In The Working Notes of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment, Stanford, CA, March 2001.

# An Overview of the Mimesis Architecture: Integrating Intelligent Narrative Control into an Existing Gaming Environment

**R. Michael Young**

Liquid Narrative Research Group
Department of Computer Science
NC State University
Raleigh, NC 27695
young@csc.ncsu.edu

## Introduction

The Liquid Narrative research group at North Carolina State University, a multidisciplinary group of faculty and students seeking to create interactive narrative environments. To do this, we are designing and building intelligent systems capable of creating structured interaction within virtual worlds that achieves the same kind of cognitive and affective responses to interactive stories as that seen in the participants of conventional narrative media such as the film or the novel. Our approach exploits a well-founded, declarative model of action and intention within virtual worlds, in combination with new computational models of narrative structure. The intended result is the production of engaging, story-based interactive applications for education, training and entertainment.

The LN group's current focus is the Mimesis (mim-E-sis) system, an implementation of an intelligent controller for virtual worlds that generates and maintains a coherent, narrative-based storyline. The system combines current research work in AI planning and natural language discourse generation with the real-time control of an existing commercial game system, Epic Games' Unreal Tournament. The remainder of the paper gives a short overview of the Mimesis system components. Many of the components of Mimesis have already been completed, and we anticipate end-to-end system integration shortly.

## Background: Unreal Tournament

Epic Games' Unreal Tournament (UT) is currently one of the most popular titles in the first-person gaming genre. The game's architecture is a client-server design (with single-user mode simply running both client and server on the same machine). UT client processes are responsible for user-side input and output (e.g., keyboard and/or joystick for input, graphics rendering and audio generation as output); the UT server maintains a consistent world model across all users by acting as the centralized controller where all functions that change the state of the world execute.

The functionality provided by the UT server executes in one of two forms, one modifiable by game licensees (i.e., anyone that purchases a copy of the game CD), the other available only to developer licensees (e.g., those who enter into special contractual agreements with the game developer). Execution of the later, called native code execution, involves running compiled C++ executables. The former involves running UnrealScript functions. UnrealScript, a specialized object-oriented scripting language developed by Epic Games to define UT object classes and control the behavior of their instances within a UT game world. UnrealScript compiles into byte code and so is platform independent. The UnrealScript process model simulates its own thread-based architecture, providing the Mimesis developer with the abstraction that each UnrealScript object runs its own code within its own thread. Fortunately, game licensees have access to the complete UnrealScript SDK as well as a class browser, a script editor and a level editor (i.e., a 3D world modeler). Built-in support for third-party development of add-on software is provided as well, making UT one of the gaming environments more accessible to AI researchers seeking to integrate their work into game environments.

## Mimesis Architectural Overview

The Mimesis system integrates AI control with off-the-shelf commercial games. The benefit of this approach for AI researchers is both immediate; use of systems like UT provide readily accessible, stable and high-quality graphics, networking, database and process execution support for virtual environments, eliminating the need for

time consuming development of these components in a research project.

As I describe below, Mimesis is composed of a number of distinct modules some that are integrated directly into the UT clients and servers, others that reside on remote machines and communicate with the UT server via socket-based connections. The UT server, extended to include Mimesis software components, is called a *Mimesis Unreal Tournament Server* (MUTS) and the intelligent controller operating on a remote machine is called the *Mimesis Controller* (MC). The role of the MUTS is to provide low-level access to the UT environment (process invocation, monitoring and control) while the role of the MC is to serve as a centralized intelligent source for the design and (high-level) control of a coherent, compelling narrative-based interaction over time.

The principal factor that allows the integration of AI research tools and technology with the existing UT server engine is the sharing between the two of a declarative representation of action and of the conditions within the UT server's virtual world. On the MC side, an HTN planner-style action representation encodes all actions that can be taken by characters in the virtual environment. On the MUTS side, each of the primitive actions in this representation is mirrored by a functional definition of UnrealScript code that directly implements the action it mirrors. Currently, the mirroring of functionality between the MUTS and the MC action representations is achieved manually by system designers, although, as I describe in the final section of this paper, much of the process is performed automatically.

## Mimesis Components

In addition to the standard UnrealScript and native code objects provided in the UT server, the MUTS consists of four main Mimesis components, implemented as instances of UnrealScript classes and integrated directly into the UT run-time environment, running essentially as independent threads within the standard UT server configuration. In contrast, the MC uses one central process, spawning special-purpose intelligent modules as needed to respond to conditions arising in the virtual world. The various Mimesis components, shown in the diagram in Figure 1, are described below.

Mimesis Controllers are currently implemented in Lisp and run on dedicated high-end machines distinct from the system on which a MUTS executes. Each MC is composed of its own socket-based communications module (for connection to remote MUTSes) , a central controller process and a number of individual intelligent support modules, spawned as needed by the MC during the creation and monitoring of an unfolding Mimesis storyline. For instance, one critical module used by the controller is the narrative planner, a variant of the Longbow planner originally developed for natural language discourse generation. Further, the controller creates a world map of the game environment's virtual space, a knowledge base of information about the current state of the environment, etc. The modular design of the controller allows the straightforward integration of new modules can the system's requirements are expanded.

