SimHazard: an agent-world exception simulator

by

David Shue

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

June 4, 1999

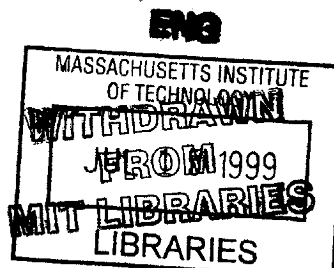
© Copyright 1999 David Shue. All rights reserved.

The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author ______________________________
Department of Electrical Engineering and Computer Science
June 4, 1999

Certified by ______________________________
Chrysanthos Dellarocas
Douglas Drane Career Development Assistant Professor of Management
Thesis Supervisor

Accepted by ______________________________
Arthur C. Smith
Chairman, Department Committee on Graduate Theses


ENG
MASSACHUSETTS INSTITUTE
OF TECHNOLOGY
WITHDRAWN
FROM
1999
MIT LIBRARIES
LIBRARIES

SimHazard: an agent-world exception simulator

by
David Shue

Submitted to the
Department of Electrical Engineering and Computer Science

June 4, 1999

in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Real-world multi-agent systems exist in dynamic, uncertain environments. Achieving robust behavior in such complex conditions often requires equally complex solutions. Traditional approaches focus on imbuing each agent with the ability to respond any exception conditions that may occur. Dellarocas and Klein [14] suggest an alternative solution to this survivalist approach: a global-shared service exception-handler. Acting as a "social institution" in the environment, the exception-handler polices the system, monitoring agent interaction and intervening when exceptions arise. Assessing the validity of the hypothesis requires a suitable test-bed environment that can inject instability in a controlled, robust manner. Direct prototyping of such a system would ascertain the most accurate results. However acquiring the results may prove frustrating at best, since the environment itself is destabilized. Although analytical modeling affords the most control, the dynamical nature of the system would render the equations intractable. Inhabiting the middle ground between these approaches, simulation offers operation under repeatable, controllable conditions, while maintaining the dynamic, stochastic nature of the target environment. This thesis presents a design for the agent-world exception simulator: SimHazard. SimHazard addresses the key issues of exception event scripting, simulation log generation, and agent-neutral design with an extensible, modular design employing a variety of object technologies and frameworks.

Thesis Supervisor: Chrysanthos Dellarocas
Title: Douglas Drane Career Development Assistant Professor of Management

Acknowledgements

I would like to acknowledge the following people for their support, guidance and overall camaraderie in helping me complete this work. Many thanks go to Professor Chris Dellarocas and Mark Klein for their mentorship in guiding me through the sometimes arduous task of design, development, and documentation, and for granting me the privilege of working on the project. A special thanks to Lijin Aryananda for spending endless hours experimenting with the system, and having immeasurable patience. My heartfelt gratitude extend to my friends who are as brothers and sisters to me in my church and fellowship, without whom I would be far more the worse for wear. To Shiu Au, Will Chen, Dave Chen, my fellow MEng buddy and roommate Richard Perng, Emily Liu and Janet Liu, more thanks than I can muster for their care and concern. A gift I would give, if I could find what might fully express my gratitude and love to my parents, who have provided this opportunity and have been my constant support. Finally, I give this thesis, and all the toil, heartache, and ultimately triumph to the One who makes all things possible, and deserves all glory, God Almighty. It is by grace alone that this journey comes to an end.

ACKNOWLEDGEMENTS **3**

CHAPTER 1: INTRODUCTION **7**

1.1 MOTIVATION AND GOAL 7
1.2 THESIS OUTLINE 9

CHAPTER 2: CONTRIBUTIONS OF SIMHAZARD **9**

2.1 SIMULATION 10
2.2 RELATED WORK 12
2.3 CONTRIBUTIONS OF SIMHAZARD 13

CHAPTER 3: DESIGN OVERVIEW **15**

3.1 DESIGN CRITERIA 15
3.2 DESIGN REQUIREMENTS 16
Automated event generation/scripting *16*
Session logging *17*
Functional user interface *17*
Robust Architecture-neutral Agent-Environment interface *17*
Flexible/extensible component model framework *17*
Environment modeling language *17*
3.3 DESIGN OVERVIEW 18
3.3.1 Simulation Issues *18*
3.3.2 Basic System Structure *20*
3.3.3 Technologies Employed *22*

CHAPTER 4: ARCHITECTURE **23**

4.1 SIMULATION MODULES 23
4.1.1 Simulation Manager *23*
4.1.2 Engine *26*
4.1.3 Model Manager *31*
4.2 MODEL FRAMEWORK 35
4.2.1 Model Interfaces *35*
4.2.2 Network Models *40*
4.2.3 Agent Adapters *43*
4.2.4 Framework Events *46*
4.3 SIMULATION INTERFACE 48
4.3.1 SimHazard UI: *48*
4.3.3 Palette *50*
4.3.4 Design Pane *50*
4.3.5 Property Editor *51*

CHAPTER 5: DESIGN EVALUATION **52**

5.1 SIMULATION PARADIGM EVALUATION 52
5.2 MODULE DESIGN EVALUATION 53

5.2.1 Engine *53*
5.2.2 Model Manager *54*
5.3 INTERFACE EVALUATION 55
5.4 FRAMEWORK EVALUATION 55
5.4.1 Object frameworks *55*
5.4.2 Event frameworks *56*
5.5 EVALUATION OF PRELIMINARY RESULTS 56
CHAPTER 6: CONCLUSIONS AND FUTURE WORK 58
MODULES: 59
FRAMEWORK: 60
INTERFACE 61
APPENDIX A: USER'S GUIDE 63
CREATING A SIMULATION 63
RUNNING A SIMULATION 65
APPENDIX B: SIMULATION MANAGER API 67
APPENDIX C: MODEL MANAGER API 69
APPENDIX D: ENGINE API 71
APPENDIX E: MODEL DEFINITION FILE DTD 72
APPENDIX F: SIMULATION PARAMETER FILE DTD 74
REFERENCES 75

List of Figures

Figure 1: SimHazard Module Dependency Diagram.. 20

Figure 2: Simulation Manager module interaction diagram 24

Figure 3: Simulation Manager state transition diagram... 25

Figure 4: Engine-model event-execution cycle 27

Figure 5: Engine state transition diagram 27

Figure 6: Agent-engine interaction diagram 29

Figure 7: Model Manager structure.. 31

Figure 8: Model Manager model life-cycle 33

Figure 9: Model Interface definition 36

Figure 10: Fallible interface definition... 37

Figure 11: Viewable interface definition.. 38

Figure 12: Textifiable interface definition 39

Figure 13: Network Model object hierarchy 40

Figure 14: Network Model event interaction diagram 41

Figure 15: AgentAdapter object hierarchy.. 45

Figure 16: SimHazard UI 49

Figure 17: Design Palette... 50

Figure 20. Effect of a "social monitoring" institution on the completion delay of supply chains where at least one subcontractor agent unexpectedly fails. 58

Chapter 1: Introduction

1.1 Motivation and Goal

Current computing environments can be hazardous environs for autonomous agents. The high degree of heterogeneity and unpredictability inherent in modern computing systems serve only to destabilize agent behavior. Robust operation in such uncertain settings is difficult at best. Add in the convolution of multi-agent communication, collaboration, and coordination and the system reaches the boiling point, ready to roil over in a foaming rage of system failures. Agents may crash, go dormant, break down or disappear without notice. Communications links may fail, sending messages into cyber-oblivion. Unforeseen inter-dependencies can lead to deadlock or resource starvation. In a system of inter-operating agents, these errors are no longer confined to the operations of a single entity, but propagate throughout the system. Left unresolved, they can wreak havoc on the system: clogged networks, inefficient resource allocation, poor performance, system shutdowns, and security vulnerabilities [14].

Most standard approaches to the multi-agent coordination problem have focused on infusing existing coordination protocols with exception-handling logic to deal with non-ideal situations [5] [19] [21]. Several problems plague this approach, however. With the inclusion of fault tolerant protocols comes the onus of increased complexity and size in both agent and message implementation. Consequently, development becomes an even more tedious and time-consuming process that is prone to human-error. Moreover, the inherent distributed nature of the protocols, though nicely complementing the distributed

paradigm of agent oriented programming, lacks a global view necessary to deal with systemic errors that involve collections of agents. To address these concerns, Dellarocas and Klein have suggested a global shared-service exception handling mechanism to act as an external watchdog for the system. Agents using the system need only implement some diagnostic interfaces, primarily for communication and monitoring. The system, an agent-based entity itself, would then use these interfaces in conjunction with a knowledge-base of exception handling strategies to detect, diagnose and resolve exceptions.

To investigate the feasibility and viability of such an approach requires a suitable test-bed that offers both realism and control. Although direct experimentation would give the most convincing results, attempting to induce exceptions in real systems would introduce too much volatility making it prohibitively difficult to test and develop. Purely theoretical models lack sufficient detail to provide the practical test results obtainable through empirical study. Moreover, theoretical models are notoriously ill-equipped to handle complex systems with dynamic interactions. Given the already intricate behaviors of multi-agent systems, injecting further instability and uncertainty would only lead to utterly intractable equations. The best option would be to construct a simulation that models the agent-world and its intrinsic unpredictable nature, i.e. exceptions. Simulation provides a stable environment in which to observe the effects of instability in the system and determine how it might be curbed. In this thesis, I intend to design and construct SimHazard , an agent-world simulator capable of generating environmental exceptions. SimHazard will provide a test bed for the exception handler to study the effectiveness of

the global shared-service exception handler as compared to traditional protocol-enhancement based methodologies.

1.2 Thesis outline

The thesis is subdivided into the following sections. Chapter 2 lays down the goal of SimHazard and its contribution to the field. The next two chapters delve into the actual design, starting with a high-level view in Chapter 3, and culminating in a full architectural design discussion in Chapter 4. Chapter 5 brings a critical eye to evaluate the design in its achievement of the goals set forth in Chapter 2. Lastly, the thesis concludes in Chapter 6 with a summary of the work done, a few closing remarks, and a view towards future work. The appendix includes a users guide and several major API listings.

Chapter 2: Contributions of SimHazard

Prior to discussing the contributions and the overall goals of SimHazard, this Chapter lays the foundation for the present work being conducted. The section builds upon the rudiments of simulation, followed by an examination of current work regarding agent-exception simulation. Once the context has been set, the chapter concludes with a discussion of the goals and contributions of the SimHazard system.

2.1 Simulation

Digital simulation is a modeling process where a dynamic reality is imitated by a computational process [9]. Haggett and Chorley describe models to be:

> A model is a simplified structuring of reality, which presents supposedly significant features or relationships in a generalized form. Models are highly subjective approximations in that they do not include all associated observations or measurements, but as such they are valuable in obscuring incidental detail and in allowing fundamental aspects of reality to appear. [12]

Unlike purely analytical models, simulation models distill real objects into well-defined components embodying the attributes and behaviors critical to operation, rather than reducing everything to a set of abstract equations. The key is to simplify while maintaining enough detail to generate tenable results. Heavyweight models that incorporate every iota of detail fall prey to the same fiends of complexity that preclude direct experimentation. Over simplified systems diverge from real system behavior, thereby obviating their validity. It is this tenuous balance that often renders model design more of an art than a science. In the end, the hope is to create a set of computational components that can model real system behavior accurately while remaining computationally tractable, controllable, and testable.

To extract the behavioral insight out of the simulation models, simulations often incorporate stochastic processes to generate environmental conditions. By adding

variability to simulated attributes and values, these processes inject non-determinism into the simulation to mimic the behavior of natural processes. In this way statistical models of system behavior can emerge from the morass of collated data generated by multiple runs. However, after all the data has been recorded and tabulated, if the discrepancy between the simulated and real systems exceeds the tolerable bounds, the results are essentially useless. For this reason, adequate validation is essential: running the simulation against a set of known situation/behavior pairs to ensure that the system models reality. Although such testing cannot guarantee perfect extrapolation, it serves as a model of believability, which gives credence to observations extracted from the simulation. [2]

At the heart simulation is the simulation clock. This device drives the forward progress of time, causing events to fire and actions to occur. Time is a continuous entity; computers are digital machines. Thus, the simulation clock must employ some method to discretize time. The isochronous approach opts to evenly divide time into uniform segments δt, i.e. days, minutes, milliseconds, etc. This scheme requires δt to be equal to or less than the duration of the shortest event, similar to digital clock chips. Multiple events occurring during the course of a single time slice fire simultaneously at the next clock edge. A major disadvantage of this approach is the potential for periods of dormancy when the system waits for a long activity to complete. This can lead to rather glaring inefficiencies, especially when a large discrepancy exists between the granularity of δt and the duration of crucial events.

