AMPS: A Simulation System for Modeling and Analyzing the Psychology of Risk-Taking

by

Lawrence C. Wang

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 [Computer] Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

June 2003

May 19, 2003

Author __________

Department of Electrical Engineering and Computer Science
May 19, 2003

Certified by ______________________________

Andrew W. Lo
Harris and Harris Group Professor
Thesis Supervisor

Certified by __________

Dmitry V. Repin
Postdoctoral Associate
Thesis Supervisor

Accepted by __________

Arthur C. Smith
Chairman, Department Committee on Graduate Thesis

AMPS: A Simulation System for Modeling and Analyzing the Psychology of Risk-Taking

by

Lawrence C. Wang

Submitted to the
Department of Electrical Engineering and Computer Science

May 19, 2003

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

Abstract

This thesis presents the design and implementation of Artificial Market Psychology Simulator (AMPS), an autonomous, programmable simulation system designed to assist research on the psychology of financial decision-making. The system enables researchers to systematically control price patterns and generate other market events in a securities market simulation, in order to analyze their impact on the emotional characteristics and subsequent trading behavior of professional securities traders. Real-time trader behavior and portfolio information is then fed back into the system to dynamically generate subsequent price patterns and contribute to the desired emotional impact on the trading subject.

AMPS enables researchers to program dynamic price patterns using rule-based scripting mechanisms and a library of customized mathematical functions. The system leverages MIT Web Market, an artificial securities market, for basic market-making operations and transaction logging. AMPS is also integrated with RStudio, a physiological data collection system, to log timestamps of critical simulation events.

Thesis Supervisor: Andrew W. Lo
Title: Harris and Harris Group Professor

Thesis Supervisor: Dmitry V. Repin
Title: Postdoctoral Associate

Acknowledgements

This thesis would not have been possible without the guidance and support from a number of people.

My thesis advisor Prof. Andrew Lo granted me this remarkable thesis opportunity, for which I am extremely grateful. His research guidance and advice have been crucial to the project. His time and understanding have been most encouraging.

My direct supervisor Dr. Dmitry Repin has worked with me through the most critical phases of the project. His collaboration in requirements and functional specifications, ongoing feedback, innovations, and general supervision have made a deciding difference.

Time and collaboration from Eric Ho have made possible the integration with the RStudio system he developed. Adlar Kim has also provided critical help and information to facilitate the integration with the Web Market system. The IT staff members of the MIT Lab for Financial Engineering have consistently and promptly provided the system and technical support I required, and their contribution must not be undermined.

My academic advisor Prof. Ron Rivest has been most encouraging, and I am grateful for his general advice on research and academics. Anne Hunter, Course VI Administrator, has also provided valuable administrative support.

Ultimately, this thesis would not have been completed without the support of my beloved family and friends. Special thanks to the following individuals for being there when I needed them the most: Mom, Dad, Fred, Grace, Icie, Elbert, Alex, Sue, Kevin, Ernie, Steve, Paul, and Joey. They have shown unbelievable care and understanding as I underwent long periods of solitary confinement during critical phases of the project. I cannot say enough to express how deeply indebted I am to them.

Contents

1 INTRODUCTION 11

1.1 BACKGROUND AND CHALLENGES 11

1.2 PROJECT OVERVIEW 13

1.3 PROJECT REQUIREMENTS 13

1.3.1 Real-Time Data Requirements 14

1.3.2 Price Pattern Computation Requirements 14

1.3.3 User Interface and Usability Requirements 14

1.3.4 Data Storage and Retrieval Requirements 14

1.3.5 Interoperability Requirements 15

1.4 PROJECT SCOPE AND DEVELOPMENT METHODOLOGY 15

1.4.1 Requirements Collection and Planning 16

1.4.2 Functional Specifications 16

1.4.3 Design Specifications 16

1.4.4 Iterative Development 17

1.4.5 Quality Engineering 17

1.4.6 Deployment and Knowledge Transfer 17

1.4.7 Project Closure 17

1.5 WRITING CONVENTIONS 18

2 DESIGN CONSIDERATIONS 19

2.1 USABILITY 19

2.2 SCALABILITY AND PERFORMANCE 19

2.3 ROBUSTNESS AND MAINTAINABILITY 20

2.4 INTEGRATION WITH EXTERNAL DATA SOURCES 20

2.5 SEPARATION OF EMOTION, BID/ASK, AND MESSAGE PROCESSES 20

3 SYSTEM DESIGN SPECIFICATIONS 22

3.1 ARCHITECTURAL OVERVIEW 22

3.2 SIMULATION DESIGN 25

3.2.1 Structure and Sessions Overview *25*

3.2.2 Simulation States and Transitions *26*

3.2.3 Process and Data Flow *29*

3.3 AMPS COMPONENTS 31

3.3.1 Simulation UI *31*

3.3.2 Emotion Engine *34*

3.3.3 Bid/Ask Engine *36*

3.3.4 Message Engine *39*

3.3.5 AMPS Trader and AMPS Trader Thread *41*

3.3.6 DB Connector *41*

3.3.7 RS Connector *42*

3.3.8 WMS Connector *43*

3.4 EXTERNAL COMPONENTS 44

3.4.1 MIT Web Market *44*

3.4.2 RStudio *45*

4 GENERAL USAGE AND CONFIGURATIONS 47

4.1 GENERAL USAGE MODEL 47

4.2 SYSTEM STARTUP AND EXIT 48

4.3 SYSTEM CONFIGURATIONS 49

4.3.1 Sessions Configurations *49*

4.3.2 File and Directory Path Configurations *50*

4.3.3 RS Connector and DB Connector Configurations *51*

4.4 USING EMOTION AND BID/ASK RULE SETS 52

4.4.1 Rule Set Structure and Syntax *52*

4.4.2 Variables *54*

4.4.3 Operators and Functions *56*

4.4.4 Vectors *57*

4.5 USING MESSAGE FILES 57

4.6 USING DATABASE ACCESS SCRIPTS 58

4.7 RS CONNECTOR LOCAL LOG FILE 60

4.8 WEB MARKET STARTUP AND EXIT 61

5 AMPS USER INTERFACES 62

5.1 SIMULATION USER INTERFACES 62

5.1.1 Simulation Panel *62*

5.1.2 Data Logging Panel *65*

5.1.3 AMPS Configuration Editor *67*

5.2 WEB MARKET SERVER AND CLIENT USER INTERFACE 74

5.2.1 Client User Interface *74*

5.2.2 Server User Interface *74*

6 SOFTWARE TEST PLAN 75

6.1 OVERVIEW 75

6.2 UNIT TESTING 75

6.3 FUNCTIONAL TESTING 76

6.4 SYSTEM TESTING 78

6.5 TEST ENVIRONMENT 79

7 FUTURE WORK 80

7.1 SUPPORT FOR PHYSIOLOGICAL DATA INPUTS 80

7.2 LOGGING SYSTEM ENHANCEMENTS 81

7.3 IMPROVED EXCEPTION AND ERROR HANDLING 81

7.4 ADDITIONAL LOGIC CONSTRUCTS AND RULE SET SYNTAX 82

7.5 SUPPORT FOR MULTIPLE CONCURRENT TRADING SUBJECTS 82

8 DISCUSSION 83

APPENDIX A: AMPS CUSTOM FUNCTIONS 85

APPENDIX B: JEP OPERATORS, FUNCTIONS, AND CONSTANTS 87

OPERATORS 87

FUNCTIONS 88

CONSTANTS 88

APPENDIX C: LIST OF AMPS CONFIGURATION AND LOG FILES 89

APPENDIX D: SYSTEM CONFIGURATION FILE AND PARAMETERS 90

APPENDIX E: SAMPLE EMOTION AND BID/ASK RULE SETS 95

APPENDIX F: AMPS SYSTEM REQUIREMENTS 101

APPENDIX G: CONFIGURATION EDITOR SCREENSHOTS 103

APPENDIX H: WEB MARKET CLIENT AND SERVER SCREENSHOTS 107

List of Figures

Figure 1-1: Emotion Generation Feedback Cycle 13

Figure 3-1: AMPS Component Architecture 23

Figure 3-2: Simulation State Transitions 27

Figure 3-3: Simulation Process and Data Flow 30

Figure 3-4: Simulation UI State Transitions 32

Figure 3-5: Sample Emotion Rule Set: *sample_emotion.emo* 36

Figure 3-6: Sample Section of a Bid/Ask Rule Set: *sample_bidask.baa* 38

Figure 4-1: Sample Section of *amps.ini*: Sessions Configurations 49

Figure 4-2: Sample Section of *amps.ini*: Path Configurations 51

Figure 4-3: Sample Section of *amps.ini*: Connectors Configurations 51

Figure 4-4: Sample Message File: *sample_message.msg* 57

Figure 4-5: Sample Local Log File: *sample_rstudio.log* 60

Figure 5-1: Simulation UI: Simulation Panel 63

Figure 5-2: Simulation UI: Data Logging Panel 66

Figure 5-3: Configuration Editor: Main Graphical Components 68

Figure 5-4: Configuration Editor: File Menu Items 69

Figure 5-5: Configuration Editor: View Menu Items 70

Figure 5-6: Configuration Editor: Internal Frame and Edit Menu Items 72

Figure D-1: Sample AMPS System Configuration File: *amps.ini* 91

Figure E-1: Sample Emotion Rule Set: *sample_emotion.emo* 95

Figure E-2: Sample Emotion Rule Set Initialization File: *sample_emotion.ini*................ 96

Figure E-3: Sample Bid/Ask Rule Set: *sample_bidask.baa* 97

Figure E-4: Sample Bid/Ask Rule Set Initialization File: *sample_bidask.ini* 100

Figure G-1: Configuration Editor: File Menu - Open.. 103

Figure G-2: Configuration Editor: File Menu – Save As 104

Figure G-3: Configuration Editor: View Menu – Minimize All.................................... 104

Figure G-4: Configuration Editor: View Menu – Show All... 105

Figure G-5: Configuration Editor: View Menu – Line Up All....................................... 105

Figure G-6: Configuration Editor: Internal Frame – Minimize Icon.............................. 106

Figure G-7: Configuration Editor: Internal Frame – Maximize Icon 106

Figure H-1: Web Market Client User Interface... 107

Figure H-2: Web Market Server User Interface 108

List of Tables

Table 3-1: Architectural Components and Functions 25

Table 3-2: Simulation States, Entry Conditions, and Consequences 29

Table 4-1: AMPS High-Level Administrator Tasks 48

Table 4-2: Rule Set Operator Order of Precedence 54

Table 4-3: Market Variables and Descriptions 55

Table 4-4: Order Variables and Descriptions 56

Table 4-5: Sample Message Delivery Schedule 58

Table 4-6: Database Access Scripts and Functions 60

Table 5-1: Simulation Panel Components and Functions 65

Table 5-2: Data Logging Panel Components and Functions 67

Table 5-3: File Menu Items and Functions 70

Table 5-4: View Menu Items and Functions 71

Table 5-5: Edit Menu Items and Functions 73

Table 5-6: Internal Frame Icons and Functions 73

Table 5-7: Internal Frame Key Bindings and Functions 74

Table 6-1: Unit Test Scripts and Descriptions 76

Table 6-2: Functional Test Cases and Descriptions 78

Table 6-3: System Test Cases and Descriptions 79

Table 6-4: Test Environment Specifications 79

Table A-1: List of AMPS Custom Functions 86

Table B-1: List of JEP Operators 87

Table B-2: List of JEP Functions 88

Table B-3: List of JEP Constants 88

Table C-1: List of AMPS Configuration and Log Files 89

Table D-1: List of System Configuration Parameters 93

Table F-1: AMPS System Requirements 102

Chapter 1

Introduction

1.1 Background and Challenges

The impact of psychology on the behavior of professional securities traders and its implications on the fundamental driving forces behind financial market movements has been a longstanding issue of controversy in the financial industry. Historically, the majority of economic and financial theories are based on the assumption that individuals act rationally and consider all available information in their risk-taking and decision-making processes. The supporters of the Efficient Market Hypothesis (EMH)[1], for instance, advocate that in a market of rational investors and efficient information flow, the price of a security directly corresponds to the security's perceivable economic value, and that there should be no ambiguity to the price, or the market full of rational, competing investors will almost instantaneously adjust the price to its appropriate level. The school of technical analysts, on the other hand, believes in the existence and predictability of market patterns and anomalies from EMH's price implications. This belief is primarily based on the assumption that investors are irrational and that their risk-taking activities are based on factors beyond their perception of fundamental economic value, such as transient emotions and cognitive states. Modern research exploring the relationship between psychology and rationality in decision-making processes has evolved into what is now known as the field of behavioral finance.

Past research in behavioral finance has provided ample evidence of the influence of emotional states on the rationality of risk-takers. The research of Raghunathan and Pham

[1] Efficient Market Hypothesis (EMH) is a theory on market efficiency founded by Eugene Fama. It suggests that, in an efficient market, the competition among rational investors ensures that the actual price of a security is a good estimate of its intrinsic value at any point in time. See [8].

suggests that anxious individuals tend to be more risk-averse – biased towards low risk and low reward investment options. Sad individuals, meanwhile, exhibit a greater liking for high risk and high reward options [9]. Mano's studies have also suggested that higher emotional arousal causes the decision maker to develop a greater affinity for risk [5].

Despite the strong professional and academic interest, research in behavioral finance has largely remained at the theoretical level because of several challenges. First of all, the correlation analysis between emotions and risk tolerance is difficult to perform due to the intricacies in systematically inducing emotions. Further, the inherent complexities in collecting and quantifying data on human emotions present another obstacle. Thirdly, emotions are transient cognitive states that must be tracked and interpreted over extended periods of time in a controlled and stable environment.

An organized research effort headed by Andrew Lo and Dmitry Repin at the MIT Lab for Financial Engineering (LFE) is in progress to explore the influence of human emotions on financial decision-making processes through systematic and controlled experiments. Their preliminary research employed professional securities traders as experiment subjects and utilized biofeedback equipment to measure real-time physiological responses while the subjects were trading. The studies so far have demonstrated statistically significant correlations between certain market events and traders' physiological characteristics that are indicative of emotional fluctuations [8]. The approach represents an important milestone in that real-time physiological data puts emotions in a more empirical and quantifiable context.