When a MUTS server is started, it contacts an MC via a socket connection, and a handshaking protocol ensues in which the MUTS describes the current (pre-game) state of its virtual world. At this time The MC generates a plan-based storyline for the action that it expect to drive in the game environment. As gameplay begins on the MUTS, the MC sends commands to the MUTS environment to drive the state changes dictated by the storyline actions present in the plan that it has created. These commands may include directives to control character action and communication, commands that manipulate the state of the virtual world directly (i.e., without the use of characters to manipulate the world) or meta-level messages that send information to one or more of the MUTS modules.

The first (and simplest) MUTS-side Mimesis component is the communications module, named the *AILink*. The AILink module handles all communication with the MC, routing messages to the MUTS through its socket-based connection and distributing incoming messages from the MC to the appropriate MUTS modules.

When the MC sends commands for action to the MUTS, these commands are routed by the AILink to an execution module called the *Funcaller*. The Funcaller is responsible for parsing the commands --- arriving via the socket connection as text *strings* --- and translating them into UnrealScript function calls (i.e., determining the function to call and invoking it using correct references to the data objects named in command string's parameters). To manage this translation process, the Funcaller uses a pre-defined hash table to link argument string names to references to the appropriate run-time data objects. Once all object references are resolved, an automatically generated dispatcher function is called, the function dispatches based on the action's command name string and calls the appropriate UnrealScript code with the correct arguments to UT data objects.

One complication in the management of action execution on the MUTS is the mismatch between the representation of primitive actions by the MC's planning system as occurring instantaneously and the procedural nature of the code that runs on the MUTS to implement the corresponding primitive actions. Because users may alter the state of the world during the execution of a primitive action's procedural implementation, it is necessary to monitor each individual system-driven action to make

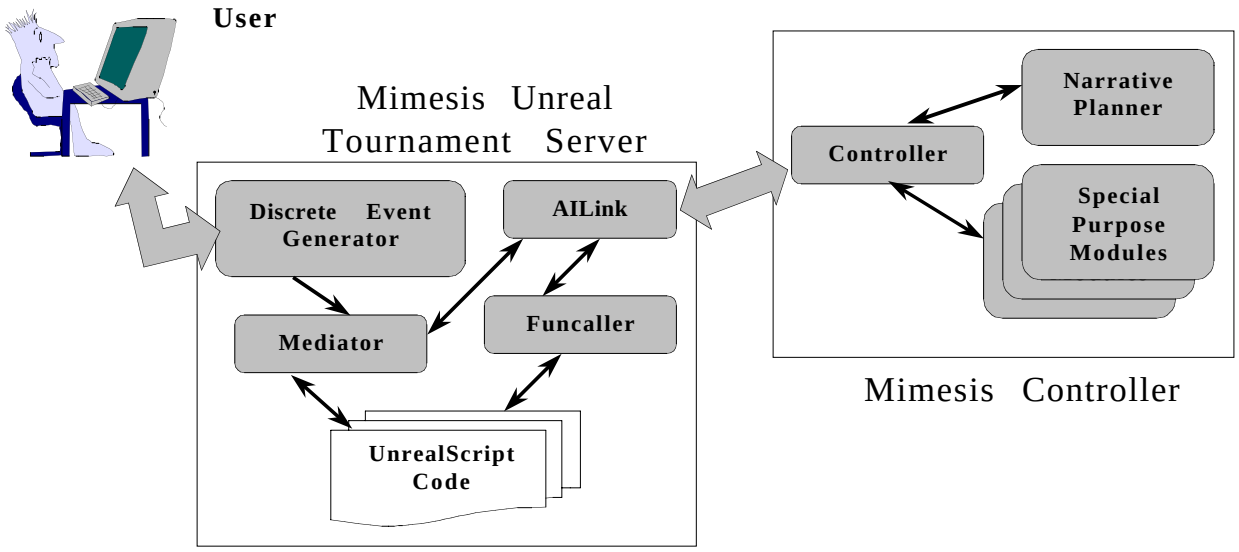


Figure 1. Mimesis Architecture

certain that user-initiated state changes do not cause the action to fail (e.g., to detect when a user closes a door in the face of a character executing the walk-through-door procedure).

Because typical planning-based representations for action (such as Longbow's) assume that primitive actions happen instantaneously, the semantics of their actions' preconditions are intended to describe the required state of the environment immediately prior to the action's execution. The Longbow extension used by Mimesis distinguishes between preconditions of this type and *persistent preconditions*, conditions in the world that must hold from the beginning of the action execution through to the point of its completion.

The UnrealScript procedures that implement each primitive action are responsible for spawning sentinel threads, processes that monitor the actions persistent preconditions during the life span of the procedure's execution. When a sentinel thread detects a change in the state of a monitored variable, it reports the change to the MC, indicating that the procedural implementation of the currently executing action may have failed. Monitoring these conditions via MUTS modules is necessary because many conditions such as the collision between character and door in the example mentioned above is handled by UT at a low-level, never signaling an additional exception.