Alternatively, the asynchronous approach focuses on state transitions rather than temporal transitions. By advancing time only when the simulation state changes, i.e. a new input arrives, a producer services a consumer, etc, the asynchronous, or Discrete-Event Simuation (DES) approach, as it is commonly known, avoids large periods of inactivity. When an event fires, the targeted model accepts the input event and processes the state transition, generating output events as result. These events then enter the simulation event list, time advances and the next scheduled event fires. One drawback with DES is the particular difficulty of implementing a wait-until mechanism, which checks for conditions within a certain time interval. Since it is not a state change, the mechanism can only make observations when the event fires, potentially skipping over the intended interval. [2]

2.2 Related work

Vincent R., Horling B., Wagner T., and Lesser V. [22] have constructed a Multi-Agent Survivability Simulator (MASS) to test and predict the performance of various coordination protocol level exception-handling mechanisms for detecting, reacting, and adapting to adverse conditions. MASS routes agent messages, synchronizes agent activity, and establishes a world model. Following a isochronous simulation scheme, the simulator synchronizes the agents using a time pulse mechanism. An agent manager receives the pulse and converts it into processing time, allowing agents to run for a specified duration corresponding to the granularity of the pulse. When the run period expires, the agent manager returns a pulse acknowledgement to the simulator. After sending a pulse, the simulator waits on the acknowledgements before proceeding to the next time tick. Allowing the agent managers to determine the pulse-to-time conversion

gives the system the ability to vary agent running speeds, effectively simulating slow hosts, long processing times, etc. MASS interposes itself between the agents, acting as a communications wrapper for routing messages. In this way, MASS can simulate high bandwidth connections using near instantaneous message delivery. To simulate bottlenecked links the simulator can incur message delays, or even message drops. MASS uses a quality/cost/duration--tuple to model the effect of actions in the world, rather than represent the entities directly. These values exist in the system, associated with various actions. When an agent decides upon a course of action, it uses its own subjective, potentially flawed, view of the world as a decision foundation. To actuate the command, the simulator obtains the true quality/cost/duration values from its objective view. The discrepancies that may arise from the difference in an agent's subjective view and the real objective view injects the necessary uncertainty into the system for testing protocol exception handling capabilities. Although a novel approach to simulating exception scenarios, MASS lacks a robust world model, relies on an inefficient simulation paradigm, and suffers from dependence on specific multi-agent coordination mechanisms and protocols that preclude the testing of agents employing different protocols.

2.3 Contributions of SimHazard

The aim of SimHazard is to provide a multi-agent simulation test-bed capable of introducing exceptions in a controlled manner. Running experiments on such a test-bed will provide insight into predicting the performance and effects of using a global shared-service exception handler vs. the traditional distributed protocol exception-handling.

Unlike [22], SimHazard adopts the Discrete Event Simulation (DES) approach to handle the interactions of a simulated agent environment. As described in section 2.1, the fundamental paradigm of DES revolves around an event-driven model: input events trigger state transitions and generate output events. The DES models follow a nearly object-oriented paradigm of encapsulating state as attributes and event-handling behavior as operations. For the most part, SimHazard follows a traditional approach to DES as taken by [15] [7]. The contributions SimHazard makes mostly deal with the areas in which SimHazard deviates from tradition.

While [15] [7] are industrial-strength general-purpose modeling languages/simulation environments, SimHazard focuses on the specialized domain of agent simulation. Actually, SimHazard is really a hybrid simulator. Although SimHazard models the agent-environment, the agents remain fully functional entities. By abstracting the agents from the simulation, SimHazard can test the behavior of real-world agents in its controlled environment. This approach significantly increases the utility of SimHazard by expanding the potential user base and reducing the development and accuracy costs of analyzing and creating agent models.

Most simulators concentrate on modeling performance or other general system behaviors under various conditions. However, SimHazard is a specialized simulator designed to model the effects of exceptions on agent coordination in the presence of different exception handling schemes. Modeling raw system performance is of ancillary concern.

The design focuses on establishing a framework for exception events and injecting instability rather than on modeling performance attributes.

The last major contribution of SimHazard bears resemblance to the approaches taken by [15]. Both of these simulators rely on an object-hierarchy framework for defining models. Users reap the benefits of such a framework in the efficacy and efficiency of creating new models through subclassing. SimHazard attempts to provide such modeling flexibility through the use of its own set of model frameworks.

To summarize, SimHazard stands among the few agent-oriented simulations geared towards exception handling. Its primary contributions include agent-pluggability, exception modeling, and extensible modeling through object frameworks.

Chapter 3: Design Overview

Starting at a high level with the design criteria and requirements, this chapter shapes the direction of the SimHazard design. After the principles are set, the chapter begins the descent into the bowels of the design with a view of the general system structure.

3.1 Design Criteria

As software design is an iterative process, SimHazard keeps its eye on the future by emphasizing flexibility and extensibility in its architecture. In turn, a flexible, open

architecture allows developers to capitalize on a general framework/infrastructure for modifying and adding functionality. Efficiency is an ancillary concern, as optimizations and performance issues can be left to later refinements. Finally, the system must be easy to use. As the basic purpose of the system is to acquire preliminary results within a limited time frame, a steep learning curve would be a prohibitive restriction.

Due to the early stage of the exception-handler research and development, several assumptions underlie the design of SimHazard. First, in the simulated realm agents must behave as single-task entities. Although the agent may queue any number of messages, only one processing task can run at any given time. Secondly, as agent interactions and coordination are the only activities of true interest in the simulation, the simulated environment focuses solely on the routing and transport of messages. Lastly, as a measure of simplicity and incremental design network resources, such as printers, i/o devices, etc. are excluded from this iteration of SimHazard.

3.2 Design Requirements

Automated event generation/scripting

the simulator must provide a mechanism for explicitly scripting exception scenarios in conjunction with the implicit generation of random anomalous events.

Session logging

to extract useful data about environmental parameters over the course of simulation runs, the simulator should enable the logging of simulation attributes and user-definable metrics.

Functional user interface

both functionality and ease of use are critical to the utility of the user interface. It must be intuitive in presentation as well as powerful and flexible in manipulation.

Robust Architecture-neutral Agent-Environment interface

the system must provide a standard API for agents to access services and process messages in the simulated environment. To avoid imposing prohibitive constraints on agent developers, the system should adopt an agent-architecture neutral interface scheme.

Flexible/extensible component model framework

to maximize the simulator's modeling acumen, the model hierarchy should be based on an easily extensible object framework that defines both entity relationships and design patterns.

Environment modeling language

Providing a well-defined textual file format for constructing agent environments confers flexibility in the system. The specification format should be human-readable and standardized.

3.3 Design Overview

3.3.1 Simulation Issues

Due to the nature of the simulation, in conjunction with the underlying assumptions in the system, several simulation related design issues arise. As a time-driven DES simulation, SimHazard requires that the execution of simulation events adhere to a strict time-ordered sequence. Such ordering is enforced by a linear serialization of events according to timestamp. However, when events with identical timestamps occur in the queue, the ordering issue becomes muddled. The proper way to execute these events would be to invoke parallel execution. Since SimHazard is a single-process simulation, the simplest solution is to handle the events in order of arrival, regardless of source. Say event A occuring at $t = 5$ triggers event C at $t = 10$ and event B occuring at $t=6$ triggers event D also with timestamp $t = 10$. Event C will have ordering precedence over event D because event A occurs prior to B, hence C enters the queue before to D. Although such ordering imposes a deterministic ordering on the inherently parallel nature of concurrency in the simulation, *sans* parallel computing, it is a simple and tractable solution.

Following from the assumption that agents are essentially single-task entities, the simulator must supplant the agent's internal message queue with an external simulated one. By queuing messages in the simulator, the simulator can ensure that an agent will be processing only one task at a time. Since most agent architectures have the capacity to handle multiple messages and tasks concurrently, imposing a single-task restriction on the agents would only bring undue requirements on the developer, violating the some of

the design goals of the system. Broadening the message queuing notion to encompass general simulation models, model multitasking adds a level of concurrency control for all models. This method works by tagging each model with an attribute indicating the maximum number of concurrent tasks it can support. Once this number is exceeded, the simulator queues subsequent events in a model-specific event queue for later scheduling after the model finishes its prior tasks. This policy enforces task serialization in accordance with the agents' abilities.

One final difficulty stems from the actuation of hazards that alter the outcome or cancel the effect of an event. Again, in the DES approach, simulation events indicate the start of a process in a model. When the event fires, the model performs the tasks specified by the event. Although from start to finish, event handling occurs in real time, simulated time remains fixed at the start of the event. All consequent output events are scheduled in the queue as *future* events. Until the original event actually completes in simulated time, the model remains in a processing state. Should a hazard event fire in the interim, and either change the output events or cancel the action altogether, the previously generated outputs events are now invalid. To maintain system consistency, the simulator must purge the queue of these invalidated events.

3.3.2 Basic System Structure

The simulation core is comprised of three object modules: the Simulation Manager (SM), the Model Manager (MM), and the Engine. Extending over the core is the User Interface (UI). Beneath the simulation core is the modeling infrastructure, the AgentAdapter and NewtorkModel object frameworks, the Model Interface framework, and the Event Framework.

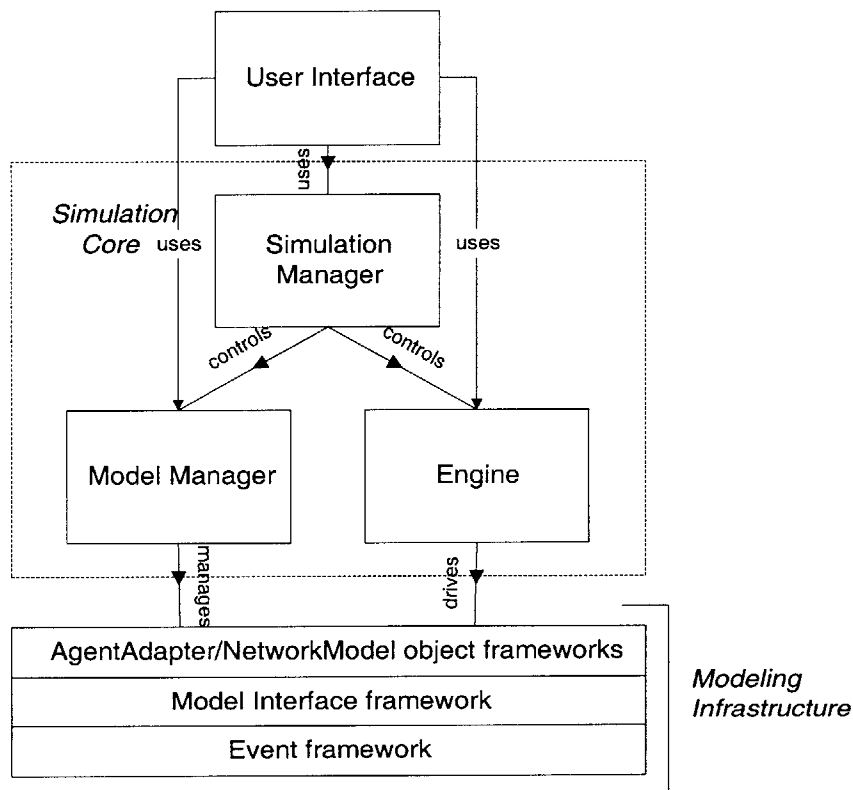


Figure 1: SimHazard Module Dependency Diagram

As shown in Figure 1, the Simulation Manager sits atop the simulation core, directing the simulation runs by defining the simulation parameters and manipulating the Engine. During the design phase, the SM remains dormant, with only its parameters exposed for modification. Here, the Model Manager takes center stage, managing the entire model

life-cycle, from instantiation to initialization and editing, to finalization and deletion/removal. Once the model is in place, and the run is ready, the Engine takes over and drives the simulation. As the simulation churns, the SM controls the simulation run by manipulating the Engine.

Above the simulation layer is the user interface. The UI wraps a shell around the simulation to facilitate access to simulator functionality for both creating and running simulations. To accomplish this, the UI taps into all three core modules.

Underlying the simulation, at the infrastructure level, is the model framework, comprised of two frameworks: NetworkModels and AgentAdapters. These entities encapsulate the basic attributes and behaviors of their "real" counterparts: network elements and software agents. NetworkModels are simulate the agent-world network topology while the AgentAdapters interface with the actual agents, acting as agent-architecture neutral proxies between agents and the simulator. These two object frameworks build off the abstract Model Interface framework and make extensive use of the Event Framework.

The partitioning of functionality into UI, Simulation Core, and Modeling Infrastructure roughly corresponds to the Model View Controller design paradigm inspired by early SmallTalk design patterns, which fosters modular design [4]. In this paradigm, the Modeling Infrastructure represents the model, while the UI acts as the view and the Simulation Core the controller. By harnessing this design pattern, the modules not only follow a logical grouping, but also provide a level of flexibility since each piece of the

MVC puzzle can be removed and replaced with minimal changes to the existing architecture.

3.3.3 Technologies Employed

Language:

Java was chosen as the object-oriented language to implement the system. Although Java performs poorly, it more than compensates for the lackluster speed with robust and cleanly designed libraries and powerful facilities for dynamic class loading and reflection. Moreover, Java's platform transparency and network orientation (network i/o libraries, servlets [10]) provide a spring board for SimHazard to migrate to other platforms or even to web based three-tiered architecture.

Events

Since SimHazard is an event-driven system, events pervade the system. Besides the main DES model, events, in the Java Bean sense [8], signal the occurrence of everything from property changes, to raised exceptions, to state changes, and beyond. By leveraging the Java Bean EventObject/EventListener design pattern, objects can register for events of interest while preserving anonymity behind the listener abstraction.

Frameworks

Object Frameworks form the basis for extending and customizing SimHazard's simulation models. Beyond establishing an object hierarchy, frameworks also specify the procedures for extending the hierarchy, and how the models plug into the system as a whole, from creation, to initialization, to running, to termination [17] [11].

XML

XML, or Extensible Markup Language, is a superset of HTML that supports data markup and specification. It is a textual, human readable format designed to encode most data types in a standard format [3]. SimHazard uses XML extensively as a text-based serialization format for models and simulation parameters.

Chapter 4: Architecture

Herein lies the core of the thesis. This chapter scrounges through the details of the SimHazard system architecture. In order of presentation, each major simulation module is described in turn, followed by the underpinning system frameworks.