The aim of this thesis project is to contribute to the effort at MIT Lab for Financial Engineering by developing a simulation system facilitating the control and monitoring of trader emotions through systematically controlled market movements. By dynamically generating emotion-impacting price patterns catered to real-time trading behavior and asset allocations of the experiment subject, the system enables researchers to not only observe the influence of emotions on the subject's risk-taking behavior, but also induce these emotions in a systematic manner.

1.2 Project Overview

The Artificial Market Psychology Simulator (AMPS) is a simulation system designed to aid in the research on the psychology of risk-taking, undertaken at the MIT Lab for Financial Engineering. The system facilitates the exploration of the relationship between market movements, the traders' emotional characteristics, and risk-taking behavior by allowing researchers to elicit emotional states on the human subject through a systematic, controlled, and dynamic process.

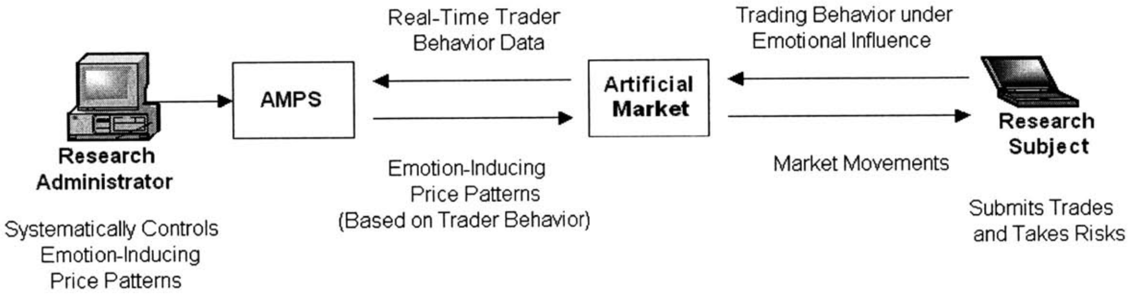

Real-Time Trader Behavior Data
Trading Behavior under Emotional Influence
AMPS
Artificial Market
Research Subject
Research Administrator
Emotion-Inducing Price Patterns (Based on Trader Behavior)
Market Movements
Systematically Controls Emotion-Inducing Price Patterns
Submits Trades and Takes Risks

Figure 1-1: Emotion Generation Feedback Cycle

The system provides the research subject, or professional securities trader, with a simulated securities market in which to perform risky trades for profit. It enables the researcher to influence the emotion of the trading subject through a series of controlled price patterns and other market events such as security news. The subject's order submissions, influenced by his or her emotions, are then monitored and collected as a form of behavior data. The real-time behavior data, along with the trader's current portfolio and other real-time market information, are continuously fed back to the system in order to dynamically adjust subsequent price movements based on the trader's observed behavior in the context of his current asset allocation and market conditions.

1.3 Project Requirements

The prototype should meet the high-level requirements discussed in the following sections.

1.3.1 Real-Time Data Requirements

The system should be able to retrieve real-time data on trader behavior and market movements, and allow the research administrators to utilize this data in price pattern generation. The challenge requires AMPS to interface with the market database directly and retrieve the data at every time step during the simulation. The design should therefore ensure that the rate of data retrieval and the overall performance of the database connection can meet the timing requirement of the simulation clock tick.

1.3.2 Price Pattern Computation Requirements

AMPS should enable the administrator to systematically and dynamically generate emotion-inducing price patterns based on real-time trader and market data at every time step during the simulation. This requirement leads to a number of finer challenges. First, the system should provide tools to facilitate the programming of dynamic price patterns based on real-time trader behavior and market data. The tools should be designed to accelerate pattern development as well as provide the extensibility to incorporate a large variety of real-time data input formats. In addition, the computation of price patterns must be fast enough to cope with the rapid pace of the simulation clock tick. Given the aforementioned requirements, the data structure and syntax of price patterns must be designed to offer an adequate balance of computation speed and usability.

1.3.3 User Interface and Usability Requirements

Intuitive graphical user interfaces must be provided for the research administrator to control and monitor the simulation process. Similarly, interfaces should be provided for the research subject to participate in trading. The user interfaces should be robust and functional for the current set of requirements and also extensible enough to incorporate potential or additional feature requirements for the future.

1.3.4 Data Storage and Retrieval Requirements

All market and trader behavior data should be collected and stored for postmortem analysis of trader behavior, emotional state, and price pattern correlations. Scripts and other tools to access the database should be provided to facilitate the data retrieval and examination after the experiment.

1.3.5 Interoperability Requirements

AMPS should work in conjunction with Web Market, MIT's existing artificial market system (see Section 3.4.1: MIT Web Market) to reduce duplicate development effort as well as to maintain compatibility among software systems regularly used by the staff of MIT Lab for Financial Engineering. The system should leverage Web Market's market-making functionality in the simulation – the trades should be submitted to and matched by the Web Market server. The transactions and trader portfolios should be maintained in the Web Market database. The trading subjects should be allowed to utilize Web Market's web-based user interface for market monitoring, portfolio tracking, and trade submissions. These requirements introduce additional complexities in the generation of price patterns because they require AMPS to submit trades using Web Market's machine trader API, thereby limiting AMPS's ability to control price movements directly.

In addition to Web Market, AMPS should also work with external data collection systems including RStudio (see Section 3.4.2: RStudio), and be able to delivery or log timestamp signals during the simulation.

1.4 Project Scope and Development Methodology

The scope of the project covers the complete development lifecycle of an initial prototype of AMPS, from the initial requirements gathering and specifications to design, development, testing, deployment, knowledge transfer, and postmortem project review.

The software development methodology underlying the project lifecycle consists of a number of phases that are briefly described below. While the methodology provides a valuable guideline for driving the development process forward and navigating it a satisfactory closure, the reader should understand that the methodology is proposed more as a conceptual guideline than a procedure to be followed. Like any sequence of software development workflow in practice, many of the outlined phases may take place concurrently and span across multiple sessions. As end user feedback is collected and technical issues arise on an ongoing basis, the overlaps and oscillations between the phases are also expected to become more complex.

1.4.1 Requirements Collection and Planning

In this initial phase of development, high-level feature requirements, general use cases, deployment environment, and timeframe requirements should be gathered from the end users or researchers that will be using the AMPS system. The requirements should be prioritized based on criticality, estimated level of effort, availability of resources, and target release schedule. The users and the developers should jointly commit to a set of high-level features that is clearly defined and understood by both parties, as well as realistic, given the resources and timeframe.

1.4.2 Functional Specifications

AMPS features and functions from the users' perspective should be specified in detail during this phase of development. The specifications should include screenshots, detailed simulation use cases, configuration file formats, development and scripting language for the user, data input and output format, and other parts of the software relating to the overall user experience. The descriptions should be detailed and precise, in order to synchronize the end product expectations between the developers and the end users.

1.4.3 Design Specifications

AMPS baseline system and subsystem architecture should be conceptualized and defined during this phase, based on the outcome of requirements collection and functional specifications. The design should include descriptions of the component relationship, data flow, process flow, communications model, and other relevant technical details. The architectural and technical specifications should also be refined on an ongoing basis up to the end of the iterative development cycle.

1.4.4 Iterative Development

AMPS application should be implemented in this phase, based on the architecture defined during design specifications and the features outlined during functional specifications. This process involves continuous iterations of new code development and testing to ensure that the new code does not adversely impact the existing code base. Source code should be fully documented, and issues and bugs should be tracked on an ongoing basis.

1.4.5 Quality Engineering

A comprehensive test plan should be developed and executed to ensure the correct functioning as well as the fulfillment of the project requirements. The test plan should cover component-level, functional-level, and system-level testing, and should be executed concurrently with development. Following the completion of code development and testing, trial experiments involving actual subjects should be performed to obtain additional field test results and feedback.

1.4.6 Deployment and Knowledge Transfer

Installation logistics and knowledge transfer should be handled in this phase. AMPS software should be packaged, documented, and delivered to the end users. The system should be fully installed, configured, and tested in the production environment. Training for end users and any supporting user documentation should also be prepared to ensure a smooth learning curve.

1.4.7 Project Closure

The developers and end users should jointly perform a post-implementation review, and identify any outstanding issues and the plans for their resolution. Proposals for future development may be formulated at this stage. Any required maintenance and support arrangement should also be made as part of a satisfactory project closure.

1.5 Writing Conventions

This thesis includes fragments of program code and sections of configuration files used as examples. The following typesetting conventions are used to help the reader:

- `Constant width` type is used for Java code and class names, with language keywords slanted.
- `Constant width` type is also used for grammars and configuration file contents.
- *`Constant width italicized`* type is used for filenames and descriptive expressions in variable names or values. When used in descriptive expressions, the expression is enclosed between '<' and '>'. For example, potential values represented by '`SESSION`<*`number`*>' include '`SESSION1`', '`SESSION2`', and so on.
- **`Constant width bold`** type is used for command line input, variable names, variable values, and other symbols not mentioned above. When these types appear inline within the main body text, they are enclosed within single quotes ('') for clarity.

In addition, several potentially ambiguous terms are defined as follows:

- **Administrator, User, and End User** – refer to a research administrator who uses AMPS to host an experiment.
- **Subject, Trader, and Trading Subject** – refer to a simulation subject or professional trader who is participating or a candidate for participating as a subject in the simulation.
- **Developer** – refers to the person or group developing AMPS.
- **Time Step and Clock Tick** – refer to one increment or pulse of the simulation's internal time counter. Each pulse generally invokes a series of simulation-related activities.
- **Runtime** – refers to time when the AMPS application or the Java code is running, or when the simulation is in progress.

Chapter 2

Design Considerations

2.1 Usability

AMPS users are researchers who may not have a technical background, therefore, it is important for the system to provide tools to facilitate the tasks an administrator must perform before, during, and after the experiment. The user interfaces should be simple and intuitive, while providing the comprehensive set of required functions. The tools for programming emotion-inducing price patterns – a critical aspect of AMPS – should be designed to be simple, familiar, and extensible, ensuring a short learning curve without compromising on capabilities. The system should provide an adequate amount of warning, error, and information messages to keep the user aware of the ongoing status of the simulation. These system notifications should be designed to be informative, relevant, and concise. Additional convenience features, including command line scripts and help files should also be considered based on available timeframe.

2.2 Scalability and Performance

AMPS experiments often involve significant volumes of data and computation within fractions of a second. Therefore, the system should be designed to be scalable in terms of both the number of emotion rules and bid/ask rules that govern the price movements, as well as the frequency of order submission (number of simulation time steps within a second). Measures should be taken to place reasonable limits on performance degradation as the number of rules and order submission frequency increase.

2.3 Robustness and Maintainability

AMPS users are researchers who may not have a technical background, and care in design should be taken to ensure that a running AMPS system is robust enough to withstand potential system failures and user mistakes. The system should also be able to handle simulation sessions with length in the magnitude of hours. While the system operates under normal use cases, any ongoing effort required to maintain the system, including data cleanup, configurations, or debugging, should be minimized by design.

2.4 Integration with External Data Sources

Depending on the needs of the experiment, the trader's real-time psychological and behavioral data may come from a variety of sources, ranging from Web Market's database to biofeedback measuring equipment that collect real-time physiological data from the human subject. Although external data sources will be limited to the Web Market database for the scope of this initial prototype of AMPS, the architecture should be designed with extensibility and ease of integration in mind, to handle different types of data sources in future development. For example, the task of incorporating new data parameters into the order generation processes should be simple and straightforward. The storage format of permanent and transient data should also be able to support potential data parameter types without loss of timeliness or precision of data.

2.5 Separation of Emotion, Bid/Ask, and Message Processes

Emotion process refers to the task of computing emotion-inducing target prices based on real-time trader and market state information. Bid/Ask process refers to the set of activities to generate actual bid and ask prices based on the target security prices computed by the emotion process. Message process involves the delivery of pre-programmed system messages to the trader regarding simulation status or market events that may influence trader emotions and behavior. The three processes should be modularized and made independent of each other, such that the researcher can mix and match processes easily, and examine their influence on the research subject with more flexibility and control.

Chapter 3

System Design Specifications

3.1 Architectural Overview

The architecture of AMPS is built from three types of components: user interface, business logic, and data components. User interface components are graphical input and display components that allow research administrators and subjects to interact with the system. Data components are structures that maintain specific types of data or metadata, and provide the mechanisms to access and modify the data. Business logic components perform the data processing and computation underlying the simulation system. For purposes of our discussion, we will categorize threads, or concurrently executable processes, under business logic. Because AMPS works in close conjunction with a number of external systems, our discussion will also cover the relevant components of these systems.

The major architectural components of AMPS and the closely related external systems are described in the following figure.

