer identifying the failed effect is returned. If all effects hold, the function returns -1.

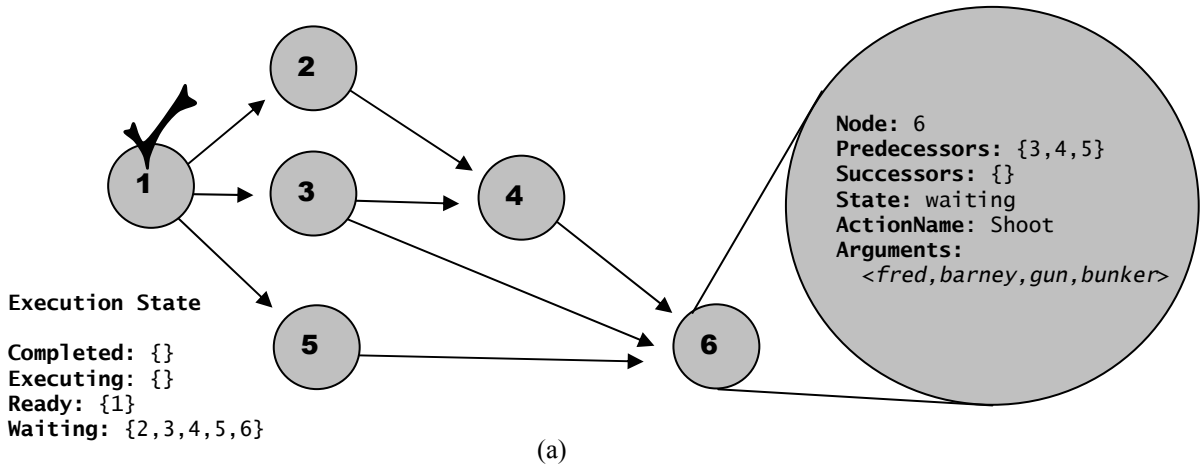
Body() implements the operator's behavior, changing the game state according to the intended meaning of the operator.

The Executing State is identical across all action classes, and so is typically defined in the top-level Action class. It is included here for reference.

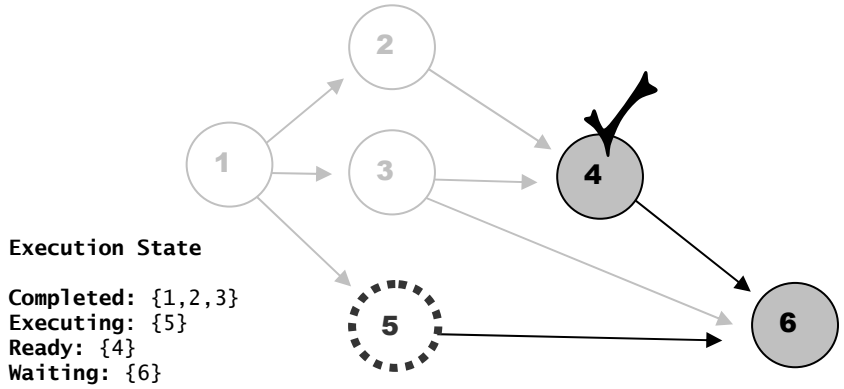
The Executing code first checks the action's preconditions. If all preconditions are met, then it runs the action's body. Next, it checks the action's effects. If all effects hold, then the action sends a message to the execution manager indicating that it has completed successfully. If an error is encountered along the way, the action sends an error report to the execution manager, facilitating re-planning.

Figure 4. An example UnrealScript action class for the Shoot operator defined in Figure 3.

classes that he or she defines. An action's CheckPreconds() function is responsible for verifying that the conditions described in the corresponding operator's preconditions currently hold in the game world. The Body() function is responsible for changing the state of the world in accordance with the meaning of the action operator. The CheckEffects() function verifies that the conditions described in the operator's effects have actually been obtained in the game world immediately after its execution.



(a)



(b)

Figure 5. Two examples of the execution DAG used by the Execution Manager, matching the plan shown in Figure 1. In these figures, gray circles indicate steps in the DAG that are as yet unexecuted. An arc between two nodes indicate that the first node must complete execution before the second node can begin execution. Nodes with grayed borders are ones that have completed execution; those with dashed borders indicate actions that are currently executing. Nodes with check marks are currently ready to execute.

In Figure 5a, execution of the plan has not yet begun. Figure 5b shows the execution DAG after Fred has gone to the armory, picked up the ammo and gun and is in the process of moving to the bunker. At this point, he's ready to load the gun, but has not yet done so.