4.1 Simulation Modules

4.1.1 Simulation Manager

The Simulation Manager controls the simulation. Much like a micro-controller, the SM takes high level commands and translates them into executive signals for the underlying components. Through its public API, described in Appendix B, the SM exposes methods for controlling simulation runs, defining environment parameters, and managing simulation logs. Internally, the simulation manager defines the state of the system, which constrains system behavior. Figure 2 depicts the Simulation Manager's basic control structure.

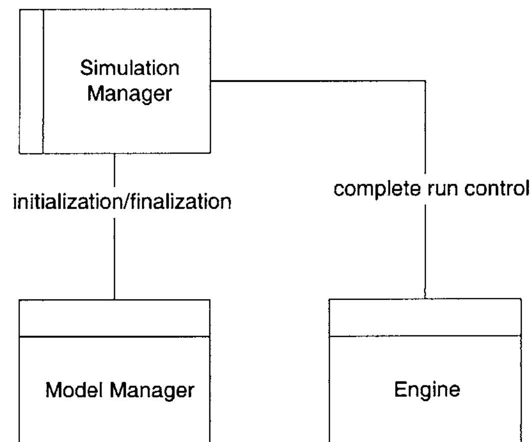


Figure 2: Simulation Manager module interaction diagram

The Simulation Manager API provides a set of basic simulation control methods for directing the progress of a simulation run. These methods initialize, start, stop, pause, resume, and reset simulation runs. Aside from the control methods, the SM exposes the simulation parameters through the beans design pattern of get/set methods. These parameters determine the course and nature of the simulation: run title, start time, duration, simulation seed, parameter file, model definition file. For serialization purposes, the SM saves the parameters in an XML format Simulation Parameter File (SPF). More detailed description of these parameters and the SPF format are found in Appendices B and F.

Over the course of a simulation run, the simulation manager exists in one of four states: IDLE, READY, PAUSED, and RUNNING. The state transitions and constraints are illustrated below in Figure 3.

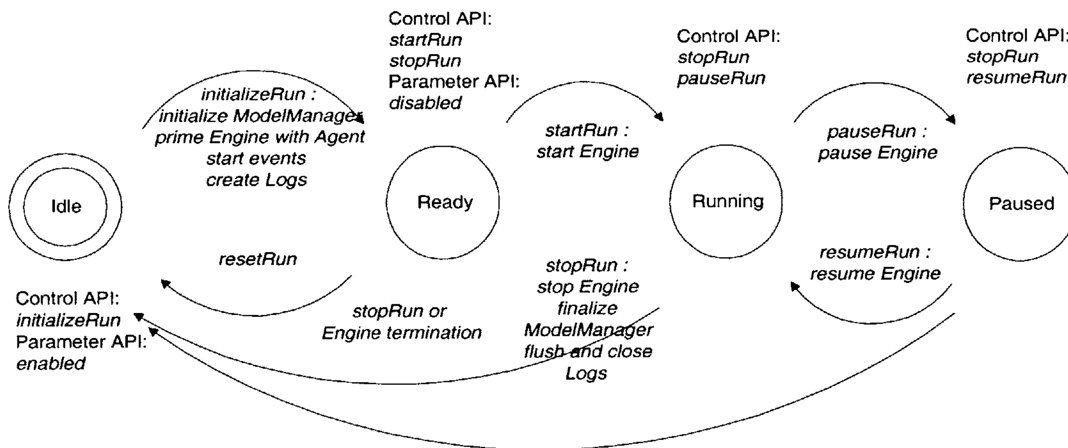


Figure 3: Simulation Manager state transition diagram

During the design phase, the Simulation Manager is in a quiescent IDLE state; the parameter API is fully accessible, while the run control methods are disabled. When the SM initializes a simulation run, it validates the simulation model and transitions to the READY state. Once in the READY state, the only permissible actions are to proceed with the run, or return to the IDLE design state. The system has entered the simulation phase. After a run starts, the SM remains, for the most part, in the RUNNING state until the run terminates. Over the course of a run, the SM can pause and resume any number of times, toggling between the RUNNING and PAUSED states. When the simulation stops, the SM returns to the IDLE state.

For pseudo-random number generator initialization, SimHazard uses a single user specified seed to generate a system wide set of seeds used to initialize the individual models' pseudo-random number generators. Although this method may propagate errors

inherent to the pseudo-random number generator, the effects should be negligible compared to the error that could arise from the actual sampling performed during a run.

As a final design feature, the Simulation Manager manages the simulation log files. During initialization, the Simulation Manager creates the four log files: message log, exception log, parameter log, and simulation log. The SM names the files using the following convention <runTitle>-<logtype>.log. The message log records agent creation and consumption of messages, the exception log tallies exceptions occurrences, the parameter log catalogues fluctuations in model parameters, and the simulation log scribes an exhaustive listing of the events executed in the system. Although the SM manages all the log files, it logs only the Simulation and Exception events. The Model Manager is responsible for logging Parameter and Message events, since these are model specific events.

4.1.2 Engine

The Engine combines the event driver and event queue data structure into one module. As the engine fires events, models respond by generating output events, which are scheduled into the event queue. This is the essence of the engine-model event cycle which drives the simulation. Figure 4 delineates the event cycle.

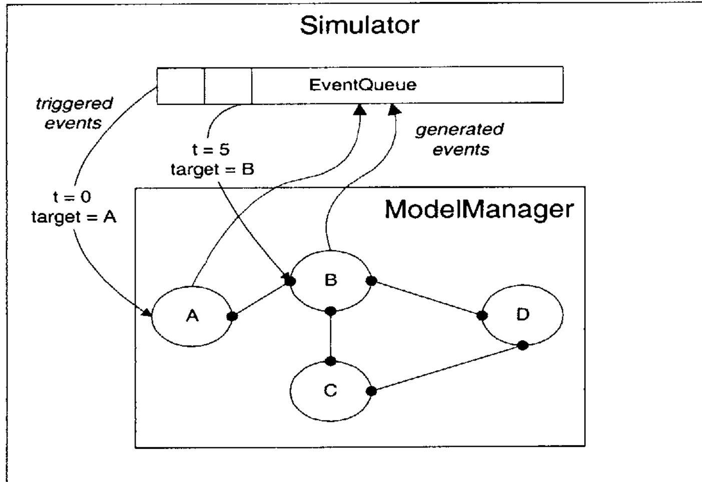


Figure 4: Engine-model event-execution cycle

At its core, the Engine is an event-execution loop. The Engine starts in the inactive IDLE state, where it remains throughout the span of the design phase. When the simulation commences, the Engine enters the RUNNING state, creating and executing a new thread to run the event-execution loop. Like any event-loop, the Engine event-execution loop is basically a large while loop. At the top of the loop, the Engine checks for termination conditions: empty event queue, expired duration, or end-states: IDLE, PAUSED. A PAUSED engine suspends the Engine event-loop thread, waiting on the Engine object until it is resumed—returns to the RUNNING state, or terminated—moves back to the IDLE state. Figure 5 details the transitions.

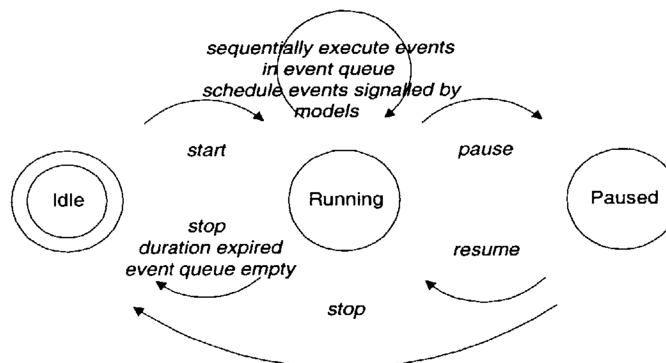


Figure 5: Engine state transition diagram

On each pass through the loop, the Engine pops the earliest event off the queue and dispatches the event to the specified target. For non-threaded models i.e. NetworkModels, the Engine simply invokes model's the execute method with the event as input. AgentAdapters, being autonomous (threaded) models, require special treatment. To preserve the single-event execution invariant, the Engine first synchronizes on the current event. Next, it calls the AgentAdapter's execute method to handle the event. Immediately following the call, the engine waits on the current event lock, sleeping until the AgentAdapter seizes the current event lock and notifies the engine—when the agent pauses or completes processing. Using this threading scheme, the Engine thread maintains a lock-step synchronization with the agent threads, ensuring that only one thread is active at any given time.

Output events generated by a model during event processing reach the Engine via the SimulationListener interface. When the model creates a new event, it notifies all registered SimulationListeners of the action through the eventFired method. Upon receiving the notification, the Engine schedules the new event. To facilitate event scheduling, the SimulationListener interface furnishes the eventFinished method. Models call this method to notify the listeners of process completion. To allow interrupts and other message events to reach agents while a task is running—in simulated time, Agent Adapters have a pause mechanism which wakes the engine without invoking eventFinished. When an agent completes its message handling, it calls the Agent Adapter done method, invoking eventFinished and waking the engine. Figure 6 depicts this process in greater detail.

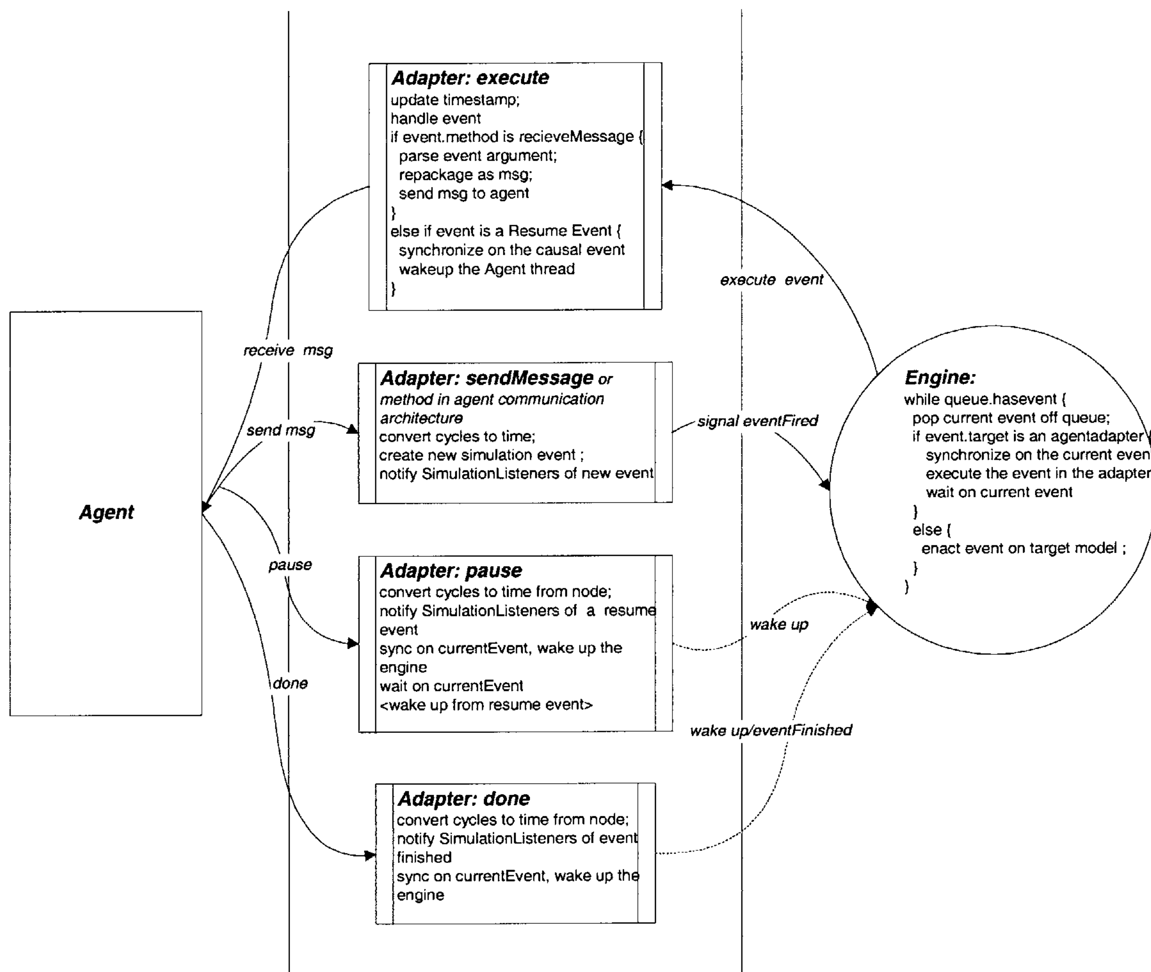


Figure 6: Agent-engine interaction diagram

Scheduling in the Simulation Engine relies on the innate ordering imposed by the event queue data structure, which is basically a red-black tree that allows multiple key-value pairs with the same key [18]. By maintaining the insertion order of key-value pairs with the same key, the event queue preserves the "order of arrival" scheduling of concurrent events. However, the need for per-model event queuing introduces some additional complexity into the scheduling process. Say an output event A emerges from a model

process with timestamp t=5. Assume the target of the event A is model M. If the number of pending events in the event queue designated for M, M_{pe} reaches the concurrent task capacity C_{cap} for M, the Engine pushes the event into the pending event queue for M, M_{peq}. When the next event for M fires, M_{pe} decrements, and a slot opens in the event queue. Upon receiving the eventFinished signal from the model, the Engine then resets extracts the earliest event in M_{peq}, resets its timestamp to the donetime of the completed event, and transfers the event to the main event queue. In this way, the Engine guarantees that no more than C_{cap} events will ever be scheduled for a model at any given moment.