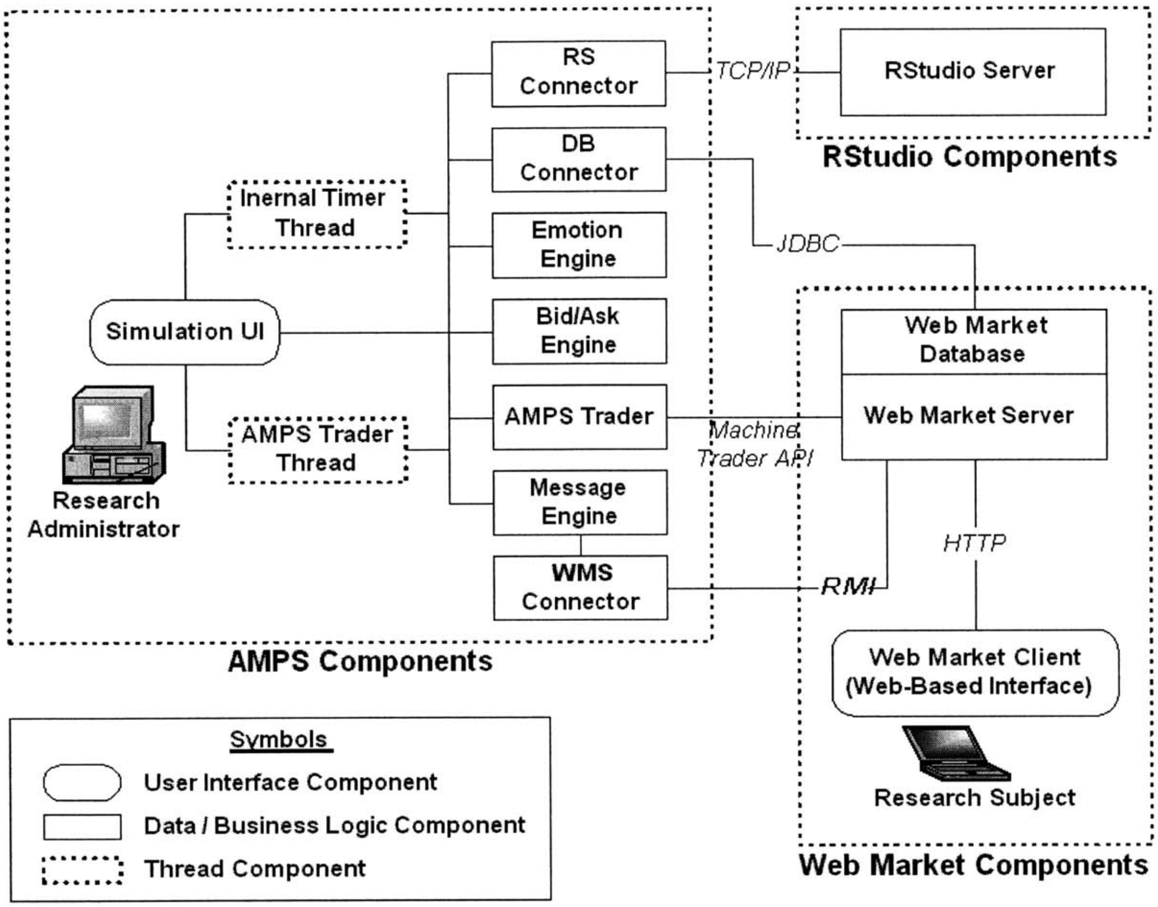


Figure 3-1: AMPS Component Architecture

The user interface components consist of Simulation UI and the Web Market client. Simulation UI enables research administrators to control the experiment, and the Web Market client allows the experiment subjects to trade within the market simulation. The main data component in our setup is the Web Market database, where all market and trader information is stored. A number of lower level data components, not shown in the figure above, also play a key role in the simulation process. These data structures include Sessions, which maintain configuration data for simulation sessions; Rule Sets, which maintain the emotion and bid/ask rules used in price computation; Timed Messages, which store the text and delivery times of broadcast messages; among others. The lower level data components are mainly designed for system modularization and data encapsulation purposes, and will be discussed in later sections.

The table below summarizes the functions of the major architectural components:

Component	*Functional Description*
Simulation UI	Main management user interface that enables the research administrator to control, monitor, and configure the simulation system.
Emotion Engine	Data processor that generates emotion-inducing target price patterns based on real-time trader and market data.
Bid/Ask Engine	Data processor that generates market-moving orders based on target prices computed by Emotion Engine.
Message Engine	Data processor that manages pre-programmed, time-driven system messages or market news messages, and delivers them to the trading subject through WMS Connector and the Web Market server.
DB Connector	Data connector that manages the connection to the Web Market database and provides mechanisms to retrieve real-time market and trader data.
RS Connector	Data connector that manages the connection to RStudio and local log file, and provides mechanisms to deliver timestamps of simulation events to RStudio and/or the local log file.
WMS Connector	Data connector that manages the connection to the Web Market server to access backend services otherwise not available through the Web Market client or machine trader API. This connector is mainly used for message broadcast in AMPS.
AMPS Trader	Data connector that manages the connection to the Web Market server, registers as a machine trader, and submits orders on behalf of AMPS to generate desired market movements.
AMPS Trader Thread	Helper thread that drives the simulation process by invoking trading-related methods in Emotion Engine, Bid/Ask Engine, and DB Connector.

Internal Timer Thread	Timer thread that helps to drive the simulation process by keeping internal simulation time and invoking methods in Simulation UI and Message Engine.
Web Market Client	Web-based user interface that allows the AMPS subject to monitor the market and submit trades to the Web Market server. This is part of the MIT Web Market system.
Web Market Server	Artificial double-auction securities market that performs basic market-making functions (order matching, price and order reporting, and transaction archiving), and keeps track of user portfolios. It allows both human traders using the Web Market client and machine traders using a specialized protocol to participate in the same market. This is part of the MIT Web Market system.
Web Market Database	Database used to store all market and trader information, including orders, sales, and portfolios. This is part of the MIT Web Market system.
RStudio	Physiological data collection system that collects and logs real-time signals from a variety of data sources. This component is external to AMPS.

Table 3-1: Architectural Components and Functions

3.2 Simulation Design

The component architecture is designed based on the structure and requirements of the simulation process discussed below.

3.2.1 Structure and Sessions Overview

An AMPS simulation composes of a series of building block sessions arranged in a linear sequence. A session is a data structure that consists of a finite time period and a set of simulation activities to perform during the period, including price pattern generation and message delivery. A run of the AMPS simulation involves carrying out one or more sessions in a sequential order, and executing the set of activities associated with each specific session. The sequence of sessions as well as individual session parameters are specified in the master configuration file (for more details, see Section 4.3: System Configurations).

In the current prototype, two types of sessions are available – trading sessions and break sessions. In a trading session, emotion-inducing orders are generated on a continuous basis, and the trading subjects are asked to actively participate in trading. In a break session, order generation process is halted, and the traders are asked to refrain from trading.

The data structure and inheritance of a session are designed for the extensibility to incorporate additional types of sessions in future development. Trading sessions and break sessions are subclasses of a generic `Session` data structure, which encapsulates general session data and accesors (get methods) that are applicable across all session types. Specific session types like trading session and break sessions inherit the generic data and methods, and provide additional configuration parameters and accessors specific to the individual session type. To illustrate, the generic `Session` class provides the following get methods for session time length and the message file name, both of which are inherited across all session types:

```
public int getLengthInSeconds()
public String getMessageFilename()
```

The `TradingSession` class, which extends the `Session` class, provides the following additional methods that are specific to a trading session's activities:

```
public String getEmotionFilename()
public String getBidAskFilename()
public String getEmotionTargetUser()
```

Through this inheritance hierarchy and data encapsulation scheme, new session types can be developed in the future with minimal changes to the existing data structures or session loading mechanisms.

3.2.2 Simulation States and Transitions

An AMPS simulation starts up in the *Stopped* state, in which no processes are running, and Simulation UI is awaiting user input to begin the simulation. During the course of an AMPS simulation, the system progresses through a number of transitional states, and eventually finishes back in the *Stopped* state. The state transition diagram below outlines the states and their entry conditions from the end user's perspective – in terms of user actions or events observable by the user through the graphical interface.

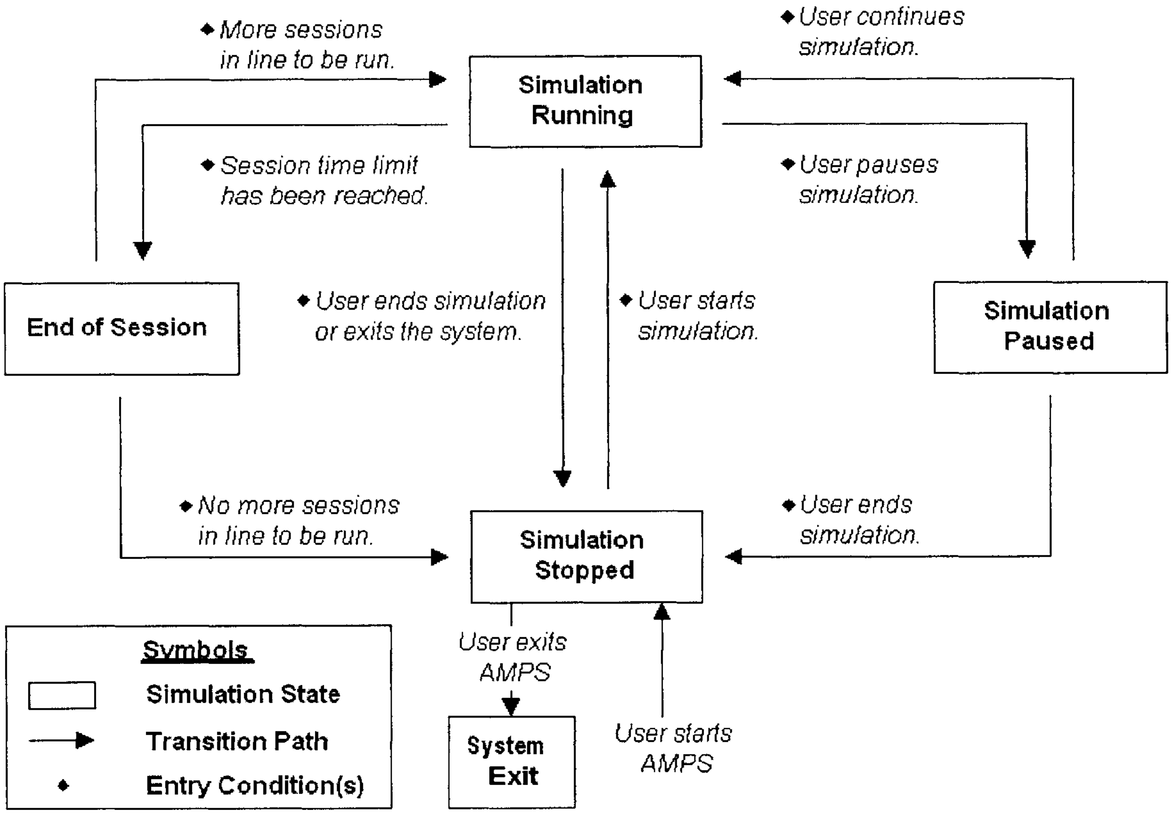

◆ More sessions in line to be run.
◆ User continues simulation.
Simulation Running
◆ Session time limit has been reached.
◆ User pauses simulation.
End of Session
◆ User ends simulation or exits the system.
◆ User starts simulation.
Simulation Paused
◆ No more sessions in line to be run.
◆ User ends simulation.
Simulation Stopped
User exits AMPS
System Exit
User starts AMPS
Symbols
Simulation State
Transition Path
◆ Entry Condition(s)

Figure 3-2: Simulation State Transitions

Underneath the hood, a number of threads are used to manage the transitions and their consequences. AMPS utilizes a lightweight timer thread component named Internal Timer Thread to drive state transitions and the overall simulation process flow. The timer thread is normally started, paused, and stopped by Simulation UI based on user input, and it keeps track of the simulation's internal time. Based on the internal clock ticks, it drives the simulation process by starting or stopping AMPS Trader Thread, invoking session changes, and triggering user interface events in Simulation UI. AMPS Trader Thread is another thread that manages the process flow behind the scenes. Controlled by the timer thread and Simulation UI, it invokes methods in Emotion Engine, Bid/Ask Engine, and DB Connector to drive the order generation process.

A comprehensive description of the simulation states, their entry conditions, and their consequences are provided below.

Simulation State	***Entry Condition(s)***	***Transition Consequence(s)***
Running	The user clicks the "Start" button and one or more sessions are lined up to be run while the simulation is in *Stopped* state. **OR** The user clicks the "Continue" button while the simulation is in *Paused* state. **OR** Emotion Engine and Bid/Ask Engine have finished loading the rule sets for the new session, and Simulation UI has finished updating time and session display while the simulation is in *End of Session* state.	Internal Timer Thread is started *. AMPS Trader Thread is started if current session is a trading session. At every clock tick, Internal Timer Thread checks for messages due for delivery to the trading subject. If a message is due, Message Engine is invoked to deliver the message.
End of Session	The current session finishes running while the simulation is in *Running* state.	Internal Timer Thread is paused. AMPS Trader Thread is stopped. **If there are one or more sessions lined up to be run:** Emotion Engine and Bid/Ask Engine are invoked to load rule sets for next session in line. Time and session display on

		Simulation UI are updated to show details for the next session in line. Simulation proceeds to *Running* state. **If there are no more sessions to be run:** Simulation proceeds to *Stopped* state.
Paused	The user clicks the "Pause" button while the simulation is in *Running* state.	Internal Timer Thread is paused. AMPS Trader Thread is stopped.
Stopped	The last session in the lineup has finished running while the simulation is in *End of Session* state. **OR** The user clicks the "End" button or exists the system while in *Running* or *Paused* state.	Internal Timer Thread and AMPS Trader Thread are stopped. Emotion Engine and Bid/Ask Engine are invoked to load rule sets for the first session in line. Time and session display on Simulation UI are reset to show the first session in line.

* Time and session display on Simulation UI are running (updating) based on the clock tick incremented by Internal Timer Thread. Thus, starting, pausing, or stopping Internal Timer Thread has the same effect on the self-updating display components. Note that when paused, the displays can be asked to resume running at a later time, but when stopped, the displays are reset to default values and cannot be resumed.

Table 3-2: Simulation States, Entry Conditions, and Consequences

3.2.3 Process and Data Flow

While the simulation is running, Internal Timer Thread and AMPS Trader Thread collectively drive the following process flow at every time step.