The Executing() function is an abstract function defined only in the parent action class. This function, shared by all action classes, first calls the action's CheckPreconds(). If one of the action's preconditions is not met, then the Executing() function stops execution and sends a failure message to the Execution Manager. Otherwise, the function calls the action's Body() and then calls the action's CheckEffects() function. If one of the action's effects does not hold, the function halts execution and reports this condition to the Execution Manager. Otherwise, if no problems were encountered, the Executing() function reports that the action has completed successfully. An

example action class definition is shown in Figure 4. This definition is written in UnrealScript, Unreal Tournament's scripting language, though APIs exist for a range of languages and are discussed in Section 5.

4.2 Execution Management, Monitoring and Mediation

4.2.1 Execution Management

In order to control and monitor the order of execution for the actions within the narrative plan, the Execution Manager builds a DAG that represents all temporal dependencies between the actions in the DAG. This temporal information is created by the story world and discourse planners when the plans are first built and extracted by the execution manager when it receives the narrative plan. An example execution DAG for the plan from Figure 2 prior to any plan execution is shown in Figure 5a.

In order to initiate the execution of an action from the execution DAG, the execution manager sends a message to the MWorld specifying the action's name and a tuple naming the action's arguments. To build this message, the narrative plan's identifiers for the action and its parameters are used to create string names that act as unique identifiers agreed upon by both the MWorld developer and the developer of the plan operators (the string names are typically created based on the symbol names used by the plan operators). As mentioned above, the MWorld uses a registration procedure to map these names into the actual arguments of function calls for action class instances.

4.2.2 Mediation

Complications to the plan execution process arise when the user performs actions within the story world that interfere with the structure of the story plan. For instance, suppose that the user

controls a character named Betty in the world of the plan in Figure 2. If the user decides to pick up the ammo in the armory before Fred moves there, Fred's subsequent pickup action would fail, as would each subsequent action that depended upon the ammo being available to Fred. Because *every* action that the user performs might potentially change the world state in a manner that could invalidate some as-yet-unexecuted portion of the narrative plan, the MWorld checks for such unwanted consequences each time that the player initiates a potentially harmful action. The MWorld maps such commands issued by the player onto the same plan operators used to create the plans by the story world planner. The MWorld monitors the user's action commands prior to executing the corresponding action's code in the game engine and signals an *exception* whenever the user attempts to perform an action that would change the world in a way that conflicts with the causal constraints of the story plan.

Exceptions are dealt with in one of two ways. The most straightforward is via *intervention*. Typically, the success or failure of an action within a game engine is determined by the code that approximates the rules of the underlying game world (e.g., setting a nuclear reactor's control dial to a particular setting may cause the reactor to overload). However, when a user's action would violate one of the story plan's constraints, Mimesis can intervene, causing the action to fail to execute. In the reactor example, this might be achieved by surreptitiously substituting an alternate set of action effects for execution, one in which the "natural" outcome is consistent with the existing plan's constraints. A control dial momentarily jamming, for instance, will preserve the apparent consistency of the user's interaction while also maintaining safe energy levels in the story world's reactor system.

The second response to an exception is to adjust the structure of the plan to *accommodate* the new activity of the user. The resolution of the conflict caused by the exception may involve only minor restructuring of the plan's causal connections, for instance, selecting a different but

compatible location for an event when the user takes an unexpected turn down a new path. Accommodation may involve more substantive changes to the story plan, however, and these types of modifications can be computationally expensive. For instance, should a user unintentionally destroy a device required to rescue a game's central character, considerable re-planning will be required on the part of the story and discourse planners.

In order to detect and respond to exceptions during the execution of a story world plan, the execution manager analyzes each plan prior to execution, looking for points where enabled user actions (that is, actions whose preconditions for execution are satisfied at that point in the plan) can threaten its plan structure. When potential exceptions are identified, the planner weighs the computational cost of re-planning required by accommodation against the potential cost incurred when intervention breaks the user's sense of agency in the virtual world. A more detailed discussion of the interaction between the MWorld and the story Planner with respect to exception detection and handling can be found in [20].