Event cancellation occurs when an actuated hazard causes a model to terminate or alter its current event processing. As a result, all output events generated by the halted event become invalid. To ensure Engine consistency via purging of the invalid events, the model notifies SimulationListeners of event cancellation through the eventCanceled method. Two types of cancellation exist: purge or cancel. Purge occurs when a model dies and can no longer process any events. As a result, all events targeted at the model must be purged from the engine, as they are no longer consistent with the model state. Cancel requires the Engine to expunge all output events caused by the canceled event. The Engine achieves this by examining the event queue and model queues. If an event should have the cancelled event as a cause, the Engine removes the invalid event. For a full Engine API listing, consult Appendix C.

4.1.3 Model Manager

The Model Manager is the central simulation model repository. It manages the entire simulation model life-cycle, from creation to deletion. The only proper way to create models is through the Model Manager API, listed in Appendix D. This centralization hides the complexities of dynamic model creation behind an abstraction barrier for simplicity. Throughout the design phase, models can be created, removed or modified. Once the system transitions to the simulation phase, the Model Manager initializes its constituent models, which includes creating logging classes for the models to record events of interest. Over the course of a run, the MM maintains model consistency as AgentAdapters terminate and NetworkModels fail. At the end of a simulation session the Model Manager performs finalization operations to clean up the models. Additionally, the Model Manager provides methods to save and load the models from a Model Definition File. The general Model Manager structure is shown below in Figure 7.

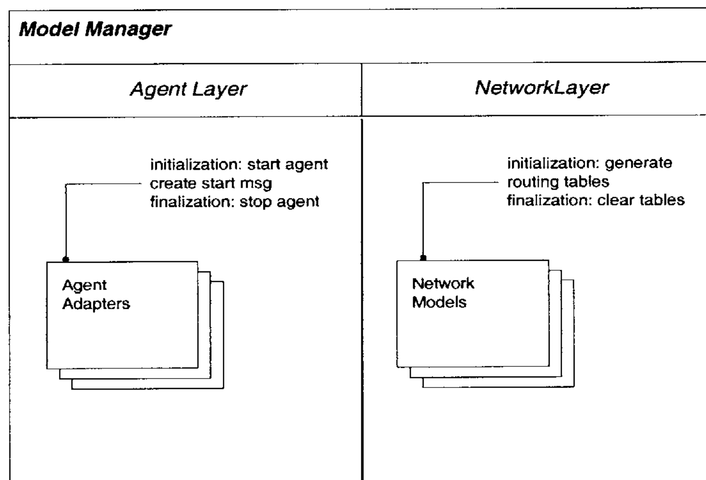


Figure 7: Model Manager structure

Model creation in the Model Manager relies heavily on Java's reflection and dynamic class loading capabilities. To create a model, the Model Manager uses the model's model Factory method. Every simulation model implements a static Factory method with the following signature <type> create<type>( ) for creating new instances of the model. Although ordinary dynamic instantiation using the model's constructor would work equally well for practical purposes, using the Factory method wrapper abstracts away the specifics of the constructor call. In this way, changes to the constructor or its arguments do not propagate to the Model Manager. Moreover, the abstraction simplifies construction by eliminating the specific knowledge needed to create a new Model. During initialization, the Model Manager uses the Classes utility class to load the models into the system. From these subclasses of the Model interface, the Model Manager extracts and stores the model Factory methods. This dynamic class loading mechanism, enables the addition of new models into the system at runtime, without the need to recompile. After creating a model, the Model Manager generates a unique model id and label delimiter for the model. Figure 8 lays out the model life-cycle in the Model Manager.

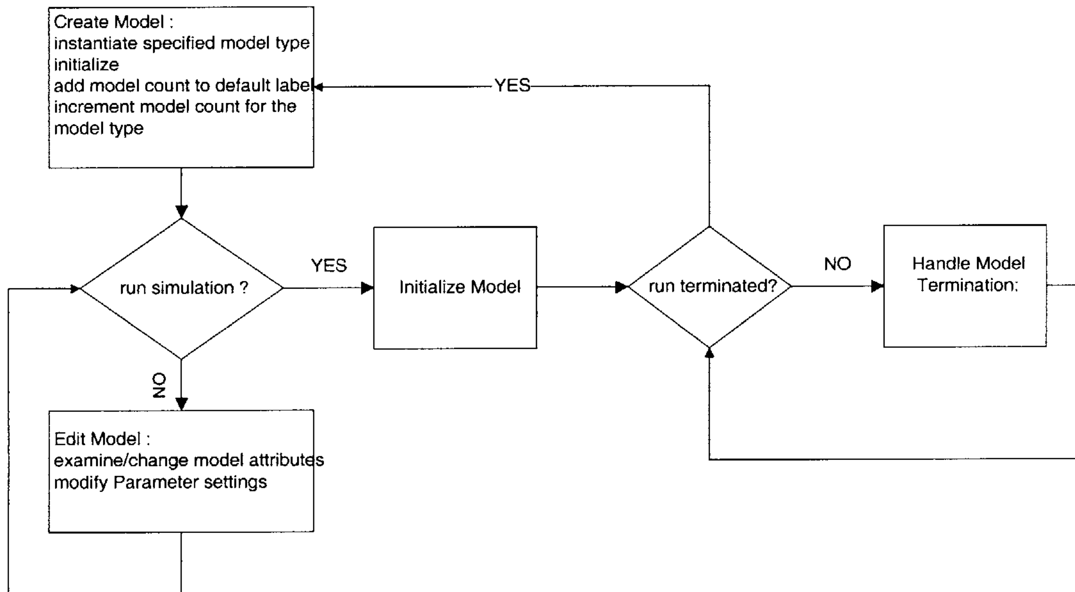


Figure 8: Model Manager model life-cycle

For model initialization and finalization, the model manager employs a framework based on the LayerManager interface. The methods defined by the LayerManager interface provide a means for performing type-specific initialization. Each LayerManager is associated with a specific root model type. AgentLayer handles AgentAdapter models and NetworkLayer handles NetworkModels. When the Model Manager initializes or finalizes a model, it checks the root model type of the model and delegates the task to the associated LayerManager.

At initialization, the AgentLayer creates a MessageLogger for each agent and starts the agent thread. MessageLoggers record the occurrence of messageSent and messageReceived events in an AgentAdapter. A different MessageLogger class exists for each agent message type used in the simulation. The text format of the log is set using a LogFormat object. Much like the Model Factory methods, these classes are loaded and

created dynamically by the Model Manager as well. When an agent dies, its associated MessageLogger is removed from the system. At finalization, the AgentLayer kills any non-terminated agents, and removes the associated MessageLoggers.

NetworkModel initialization revolves around generating a global routing table. Currently, the NetworkLayer employs the Floyd-Warshall all-points shortest path algorithm [6] to initialize node routers. To compute the algorithm, the NetworkLayer records the nodes and vertices of the network topology as NetworkModels are added and removed during the design phase. When the system is ready to initiate a simulation run, the Model Manager initializes the NetworkLayer, triggering the computation of the global routing table, and uses it to initialize the routing table of RoutingService models. One limitation to the current routing scheme is the lack of dynamic rerouting in the event of node or link failure. In addition to routing initialization, the NetworkLayer also creates a parameter Monitor for each NetworkModel. The Monitor logs parameter changes for each NetworkModel. Upon model termination or finalization, the NetworkLayer removes the Monitors from service.

The Model Definition File (MDF) contains the complete XML specification of the simulation models. Each model is responsible for serializing and loading its state from the XML parse tree. The full MDF format is descibed in Appendix E.

4.2 Model Framework

4.2.1 Model Interfaces

The Model Framework is a set of interfaces designed to abstract simulation behavior into functional sets. Comprising this framework are the following interfaces: Model, ProbabilisticModel, PendingModel, Host, Resident, Fallible, Viewable, Textifiable, and ModelResolution. As described in section 4.1.2, the Engine drives the simulation by executing an event on the specified target model. This single execute method defines the essence of the model's DES characterization. Beyond implementing the event execution method, a simulation model needs only a few additional support methods and attributes to define its simulation identity. The base Model interface defines these methods, while the other model interfaces indicate more specialized model types. The remaining interfaces provide system support. By implementing these interfaces, objects of any kind can serve as simulation models. This high level of polymorphic abstraction allows the simulation to handle almost any form or implementation of model.

Model
execute(event : SimulationEvent) : void *getConcurrentTasks() : int* *getLabel() : String* *getLastTime() : long* *getModelId() : long* *initialize() : void* *isMultitasking() : boolean* *reset() : void* *setLabel(label : String) : void* *setModelId(mid : long) : void* *shutdown() : void* *addSimulationListener(sl : SimulationListener) : void* *removeSimulationListener(sl : SimulationListener) : void*

Figure 9: Model Interface definition

The most important method in the Model interface is the execute method. This is the basic event input method that tells the model to execute an event. On the attribute side, there are two critical identification attributes a model must implement: model id and model label. The model id is a long integer unique to the model in the scope of the simulation, which the system uses to identify the model. The label is a human readable string for visual identification. The model must also specify whether it is multitasking and how many concurrent tasks it can support. Additionally, the Model interface specifies several support methods for creation and finalization. When a model is created, initialize is called to set the model's attributes to initial values. Reset clears the model's non-identification attributes. This is called prior to setting the model's attributes to default values or for finalization to return the models to design time settings. Shutdown is called when a model terminates during a run to perform necessary dereferencing clean up e.g. removing dead agents from nodes. Additionally, Model has methods for adding and removing SimulationListeners, e.g. the Engine.

Probabilistic Model extends the Model interface with methods to get/set simulation seeds and updateParameters for probabilistic modeling. PendingModel defines a set of support methods for models that need to track events running the model to calculate load. Host indicates that a model is capable of housing residents. Models implementing the Resident interface specify whether they are dependent on the host for operation. Dependent models rely on hosts for time calculations e.g. agent residents expending processor cycles that the host node converts into execution time.

Fallible
actuateHazard(hevt : HazardEvent) : void *clearHazards() : void* *getHazardTypes() : Enumeration* *hazards() : Enumeration* *insertHazard(hevt : HazardEvent) : void* *removeHazard(hevt : HazardEvent) : void* *addExceptionListener(el : ExceptionListener) : void* *removeExceptionListener(el : ExceptionListener) : void*

Figure 10: Fallible interface definition

Fallible specifies methods necessary for implementing exceptions. Exceptions in the system are divided into two categories: implicit and explicit. Implicit exceptions arise from probability distributions associated with Model parameters, i.e. processor load, network congestion, misrouting, etc. Explicit exceptions are deterministically scheduled hazard events. Fallible provides methods for inserting, viewing, removing, and clearing hazard events. Most importantly, actuateHazard enacts the effects of the hazard event on the Model. By making these exceptions explicit, discrete exceptions can be pre-determined to occur, infusing the system with deliberate instability. Additionally,

getHazardTypes allows a model to specify the types of hazards that it supports, and objects interested in ExceptionEvents can be registered and removed as well.

Viewable
getParameter(label : String) : Parameter *parameters() : Enumeration* *setParameter(label : String, param : Parameter) : void* *addPropertyChangeListener(pcl : PropertyChangeListener) : void* *removePropertyChangeListener(pcl : PropertyChangeListener) : void*

Figure 11: Viewable interface definition

All probabilistic Models use psuedo-random univariate number generators as the basis for emulating stochastic behavior. This randomized behavior, in turn, simulates fluctuations in performance and gives rise to implicit exceptions. Randomized attributes are embodied in the Parameter class. Each Parameter specifies a value and distribution description which includes distribution type and distribution parameters. Using the Distribution utility class, which implements a collection of univariate distributions in [16], models can sample parameter values from a variety probability distributions. The Viewable interface provides a window onto these Parameters and enables Monitors to survey Parameter changes via the PropertyChangeListener interface.

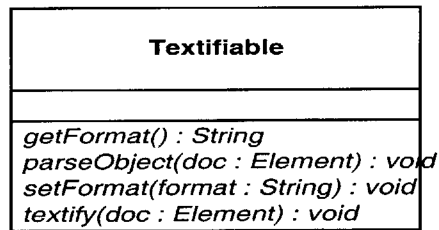


Figure 12: Textifiable interface definition

To support saving and loading in XML, the Textifiable interface provides two essential methods: parseObject and textify. For SimHazard simulation models, saving to an XML MDF file through the textify method entails using VarTextify, a utility wrapper class around the IBM XML4J DOM/XML implementation. Loading entails calling parseObject and uses VarParser, another utility wrapper class. A more detailed description of the textification process is given below in the Model Extension subsection.