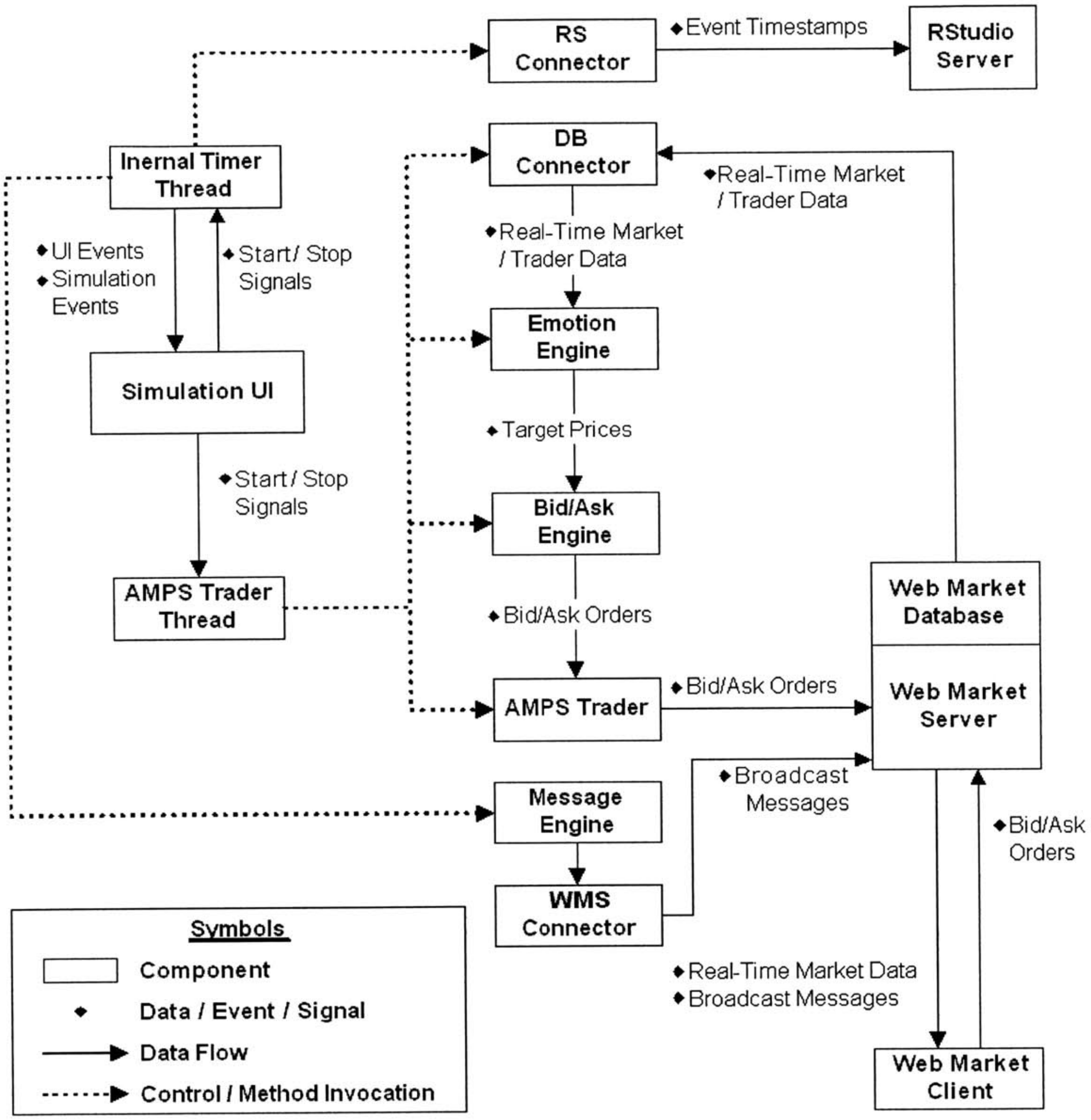


Figure 3-3: Simulation Process and Data Flow

At each time step, Internal Timer Thread and AMPS Trader Thread invoke the business logic components to perform their respective functions in the correct order. DB Connector retrieves real-time market and trader information from the Web Market database. Emotion Engine takes this real-time data as input, performs computation based on the emotion rule set, and dynamically generates a new target price designed to elicit the desired emotional state in the trader. Based on this target price, Bid/Ask Engine generates new bid and ask orders to move the actual market price toward the target price. AMPS Trader then submits the final orders to the Web Market server. Message Engine provides Internal Timer Thread with the message delivery times, and if a message needs to be delivered in the current time step, Internal Timer Thread invokes Message Engine to deliver the messages to the Web Market server. As the simulation progresses, selected simulation events including session start and session stop are also logged to RStudio and the local log file through RS Connector.

The workflow presented above describes activities that are performed during a trading session, one of the two types of sessions in which the trader is asked to participate in active trading and price patterns are continuously generated by AMPS (for more details on sessions, see Section 3.2.1: Structure and Sessions Overview). If the current session is a break session, in which the trader is asked to refrain from trading and price patterns are not generated, only Internal Timer Thread will run while AMPS Trader Thread remains stopped. This thread configuration during the break session ensures that all the user interface displays, message deliveries, and state changes are invoked properly while all price generation processes are stopped.

3.3 AMPS Components

3.3.1 Simulation UI

Simulation UI is the main management console that enables the research administrator to control the simulation system. It abstracts away the difficulties of managing threads and simulation state changes, and enables the user to perform administrative tasks through a simple and intuitive interface. The administrative functions enabled by Simulation UI include controlling, configuring, and monitoring the simulation process; managing connectivity to data logging components including RStudio and the local log file; and modifying configuration files.

Simulation UI is opened as soon as the user starts the AMPS system, and is accessible to the user as long as the system is running. While running, Simulation UI can be in a number of states, shown as follows:

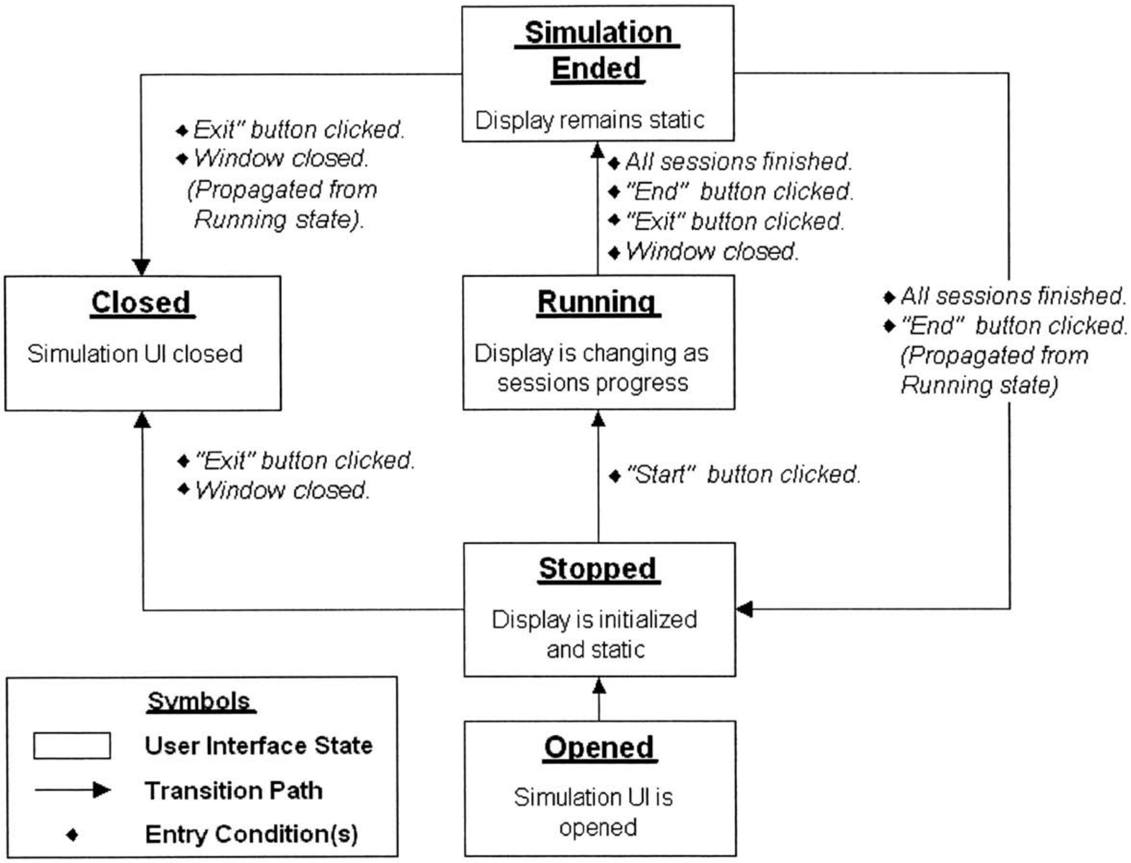


Figure 3-4: Simulation UI State Transitions

Once the configurations have been loaded and system initialization is complete, Simulation UI starts up in the *Stopped* state, in which it displays the sequence of sessions, including the following information for each session: session type, session length, emotion rule set, bid/ask rule set, message file, and target username (for parameter descriptions, see Section 3.2.1: Structure and Sessions Overview). The session display and time display are initialized to reflect the newly loaded sessions. The first session in the linear display sequence is highlighted as the default starting session. For more details on functions, usage, and screenshots of Simulation UI, see Section 5.1: Simulation User Interfaces.

The user interface allows the administrator to control the simulation process by using the "Start", "Pause", and "End" buttons. The administrator can also select the starting session in the sequence by highlighting the session in the sequence displayed.

Once the simulation run is started, the interface goes into *Running* state, in which the sessions are run one after another in the linear sequence displayed. The displays will update themselves to highlight the session in progress as well as show the time left in the current session and the total time remaining. Based on the session currently in progress, Simulation UI starts and stops Internal Timer Thread and AMPS Trader Thread, which in turn invoke processes in the other components to perform the simulation activities. When the end of the sessions or the end of the simulation run has been reached, the interface goes into *Stopped* state and reinitializes its displays to the default state before the run.

At any point in time while the sessions are running (user interface is in *Running* state), the user can stop the run by ending the simulation using the "End" button. Exiting the system, either by using the "Exit" button or by simply closing the window, also terminates the current run. If the run is stopped without exiting the system, the user interface reinitializes the sessions and display (goes into *Stopped* state). If the run is stopped because of system exit, the user interface is closed (goes into *Closed* state) after terminating the current simulation.

Another functionality provided by the user interface includes the ability to reload configuration files and reinitialize the system. This feature, accessible using a "Reload Configurations" button, is useful for dynamically updating the system configurations after AMPS is started, while the simulation is not running. The user interface also provides a "Configurations" button, which launches the AMPS Configuration Editor, a simple configuration file editor with basic text editing and windowing functionality. The Editor is provided as a convenience feature to allow quick modifications to system configurations during runtime. For screenshots or details on the usage of Configuration Editor, see Section 5.1.3: AMPS Configuration Editor.

Simulation UI has a second viewable page that allows the user to monitor, configure, and test the status of data logging connections to RStudio and the local log file. The page displays current connection settings used as well as the real-time connection status for RStudio connection and for the log file. Simulation UI is registered as a listener for connection state changes in the RStudio connection (for details on event listener model, see Section 3.3.7: RS Connector), and it updates the display of RStudio connection status dynamically. In addition, a "Connect" button, a "Reconnect" button, and a "Test" button are provided to allow the user to connect, reconnect, and test the connection to RStudio, respectively.

3.3.2 Emotion Engine

Emotion Engine generates target prices designed to move the current security price in a way that elicits emotional states in the research subject. Target price computation is based on mathematical expressions referred to as emotion rules, which are programmed by the user prior to running the simulation. Emotion Engine manages the set of emotion rules for each session, and provides methods to run them and perform the target price computation.

At the start of each trading session, Simulation UI, driven by Internal Timer Thread, initializes Emotion Engine by invoking the following methods:

```
public boolean loadEmotionFileByName(String filename)[2]
public void setEmotionTargetUser(String username)
```

The method `loadEmotionFileByName()` initializes the session's emotion rule set for target price computation. The second method, `setEmotionTargetUser()`, initializes the target of emotional influence by configuring the Web Market login name of the trading subject.

During each time step of the session, Emotion Engine invokes DB Connector to query the Web Market database for updated values for Market Variables, the set of data variables representing the current status of the market and user (for more details on Market Variables, see Section 4.4.2: Variables). Market Variables include time, price, volume, and type of the latest order submitted by the trading subject. Using these real-time values, Emotion Engine runs the emotion rule set for the current session and computes the target price dynamically. The target price is stored in a system variable (Target Price Variable, denoted by '**P**'), to be picked up later by Bid/Ask Engine for further processing. The series of activities to update the target price is started by the following method:

[2] The Boolean return value indicates whether the rule set loading process has completed without errors.

public boolean `updateNextOrder()`[3]

Emotion rule sets are ASCII text files identified by their filename extension – *.emo*. The name of the emotion rule set associated with each session is specified in the system configuration file (for more details, see Section 4.3: System Configurations). These rule sets are composed using a simple scripting language designed for AMPS. The scripting language allows the user to develop emotion rules using basic logic constructs (for example, if-then statements), a series of AMPS-specific mathematical functions, and a variety of basic mathematical functions and operators. The set of functions and operators as well as the tools to parse and evaluate them are provided by an open source mathematical expression parser package named JEP (Java Mathematical Expression Parser)[4]. While the JEP functions and the custom AMPS functions are available to the user for composing emotion rule sets, Emotion Engine also uses these libraries to process the emotion rule sets. The syntax, variables, and functions available for scripting emotion rule sets are explained in more detail in Section 4.4: Using Emotion and Bid/Ask Rule Sets.

A sample emotion rule set is provided below.

```
if (LTYPE==1) {
PP_temp = PP*0.98;
}
if (LTYPE==2) {
PP_temp = PP*1.02;
}
if ((LTYPE==3) && (LPRICE < PP)) {
PP_temp = PP*1.01;
}
if ((LTYPE==4) && (LPRICE > PP)) {
PP_temp = PP*0.99;
}
if (1==1) {
PP = P;
p_hat = max(p_min, PP_temp * (e^((mu-0.5*sigma_p*sigma_p)*dt) +
sigma_p*sqrt(dt)*randn()));
P = max(p_min, Ap*p_hat + Bp*p_hat*p_hat + Cp*e^(p_hat));
```

[3] The Boolean return value indicates whether the target price computation has completed without errors.