One central research question for the Liquid narrative group is how to detect and respond to situations where a user performs an action that seriously conflicts with the storyline that the MC narrative planner has created. While the Mimesis system can create compelling action sequences for a UT world and then issue commands that drive the UT world according to the intended storyline, users in the environment may attempt to make changes to the world that can substantively affect the executability of the plan behind the action sequence.

To address this issue, we exploit the structure inherent in narrative plans that we create when mediating between the user's commands and the actual execution of those commands in the virtual world. To manage potential conflicts between a user's actions and the MC's storyline, we mediate between the commands issued by the user and their execution by the underlying UT environment. This mediation is a two-stage process. First, a *Discrete Event Generator* maps relevant input from the user's continuous event stream (mouse motion and keyboard events, for instance) into discrete action commands matching up with the MC's declarative representation of the actions available in the world.

.

Next, we compare each action that the user attempts to execute to the plan structure of the storyline that the MC has created, looking for possible conflicts. The plans we create contain a rich causal structure: all causal relationships between steps in the plans are specifically marked by data structures called causal links. To ensure that the plans are functionally correct, these links are originally added to a plan at construction time. We put them to use at execution time; in our system, when a user attempts to perform an action. The declarative representation of that action is checked against the causal links present in the plan. If the successful completion of the user's action poses a threat to any of the causal links, an *exception* is raised.

Exceptions are dealt with in one of two ways. The most straightforward is via *intervention*. Because all of a user's actions in the environment pass through the mediator prior to execution, it is Mimesis itself that determines whether an action succeeds or fails. Typically, the success or failure of an action within a virtual environment is determined by software that approximates the rules of the underlying story 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 narrative plan's constraints, Mimesis can intervene, causing the action to fail. In the reactor example, this might be achieved by surreptitiously substituting an alternate action 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 storyworld's reactor system.

The second response to an exception is to adjust the narrative 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, for instance, selecting a different but compatible location for an event when the user takes an unexpected turn down a new path. Alternatively, this may involve more substantive changes to the plan, for instance, should a user stumble upon the key to a mystery early in a narrative or unintentionally destroy a device required to rescue a narrative's central character.

Clearly, detecting exceptions is a time-critical task, particularly when intervention is a potential response. Fortunately, there are several techniques that we employ in Mimesis to make exception detection more straightforward. First, the Mediator holds a cache of causal links that are currently relevant to the unfolding plan (i.e., all conditions in the current world state that future steps in the plan are dependent upon). As the storyline unfolds, the MC adds and deletes elements in this cache as appropriate, so that exceptions can be quickly detected by comparing the effects of a user's current action to the elements of this cache.

Second, responses to potential exceptions are pre-computed as time permits, allowing the Mediator to select from a list of responses to an exception without having to wait for the MC to generate an appropriate response. Responses are pre-computed based on a dynamically updated list of actions that a user can potentially carry out at the current moment. At any given time, the game's world state will be such that a subset of all possible actions will have all their preconditions satisfied; these actions are precisely the set of actions that are currently available to the user for immediate execution. At times when the MC's processing load is low, it considers each of the actions in this list in turn, determining the appropriate response to potential exceptions raised by the action's execution by the user.

## Current Implementation Status and Near-Term Extensions

The Mimesis system is still under development, however, substantial parts of the implementation have been completed. We are currently able to develop plans for interaction using the MC, send commands for activity based on those plans to the Funcaller and have the appropriate UnrealScript procedure calls executed. However, execution-monitoring is not yet complete, and the Mediator's functionality is still only partially implemented. Nevertheless, we expect overall system integration will be completed by the time of this workshop.

Currently, our group is also working on additional MUTS and MC modules. For example, we have completed a translator module that takes as input Longbow primitive action operators and generates UnrealScript class files containing procedure stubs for each action procedure that will need to be implemented for a given domain. The module, named *LBUS* (LongBow-UnrealScript) is used to generate a first-pass automatic class definition; designers are then responsible for filling out the stub functions to implement the low-level behaviors for the various actions in a given world.

Mimesis project members are also implementing a virtual camera controller that will integrate pre-designed idioms for shot composition taken from film with discourse-level direction included in the storyline plans generated by the MC. This work is described in more detail in a separate submission to this workshop.

Much of the Mimesis design has intentionally been limited to either the MC or the server-side Unreal Tournament environment. In this manner, no specialized UT client modifications are necessary to connect to and participate in a Mimesis world. However, client-side extensions (called *mods* by the gaming community) are also easily created for UT, and we have experimented with the addition of text-to-speech capability in UT clients by creating a mod that makes system-level calls to Microsoft's test-to-speech API. The mod allows our system to generate custom text to communicate with each user, then to speak the text as a character or narrator within the Mimesis storyline.

## Acknowledgements

The work of the Liquid Narrative group has been supported by a number of sources, including a Faculty

Research and Development grant from the NC State Office of the Provost, and equipment grants and gifts from the NC State Department of Computer Science and Microsoft Research's University Grants Program. The Mimesis system could not have been designed or constructed without the constant help of the student developers that have contributed many hours learning by doing. And doing. And re-doing.