4.2.2 Network Models

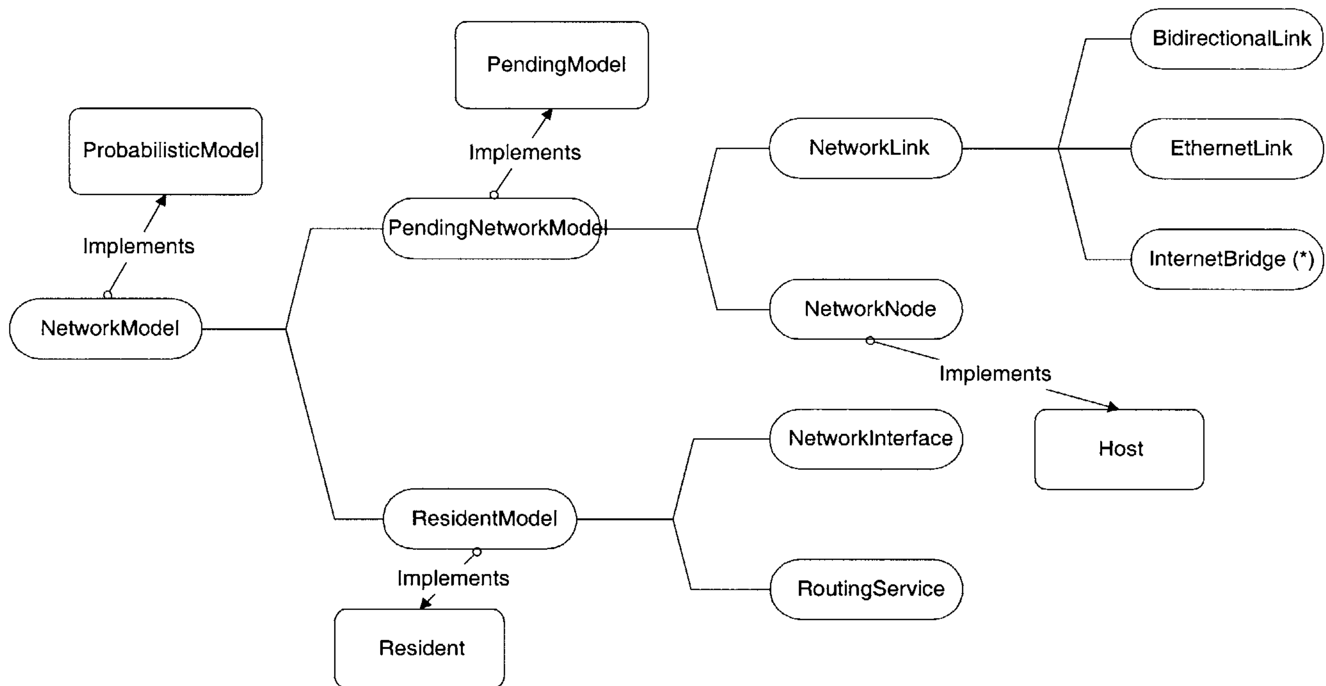


Figure 13: Network Model object hierarchy

As a set of foundation classes, the simhazard Model Hierarchy provides the core Models for simulating a network topology. At the base of the hierarchy is the NetworkModel, which implements the basic Model interfaces and provides convenience methods for initialization, hazard triggering, and sampling. Directly extending the NetworkModel are PendingNetworkModel and ResidentModel. PendingNetworkModel implements the PendingModel interface, and ResidentModel the Resident interface. Past these base models, NetworkNode and NetworkLink represent nodes and links in the network, and NetworkInterface and RoutingService embody the NIC and Routing Table of a network protocol stack. Figure 14 displays the function of these models and their simulated interaction.

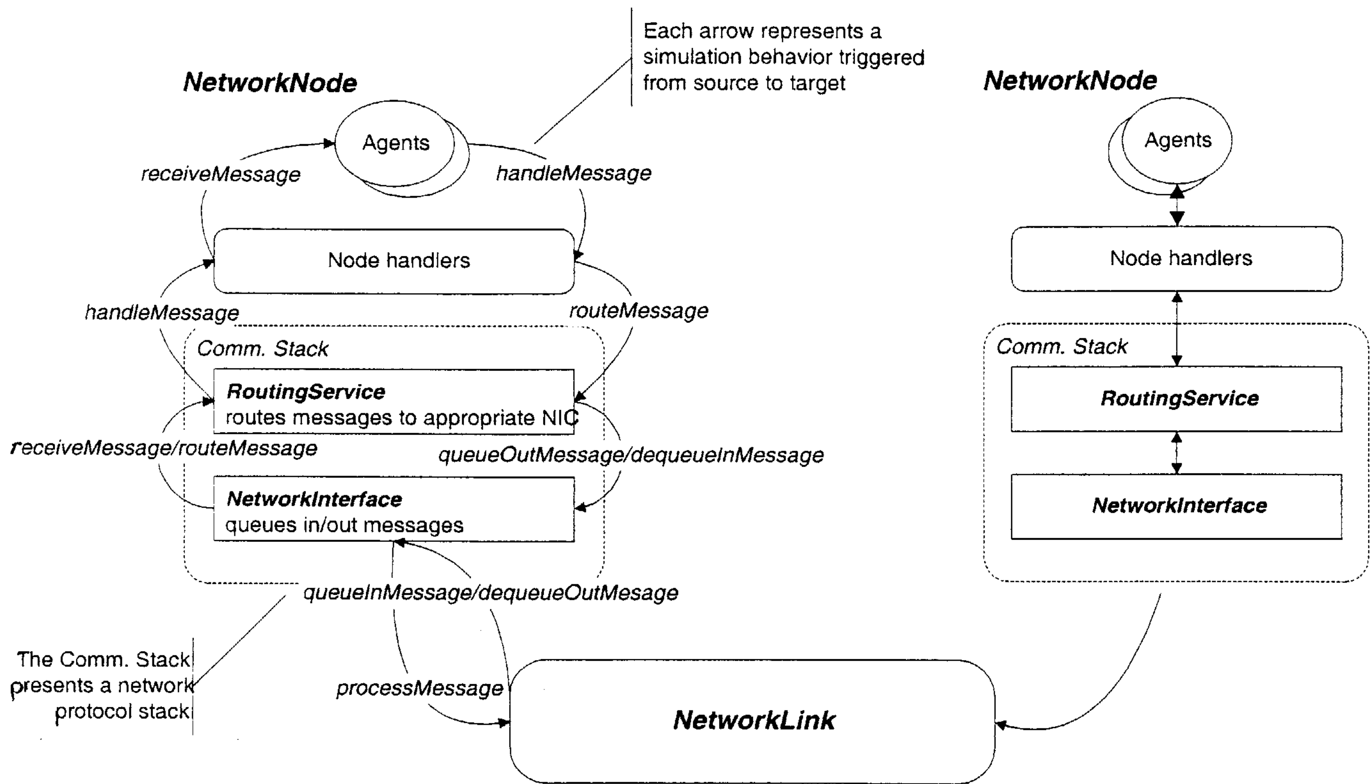


Figure 14: Network Model event interaction diagram

At the present moment, only the basic set of network elements are implemented, but there remains ample space for extension from any of the basic classes: Workstation, Server node specializations, InternetBridge, ATMRelay as BroadBand links, various NIC's to support them, and RoutingServices to support dynamic or cell-based routing schemes. To extend the Model Framework, there are several design patterns to adhere to, and a few issues to keep in mind. At the very least, any new model in the framework must implement initialize, reset, shutdown, textify, selectVars, and specifyVars methods, if it contains any state variables of its own. These new subclassed versions should follow the structure used in the base class implementations. Initialize, reset, and shutdown follow the creation-finalization routines described in section 4.1.3. Textify writes the model's state to XML format. SelectVars and specifyVars are methods to aid in XML loading.

To prevent rescanning of the entire XML DOM parse tree at every class in the hierarchy chain, the utility class VarParser enforces a parse ordering. First, all the desired XML variables must be specified. Then the variables are "parsed" from the tree. Finally the variables can be extracted to obtain the serialized values. In NetworkModel, parseObject calls specifyVars to allow the entire class chain to specify the desired variables. It invokes the VarParser's parse method and calls selectVars for the class chain to extract the values. All of these subclassed methods must call the superclass versions as well to ensure completion of the corresponding process.

Models that will handle new simulation events must override the dispatch method. This method dispatches the simulation event to the appropriate handler based on method. Dispatch is used as the subclassed method instead of execute because execute performs event pre-processing and post-processing operations which preclude method chaining. Along with dispatch, the new Model must also provide the actual event handlers. These methods will perform the actual work, and add the new simulated behavior. Any events generated by the handler methods are sent to the SimulationListeners using the fireEventFired method. Exception events can be reported with the fireExceptionTriggered method. Following the handler pattern, new handlers should be protected methods. Reducing handler method visibility ensures that execute remains the only entry point for event handling.

To add new hazards or extend old ones, the new model need only override actuateHazard, the hazard dispatch method, and implement new hazard handlers. ActuateHazard should

fire the appropriate ExceptionEvent when a given hazard occurs. Following a design pattern much like the simulation event handlers, hazard handlers must be declared as protected methods. The only additional aspect is adding new hazard types in the initialization method.

4.2.3 Agent Adapters

Building off the Model Framework interfaces, the AgentAdapter Framework establishes an architecture neutral API interface for agent-simulator interaction. Essentially a proxy for communication between agents and the simulator, each AgentAdapter uses architecture-specific code to integrate an agent system into the simulation. Basically, an AgentAdapter implements the Adapter interface for communication with the agent on one end, and translates agent requests into simulation events on the other. Another Adapter interface, the SystemAdapter exists to provide additional functionality to privileged agents, such as the ExceptionHandler. In this way, different agent systems can be used in the simulation without significantly modifying either the agents or the simulator, but by merely extending the AgentAdapter Framework. This extensibility is crucial to augmenting the viability of the simulator and its capabilities to run simulations containing a heterogeneous mix of agents.

The Adapter specifies the following set of methods for agent-simulator interaction.

> `void sendMessage(long cycles, Object msg)`: translate agent communication into simulator events for message sending. The msg is handled in an architecture specific way. This method may be ignored in favor of an existing agent-communications interface specified by the architecture.
>
> `pause(long cycles)`: yield to the simulator so that other pending messages and interrupts can be handled

timeout(long timestamp): notify the agent when a certain amount of time has elapsed.

die(long cycles): notify the AgentAdapter that the agent is dead and should be cleaned up/shutdown.

done(long cycles): signals message handling completion

long getLastTime(): returns the agent host's local time stamp

These methods provide a standard way for agents to interact with the simulator. All actions requiring actual simulated time include a cycles field. Cycles represent the number of processor cycles that the agent has expended since it first started processing its current message. The AgentAdapter computes actual simulation timestamps. During message processing, agents can send any number of messages and set timeouts. However, agents should periodically call the pause method to relinquish control to the simulator. This ensures that high-priority messages are received in a timely fashion. Otherwise, the agent will have completed its task by the time the high-priority message arrives, which may cause problems if the message was meant to terminate the agents activity. When the agent is finished computing, it calls done to signal event completion, transferring control back to the simulator.

The SystemAdapter interface is for use by uber-agents only. It provides a set of network monitoring methods for determining network performance. Currently only the exception handler can use this interface for diagnostic purposes.

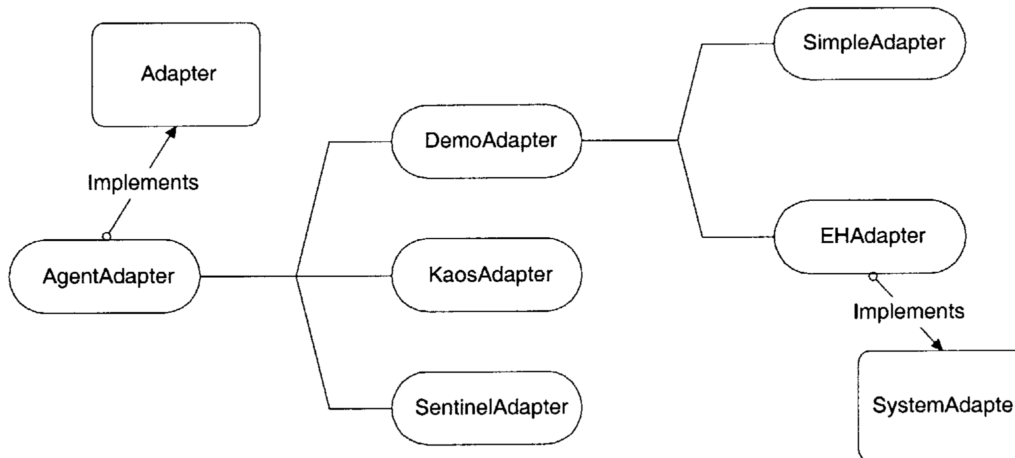


Figure 15: AgentAdapter object hierarchy

Similar to the NetworkModel base class in the model Framework, the AgentAdapter class furnishes implementations of the Model Framework interfaces, making the adapter full-fledged models as well. Furthermore, AgentAdapter implements all the Adapter interface methods except for sendMessage, which is architecture specific. For interfacing with the simulator, the AgentAdapter provides a skeletal implementation of the receiveMessage event handler, which sends incoming messages to the agents, the wakeUp event handler, which notifies agents of timer expiration, and the terminate hazard handler, which kills the agents.

Subclasses tailor the AgentAdapters to the specific architectures by implementing sendMessage or an architecture specific communication interface, receiveMessage, wakeUp and terminate. By using the existing agent-architecture communications infrastructures, agents can communicatedwith the simulator as if it were another agent. Additionally, the AgentAdapters must also implement agent specific start methods for creating and initializing the agents. The AgentAdapter should expose all agent attributes that need to be modified using the get/set design pattern. For simulation initialization

purposes, the agentadapters must implement createStartMessage which is used by the Simulation Manager to prime the Engine. Although agents are created at simulation initialization, they can commence activity only upon receiving the start messages. Again, much like the extension of NetworkModels, AgentAdapters with new attributes are responsible for initialization and textification.

4.2.4 Framework Events

As seen in the various parts of the SimHazard design, Java EventObjects, as specified in the Beans architecture [8], are central inter-object communication. These seemingly ubiquitous objects define events of interest in the system, from simulation events to hazard events to exception events. The beans EventObject-EventListener architecture provides a means to register call-back functions anonymously, hidden behind an abstraction barrier. In the spirit of beans design, this design promotes software reuse, while complimenting the event-driven nature of the simulation.

Simulation Events are the bread and butter of the simulation. These events specify the model execution parameters: source, target, method, arguments, timestamp and priority, and the engine firing parameters: timestamp, donetime, and cause. The model dispatches the event to the appropriate handler indicated by the method with the given arguments. Donetime indicates when the event finishes, and is used for rescheduling (see section 4.1.2) and event cancellation (see section 4.1.2). Additionally, Simulation Events implement a done method that deferences completed events in the causal chain. This prevents arbitrarily long event chains from existing in the system and wasting memory. In large, long simulations, such waste may prove prohibitive.