5. BUILDING GAMES USING MIMESIS

We have created two sets of APIs for use by developers integrating Mimesis functionality into their games. The first API, the *mclib*, is a library of functions that provides registration, message passing and parsing functionality for new Mimesis components. APIs are available for C++, Java 2 Standard Edition and Allegro Common Lisp version 6.2. The second, called the *mwlib*, provides the functionality needed to integrate new or existing game engines with the rest of the Mimesis components. The mwlib includes functions that allow an MWorld to register with the MC, to record unique names for game objects (facilitating the translation between XML strings and action class method invocation) and to report the progress of action class execution

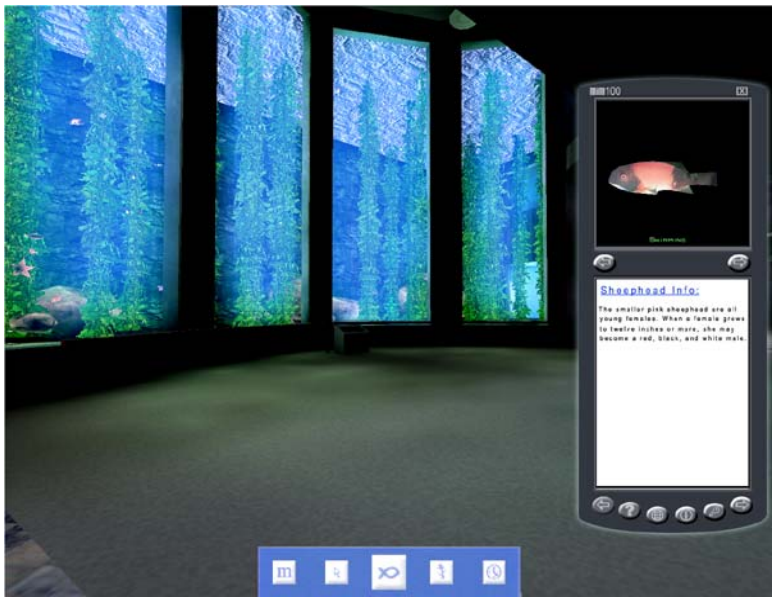


Figure 6. A screenshot of the Mimesis Virtual Aquarium, a simulation of a portion of the Monterey Bay Aquarium, a large marine-science learning center in Monterey, CA.

(via the ReportPrecondFailure(), ReportEffectFailure() and ReportActionSuccess() functions described in Section 4).

We have developed mwlib libraries for all the environments for which mclibs exist. Further, we have created mwlib APIs for Unreal Tournament, Unreal Tournament 2003 and two text-based virtual reality environments: MudOS [18] and LambdaMOO [13]. To see how these APIs may be put to use, consider the following example in which a university research team partners with a game development studio to create an educational game in which players move through a simulation of the Monterey Bay Aquarium, a real-world marine-science learning center. The university team seeks to integrate their research on intelligent tutoring systems into the 3D game engine used by the game developers, in this case, Epic Games' Unreal Tournament 2003. The game developers have created levels that correspond to the Aquarium's exhibits (see Figure 6) and have written UnrealScript code to control both the Aquarium's inhabitants (i.e., marine life, visitors and tour guides) and its physical environment (e.g., elevators, lights, TV monitors).

The research group has developed effective pedagogical techniques for teaching material regarding the behaviors of marine animals that involve the tight coordination between animated pedagogical agents [14] (e.g., tour guide characters), the fixed information resources available in the environment (e.g., the explanatory labels associated with each habitat in the aquarium) and the behavior of the animals themselves (e.g., the schooling behavior of tiger sharks before feeding). The team wants to integrate a planning system into the game in order to generate plans that control the tour guide and the marine animals living inside the Kelp Forest, one of the Aquarium's larger habitats. These plans will be customized according to a) queries about the environment posed by the player and b) the state of the game world, including the position of the tour guide relative to the player, the composition of marine life present in the Kelp Forest at the time of the query and the textual content of visible labels on the Kelp Forest window through which the player is looking.

To integrate the work of both project groups, the team first defines a library of plan operators using a GUI provided with the planning systems used by Mimesis. These plan operators specify the preconditions and effects for the primitive actions used within the planner to create story world plans. In the aquarium example, these might include actions such as gesturing at objects, speaking to the player, turning on and off displays (for tour guide characters) or eating food, retracting into one's shell and self-grooming (for animals such as sea turtles, snails, rockfish or otters).

Second, the team defines the set of UnrealScript action classes that will execute within the MWorld. One action class is defined for each action operator used by the planner; in each action, the CheckPreconds(), Body() and CheckEffects() methods are written so that they correctly check and manipulate the engine state captured by the semantics of the preconditions and effects

in the corresponding action operator. These methods will, in turn, typically call functions already defined in the developer's game-specific code or within the engine itself.

