An Architecture for Integrating Plan-Based Behavior Generation with Interactive Game Environments

P. Michael Young, Mark O. Riedl, Mark Brudy, Amanda Jeffel
North Carolina State University
Raleigh, NC 27695
young@csc.ncsu.edu

R.J. Hartie
Knowwonder Digital Mediaworks
rockdeity@mindspring.com

C.J. Skarda
Microsoft Corporation
cjcar@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 by the requirement to anticipate game state at design time. Research in artificial intelligence (AI) has developed a number of techniques for automatic plan generation that create novel action sequences that are highly sensitive to runtime context.

In this article, 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 AI that focuses on the automatic creation of novel and effective action sequences. Users of the system construct two parallel modes 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 runtime 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 article, we describe the architecture, its main components, and the APIs available for integrating it into existing or new game engines.
One central challenge for game developers is the need to create compelling behavior for a game’s characters and objects. For example, 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 precompiled scripts or the development of control software that implements a finite-state machine. While scripting approaches may require less complicated design than that needed to construct a finite-state behavioral model, a scripting language’s runtime 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 [Pettinger 2003]. 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 runtime context. Recent work [Cavazza 2002; Hill 2003] 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 article, 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 runtime 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, and so forth. 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, although 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 use 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 article is organized as follows. In the next section, we summarize work integrating AI research and development with computer game engines. Then, we describe the Mimesis architecture further. Following that, 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 maintain their coherence effectively during execution. Next, we give a high-level discussion of the methodology for developing games that use the Mimesis architecture. Finally, we summarize the work described here and discuss our future work extending and employing the Mimesis architecture.

**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 to generate novel action sequences—that is, action sequences that have not been pre-scripted or otherwise defined ahead of time—for characters inside existing game engines. For example, [Cavazza 2002; Charles 2003] have integrated both hierarchical task planning and heuristic search planning with *Unreal Tournament* to explore the creation of interactive narrative within games. However, they have not yet extended their work to generate API across a range of development environments. The work on the Mission Rehearsal Exercise by Hill, et al [Hill 1995] integrates an intelligent system based on the Soar cognitive architecture with a sophisticated virtual reality training environment. This system, too, is specific to the high-end runtime environment used to create the training modules.

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 has been designed or adapted for use across more than one game environment. For example, work by Laird and his students [Laird 2001] has integrated the Soar architecture mentioned previously into games such as *Quake* and
Decent 3. In this work, Soar models the cognitive state and reasoning performed by individual agents acting within the game world; global coherence of action is limited to that which emerges based on loosely coordinated individual action. The work by Atkins, et al [Atkins 1998] on the Hierarchical Agent Control system has been integrated with real-time strategy games, although the system has specifically been designed for use with a range of agent environments. Because their approach is quite general, they assume no direct access to the internal state of the runtime environment. Instead, action sequences must include specific sensing actions, computation that is often unnecessary when a game environment is generating actions for itself.

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 [Adobbbati 2001], is a game-specific architecture that defines a socket-based interface allowing intelligent agents to control bots within Unreal Tournament. While the API developed by this project allows easy integration of external code with UT, no facilities for generating or controlling the execution of action sequences is provided.

THE MIMESIS ARCHITECTURE

Design Overview

In this article, 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 runtime.
- The automatic execution-monitoring and response generation within the context of the plans that it creates.

This article focuses on the former rather than the latter, although execution monitoring and replanning are discussed in the section Mediation.

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 use 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 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 startup and communicate via socket-based message passing, using a set of predefined XML message types. This approach facilitates the use of a collection of special-purpose processes that can be easily extended.
Work to date on the Mimesis architecture has been performed in the context of research integrating theories of interactive narrative with computer game development (Christian 2003; Riedl 2003; Young 1999; Young 2003). 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, and so forth. 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 (Figure 3.1).

![Diagram](image)

**FIGURE 3.1** 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 *Components*. 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).

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; for example, what goals the story actions must achieve and what library of actions are available to build the action sequence from (more detail about the nature of plan goals used by the story world planner is provided in the section *Components*). 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 3.2. 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, background music, and other media resources within the game world during the execution of the story world plan.
FIGURE 3.2 A Nimesis story world 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 upper right indicates the current planning problem’s goal description. This plan involves one nonplayer 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 6), and moving to the bunker (Step 5). There, Fred uses the gun to shoot another character, Barney, wounding him (Step 6).