ResumeEvents extend simulation events, adding the resume method. This event is posted by AgentAdapters when an agent pauses. The resume method toggles an internal flag to indicate that the agent should resume when it is woken up from by Engine.

Hazard Events implement explicit hazards. Currently they only include a type, start time, and end time. These are simple hazards that have, minimally, a timestamp and duration of effect.

PropertyChangeEvents are defined by in the java.bean.event package. This event signals a change in an object's attribute, or property, state. In SimHazard, these events are used extensively in its original form to update listeners when a model's state changes, including changes in Parameters and message sending/receiving, for logging purposes.

EngineEvents notify the listener when the engine state changes. These events ensure that listeners, such as the Simulation Manager and UI know when the engine has actually changed state, since it runs in its own thread. Moreover, the EngineEvents are also triggered whenever a simulation event is fired in the Engine, for logging purposes.

ExceptionEvents occur when models trigger hazards either explicitly or implicitly. These events are used solely to log these anomalous occurences.

ModelStateEvents indicate when the state of the ModelManager changes during a simulation run. These events occur only when a new simulation entity occurs in the system.

4.3 Simulation Interface

4.3.1 SimHazard UI:

Aside from the raw simulator functionality, there is a distinct need for a user interface that brings the full utility of system to bear while maintaining ease of use. The aim of the SimHazard UI (SUI) is to engender a design/run environment that is both intuitive and powerful. To this end, the SUI borrows from the visual metaphor found in many visual builder applications like IBM's Visual Age for Java and Microsoft's Visual Basic [13]. These development environments enable the construction of programs from visual component, with minimal hand coding. Similarly, the SUI provides a graphical means of constructing simulations without having to manually compose the Model Definition File. The SUI encapsulates the View and parts of the Controller in an MVC architecture. Any number of views can be used to interface with the simulator, though currently there is only one. The basic GUI widgets/components are derived from the Swing library. Figure 16 is a screenshot of the SimHazard UI.

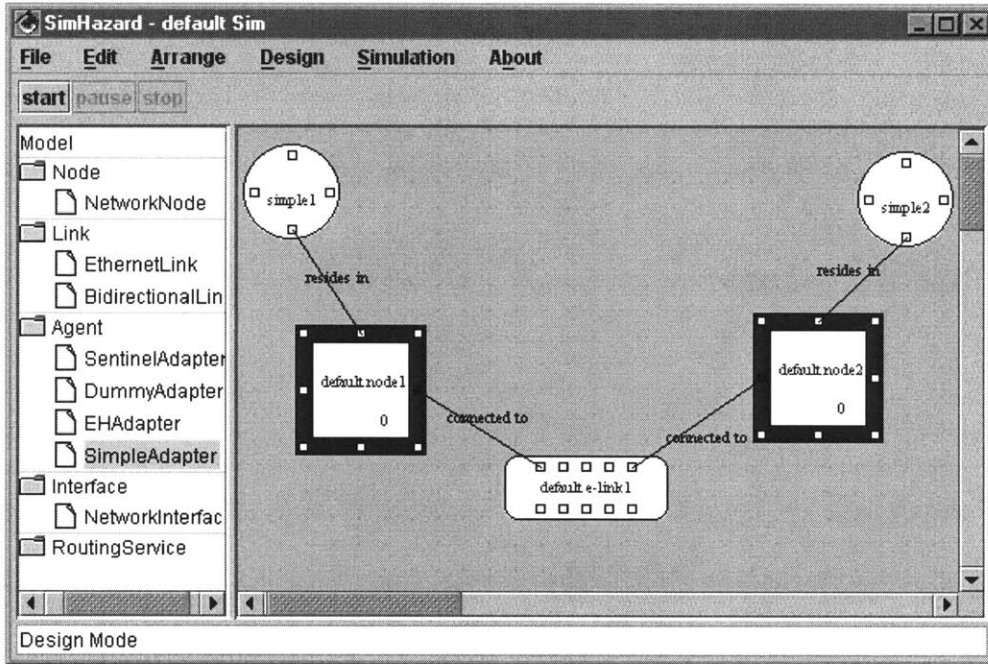


Figure 16: SimHazard UI

The SUI operates in two modes: design mode and simulation mode. Like most visual composers, in design mode, the user can build the simulation by selecting models from the Palette and adding them to the Design Pane. Once a model is created, its properties can be edited using the Property Editor, similar to a beans or Visual Basic control property sheet. Simulation parameters also appear in the Property Editor, under the simulation runtitle, and can be edited as well. The File menu contains options for saving, loading and creating a new simulation. These actions modify the simulation parameters. Under the Design menu, there are additional features to load models from a user-specified Model Definition File and refresh the Palette. Either pressing the start button or selecting start from the simulation menu transitions to simulation mode. In simulation mode, the interface lies dormant, allowing only a static view of the system as it runs. The components, however, do change in response to model termination and pending events.

If specified, the Simulation Progress window appears during simulation mode displaying simulation and exception events as they emerge.

4.3.3 Palette

The Palette acts as a graphical repository for the models in the simulation. When the SUI initializes, it uses the Classes util class to search for the models in the system. These models are then organized in the palette tree according to type. Selecting a model in the palette prepares the SUI for adding the indicated model into the simulation.

Figure 17: Design Palette

4.3.4 Design Pane

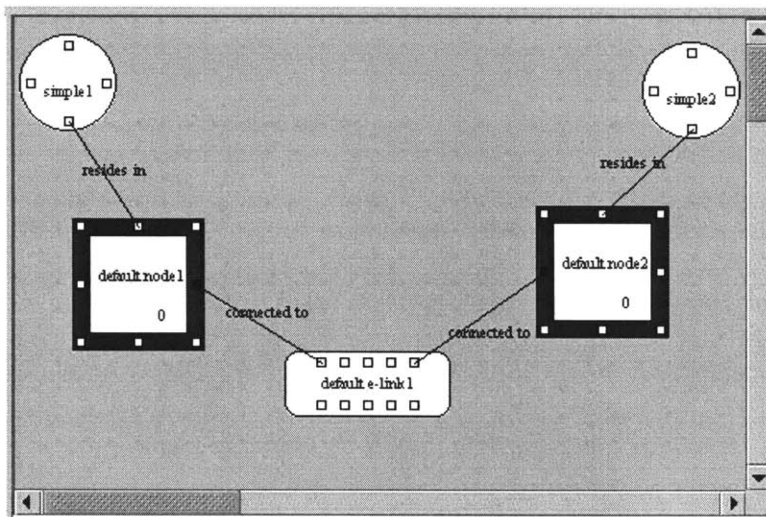


Figure 18: Design Pane

The Design Pane is essentially a display of the models in the simulated topology. The Design Pane is based on GEF, Graph Editing Framework [20]. GEF provides a basic set of graph objects that SimHazard subclasses to construct the visual components. In addition, GEF includes features for selecting, grouping, aligning, distributing, and reordering elements. Models are added by choosing a model from the Palette and clicking on the design pane. Once

the model is created and appears, it can be selected and positioned. Right-clicking on a model brings up a popup menu. The menu contains some generic graph editing features as well as model specific options for editing hazards, message logs, monitors, etc, depending on the model's capabilities. Selecting an editing option spawns an editor dialog for the chosen option.

4.3.5 Property Editor

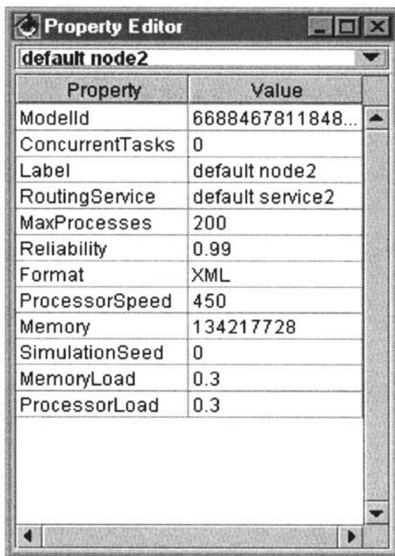


Figure 19: Property Editor

The Property Editor emulates the behavior of component property sheets. By using Java reflection, the editor displays all model attributes that are exposed via get/set methods. Moreover, the Property Editor also displays Parameters for objects/models that implement the Viewable interface. Using this tabular format maximizes information display, as all the attributes can be viewed and edited on the same panel. Double clicking on an editable field engages the editing mode. For most simple values, the user can edit the value in place. However, for more complex data types like Files and Parameters, an editor Dialog is started to handle the editing. Users can select the model to edit by using the combo box list of simulation models, or by selecting the desired model in the Design Pane. To show the Property Editor, users must choose the Edit Properties option in the Design menu.

Chapter 5: Design Evaluation

Having lain down the ironwork, this section proceeds to evaluate the design piece and discuss the decisions and trade-offs made. At the end of the section, a brief summary of preliminary results demonstrates system viability in the context of simple experiment.

5.1 Simulation paradigm evaluation

Using a DES approach instead of the isochronous time-sliced brings benefits on both the performance and design level. As explained in section 2.1, isochronous methods that rely on a fixed time-slice δt may incur heavy performance penalties due to periods of inactivity which may result in useless simulation cycles. The problem is most visible in the case where model behaviors vary greatly in duration. The size of dt is fixed by the duration of the shortest action to disallow multiple actions in a single simulation cycle. Given this restriction, if only long duration actions are running in the simulation, or if the system is waiting for a timer to expire on the order of hundreds of δt, an incredible number of simulation cycles would be wasted as the simulation churns toward the completion of those activities. Since the nature of the simulation mimics this scenario, using time-slicing is prohibitively expensive.

Moreover, using the isochronous approach would require all the models to record the progress of its actions at each cycle. Behaviors with multiple effects would need to be divided into smaller actions, atomizing the structure of the models. Imposing this requirement on agents would incur an unaccepatably high development penalty for modifying the agents to fit the schema. DES has its own set of limitations in its inability

to handle true "interrupts". Once an event fires and finishes processing, the execution state no longer exists. Should an interrupt, such as a hazard, arrive prior to the done time of the event, there is no way to fully reverse or redirect the event processing since it has already completed in real time. For normal models, this does not pose a large problem, as event cancellations remedy the invalidation of results. However, for agents, it can prove to be troublesome, since a high priority message may request certain changes in behavior, not just task termination. SimHazard attempts to circumvent this problem by using the pause mechanism as described in section 4.2.3. For most simulations, this should be a sufficient fix. Thus, for its performance and ease of development, DES still emerges as the paradigm of choice for the SimHazard system.

5.2 Module Design evaluation

5.2.1 Engine

Several issues exist in the design of the Engine. Engine-agent synchronization is necessary to ensure event serialization for agent message handling. However synchronization introduces a model-specific event triggering in the Engine event loop. To avoid being tied to a specific model hierarchy, the Model interface should be extended to include Threaded or AutonomousModel interfaces. Such an interface would notify the engine to use synchronization without binding the process to a specific object hierarchy. A related issue is the use of message/event queues in the Engine. With the restriction that agents can only handle one message at a time, comes the need for agent message/event queuing. However, there is no fundamental constraint on agents that would prevent simulated multi-tasking. Although multitasking could lead to multiple task threads

existing at the same time in the agent, only one will ever be executing. The agent would only need to manage its tasks thread. Lastly, it would probably be more appropriate if Hazards were treated as full-fledged simulation events, rather than special events associated with a given model. This would consolidate the event execution mechanisms and improve modularity.

5.2.2 Model Manager

The intent in designing the Model Manager to be a centralized model "factory" is both to hide the complexities of model construction and to make the Model Manager more bean-like and self contained. Objects interfacing with the Model Manager and the simulation as a whole should not have to deal with internal simulator issues. Moreover, the simulation should have control over what models can be used in the system. The use of the LayerManager interface is part of an attempt to maintain a model neutral design overall. Enforcing a model neutral design means less work when adding new models or model types e.g. physical models (buildings, etc). Using pluggable interfaces like the LayerManager enhances extensibility and allows the Model Manager to be used with a wide array of models for different types of simulations. Coupled with the extensive use of dynamic class loading, it gives the simulator extreme flexibility. Building off this foundation, the simulator could even provide simulation templates to define the type of simulation desired, and switch between templates on the fly.

Logging in the system follows the same extensible design pattern of using pluggable classes and object hierarchies to achieve extensibility. Currently there are only a few

classes in the hierarchy, and the addition of more powerful subclasses for creating user-defined metrics would significantly benefit the system.

5.3 Interface evaluation

As a whole, the User Interface succeeds in providing a useful and intuitive front-end to the simulator. The visual composition metaphor is a relatively familiar for the intended user audience of SimHazard: Agent Researchers and Developers. Leveraging the GEF libraries reduced the design workload by uncountable degrees. However, limitations in the GEF design may predicate the re-design of the Design Pane and visual layout modules. Further adjustments to the interface are needed as well, to improve on the overall utility and look and feel of the system.

5.4 Framework evaluation

5.4.1 Object frameworks

Both the Model and AgentAdapter frameworks stress the use of extensible architectures in their design. Again, this is to allow the system to grow and adapt to the needs of the user and the simulation domain. The extensive use of interfaces to define models and functionality ensures that the system can accommodate most any type of simulation model. Using method overriding, in network models and agent adapters for initialization, finalization, and textification, leverages the method chaining design pattern to simplify framework extension.