Third, the team writes initialization code that will register all of the game objects in the MWorld, allowing a mapping to be made from the names used to refer to these objects by the planner to the object instances in the game engine. The team extends the game's start-up code to call the mwlib registration function in order to make the initial connection with the MC.

Finally, the team defines the points in the game where requests for plan structures will be made. Plan requests can be initiated – and the resulting plans executed – at the very beginning of the player's session, at fixed trigger points throughout game play or dynamically in response to user input or arbitrary game engine code.

We have successfully used the process described above to create a prototype intelligent interactive tour of the Monterey Bay Aquarium (except that we have filled the roles of both researchers and game developers). Additional small-scale games demonstrating features of our research on interactive narrative have been developed using the same methodology for Unreal Tournament 2003 and for OpenGL worlds running on the Playstation 2 (PS2Linux). More details can be found at the Mimesis project website, http://mimesis.csc.ncsu.edu/.

6. SUMMARY AND CONCLUSIONS

We are extending the work described here along several dimensions. First, the current mclib and mwlib APIs are being implemented in Java 2 Micro Edition in order to integrate with games running on mobile platforms like PDAs and mobile telephones. We are considering creating mwlibs for other 3D games engines as well. The APIs and several of the existing components are being implemented within the .NET framework in order to increase platform coverage.

We are also exploring ways to extend the architecture to handle control of multi-player games. Even though the components of the current system can run in a distributed manner, with each component executing on distinct processors, the run-time performance of the planning processes limits Mimesis' application to single-user games. To address this limitation, we are exploring the performance trade-offs involves in the execution of the architecture on a grid-based gaming platform [15, 24] and addition of inter-planner communication needed to coordinate plans for multiple players.

As we mentioned above, Mimesis is being used in our research group to investigate new models of interactive narrative in games [20, 30]. In this work, we are building new intelligent components for modeling the user's knowledge about the stage of the game world, for drawing inferences about a user's plans and goals based on observation of their game actions [3,12] and for cinematic control of the game's camera [2,7], for use, for instance, in the automatic generation of cut-scenes or fly-throughs.

In summary, the Mimesis system defines an architecture for integrating intelligent plan-based control of action into interactive worlds created by conventional game engines. To this end, it bridges the gap between the representations used by game developers and those of AI researchers. It provides an API for game developers that can be readily integrated in a typical game engine design and its component-based architecture makes the addition of new intelligent modules straightforward.

A detailed design document used for implementing new Mimesis components can be found on the Mimesis mclib page (http://mimesis.csc.ncsu.edu/mclib). All APIs described here are available for download from the Mimesis project home page (http://mimesis.csc.ncsu.edu/).

7. ACKNOWLEDGEMENTS

The work of the Liquid Narrative group has been supported by National Science Foundation CAREER award 0092586 and by Microsoft Research's University Grants Program. Contributions to the Mimesis architecture have been made by a number of students, including Dan Amerson, William Bares, Charles Callaway, D'Arcey Carol, David Christian, Shaun Kime, Milind Kulkarni and the many students of CSC495, *Research Projects in Intelligent Interactive Entertainment*. Thanks also goes to Epic Games for their support of our work.

8. REFERENCES

[1] Adobbati, R., Marshall, A.N, Scholer, A., Tejada, S., Kaminka, G.A., Schaffer, S., Sollitto, C., Gamebots: A 3D virtual world test bed for mult-iagent research, In *Proceedings of the Second International Workshop on Infrastructure for Agents, MAS and Scaleable MAS*, Montreal, Canada, 2001.

[2] Bares, W., Gregoire, J., and Lester, J., Realtime Constraint-Based Cinematography for Complex Interactive 3D Worlds, in the Proceedings of the Conference on Innovative Applications of Artificial Intelligence, 1998.

[3] Sandra Carberry. Techniques for Plan Recognition. *User Modeling and User-Adapted Interaction,* 11(1-2), pp. 31-48, 2001

[4] Cavazza, M., Charles, F., Mead, S., Planning characters' behaviour in interactive storytelling. In the *Journal of Visualization and Computer Animation*, 13(2): 121-131, 2002

[5] Charles, F. , Lozano, M. Mead, S., Bisquerra, A., and Cavazza, M. , Planning Formalisms and Authoring in Interactive Storytelling, *in the Proceedings of the First International Conference on Technologies for Interactive Digital Storytelling and Entertainment*, 2003.

[6] Christian, D. and Young, R. M. *Comparing Cognitive and Computational Models of Narrative Structure.* Liquid Narrative Technical Report TR03-001, Liquid Narrative Group, Department of Computer Science, North Carolina State University, Raleigh, NC. 2003.