[4] JEP (Java Mathematical Expression Parser), developed and maintained by Nathan Funk at Singular Systems, is an open source Java API for parsing and evaluating math expressions. See [3].

```
}
```

Figure 3-5: Sample Emotion Rule Set: *sample_emotion.emo*

As demonstrated in the example above, Market Variables '**LTYPE**' and '**LPRICE**', denoting last order type and last order price respectively, are key inputs used in the computation of Target Price Variable, '**P**'. In this simplified rule set, the target price '**P**' is adjusted downward if the subject's last order is a market buy (denoted by '**LTYPE==1**'), or if the last order is an ask where the asking price is greater than the previous target price (denoted by '**(LTYPE==4) && (LPRICE > PP)**', where '**PP**' represents the previous target price)[5]. The target price is adjusted upward if the subject's last order is a market sell ('denoted by '**LTYPE==2**'), or if the last order is a bid where the bid price is less than the previous target price (denoted by '**(LTYPE==3) && (LPRICE < PP)**', where '**PP**' represents the previous target price). The overall effect of this rule set is to elicit, without loss of generality, a negative emotion in the subject by adjusting the market price in a way that decreases the subject's portfolio value in the short-term and makes it difficult for the subject to complete limit orders (bids and asks) at the desired prices.

The variables in an emotion rule set are initialized in a separate file under the same name as the rule set but with a different filename extension of *.ini*. For a sample of the emotion rule set configuration file, see Appendix E: Sample Emotion and Bid/Ask Rule Sets. Both emotion rule sets and their corresponding configuration files are located in the emotion file folder specified in the system configuration file, allowing AMPS to locate them during runtime.

3.3.3 Bid/Ask Engine

Bid/Ask Engine generates actual buy and sell orders as well as bid and ask prices to move the market price based on the target price computed by Emotion Engine. Similar to Emotion Engine, Bid/Ask Engine uses mathematical expressions referred to as bid/ask

[5] In Figure 3-5, the final target price '**P**' is computed based on the value of '**PP_temp**', a temporary variable based on the value of the previous target price '**PP**'. By adjusting the value of '**PP_temp**' upward or downward, we effectively adjust the value of '**P**' upward or downward from its previous value.

rule sets for its order price computation. Bid/Ask Engine manages the set of bid/ask rules for each session, and provides methods to run them and perform the computation. Bid/Ask Engine is necessary on top of Emotion Engine in order to create a realistic trading experience for the subject that is close to or indistinguishable from actual trading with other competing traders in the Web Market system. Instead of adjusting the current security price directly to the new target price computed by Emotion Engine, Bid/Ask Engine allows AMPS to adjust the security price in a way that more closely resembles how prices are normally moved in a securities market – through the matching and fulfillment of bids and asks around the current market price.

At the beginning of each trading session, Bid/Ask Engine is initialized by invoking the following method:

public boolean `loadBidAskFileByName(String filename)`[6]

The call updates Bid/Ask Engine with the bid/ask rule set associated with the new session.

At each time step after Emotion Engine generates the new target price, Bid/Ask Engine runs the session-specific rule set to compute and assign new values to Order Variables, the set of system variables whose values collectively specify an actual order (for more details on Order Variables, see Section 4.4.2: Variables). These variables include bid price, ask price, volume, and order type. At the end of each time step, the updated Order Variables are picked up by AMPS Trader to create and place the new order in the Web Market system. If Order Variables are not modified during the time step, they default to their values from the previous time step. Similar to Emotion Engine, the series of activities performed by Bid/Ask Engine is started by the following method:

public boolean `updateNextOrder()`[7]

Bid/Ask rule sets are stored in ASCII text files identified by their filename extension – *.baa*. The name of the bid/ask rule set associated with each session is specified in the system configuration file (see Section 4.3: System Configurations). Similar to the emotion rule sets, bid/ask rule sets are composed using the same scripting language, and

[6] The Boolean return value indicates whether the rule set loading process has completed without errors.
[7] The Boolean return value indicates whether the Order Variable computation has completed without errors.

the same set of mathematical functions is also available from the library of custom AMPS functions and the JEP package (see Section 3.3.2: Emotion Engine). Like Emotion Engine, Bid/Ask Engine also uses the two libraries in processing bid/ask rule sets. The syntax, variables, and functions available for scripting bid/ask rule sets are the same as those for emotion rule sets, and they are explained in Section 4.4: Using Emotion and Bid/Ask Rule Sets.

To illustrate, the main sections of a sample bid/ask rule set are provided below with some lines removed for better clarify (for the complete file, see Appendix E: Sample Emotion and Bid/Ask Rule Sets).

```
if (1==1) {
SP = S;
BIDP = BID;
ASKP = ASK;
V_vec = vecAddToSize(V_vec, num_vlags, V);
}
if (vecAvg(V_vec) > 0) {
inv_volume = 1/vecAvg(V_vec);
S = max(dp, SP*(As*dp + Bs*inv_volume + sigma_s*sqrt(dt))*randn());
}
if (vecAvg(V_vec) <= 0) {
S = max(dp, SP * (As*dp + sigma_s*sqrt(dt))*randn());
}
if (1==1) {
V = round_lot(Av*abs(randn()),dv);
p_cur = P;
BID = round_to_tick(p_cur-0.5*S,p_cur,dp,1);
ASK = round_to_tick(p_cur-0.5*S,p_cur,dp,0);
}
if (BID >= ASK) {
rand_var = randn();
}
if ((BID >= ASK) && (rand_var >= 0)) {
BID = ASK - S;
}
if ((BID >= ASK) && (!(rand_var >= 0))) {
ASK = BID + S;
}
```

Figure 3-6: Sample Section of a Bid/Ask Rule Set: *sample_bidask.baa*

In the example above, Order Variables '**BID**' and '**ASK**', denoting bid price and ask price respectively, are computed during the run based on the new target price '**P**', which has been computed by Emotion Engine before Bid/Ask Engine started processing. Order Variable '**V**', denoting order volume, is also assigned a value here, although not directly based on the value of '**P**' in this particular example. The general rule for the reader to note is that all Order Variables should be updated in the bid/ask rule set, using Target Price Variable '**P**' and other user-defined variables as input.

Similar to variables in emotion rule sets, those in a bid/ask rule set are initialized in a separate file under the same name as the rule set but with a different filename extension of *.ini*. For an example of a bid/ask rule set initialization file, see Appendix E: Sample Emotion and Bid/Ask Rule Sets.

Both bid/ask rule sets and their corresponding configuration files are located in the bid/ask file folder specified in the system configuration file, enabling AMPS to locate them during runtime.

3.3.4 Message Engine

Message Engine manages system messages and market event messages that are delivered to the trader at specific times during the session. This messaging mechanism is designed to aid the procession of the simulation (for example, by notifying the trader of session start and stop) as well as to contribute to the desired emotional impact on the human trader (for example, by delivering news on the traded security).

Each session has an associated message file specifying the set of messages for the session. The name of the message file corresponding to each session is specified in the system configuration file. The message file is an ASCII text file identified by its filename extension – *.msg*. The file contains one or more messages, including the delivery time during the session and the message text to be displayed. The delivery time specified in a message file can be relative to the start or the end of the session, depending on the sign of the number. A positive number is interpreted as a time relative to the start of the session, and a negative number is interpreted as a time relative to the end of the session. This syntax using relative time enables the user to combine message files with sessions of different length without having to change the delivery time specifications in the files. For more details on the usage of message files, see Section 4.5: Using Message Files.

Each message is maintained in a separate instance of a lower level data structure, `TimedMessage`. `TimedMessage` class provides the methods to initialize and access the message text and delivery time of the underlying message. The data structure is designed to enhance modularization and data encapsulation of the message process, such that the message storage format can change in the future without affecting the rest of Message Engine.

At the start of a session, Message Engine is initialized by the following method call:

```
public boolean loadMessageFileByName(String fileName,
                                     int sessionLength)[8]
```

The method above loads the messages in the file referenced by the given filename, and computes the message delivery times based on the relative time specifications in the file and the input session length provided. It then queues up the messages ordered by delivery time from the start to the end of the session.

Internal Timer Thread helps Simulation UI keep track of when to deliver messages by maintaining the next delivery time and checking whether the delivery is due at each time step. At the start of the session after Message Engine has been initialized, Internal Timer Thread retrieves the delivery time of the first message by invoking the following method provided by Message Engine:

```
public int getNextMessageTime()
```

At every time step, the timer thread checks whether the current session time matches the incoming delivery time. If it is time for delivery, the following method is invoked to send the message:

```
public boolean sendNextMessage()[9]
```

Immediately after calling `sendNextMessage()`, the timer thread calls `getNextMessageTime()` to update the next delivery time to the delivery time of the

[8] The Boolean return value indicates whether the message loading process has completed without errors.
[9] The Boolean return value indicates whether the message delivery has completed without errors.

following message. The reader should note that sendNextMessage() not only delivers the next message in the queue but also increments the index of the message queue such that the next time getNextMessageTime() or sendNextMessage() is called, the actions invoked will refer to the next undelivered message in the queue.

Web Market server has an existing feature that allows administrators to send broadcast messages to one or all traders currently logged on, and Message Engine utilizes this function for its message delivery. However, because the administrative function is not accessible through the Web Market client or the API for machine traders, Message Engine leverages WMS Connector to access this feature through a Java RMI (Remote Method Invocation) connection to the server. For more details on WMS Connector and RMI, see Section 3.3.8: WMS Connector.

3.3.5 AMPS Trader and AMPS Trader Thread

AMPS Trader maintains the connection with Web Market server and submits the orders generated by Bid/Ask Engine. At the start of the simulation, it registers with the Web Market server as a machine trader with the same trading privileges as a normal human trader. During a trading session, AMPS Trader submits orders on behalf of AMPS using the machine trader API. This design allows AMPS Trader to move prices by submitting orders in the same way as the subject or any other market participant, creating a more realistic trading experience for the AMPS subject because security prices are moved in the same manner as in a normal run of Web Market without AMPS.

AMPS Trader works in conjunction with AMPS Trader Thread, a timer process that governs the trading frequency. Controlled by the administrator through Simulation UI, AMPS Trader Thread continuously invokes AMPS Trader methods to initiate trades at each simulation time step.

3.3.6 DB Connector

DB Connector is AMPS's interface to the Web Market database. It maintains a JDBC (Java Database Connectivity) connection to the database, and enables AMPS to query for real-time trader and market information at each time step. The supported queries include latest bid price, ask price, sale price, and orders placed by the subject trader, as well as the current portfolio of the trader. The Connector embodies the SQL (Structured Query Language) statements pertaining to the specific query based on the table structures of the Web Market database, and it provides methods to submit these queries and retrieve the target data.

In addition, DB Connector provides methods and corresponding structured queries for the AMPS database access scripts to retrieve Web Market data. These data scripts are developed as a supplement to the simulation system to help the research administrators study the market and trader data after the experiment (for more details on database access scripts, see Section 4.6: Using Database Access Scripts).

3.3.7 RS Connector

RS Connector manages and monitors the connection to RStudio, a real-time physiological data collection system (for more details, see Section 3.4.2 RStudio). It allows AMPS to send timestamps of selected simulation events to RStudio, which logs the timestamps along with other physiological data for postmortem analysis. Upon connecting, RS Connector opens a byte stream over a TCP/IP socket connection to the listening RStudio server. Once the connection is established, AMPS components can invoke methods in RS Connector to send time signals to RStudio. If the initial connection attempt is unsuccessful or the connection is interrupted for any reason during the simulation, RS Connector provides the ability to retry the connection, once every given timeout period. The timeout period can be configured in the system configuration file.

A time signal is essentially a byte of data delivered to RStudio over the byte stream. Upon receiving the byte of data, RStudio logs the time of its arrival. In the current prototype, the data logged by RStudio is limited to the event timestamps and contains no event descriptions.

RS Connector keeps the other AMPS components (and thus the administrator) informed of the status of the connection to RStudio using an event notification model based on the Observer Pattern[10]. The model enables relevant AMPS components to dynamically register as observers, or listeners, to be notified of changes in the state of the connection. The observable component, RS Connector, maintains a dynamic list of listeners for its state changes, and delivers real-time event notifications to these listeners. The state change events include connection establishment, connection drop, and connection attempt or retry in progress. The design allows AMPS components, such as Simulation UI, to keep informed of connection state changes and respond accordingly, for example, by changing the display and notifying the administrator, redirecting event notifications to other relevant components, and starting or stopping the delivery of time signals to RStudio.

RS Connector also manages the connection to a local log file and provides methods to log simulation events in the file. Designed as a supplement and backup to RStudio logging, the local file logging mechanism provides additional redundancy in timestamp logging. For more details or a sample of the local log file, see Section 4.7: RS Connector Local Log File).

3.3.8 WMS Connector

WMS Connector manages a Java RMI (Remote Method Invocation) connection to the Web Market server components, and provides methods to access the server's backend functions that are available only through the Web market server user interface[11]. Specifically, RMI allows WMS Connector to dynamically obtain references to Web Market server components, and to invoke methods on the remote components to perform the desired administrator functions. For more details on RMI and its usage, see [12].

WMS Connector allows Message Engine to send broadcast messages either to an individual trader or to all traders in the market. Depending on the delivery target, WMS

[10] Observer Pattern is a software design pattern that models a one-to-many dependency between objects so that when an object changes state, all its dependents are notified and updated automatically [4].

[11] AMPS is designed to minimize dependencies on and changes to the Web Market source code, in order to keep AMPS development effort independent from that of Web Market and to limit the scope of the prototype. As a result, AMPS is not fully integrated with Web Market and cannot directly invoke Web Market's backend functionality available only to its server administrator.

Connector calls upon one the following two methods in Web Market's ServerMain class:

```
public static void sendAsyncNotification(int clientId,
                                          Serializable obj)
public static void sendAsyncNotification(Serializable obj)
```

To deliver a broadcast message, WMS Connector simply invokes one of the methods above and passes to it the message in String format.

As of the writing of this thesis, the development of WMS Connector has not been completed because the Web Market source code is undergoing a major revision and stable code for some crucial components is not available. For the time being, WMS Connector displays all broadcast messages on screen for testing and debugging purposes.

Armed with the ability to make remote method calls, WMS Connector opens up a large variety of Web Market backend functions that are normally unavailable through the Web Market client or the machine trader API. In this initial prototype of AMPS, the use of WMS Connector is limited to the delivery of broadcast messages. In future development, however, WMS Connector can be a promising interface for accessing Web Market's server-side functionality including market price adjustments and database queries.