5.4.2 Event frameworks

Overall the hazards, along with the probabilistic parameters provide a simple exception scripting mechanism suitable for this initial design. The most interesting exception events in the current system deal with model death and performance issues which are easily implemented by this design. However, more complicated simulations may require more sophisticated approaches to hazard scripting. Using the event-listener design pattern to model many of the module interactions increases the modularity of the design by focusing attention on event-driven behavior: entities act and respond to events generated by other entities. This pattern can be exploited, however, to render the system fully component-based and event-driven.

5.5 Evaluation of preliminary results

To test functionality and garner preliminary results, a simple experiment comparing shared-service exception-handling vs. heavyweight protocol exception-handling performance was devised. The agents themselves are simple single-threaded entities implementing a basic version of the Contract Net protocol. To interface with these agents, the Demo and Simple Adapters were constructed. An exception handler prototype, based on the same agent architecture, provided the shared-service exception handling, while an internal switch activated the agents' heavyweight protocol. As a simplification, SentinelAdapters were created at every node to act as a message rerouting wrapper. Every message sent by an agent was first routed to the exception handler, then sent on its way to the intended recipient. In a more realistic simulation, these SentinelAdapters would embody agents themselves, performing distributed monitoring

tasks and communicating with the exception-handler. High priority exception-handler messages were rendered immune to network and node delays as a simplification as well.

The goal of the experiment was to measure the usefulness of the shared exception handling approach in dealing with subcontractor agents that may crash unexpectedly after having been awarded a task but prior to completing the work. It works by periodically monitoring the "health" of subcontractors and assisting in the immediate reassignment of tasks performed by failed subcontractors. In the absence of this "social monitoring", the heavyweight variant of the contract net protocol used in the experiment only checks for subcontractor death after a task result fails to arrive by a specified deadline. The experiment, itself, involves agent tasks of varying complexity for which a single agent in the contracting chain dies prior to completing its task. Figure 20 depicts the results of the experiment, showing that the exception handling service does reduce the average contract completion delay by a greater factor than the normal heavyweight protocol [1].

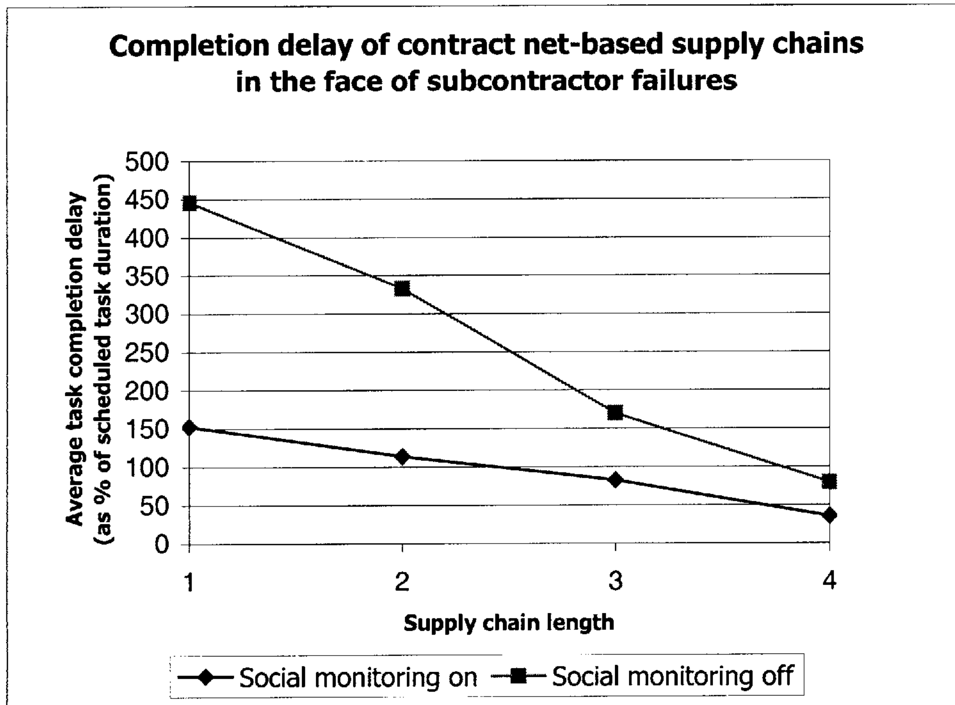


Figure 20. Effect of a "social monitoring" institution on the completion delay of supply chains where at least one subcontractor agent unexpectedly fails.

Chapter 6: Conclusions and Future Work

Designed for ease-of-use and extensibility, SimHazard uses a novel blend of DES simulation and flexible, framework based design to engender an agent-architecture neutral simulation test-bed for observing and extrapolating agent behavior in an exception-riddled environment. Exploiting the generality of Model interface, the simulator core modules can control, drive, and store a variety of simulation types. The NetworkModel and AgentAdapter hierarchies provide an easily extensible object model foundation for creating new simulation elements and incorporating agents of different architectures. By exploiting a familiar development interface metaphor, the SimHazard User Interface exposes simulator functionality in an easily accessible way. Being a first

design implementation, the current system benefits greatly from the extensibility of the SimHazard design. Extending the system to include additional functionality and expand on various aspects of the simulation is simple and requires no changes to the overall system architecture. This insures that changes to one module will not propagate to others. As a final note, throughout the implementation of SimHazard, leveraging of existing software libraries expedited development immeasurably.

There are many areas in which the SimHazard system can be improved and extended. Listed below are several possible additions and modifications along with a brief description for each.

Modules:

Beanify: to support component-based software and augment the utility of SimHazard, realizing the modules as certified Java Beans would allow the system to be easily embedded and integrated in a larger system/environment.

Client/server: As mentioned in section 4.1.1, the Simulation Manager has the potential for moving to a server or servlet design as part of client-server/web-based application. This would greatly increase the viability/accessibility of the system.

Run scripting: One very useful feature would be to incorporate a generic run scripting mechanism in the Simulation Manager. Many of the experiments intended for SimHazard are permutations of a basic simulation template. Providing a way to specify a

template and the variation of parameters would greatly expedite the simulation process and reduce tedious user interaction.

Consolidated logging: Currently the system partitions logging into the Simulation Manager and Model Manager. A more centralized, unified logging pool would streamline the design.

Simulation templates: As mentioned in the Run scripting item, simulation templates would greatly facilitate the simulation design process. These templates would define the basic simulation behaviors that would vary from run to run. The LayerManager used by the Model Manager can form the foundation of such a system since it provides a pluggable means of specifying initialization and finalization behaviors.

Hazard engine: More complicated simulations may require a more sophisticated explicit hazard specification system. A Hazard Engine would provide an automated/scriptable way of incorporating hazards on a behavioral level. Instead of specifying exact hazards at certain times, the scripts would define what kinds of hazards should occur and under what conditions. An inference engine would be a suitable candidate for the task.

Framework:

Network protocol stacks: Currently the protocol stack defined by the NetworkNode is static and rigid. Generalizing the stack to allow an arbitrary number of layers adds the possibilities for different/multiple routing schemes and end-layer wrappers.

Model verification: Conferring to models the ability to perform self-verification would enable the enforcement of different simulation constraints, e.g. all interfaces must be connected to a link, no node self-loops, at least one agent must be present in the system, etc

General resource interface: As mentioned in section 3.1, SimHazard currently has no support for general network resources. Including such elements as I/O devices and other resources would broaden the simulated domain by introduce new simulated exception behaviors such as resource deadlocking and resource poaching.

XML DTD's: DTD, or Document Type Definition, files define the structure of a specific XML format. Creating DTD's for the Model Definition File and the Simulation Parameter File would provide a more rigorous specification of the file formats.

Distributed agents: for scalability, enabling distributed agents would greatly reduce the strain of running large numbers of agents. Currently, every agent runs in the simulator process. Moving to a distributed paradigm would spread the load onto other machines and possibly allow true event parallelism as well.

Interface

Button palette: Most visual builder environments represent components as buttons in the palette. Emulating the design would make the interface even more accessible.

Interface settings: As with most design environments, giving the user the ability to customize system settings like font, color, etc., makes for a less rigid environment, enhancing the user's UI experience.

Simulation indicators: Adding more indicators of simulation progress during a run would improve the interactivity of the interface and provide the user with useful info.

Layout algorithm: Currently, the system does not save the layout of the simulation models. When the models are loaded, they appear in a tangled mess that the user must navigate manually. Providing an automatic layout algorithm would enhance the practical aesthetic of the interface.

Appendix A: User's Guide

Creating a Simulation

The best way to explain how to use the system is through an example. The goal of this example is to demonstrate how to create a simulation, add models, edit parameters, and save the results. To keep things simple, this example contains a small network topology comprised of a few models.

1. *Start SimHazard from the command line*: java edu.mit.ases.simhazard.ui.SimHazard
2. *Edit the simulation parameters*:
   - 2.1. open the Property Editor by selecting the Edit Properties option from the Design Menu. The Property Editor should be displaying the simulation parameters.
   - 2.2. Edit the parameters by double-clicking on the value field.
     - 2.2.1. Set the runtitle to "TestSim", and set the duration to 100000.
     - 2.2.2. For ParameterFile and ModelFile, press the edit button in the field. This brings up a file chooser dialog. Go to the SimFiles directory and specify the filename TestSim.spf for the ParameterFile. For ModelFile specify TestSim.mdf
3. *Create the models*:
   - 3.1. Select NetworkNode under the Node "folder" in the Palette. Add a node to the Design Pane by clicking on the pane.
   - 3.2. Click on the model and edit the parameters in the Property Editor. To keep things simple, just change the name to Node A.

3.3. Create another node and set it to Node B.

3.4. Add an EthernetLink by choosing the corresponding item in the Palette and clicking on the Design Pane. Set the link name to Link 1.

3.5. To connect the nodes to the link, interfaces must be added to the nodes.

3.5.1. Select the Networkinterface Palette item and click on a port in Node A. Ports are the small squares bordering the node. The port should turn black, signifying the presence of an active interface port.

3.5.2. Click on the interface port and drag the line to a link port. This connects Node A to Link 1.

3.5.3. Repeat the process to connect Node B to Link 1.

3.6. To add Agents into the system, select an AgentAdapter from the Palette, under the agent branch. Again, adding the chosen agent requires just a click on the design pane.

3.7. Change the agent's name to Agent A and add it to Node A by clicking on an agent port and dragging to a non-interface node port.

3.8. To complete the sample model set, add one more agent, Agent B, and connect it to Node B.

4. *Save the simulation and exit:*

4.1. select the save option in the File menu, to save the simulation. This saves Simulation Parameter File and Model Definition File as specified through the Property Editor: TestSim.spf and TestSim.mdf.

4.2. To save the SPF under a new name, use the Save As menu item. For the MDF, either change the ModelFile simulation parameter or use the Save Models As item in the Design menu.

4.3. Once the files have been saved, select Exit from the File menu to quit.

Running a Simulation

Continuing from the previous example, this segment explains how to run a simulation.

1. *Start SimHazard*: from the command line: as above

2. *Load the Simulation Parameter File*:

   2.1. choose Open in the File Menu and select TestSim.spf in the File Chooser from the SimFiles directory.

   2.2. To load a different model set, select Load Models in the Design menu to choose a new model file.

3. *Start the simulation*:

   3.1. To start the simulation, press the start button or select start from the Simulation menu. The start options should now be disabled, while the other simulation controls should be online: pause, stop, reset. The Simulation Progress window should also be visible now as the simulation runs.

4. *Pause and Resume the simulation*:

   4.1. As the simulation runs, click on the pause button to suspend the run. Select a model and view its properties in the Property Editor. At this point only Parameters can be edited. Hazards can also be added via the popup menu editor.

   4.2. Press resume to continue the run.

5. *Stop/reset the simulation and exit*: a run can be manually terminated using the stop button. However, this is a short simulation and the run should be over quickly.
   5.1. After the run stops, select reset from the Simulation menu to bring the system back to Design mode.
   5.2. The run is complete, now select exit from the File menu and quit.

These examples cover the rudiments of creating and running a simulation using SimHazard. Alternatively, to design a simulation, the model and simulation parameter files can manually edited. After a simulation run, the log files can be examined and processed. These include TestSim-sim.log, TestSim-exception.log, TestSim-message.log, and TestSim-parameter.log.