The discourse planner synchronizes the communicative actions of the discourse plan with all character and environmental actions of the story world plan, creating an integrated narrative plan describing all system, nonplayer character, 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 (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 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.
Components

Mimesis uses five core components, which we describe in more detail in this section. The *mimesis controller* (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. This encoding is managed by special-purpose functions that translate values stored in the game data structures into tuples (e.g., first-order logic atomic sentences). Second, it names one of a set of predefined 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. These goals are also defined using the same tuple language that is used to characterize the current game state. An example entry from an action library is shown in Figure 3.3. Tuples appearing in the current state description and the story world plan’s goals might include the following:

<table>
<thead>
<tr>
<th>Current State Description (partial)</th>
<th>Story World Plan Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. The section *Mimesis Control Model* 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 article. Readers should see [Weld 1994] 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 example, should a player prefer stealthy opponents, plans that use covert movement and stealth attacks can be selected over ones that employ frontal attacks. These types of preferences can be acquired by settings explicitly set by each user at startup or stored in a profile. (Future work will investigate the acquisition of these preferences by tracking the user's actions in similar contexts and exploring the use of learning algorithms.) Third, as we discuss briefly in the section Execution Management, Monitoring, and Mediation, 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 replanning 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 [Swarout 1991].

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 example, 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 this article focuses on the use of story world plans rather than discourse plans. [Young 1994] provides a discussion of Longbow, the discourse planner used in Mimesis. Bares [Bares 1998] and Christianson, et al [Christianson 1996] 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 action 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 later in the article). 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 lookup 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 previously, Mimesis can be configured with additional components to provide extended functionality. For example, components that provide SQL database access, natural language generation capabilities [Elhadad 1992 & Young 1994], and HTTP services 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#, C++, and Allegro Common Lisp.

**Mimesis Control Model**

**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.

**Model-Based Action Generation**

Declarative representations of actions typically used by AI systems characterize the properties of actions—for example, 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 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 3.3 shows an example plan operator for the action of one character shooting another with a weapon.

**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 3.3** A DPOCL plan operator for the shoot action used in the plan in Figure 3.2. In this operator, the Constraints section is used to provide bindings between the relevant game world objects and the operator'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.

Mimesis uses DPOCL [Young 1994] 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 in which the steps must be executed. 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 subplan. (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 algorithm, however, along with the plan-based techniques that we present here, are applicable to planning problems using more expressive representations (i.e., casual as well as decompositional structure). 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 3.3 uses additional category of constraints, used to query the world state to obtain state variable values for use in the context of the operator.
DPOCL uses refinement search [Kambhampati 1995] 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 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 to, for example, 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 toward (or away from) plans that use certain objects, tools, routes, characters, or types of action.

Procedural Representations for Action
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 that the game developer must provide definitions for in the action 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.

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, the Executing() function stops execution and sends a failure message to the execution manager; the game engine and the story world planner then synchronize their views of the current game state, and replan the story’s action. Otherwise, the function calls the action’s Body() and then calls the action’s CheckEffect() 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 3.4. This definition is written in UnrealScript, *Unreal® Tournament*’s scripting language, although APIs exist for a range of languages and are discussed in the section *Building Games Using Mimesis*. 
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 (AgentPawn.Weapon != MyWeapon)
        {return 0;}
    else if (!((AgentPawn.Weapon.HasAmmo())))
        {return 1;}
    else if (AgentRoom != 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;
    AgentPawn.SetPhysics(PHYS_None);
    AgentPawn.SetViewRotation(rotator(MyTarget.Location -
                                       AgentPawn.Location));
    fireMode = AgentPawn.Weapon.RestMode();
    AgentPawn.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();

FIGURE 3.4 An example UnrealScript action class for the Shoot operator defined in Figure 3.3.
**Execution Management, Monitoring, and Mediation**

**Execution Management**

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 3.2 prior to any plan execution is shown in Figure 3.5a.

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.

![Diagram of execution DAG](image)

**FIGURE 3.5** Two examples of the execution DAG used by the execution manager, matching the plan shown in Figure 3.2. In these figures, gray circles indicate steps in the DAG that are as yet unexecuted. An arc between two nodes indicates 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 3.5a, execution of the plan has not yet begun. Figure 3.5b 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.
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 previously, the MWorld uses a registration procedure to map these names into the actual arguments of function calls for action class instances.