3.4 External Components

3.4.1 MIT Web Market

Web Market is a web-based securities market simulation system developed at MIT for research in financial engineering and autonomous agents. It is a Java-based, client-server system that enables human and machine traders to log in and trade with each other in a simulated double-auction market. The system provides the human traders with a web-based client, or user interface, to monitor the market, track trader portfolios, and submit orders. The system also provides API's to allow machine traders, or artificially programmed trading agents, to participate in the market simulation. All transactions, portfolios, and other market information are logged to the Web Market database.

AMPS is integrated with Web Market at three different levels: server, client, and database. The system uses Web Market's API's for machine traders to interact with the market server – it registers with the server as an autonomous trading agent and submits orders using the machine trader API. At this level, AMPS acts as a market maker with unlimited purchasing power, and moves the market price through the submission of a large number of orders around the market price. At the client level, the human subject in an AMPS simulation is asked to use the Web Market client interface to participate in the market. All trades placed by the human subject are thus directly submitted to the market server and indirectly logged to the Web Market database. Programmed text messages delivered by Message Engine are also displayed through the client interface. Lastly, AMPS connects to the market database to retrieve real-time market and trader information for order generation as well as for port mortem analysis by the researcher administrator.

The design decision to leverage Web Market in the above manner is based on a number of considerations. First, to maximize the precision and extent of the emotional impact on the trading subject, AMPS must have direct and timely control over the price movements. On the other hand, in order to allow the user to experience realistic market movements, prices should be changed only through the normal order submission and matching process, as opposed to absolute and brute-forced adjustments. Both objectives are achieved by allowing AMPS to leverage the machine trader API and submit orders utilizing the same functions a normal trader would. AMPS can generate the desired price movements quickly and quite precisely by submitting a series of orders around the target price, and the price movements would appear natural and realistic to the subject because they are the result of the market's normal order matching and fulfillment processes. Further, AMPS subjects are asked to use the existing web-based interface in order to maximize platform independence in hardware and experiment location. The approach also reduces duplicate development effort. Lastly, data is retrieved directly from the Web Market database, independent of the market server, in order to allow faster retrieval and greater flexibility in query structures.

3.4.2 RStudio

RStudio is a data collection system developed to aid in the research of risk-taking physiology by providing tools to monitor and analyze the physiological states of securities traders. It collects and displays real-time physiological data feed and market events data from a variety of data sources, ranging from portable biofeedback data collection equipment to software simulation systems. The data collected enables researchers to analyze the correlations between market events and physiological characteristics that may be indicative of emotional state changes.

One of the design goals of AMPS is to integrate with RStudio and deliver timestamp signals of selected simulation events, including simulation start and stop, subject trade submissions, and subject key presses. RS Connector manages the connection to RStudio, keep related AMPS components informed of the connection status, and handle the delivery of event signals.

Chapter 4

General Usage and Configurations

This chapter discusses the functional specifications of AMPS and the relevant components of Web Market from an end user's perspective. It covers the usage model and configurations of AMPS and Web Market features, and provides specific instructions on performing the tasks required for a complete simulation, serving as a user manual and a guideline. The usage of AMPS user interfaces is explained in detail in Chapter 5: AMPS User Interfaces.

4.1 General Usage Model

For the administrator, running an AMPS experiment generally comprises of a number of high-level tasks, outlined as follows:

Phase	***Task / Description***	***Interface / Tool***
Configurations ***(Prior to Run)***	Set up session sequence and configure other system variables in *amps.ini.* Set up and configure emotion processes in *<emotion file>.emo* and *<emotion file>.ini.* Set up and configure bid/ask processes in *<bid/ask file>.baa* and *<bid/ask file>.ini.* Set up and configure message processes in *<message file>.msg.*	AMPS Configuration Editor or A general-purpose text editor

Simulation Control *(During Run)*	Control and monitor simulation and session processions. Load / Reload configuration files (if changed after system startup). Connect / Reconnect to RStudio or local log file.	Simulation UI
Analysis *(After Run)*	Retrieve market and trader data from Web Market database for analysis.	AMPS Database Access Scripts

Table 4-1: AMPS High-Level Administrator Tasks

The following sections discuss the tasks before and after the run. The administrative tasks during the experiment run are primarily performed through Simulation UI, and are thus covered in Chapter 5: AMPS User Interfaces.

4.2 System Startup and Exit

AMPS works in conjunction with Web Market, and starting the Web Market server is a prerequisite to starting AMPS (for details on starting Web Market server, see Section 4.8: Web Market Startup and Exit). Once the Web Market server is up and running, AMPS and Simulation UI can be started simultaneously from the command line with the following "Java" command:

```
java rst.amps.AMPSMain <server name> <port number>
```

For Unix and Linux platforms, a run script named *runamps.sh* located in the AMPS application directory is provided to for the user's convenience. It runs the command above using default parameters. For details on system requirements for running or installing AMPS, see Appendix F: AMPS System Requirements.

To exit AMPS, the user can either close the Simulation UI window or click the "Exit" button (for details on Simulation UI, see Chapter 5: AMPS User Interfaces). Exiting AMPS while a simulation is in progress will terminate the current simulation.

4.3 System Configurations

At startup, AMPS initializes itself using a system configuration file named *amps.ini*, which contains configuration parameters for sessions, RS Connector, DB Connector, as well as paths to other configuration files. For a sample *amps.ini* file or a comprehensive list of system configuration parameters, see Appendix D: System Configuration File and Parameters.

4.3.1 Sessions Configurations

A crucial section of the system configuration file is the configuration of simulation sessions. Sessions are the building blocks of an AMPS simulation – a run of the simulation involves executing the sessions in a linear order (for more details on sessions, see Section 3.2.1: Structure and Sessions Overview). Each session is a finite time period associated with a set of parameters specifying the activities that should be carried out during the session. The sequence of sessions as well as individual session parameters are specified in the sessions configuration section of the master configuration file, as shown below:

```
NUM_SESSIONS=7
SESSION1=trading 10 trading_default.msg CAPM.emo default.baa lwang
SESSION2=trading 30 trading_default.msg pos_affect_20s.emo
default.baa lwang
SESSION3=break 10 break10sec.msg
SESSION4=trading 300 trading300sec.msg distress_20s.emo high_vol.baa
lwang
SESSION5=break 10 break10sec.msg
SESSION6=trading 60 trading60sec.msg arousal_20s.emo high_freq.baa
lwang
SESSION7=break 10 break_final.msg
```

Figure 4-1: Sample Section of *amps.ini*: Sessions Configurations

In the sample section above, there are seven sessions lined up in the run sequence, specified by the NUM_SESSIONS parameter. Each line starting with SESSION<*number*>

specifies a single session, and the <*number*> field indicates the session's order in the run sequence. Each session declaration should be a separate line by itself, and should specify, at a minimum, three required parameters. With respect to the order they appear in the system configuration file, from left to right, the three required parameters are:

Session Type – Specifies whether the session is a "Trading" session, in which the human subject is asked to participate in active trading, or a "Break" session, in which the subject is asked to refrain from trading.

Time Length – Specifies the amount of time in seconds that the session should be run.

Message File – Specifies the name of the file containing the set of broadcast messages and the times to deliver them during the session (for details, see Section 3.3.4: Message Engine).

Additionally, trading sessions must also have the following parameters:

Emotion File – Specifies the name of the file containing the emotion rule set for this trading session.

Bid/Ask File – Specifies the name of the file containing the bid/ask rule set for this trading session.

Target User – Specifies the username of the experiment subject in Web Market. This is used to retrieve real-time trader information from the Web Market database.

4.3.2 File and Directory Path Configurations

Another section of *amps.ini* specifies the directories or paths under which the emotion rule sets, bid/ask rule sets, message files, and RStudio log file can be found. These parameters allow the administrator to change the default location of these configuration files and log files, and enable the system to dynamically locate the files during runtime. The figure below shows the section of a sample file that specifies the file paths:

```
EMOTION_FOLDER_PATH = /config/emotions
BIDASK_FOLDER_PATH = /config/bidask
MESSAGE_FOLDER_PATH = /config/messages
RS_LOGFILE_PATH = /tmp/rstudio.log
```

Figure 4-2: Sample Section of *amps.ini*: Path Configurations

Note that, by design, these paths should be specified relative to the user's current working directory, or the directory from which the administrator started AMPS.

4.3.3 RS Connector and DB Connector Configurations

One section of the system configuration file allows the user to configure the parameters for RS Connector and DB Connector. Some of these parameters specify the information required to establish the connections, and others provide the user with configurable options, including automatic connect, retry delay, and log file backup for RStudio Connection.

The figure below shows the section of a sample file that specifies these configurations:

```
RS_HOSTNAME=18.193.0.69
RS_PORT=7777
RS_RETRY_DELAY=5
RS_AUTO_CONNECT=0
RS_LOG_TO_FILE=1
RS_LOG_TO_RS=0
. . .
DB_URL = jdbc:oracle:thin:@wang:1521:OMKT
DB_USERNAME = dummy_user
DB_PASSWORD = dummy
```

Figure 4-3: Sample Section of *amps.ini*: Connectors Configurations

4.4 Using Emotion and Bid/Ask Rule Sets

Emotion rule sets and bid/ask rule sets are critical components of the simulation used to generate emotion-inducing price patterns. Emotion rule sets are used to compute target prices designed to influence trader emotions, and bid/ask rule sets are used to generate actual orders based on the target prices.

4.4.1 Rule Set Structure and Syntax

The structure of emotion and bid/ask rule sets composes of a series of if-then constructs, and the statements are executed in the sequential order they appear in the rule set file, downward from the top. The syntax allows a one-to-many relationship between the if-clause (the predicate) and the then-clause, and the then-clauses within the same if-then statement are executed in the sequential order they appear in the statement, downward from the top. To illustrate, consider the following example:

```
if (<if-clause A>) {
<then-clause A1>;
<then-clause A2>;
          .
          .
}
if (<if-clause B>) {
<then-clause B1>;
<then-clause B2>;
          .
          .
}
```

In the example above, ***<then-clause A1>***, ***<then-clause A2>***, and all subsequent then-clauses within the enclosing curly brackets will be executed in the order they appear, if ***<if-clause A>*** is evaluated to be true. Afterwards, ***<if-clause B>*** is evaluated, and depending on the evaluation result, ***<then-clause B1>***, ***<then-clause B2>***, and all subsequent then-clauses within the enclosing statement will be executed.

The complete set of semantic rules for the statements and expressions in rule sets are summarized as follows:

- Each if-then statement must begin with ‘if’, followed by the predicate enclosed within a pair of parenthesis, ‘ (‘ and ‘)’, and end with a set of zero or more then-clauses enclosed within a pair of curly brackets, ‘ {‘ and ‘ }’.
- The predicate of each if-then statement must be a Boolean expression that is evaluated to be either *True* or *False*. Exactly one conditional operator should be used within each expression, but multiple expressions can be grouped together into compound expressions using Logical AND (‘&&’) and Logical OR (‘||’). Conditional operators allowed include the relational operators (‘<’, ‘>’, ‘<=’, ‘>=’) and the equality operators (‘==’, ‘ !=’). The use of the assignment operator (‘=’) is not allowed within the predicate.
- Each then-clause of an if-then statement must contain exactly one assignment operator (‘=’), and end with a semi-colon, ‘ ;’.
- In compound expressions involving more than one operator or function, the order of evaluation is dictated by the parentheses, with expressions in the inner most pair of parentheses evaluated before outer pairs. For example, in the expression ‘ (a*(b-c))’, ‘ (b-c)’ is evaluated before everything else. Explicit use of parentheses is the recommended practice for readability and maintainability of the scripts.
- Compound expressions without parentheses are evaluated based on the operator precedence order below. The operators are listed in order or precedence from top to bottom, where the higher in the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with a relatively lower precedence. Operators with same precedence are evaluated from left to right, except the assignment operator, which is evaluated right to left.

Operator Type	*Symbols*
Unary	+X, -X
Multiplicative	*, /, %
Additive	+, -
Relational	<, >, <=, >=
Equality	==, !=

Logical AND	&&
Logical OR	||
Assignment	=

Table 4-2: Rule Set Operator Order of Precedence

For samples of correct rule set syntax, see Appendix E: Sample Emotion and Bid/Ask Rule Sets.

4.4.2 Variables

The emotion rule set and the bid/ask rule set share a common set of variables, whose values persist throughout the session. Each variable is a real number whose value can range between ±1.7976931348623157E+308, with a decimal precision of ±4.9E-324[12].

Each emotion rule set and bid/ask rule set has a corresponding initialization file under the same name as the rule set file, but with a different extension of *.ini*. The initialization file specifies the default values of the rule set variables. Any variable not initialized in the *.ini* files will have a default value of zero at the start of the session. Although the system does not explicitly require that all variables be declared in their corresponding initialization files, the recommended practice is to declare the variables in the *.ini* file corresponding to the rule set file that uses the variables – variables used in the emotion rule set should be declared in the *.ini* file for the emotion rule set, and variables used in the bid/ask rule set should be declared in the *.ini* file for the bid/ask rule set.

The variables available in the rule sets fall into several categories, as described below.

Market Variables are system-defined variables that provide real-time data on market status and trader actions. These variables are updated by AMPS with real-time values from the market database at every time step. Market Variables are utilized in emotion rule sets to generate emotion-inducing price patterns. The set of Market Variables is pre-

[12] Each variable is a 64-bit, double-precision floating-point value that ranges from ±4.9E-324 (decimal precision) to ±1.7976931348623157E+308 (absolute maximum/minimum). The variable adheres to the IEEE 754-1985 standard, which specifies both the format and arithmetic behavior of the real number.

determined, static, and persistent, and their names are reserved names that cannot be used for User Variables.

The names and descriptions of Market Variables are outlined as follows:

Market Variable	*Description*
LTIME	Time of last order placed by the target user, in number of milliseconds.
LTYPE	Type of last order placed by the target user, where 1=*Market Buy*, 2=*Market Sell*, 3=*Bid*, and 4=*Ask*.
LPRICE	Price of last order placed by the target user, in number of ticks. *
LVOL	Volume of last order placed by the target user, in number of shares.

* The minimum tick used in Web Market is 1/16, or 0.0625 of a currency unit. For example, 160 ticks are equivalent to 10 currency units (i.e. dollars).