Appendix B: Simulation manager API

SimulationManager** implements **ExceptionListener, EngineListener	
`SimulationManager(String runtitle)`	Constructs a SimulationManager with the given runtitle
`void addPropertyChangeListener(PropertyChangeListener pcl)`	Adds a property change listener
`void closeLogs()`	Closes the simulation logs
`void engineStateChange(EngineEvent eevt)`	EngineListener method, handles changes in engine state
`void exceptionTriggered(ExceptionEvent evt)`	ExceptionListener method, logs exceptions
`void flushLogs()`	Flushs the simulation logs
`Engine getEngine()`	Returns the simulation engine
`ModelManager getModel()`	Retuns the model manager
`void initialize()`	Initializes the SimulationManager and its constituent modules
`void initializeRun()`	Initializes a simulation run
`void initLogs(String path)`	Creates and initializes the simulation logs
`void loadModelFile()`	Loads a Model Definition File
`void loadParameterFile()`	Loads a SimulationParameterFile
`void loadRun(String modelfile, String paramfile)`	Loads a saved simulation run (not implemented)
`void pauseRun()`	Pauses a simulation run
`void removePropertyChangeListener(PropertyChangeListener pcl)`	Removes a property change listener
`void reset()`	Resets the Simluation Manager and its modules
`void resetRun()`	Resets a simulation run
`void resumeRun()`	Resumes a paused simulation run
`void saveModelFile()`	Saves the models into a ModelDefinitionFile
`void saveParameterFile()`	Saves the simulation parameters into a SimulationParameterFile
`void saveRun()`	Serializes a simulation run (not implemented)
`void startRun()`	Starts a simulation run
`void stopRun()`	Stops a simulation run

***Simulation Parameters**: for each of these parameters, SimulationManager implements a get/set method pair*	
`Duration`	Duration of the simulation session in microseconds
`StartTime`	Starting time of the simulation session in microseconds
`ModelFile`	Model Definition File handle

`ParameterFile`	Simulation Parameter File handle
`RunTitle`	Simulation RunTitle
`SimulationSeed`	Simuation random number generator seed

Appendix C: Model Manager API

ModelManager implements ***PropertyChangeListener***	
`void addExceptionListenerToModels(ExceptionListener el)`	Convenience method for adding ExceptionListeners to the models
`void addLayerManager(LayerManager layer)`	Adds a LayerManager to the ModelManager init/finalization framework
`void addModelStateListener(ModelStateListener msl)`	Adds a ModelStateListener
`void addSimulationListenerToModels(SimulationListener sl)`	Convenience method for adding SimulationListener to the models
`void clear()`	Clears the models from the ModelManager
`void createMessageLoggerFor(long id)`	Creates a message logger for a particular AgentAdapter
`long createModel(String modeltype)`	Create a model specified by modeltype, returns the new model's id
`long createMonitor(long logmod)`	Creates a standalone Model Monitor
`public void createMonitorFor(long model)`	Create a Monitor for a specific model
`public void finish()`	Finalize the models at the end of a simulation run
`void generateModel(String mdf)`	Generates the model from a Model Definition File
`LayerManager getLayerManager(Class type)`	Returns the LayerManager associated with the baseclass type
`MessageLogger getMessageLogFor(long modelid)`	Returns the MessageLogger associated with the given modelid
`Model getModel(long mid)`	Returns the model with the given modelid.
`long getModelSeed()`	Retrieves the model simulation seed
`Monitor getMonitor(long mid)`	Returns a standalone monitor
`Monitor getMonitorFor(long mid)`	Retrieves the Monitor created for the Model specified by mid
`boolean hasModel(long mid)`	Whether a model exists in the Model Manager or not
`void initialize()`	Initializes the Model Manager
`void initMessageLog(PrintWriter pw)`	Initializes the MessageLoggers with the PrintWriter file stream
`void initModels()`	Initializes the models for a simulation run
`void initMonitors(PrintWriter pw)`	Initializes the Monitors with the PrintWriter file stream
`Enumeration layers()`	Returns the layers in the ModelManager
`void loadModel(String msf)`	Loads a serialized model (not implemented)
`int modelCount()`	Returns the number of Models in the

	repository
`int messageLogCount()`	Returns the number of MessageLoggers in the repository
`int monitorCount()`	Returns the number of Monitors in the repository
`Enumeration models()`	Returns the Models
`Enumeration models(Class type)`	Returns the Models in the ModelManager repository that subclass type
`Enumeration modelTypes()`	Returns the types of Models supported by the ModelManager
`void propertyChange(PropertyChangeEvent pce)`	Implements the PropertyChangeListener method. Used for handling model death
`void refreshFactories()`	Refreshes the Model Factories dynamically
`void removeExceptionListenerFromModels(ExceptionListener el)`	Convenience method for removing the ExceptionListener from the Models
`void removeLayer(LayerManager lay)`	Removes the LayerManger from the ModelManager
`void removeMessageLogger(long mid)`	Removes the MessageLogger from the ModelManager
`void removeModel(long mid)`	Removes the Model from the ModelManager
`void removeModelStateListener(ModelStateListener msl)`	Removes the ModelStateListener from the ModelManager
`void removeMonitor(long mid)`	Removes the Monitor from the ModelManager
`void removeSimulationListenerFromModels(SimulationListener sl)`	Removes the SimulationListener from the ModelManager
`Void resetFromFile(String resfile)`	Resets the Models from the specified Model Definition File
`Void resetModel(long mid)`	Resets the specified Model to default values
`Void setModelSeed(long seed)`	Sets the Model Seed
`Void textify(String mdf)`	Textifies the Models in the specified Model Definition File

Appendix D: Engine API

Engine implements ***Runnable***, ***SimulationListener***	
`Engine()`	Constructs the Engine object
`void addEngineListener(EngineListener el)`	Adds an EngineListener
`void cancelEvent(SimulationEvent cevt)`	Cancels all events caused by the specified event
`void eventFinished(SimulationEvent evt)`	Implements the SimulationListener method. Handles the completion of the given event. Reschedules the event target.
`void eventFired(SimulationEvent evt)`	Implements the SimulationListener method. Handles the generation of new events. Schedules the event.
`long getEndTime()`	Returns the end time of the session
`void insertEvent(SimulationEvent evt)`	Insert an event into the Engine manually
`void pause()`	Pauses Engine processing. Fires an EngineEvent to signal the change in Engine RunState.
`void purgeModelEvents(Model targ)`	Purges all the events for the specified model
`int queuedEvents()`	Returns the count of events in the event queue
`void removeEngineListener(EngineListener el)`	Removes the EngineListener
`void removeEvent(SimulationEvent revt)`	Removes the specified event from the event queue
`void rescheduleModel(Model mod, long timestamp)`	Reschedules the specified model with at the given timestamp.
`void reset()`	Resets the Engine. Fires an EngineEvent to signal the change in Engine RunState.
`void resume()`	Resumes Engine processing. Fires an EngineEvent to signal the change in Engine RunState.
`run()`	Implements the Runnable method. Executes the event loop.
`int runState()`	Returns the RunState of the Engine
`scheduleEvent(SimulationEvent evt)`	Schedules the specified event in the event queue
`void setEndTime(long endtime)`	Sets the EndTime of the session
`void start()`	Starts the Engine. Creates a new thread for the session and runs it. Fires an EngineEvent to signal the change in Engine RunState.
`void stop()`	Terminates Engine processing. Stops the current session. Fires an EngineEvent to signal the change in Engine RunState.

Appendix E: Model Definition File DTD

```
<?xml encoding="US-ASCII"?>

<!ELEMENT mdf (model+, adapter+)>
<!ATTLIST mdf version CDATA #REQUIRED>

<!ELEMENT model
          (processorspeed?, memory?, bandwidth?,latency?,
           defaultinterface?, link?, service?, maxsize?,
           queueadd?, queuecost?, routecost?, handlecost?,
           sendcost?, maxprocesses?, router?, interface*,
           resident*, host?, multitasking, concurrent_tasks,
           reliability, parameter*)>
<!ATTLIST model
          type CDATA      #REQUIRED
          id ID           #REQUIRED
          simseed CDATA   #REQUIRED>

<!ELEMENT processorspeed (#PCDATA)>
<!ATTLIST processorspeed type CDATA #REQUIRED>

<!ELEMENT memory (#PCDATA)>
<!ATTLIST memory type CDATA #REQUIRED>

<!ELEMENT bandwidth (#PCDATA)>
<!ATTLIST bandwidth type CDATA #REQUIRED>

<!ELEMENT latency (#PCDATA)>
<!ATTLIST latency type CDATA #REQUIRED>

<!ELEMENT defaultinterface (IDREF)>

<!ELEMENT link (IDREF)>

<!ELEMENT service (IDREF)>

<!ELEMENT maxsize (#PCDATA)>
<!ATTLIST maxsize type CDATA #REQUIRED>

<!ELEMENT queueadd (#PCDATA)>
<!ATTLIST queueadd type CDATA #REQUIRED>

<!ELEMENT queuecost (#PCDATA)>
<!ATTLIST queuecost type CDATA #REQUIRED>

<!ELEMENT routecost (#PCDATA)>
<!ATTLIST routecost type CDATA #REQUIRED>

<!ELEMENT handlecost (#PCDATA)>
<!ATTLIST handlecost type CDATA #REQUIRED>

<!ELEMENT sendcost (#PCDATA)>
<!ATTLIST sendcost type CDATA #REQUIRED>

<!ELEMENT maxprocesses (#PCDATA)>
<!ATTLIST maxprocesses type CDATA #REQUIRED>

<!ELEMENT router (IDREF)>

<!ELEMENT interface (IDREF)>

<!ELEMENT resident (IDREF)>

<!ELEMENT host (IDREF)>

<!ELEMENT multitasking (#PCDATA)>
```

```
<!ATTLIST multitasking type CDATA #REQUIRED>

<!ELEMENT concurrent_tasks (#PCDATA)>
<!ATTLIST concurrent_tasks type CDATA #REQUIRED>

<!ELEMENT reliability (#PCDATA)>
<!ATTLIST reliability type CDATA #REQUIRED>

<!ELEMENT parameter (value, distribution)>
<!ATTLIST parameter
          type CDATA #REQUIRED
          label CDATA #REQUIRED>

<!ELEMENT value (#PCDATA)>
<!ATTLIST value type CDATA #REQUIRED>

<!ELEMENT distribution (distribution_parameter+)>
<!ATTLIST distribution type CDATA #REQUIRED>

<!ELEMENT distribution_parameter (#PCDATA)>
<!ATTLIST distribution_parameter type CDATA #REQUIRED>

<!ELEMENT adapter (taskfile?, parameterfile?, host, reliability)>
<!ATTLIST adapter
          type CDATA #REQUIRED
          label CDATA #REQUIRED
          id ID #REQUIRED
          simseed CDATA #REQUIRED>

<!ELEMENT taskfile (#PCDATA)>

<!ELEMENT parameterfile (#PCDATA)>

<!ELEMENT taskfile (#PCDATA)>
```

Appendix F: Simulation Parameter File DTD

```
<?xml encoding="US-ASCII"?>

<!ELEMENT spf (title, modelfile, parameterfile, start_time, duration,
simulation_seed)>
<!ATTLIST spf version CDATA #REQUIRED>

<!ELEMENT title (#PCDATA)>

<!ELEMENT modelfile (#PCDATA)>

<!ELEMENT parameterfile (#PCDATA)>

<!ELEMENT start_time (#PCDATA)>

<!ELEMENT duration (#PCDATA)>

<!ELEMENT modelfile (#PCDATA)>

<!ELEMENT simulation_seed (#PCDATA)>
```

References

1. Aryananda L. "An Exception Handling Service for the Contract Net Protocol Family" Master's of Engineering Thesis, Massachusetts Institute of Technology, 1999.

2. Bratley P., Fox B., Schrage L., "A Guide to Simulation 2nd edition" Springer-Verlag, 1987 New York.

3. Bray T., Paoli J., Sperberg-McQueen C. M., "Extensible Markup Language (XML) 1.0" http://www.w3.org/TR/REC-xml

4. Burbeck S. "Applications Programming in Smalltalk-80 ™: How to use Model-View-Controller (MVC)" http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/mvc.html

5. Chia M.H., Neiman D.E., Lesser V.R. "Poaching and Distraction in Asynchronous Agent Activities." Proceedings International Conference on Multi Agent Systems. IEEE Comput. Soc., (1998) pp.88-95. Los Alamitos, CA.

6. Cormen T., Leiserson C., Rivest R. "Introduction to Algorithms" MIT Press, 1990 Cambridge

7. Crain R., Smith D. "Industrial strength simulation using GPSS/H" Proceedings of the 1994 Winter Simulation Conference.

8. Englander R. "Developing Java Beans" O'Reilly, 1997 Sebastopol

9. Evans J. "Structures of Discrete Event Simulation: an introduction to the engagement strategy" Ellis Horwood Limited, 1988 Chichester

10. Flanagan D. "Java in a Nutshell" O'Reilly, 1997 Sebastopol

11. Gamma E., Helm R, Johnson R, and Vlissides J. "Design Patterns Elements of Reusable Object-Orientated Software" Addison-Wesley, 1995

12. Haggett P. and Chorley, R. J. "Models, Paradigms, and the New Geography", Chapter 1 of Chorley, R. J. and Haggett, P. (Eds) Models in Geography, pgs. 19-41, Methuen, 1967 London

13. IBM corporation "Visual Age for Java, Version 2.0: IDE Basics"

14. Klein M. and Dellarocas C. "Exception Handling in Agent Systems" Proceedings of the Third International Conference on AUTONOMOUS AGENTS, Seattle, Washington, 1999.

15. Mikler, A.R., Wong, J.S.K., and Honavar, V.G. "An Object-Oriented Approach to Simulating Large Communication Networks." The Journal of Systems and Software. Volume 40, No.2 (1998) pp. 151-164.

16. Miller A. J. "Random Number Generation: randomF90" http://www.ozemail.com.au/~milleraj/misc/random.f90

17. Nemirovsky M. A., "Building Object-Oriented Frameworks" http://www7.software.ibm.com/vad.nsf/Data/Document1569?OpenDocument&p=1&BCT=1&Footer=1

18. Objectspace JGL Version 3.1 User's Guide http://www.objectspace.com/developers/jgl/white/jglsupport.dll?2

19. Ohko T., Hiraki K., and Anzai Y. ``Reducing Communication Load on Contract Net by Case-Based Reasoning -- Extension with Directed Contract and Forgetting --", ICMAS-96,Dec.10-13,pp.244--251, 1996

20. Robbins J., Hilbert D., "Graph Editing Framework" http://www.ics.uci.edu/pub/arch/gef/index.html

21. Saad A., Biswas G., Kawamura K., " Performance Evaluation of Contract Net-Based Heterarchical Scheduling for Flexible Manufacturing Systems", International Journal of Automation and Soft Computing, Special Issue on Intelligent Manufacturing Planning and Shop Floor Control, 1996.

22. Vincent R., Horling B., Wagner T., and Lesser R. "Survivability Simulator for Multi-Agent Adaptive Coordination" http://dis.cs.umass.edu/~vincent/wmc-98/aritcle.html