[7] Christianson, D., Anderson, S., He, L., Salesin, D., Weld, S., and Cohen, F., Declarative Camera Control for Automatic Cinematography, in *the Proceedings of the Conference of the American Association for Articifical Intelligence,* page 148-155, 1996.

[8] Elhadad, M. *Using argumentation to control lexical choice: a unification-based implementation.* PhD thesis, Computer Science Department, Columbia University, 1992.

[9] Hill, R. W., Gratch, J., Marsella, S., Rickel, J., Swartout, W., and Traum, D. Virtual Humans in the Mission Rehearsal Exercise System, *Kunstliche Intelligenz* (Special Issue on Embodied Conversational Agents), 2003.

[10] Kambhampati, S., Knoblock, C.A. and Qiang, Y. Planning as refinement search: a unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence*, 76:167-238, 1995.

[11] Laird, J.E., Using a computer game to develop advanced AI. *Computer*, July 2001, 70-75.

[12] Laird, J., It knows what you're going to do: adding anticipation to a Quakebot, in *the Proceedings of the Fifth International Conference on Autonomous Agents*, 385-392, 2001

[13] LambdaMoo, LambdaMOO Community Page, available at http://www.lambdamoo.info/.

[14] Lester, J., Stone, B., and Stelling, G., Lifelike pedagogical agents for mixed-initiative constructivist learning environments, in the *International Journal of User Modeling and User-Adapted Interaction*, 9(1-2), pages 1-44, 1999.

[15] Levine, D., Wirt, M., Whitebook, B., *Advances in Computer Graphics and Game Development: Practical Grid Computing For Massively Multiplayer Games*, Charles River Media, 2003.

[16] Lewis, M., and Johnson, L., Game Engines in Scientific research, *Communications of the ACM*, page 27 – 31, vol 45, no 1, January 2002.

[17] Marsella, S., Johnson, W., and LaBore, C. Interactive pedagogical drama for health interventions, In *Proceeding of the 11th International Conference on Artificial Intelligence in Education*, Sydney, Australia, July 2003.

[18] MudOS, *The Official MudOS Support Page*, available at: http://www.mudos.org

[19] Pottinger, D., Computer Player AI: The New Age, *Game Developer Magazine*, July, 2003. by

[20] Riedl, M., Saretto, C.J. and Young, R. M. Managing interaction between users and agents in a multi-agent storytelling environment In: *Proceedings of the Second International Conference on Autonomous Agents and Multi-Agent Systems*, June, 2003.

[21] Rosenbloom, Andrew, A Game experience in every application, *Communication of the ACM*, page 29-31, vol 46 no 7, July 2003.

[22] Seneff, S., Hurley, E., Lau, R., Pao, C., Schmid, P. and Zue, V, "Galaxy-II: A Reference Architecture for Conversational System Development," *ICSLP '98*, pp. 931-934, Sydney, Australia, December, 1998.

[23] Swartout, W., Paris, C., and Moore, J., Design for explainable expert systems. *IEEE Expert*, 6(3):58--64, 1991.

[24] Tapper, D, and Olhava, S., IBM Global Services: Scoring Big on Utility-Based Gaming, IDC Report #27453, Jun3 2002

[25] van Lent, M., Laird, J., Buckman, J., Hartford, J., Houchard, S., Steinkraus, K. and Tedrake, R., Intelligent Agents in Computer Games, in *the Proceedings of the National Conference of the American Association for Artificial Intelligence*, 2001.

[26] Weld, D. An Introduction to Least-Commitment Planning. *AI Magazine*, 27-61, 1994.

[27] Young, R. M. Notes on the Use of Plan Structures in the Creation of Interactive Plot. In: *The Working Notes of the AAAI Fall Symposium on Narrative Intelligence*, Cape Cod, MA, 1999.

[28] Young, R. M., Moore, J. D., and Pollack, M., Towards a Principled Representation of Discourse Plans, In *the Proceedings of the Sixteenth Conference of the Cognitive Science Society*, Atlanta, GA, 1994.

[29] Young, R. M., , Pollack, M. E., and Moore, J. D., Decomposition and causality in partial-order planning, in *Proceedings of the Second International Conference on AI and Planning Systems,* Chicago, IL, pages 188-193, July, 1994.

[30] Young, R. M. And Riedl, M. Towards an Architecture for Intelligent Control of Narrative in Interactive Virtual Worlds. In: *Proceedings of the International Conference on Intelligent User Interfaces*, January, 2003.