Table 4-3: Market Variables and Descriptions

Order Variables are system-defined variables whose values collectively describe the order to be placed by AMPS at the end of each time step. Order Variables are primarily modified in bid/ask rule sets based on the Target Price Variable updated by Emotion Engine. At the end of each time step, their values are utilized by AMPS Trader to create the new order for submission. As system variables, the set of Order Variables is predetermined, static, and persistent, and their names are reserved names that cannot be used for User Variables.

The names and descriptions of Order Variables are outlined as follows:

Order Variable	*Description*
BID	Bid price of the new order for the current time step, in number of ticks. See * in Table 4-3 for details on ticks.
ASK	Ask price of the new order for the current time step, in number of ticks. See * in Table 4-3 for details on ticks.
V	Volume of the new order for the current time step, in number of

	shares.
R	Flag indicating whether or not to place an order this time step (1=yes, 0=no). If R=1, AMPS randomly selects between bid and ask order.

Table 4-4: Order Variables and Descriptions

Target Price Variable is a system-defined variable denoted by '**P**', which represents the emotion-inducing price target for the current time step. Its value is updated by Emotion Engine based on Market Variables, and is used by Bid/Ask Engine to update Order Variables. The value of '**P**' should be in terms of the number of minimum ticks used in Web Market, which is 1/16 of a currency unit[13]. As a system variable, its name '**P**' is a reserved name that cannot be used for User Variables.

User Variables are variables defined by the user for general purposes and for convenience in rule set scripting. These variables can be dynamically declared by the user any place within a rule set, and their values can only be modified by the user within the rule set. For these reasons, User Variables are commonly used as time counters, indices, and temporary data variables for the user's convenience.

JEP Constants are a pre-determined, static, and persistent set of common mathematical constants provided by the JEP package, including '**e**' (2.718281828459045) and '**pi**' (3.141592653589793). They are provided for the convenience of the user. The values of JEP Constants are preset by the JEP package at startup time, and the names of JEP Constants are reserved names that cannot be used for User Variables. For a list of JEP constants, see Appendix B: JEP Operators, Functions, and Constants.

4.4.3 Operators and Functions

[13] The minimum tick used in Web Market is 1/16, or 0.0625 of a currency unit. For example, 160 ticks are the equivalent of 10 currency units (i.e. dollars).

A library of common mathematical operators and functions provided by the JEP package can be used within the rules sets. For a comprehensive list of these functions, see Appendix B: JEP Operators, Functions, and Constants. In addition, a custom library has been developed to provide a number of specific functions required by AMPS users for rule set scripting and price computations. For a list of these custom functions and their descriptions, see Appendix A: AMPS Custom Functions.

4.4.4 Vectors

Vectors are a special category of variables in that many primitive operators are not applicable to them. This means that vector declarations and operations require vector-specific functions. A variety of vector functions have been developed for the convenience of AMPS users, and descriptions and usage of these functions are provided in Appendix A: AMPS Custom Functions.

4.5 Using Message Files

Message files contain messages that are broadcasted to the trading subject during the simulation, and each file contains the set of messages for a single session. Each session can be configured to use a different message file, based on session settings in the system configuration file, *amps.ini*. Message files should be placed in the message file directory also specified in the system configuration file.

A sample message file is as follows:

```
0 Session starting.  Please begin trading.
60 1 minute into session.  10 addition traders will start trading.
-30 Session ending in 30 seconds.
-10 Session ending in 10 seconds.
-1 Session has ended.
```

Figure 4-4: Sample Message File: *sample_message.msg*

Each line of a message file specifies a single message to be delivered. It contains the delivery time in number of seconds followed by the message in plain text. The delivery time is relative to the beginning of the session if it is a positive number; it is relative to the end of the session if it is a negative number. For example, a time specification of '60' indicates a delivery time of 60 seconds after the session begins, and a time specification of '-10' indicates a delivery time of 10 seconds before the end of the session. A time specification of '0' indicates a delivery time at the beginning of the session. Using the sample file above in a 2-minute long session, the resulting message delivery sequence is as follows:

Time in Session	***Message Delivered to Subject***
0	`"Session starting.  Please begin trading."`
60	`"1 minute into session.  10 addition traders will start trading."`
90	`"Trading session ending in 30 seconds."`
110	`"Trading session ending in 10 seconds."`
119	`"Trading session has ended.  Please stop trading."`

Table 4-5: Sample Message Delivery Schedule

The time syntax in relative terms provides the research administrator the flexibility to combine message files with sessions of different time length, without having to adjust the time specifications in the file.

4.6 Using Database Access Scripts

AMPS provides a number of database scripts that enable the research administrator to retrieve market and trader data from the Web Market database. The data retrieved is either displayed on screen or stored locally for analysis or debugging. The list of database access scripts and their functions are as follows:

Database Access Script	***Description***
saveRegUser.sh	Retrieves the login information for all registered users of Web Market, and saves it in the given directory under a file named *regusers.dat*.

Usage: `saveRegUser.sh` *<save dir>*

saveUserData.sh

Retrieves user and portfolio information for all registered Web Market users that have traded since registration, and saves it in the given directory under the following files:

- *users.dat* – ID and username of all users who have traded since registration.
- *portfolios.dat* – Portfolio value and asset positions of all users who have traded since registration.

Usage: `saveUserData.sh` *<save dir>*

saveStockData.sh

Retrieves information for all Web Market securities and saves it in the given directory under the following files:

- *securities.dat* – Security ID, symbol, minimum tick, and last traded price of all securities.
- *orders.dat* – Time, transaction ID, type, trader, security ID, price, volume, and fulfillment status of all orders submitted.
- *trades.dat* – Time, transaction ID, security ID, buyer, seller, volume, and price of all security sales (orders fulfilled).
- *p_<security ID>.dat* - Time, price, and volume of all sales of the security identified by *<security ID>*.
- *q_<security ID>.dat* - Time, volume, and order type of all orders submitted for the security identified by *<security ID>*.

Usage: `saveStockData.sh` *<save dir>*

showLastOrderForUser.sh

Retrieves the last order submitted by the given Web Market user and displays the information on screen. Information displayed includes time, type, volume, and price of the order.

Usage: `showLastOrderForUser.sh` *<username>*

showLastPrices.sh

Displays the last bid price, last bid price, and last sale price for the given security on screen.

Usage: showLastPrices.sh <*security ID*>

Table 4-6: Database Access Scripts and Functions

The scripts are developed as convenience mechanisms to invoke DB Connector methods to retrieve the data from the Web Market database. As C-shell scripts, they must be ported before use on non-Unix or non-Linux platforms.

4.7 RS Connector Local Log File

RS Connector uses a local log file as an alternative and backup to RStudio logging over the network. When local file logging is enabled, the timestamp and description of selected simulation events are appended to the end of the file as the events take place.

A sample local log file is provided below.

```
2003/04/07 21:04:35.710 - Log File Opened.
2003/04/07 21:05:24.765 - Simulation Started.
2003/04/07 21:05:24.765 - Current session: 1
2003/04/07 21:05:34.895 - Current session: 2
2003/04/07 21:05:46.804 - Simulation Paused.
2003/04/07 21:05:48.733 - Simulation Continuing.
2003/04/07 21:05:48.733 - Current session: 2
2003/04/07 21:06:38.236 - Current session: 3
2003/04/07 21:06:49.804 - Simulation Paused.
2003/04/07 21:06:52.413 - Simulation Ended.
2003/04/07 21:13:52.516 - Log File Closing.
```

Figure 4-5: Sample Local Log File: *sample_rstudio.log*

Each line in the local log file is a separate event entry, containing the date and time formatted for better viewing, and a short text description of the event. The sample above shows a variety of simulation events, including simulation state changes ("Started", "Ended", "Paused", and "Continuing"), session changes ("Current session: <*session number*>"), and log file connection status changes ("Opened" and "Closing"). The timestamp and event descriptions in the log file are useful for both postmortem analysis as well as debugging. The timestamps are almost identical to those logged by RStudio, if RStudio logging is also enabled[14].

The administrator can enable or disable logging to the local file, or change the location and name of the log file within the system configuration file (see Section 4.3: System Configurations).

4.8 Web Market Startup and Exit

Web Market server must be up and running in order to start AMPS[15]. Assuming the environment variables and paths are set up correctly[16], the server can be started with the following "Java" command at the command line:

```
java rst.market.ServerMain jdbc:<subprotocol>:<subname> <jdbc driver>
<db username> <db user pwd> <market name> <market port>
```

For Unix and Linux platforms, AMPS provides a simplified run script named *runserver.sh* located in the AMPS application directory for the user's convenience. It invokes a Web Market startup script using default parameters, and the Web Market script in turn runs the "Java" command above.

[14] For the same event, the timestamp recorded by RStudio slightly delayed due to network latency. The delay is typically a small fraction of a second between 50ms to 200ms if the RStudio server is on the same local area network as the AMPS server. This seems tolerable for our current purposes because the length of the delay is relatively consistent and small compared to the periods between critical events, which is our primary interest.

[15] For AMPS to start up, the Web Market server must be up and running, but the market itself may be open or closed.

[16] For instructions on installing and configuring Web Market, see the Web Market documentation.

Chapter 5

AMPS User Interfaces

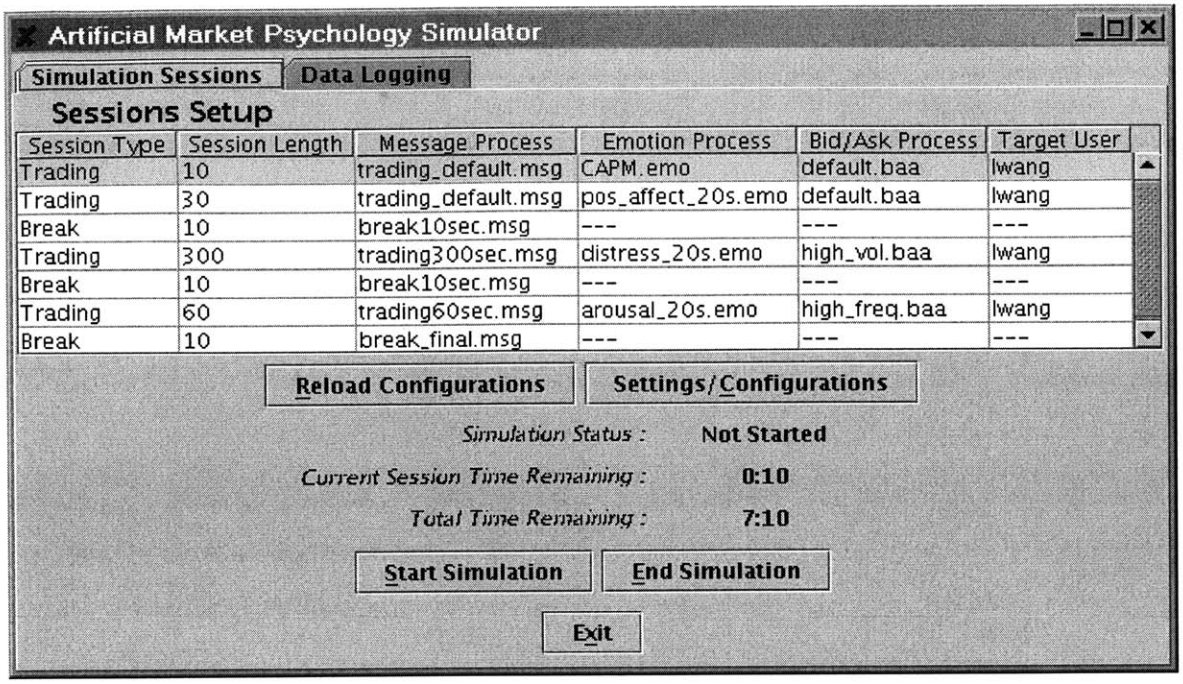
Simulation UI is the main user interface for AMPS administrators. This chapter discusses the functions and usage of Simulation UI and its supplementary component, AMPS Configuration Editor. In addition, Web Market user interfaces and their functions are also summarized, as they work in close conjunction with AMPS user interfaces to provide a complete simulation experience for the administrator and the subject.

5.1 Simulation User Interfaces

Simulation UI is the main graphical interface allowing the user to control the different aspects of an AMPS experiment. The interface is opened upon starting AMPS, and closed when the user exits AMPS. In this prototype of AMPS, Simulation UI has two main pages, or tab panels: Simulation Panel and Data Logging Panel. Simulation UI also allows the user to launch AMPS Configuration Editor, an editing tool for AMPS configuration files. The functions of these interfaces are described in detail below.

5.1.1 Simulation Panel

Simulation Panel is the main AMPS control panel that enables the user to configure, control, and monitor the simulation sessions as well as launching AMPS Configuration Editor.



Sessions Setup

Session Type	Session Length	Message Process	Emotion Process	Bid/Ask Process	Target User
Trading	10	trading_default.msg	CAPM.emo	default.baa	lwang
Trading	30	trading_default.msg	pos_affect_20s.emo	default.baa	lwang
Break	10	break10sec.msg	---	---	---
Trading	300	trading300sec.msg	distress_20s.emo	high_vol.baa	lwang
Break	10	break10sec.msg	---	---	---
Trading	60	trading60sec.msg	arousal_20s.emo	high_freq.baa	lwang
Break	10	break_final.msg	---	---	---

Simulation Status : **Not Started**

Current Session Time Remaining : **0:10**

Total Time Remaining : **7:10**

Figure 5-1: Simulation UI: Simulation Panel

The "Start Simulation" and "End Simulation" buttons enable the administrator to start, pause, and stop the simulation process. Clicking on the "Start Simulation" button begins running the sessions in the sequence shown, starting with the session current highlighted in Session Sequence Table (for details on Session Sequence Table, see Table 5-1 below).

While the simulation is running, the display on the "Start Simulation" button is changed to "Pause Simulation", and clicking on the button during such period allows the user to pause the simulation. While the simulation is paused, the display on the "Pause Simulation" button changes to "Continue Simulation", and clicking on the button during such period allows the user to continue the simulation, or recover from pause.

At any point during the simulation, clicking the "End" button will terminate the current session and end the simulation. Exiting the system by clicking the "Exit" button or simply closing the window also ends the current simulation run.