**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 example, suppose that the user controls a character named Betty in the world of the plan in Figure 3.2. 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 on 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 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 example, 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 example, 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 example, should a user unintentionally destroy a device required to rescue a game's central character, considerable replanning will be required on the part of the story and discourse planners.

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 replanning 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 [Riedl 2003].
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 *mcilib*, 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 (via the `ReportPrecondFailure()`, `ReportEffectFailure()` and `ReportActionSuccess()` functions described previously).

We have developed mwlib libraries for all the environments for which mcilibs exist. Further, we have created mwlib APIs for Unreal Tournament, Unreal Tournament 2003, and two text-based virtual reality environments: [MudOS; LambdaMOO]. 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 3.6) and have written UnrealScript

![FIGURE 3.6](image-url) 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.
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 [Lester 1999] (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 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 displays on and off (for tour guide characters), eating food, or 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 CheckPreconditions(), 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 those objects by the planner to the object instances in the game engine. The team extends the game’s startup code to call the mwlib registration function 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 gameplay, or dynamically in response to user input or arbitrary game engine code.

We have successfully used the process just described 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 Web site http://mimesis.csc.ncsu.edu/.

**SUMMARY AND CONCLUSIONS**

We are extending the work described here along several dimensions. First, the current mwl and mwl2 APIs are being implemented in Java 2 Micro Edition to integrate with games running on
mobile platforms like PDAs and mobile telephones. We are considering creating mwiibs for other 3D game engines as well. The APIs and several of the existing components are being implemented within the .NET framework to increase platform coverage.

We are also exploring ways to extend the architecture to handle control of multiplayer games. Even though the components of the current system can run in a distributed manner, with each component executing on distinct processors, the runtime performance of the planning processes currently limits Mimesis' application to single-user games. To address this limitation, we are exploring the performance trade-offs involved in the execution of the architecture on a grid-based gaming platform [Levine 2003; Tapper 2002] and addition of inter-planner communication needed to coordinate plans for multiple players.

As mentioned earlier, Mimesis is being used in our research group to investigate new models of interactive narrative in games [Riedl 2003; Young 2003]. 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 the user's game actions [Carberry 2001; Laird 2001], and for cinematic control of the game's camera [Bares 1998; Christianson 1996] for use, for example, in the automatic generation of cut-scenes or fly-throughs.

The central feature of the architecture is the effective integration of plan-based techniques with a range of game engines. Under the assumption that their representations of actions in the story world is correct, planners can compose action sequences that achieve story world goals in ways that vary according to context, providing new action sequences for each run of a given story and generating action sequences for new stories not originally defined by a game's designers. Further, assuming the accurate implementation of the semantics of the planner's operators within the game engine's native code, the plans can be shown to be provably sound; that is, they can be guaranteed to execute correctly as long as users do not interfere with their progress. Finally, the structure of the plans created by the planner contain temporal and causal annotations sufficient to detect when a user's actions will invalidate the plan, and these annotations guide replanning in ways that result in minimal effort on the part of the system.

While the architecture is now complete, evaluation of the system is ongoing. In particular, we are hoping to evaluate the computational effectiveness of the plan-based approach. A central aspect of the computational effectiveness of the system involves the time taken by the planning system to generate complex action sequences for story world plans. Currently, our test-bed scenarios involve libraries of 20–40 abstract actions (e.g., rob-bank, open-vault, disarm-alarm) and as many primitive actions (e.g., pick-up, turn-to-face, move-to, shoot, swipe-cardkey), and generate plans with roughly 50 primitive actions in under one second (running in Allegro Common Lisp on a Pentium 4 1GHz machine). While these plans are relatively short, they are adequate for many cut-scenes and other short action sequences.

We anticipate that the use of plan-generation techniques that interleave planning and execution will facilitate the effective scaling of our algorithms. In this approach, we will exploit the hierarchical structure of the plans used by Mimesis by a) first generating the complete plan down to the primitive level for just that portion of the plan that must execute first, and b) deferring the completion of the later portions of the plan until the initial portion of the plan is executing. In addition, we are reimplementing the DPOCL planning algorithm in C# to exploit the runtime efficiency of the .NET framework. Because the primary use of the planner comes at system startup (to generate the initial story world plan), the impact of the delay caused by the planning system can be folded in to the overall system initialization time, minimizing its impact on overall gameplay.
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).

ACKNOWLEDGMENTS

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 go to Epic Games for their support of our work.

REFERENCES