Explanations of all the graphical components and their functions are as follows:

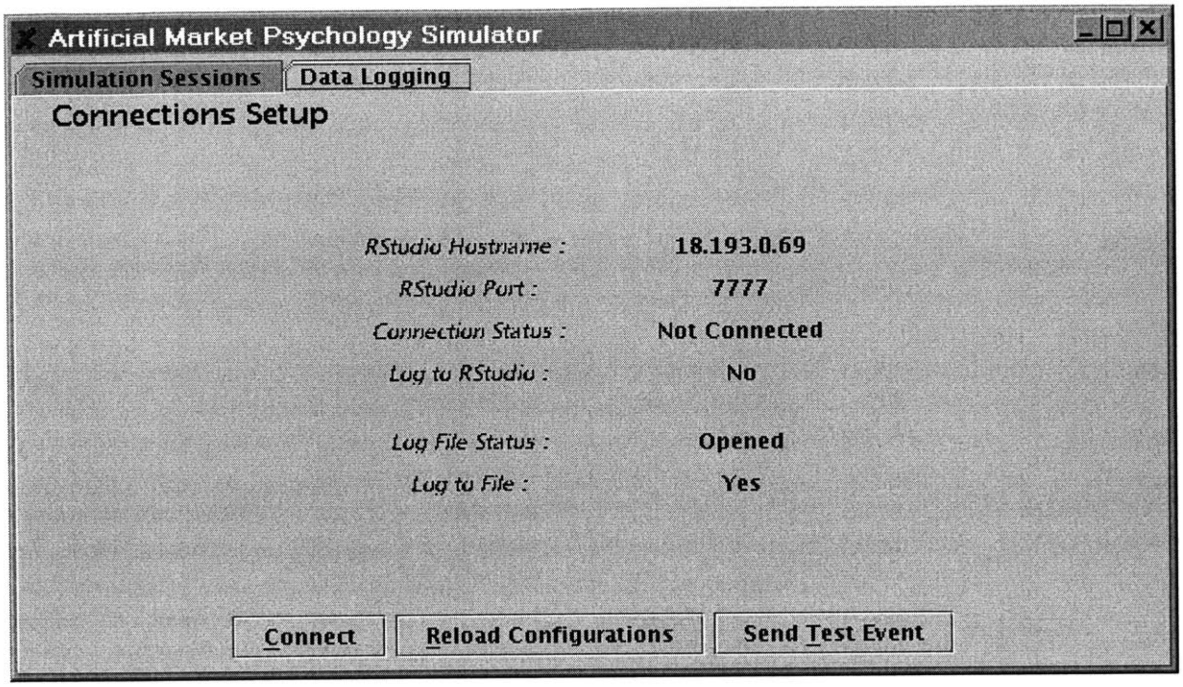
Component	*Function / Description*
Session Sequence Table	Session Sequence Table is the two-dimensional table occupying the top half of Simulation Panel. It displays the list of sessions loaded from the system configuration file in their specified sequence, as well as the session parameters. Each row is a separate session, and the session's parameter values are displayed in the columns. The session parameters displayed in the columns are as follows, ordered from left to right: **Session Type** – Type of session, denoted by "Trading" or "Break". **Session Length** – Length of session in number of seconds. **Message Process** – Name of message file to be used. **Emotion Process** – Name of emotion rule set file to be used (Trading session only). **Bid/Ask Process** – Name of bid/ask rule set file to be used (Trading session only). **Target User** – Username of the trading subject in the Web Market system (Trading session only). When the simulation is stopped, the user can select the session to start the run from by clicking on the session and highlighting the corresponding row. When the simulation is running, the display continuously updates itself, highlighting the row corresponding to the session currently in progress.
Reload Configurations Button	Reloads the system configuration file and rule sets, and reinitializes sessions and all other AMPS configurations. Session Sequence Table, time displays, DB Connector, RS Connector, and log file connection are also reinitialized.
Settings / Configurations Button	Launches the AMPS Configuration Editor, a file editor with basic text editing functionality allowing the user to modify configuration files quickly and easily while AMPS is running.
Simulation Status Display	Indicates the status of the simulation by displaying one of the following: **"Stopped"** – Simulation has been initialized but has not started

	running. "**Running**" – Simulation is currently running. "**Paused**" – Simulation is paused.
Session Time Display	Displays the time left in the current session in the following format: *<minutes>* : *<seconds>*.
Total Time Display	Displays the time left in the entire simulation run or sequence of sessions (sum of time left in current session and in all subsequent sessions) in the following format: *<minutes>* : *<seconds>*.
Start / Pause / Continue Button ("Start Simulation")	If the simulation is not running, this button displays "**Start Simulation**" and allows the user to start the run, beginning with the session currently highlighted. If the simulation is currently running, this button displays "**Pause Simulation**" and allows the user to pause the simulation. If the simulation is paused, the button displays "**Continue Simulation**" and allows the user to continue the simulation or recover from pause.
End Button ("End Simulation")	Stops the simulation, terminating the current session.
Exit Button	Exits AMPS and Simulation UI, terminating any session in progress.

Table 5-1: Simulation Panel Components and Functions

5.1.2 Data Logging Panel

The Data Logging Panel allows the user to monitor the current configurations and status of the connection to RStudio and to the local log file. It also allows the user to manually start or restart a connection attempt to RStudio, send test signals, and reload configurations during runtime.



Connections Setup

RStudio Hostname : 18.193.0.69
RStudio Port : 7777
Connection Status : Not Connected
Log to RStudio : No

Log File Status : Opened
Log to File : Yes

Figure 5-2: Simulation UI: Data Logging Panel

Explanations of the graphical components and their functions are as follows:

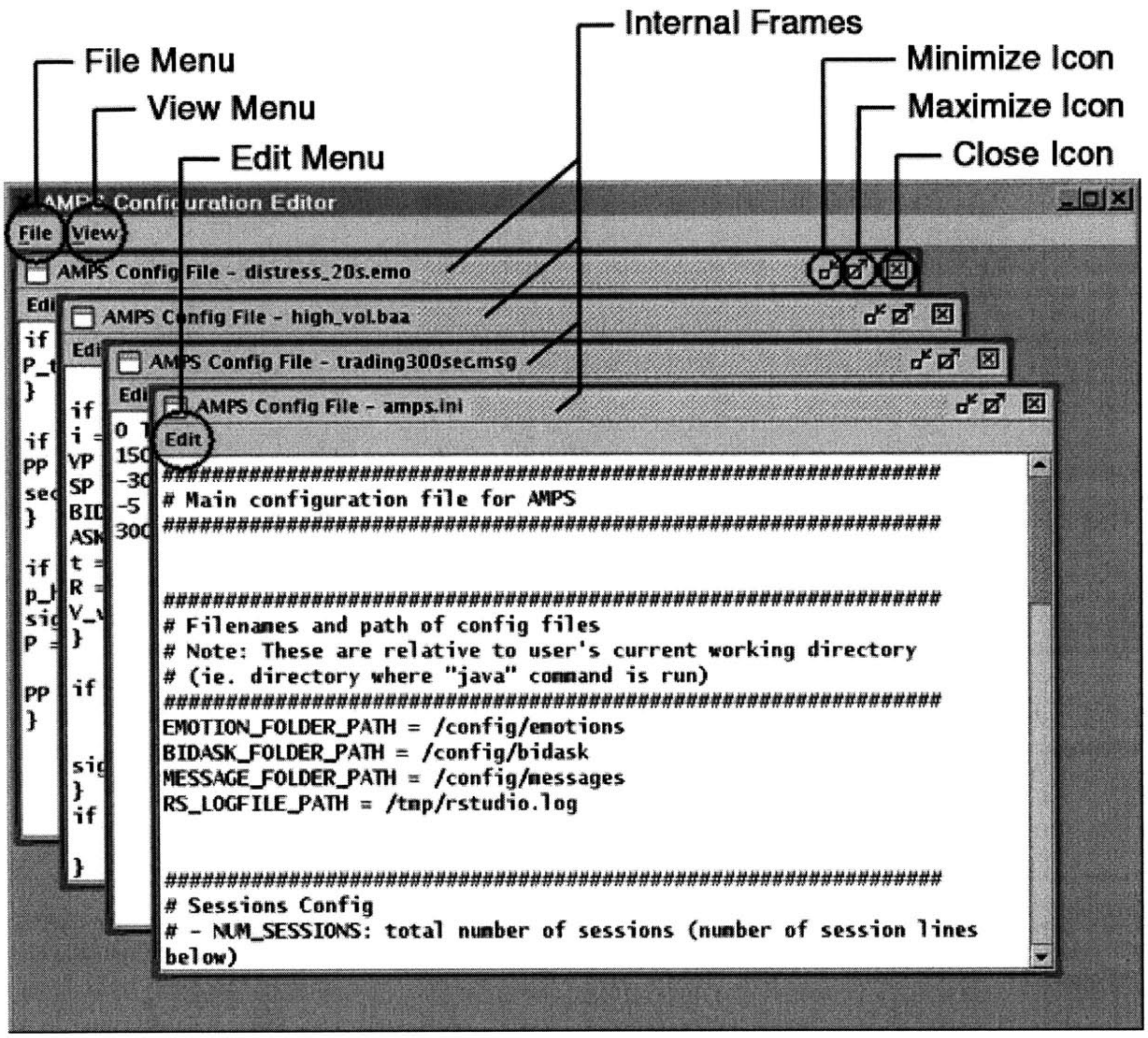
Component	*Function / Description*
RStudio Hostname Display	Displays the RStudio hostname or IP address used by RS Connector.
RStudio Port Display	Displays the RStudio port number used by RS Connector
Connection Status Display	Indicates the current status of the RStudio connection by displaying one of the following: **"Not Connected"** – RStudio connection is not established and not available for logging. **"Connected"** – RStudio connection is established and available for logging. **"Connecting... (Trying every *<number>* sec.)"** – RS Connector is trying to connect every *<number>* seconds. The *<number>* shown is the retry delay specified in the system configuration file.
Log to RStudio	Indicates whether RS Connector is currently configured to send

Display	log signals to RStudio by displaying either "**Yes**" or "**No**".
Log File Status Display	Indicates the current status of the connection to the local log file by displaying one of the following: **"Opened"** – Local log file is opened and available for writing. **"Closed"** – Local log file is closed and not available for writing.
Log to File Display	Indicates whether RS Connector is currently configured to write log events to the local log file, by displaying either "**Yes**" or "**No**".
Connect Button	Starts or restarts an attempt to connect to the RStudio server, terminating any attempt currently in progress.
Reload Button ("Reload Configurations")	Reloads RS Connector configurations from the system configuration file.
Send Test Event Button	Sends a test log event to RStudio and/or the local log file, depending on which one(s) RS Connector is currently configured to write to. The test event contains a description of "Test Event", for the local log file.

Table 5-2: Data Logging Panel Components and Functions

5.1.3 AMPS Configuration Editor

AMPS Configuration Editor is a simple configuration file editor with basic text editing and windowing functionality. It is provided as a convenience mechanism to facilitate quick and simple changes to the configuration files while AMPS is running.


Internal Frames
File Menu
View Menu
Edit Menu
Minimize Icon
Maximize Icon
Close Icon
AMPS Configuration Editor
File
View
AMPS Config File - distress_20s.emo
AMPS Config File - high_vol.baa
AMPS Config File - trading300sec.msg
AMPS Config File - amps.ini
Edit

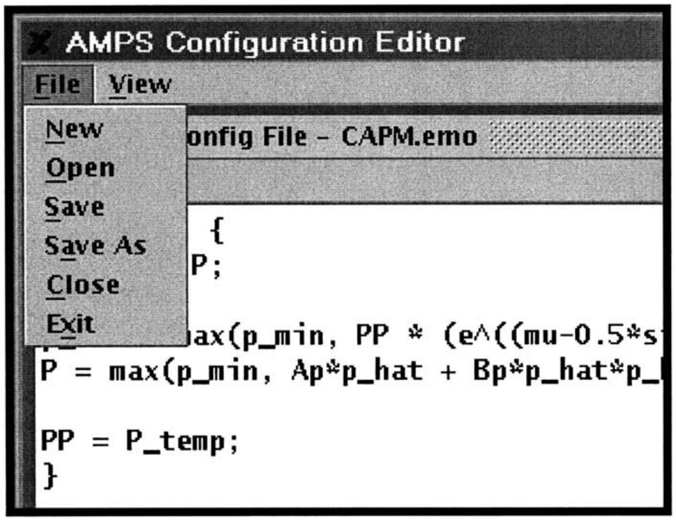
```
##################################################################
# Main configuration file for AMPS
##################################################################


##################################################################
# Filenames and path of config files
# Note: These are relative to user's current working directory
# (ie. directory where "java" command is run)
##################################################################
EMOTION_FOLDER_PATH = /config/emotions
BIDASK_FOLDER_PATH = /config/bidask
MESSAGE_FOLDER_PATH = /config/messages
RS_LOGFILE_PATH = /tmp/rstudio.log


##################################################################
# Sessions Config
# - NUM_SESSIONS: total number of sessions (number of session lines
below)
```

Figure 5-3: Configuration Editor: Main Graphical Components

Configuration Editor is a master window that allows the user to open configuration files in internal frames, or sub-windows inside itself, as shown in the figure above. Clicking on the "Settings/Configurations" button in Simulation Panel launches Configuration Editor. When the Editor is launched, all configuration files specific to the current session are opened by default. This includes the emotion rule set file, the bid/ask rule set file, and the message file for the session in progress, as well as the system configuration file *amps.ini*. The title bar at the top of each internal frame displays the name of the file whose contents are shown in the frame.

To exit the Editor, the user can either select "Exit" under File Menu, or simply close the master window. If there are files or internal frames open when exiting, the file, location, and size of the internal frames will be remembered, such that the next time Configuration Editor is started, it will recover to the same appearance it had before the exit.

File Menu in the master window provides a number of common file-related functions, allowing the user to create, open, save, and close individual files. The items found under File Menu are shown below.



```
{
P;

ax(p_min, PP * (e^((mu-0.5*s
P = max(p_min, Ap*p_hat + Bp*p_hat*p_

PP = P_temp;
}
```

Figure 5-4: Configuration Editor: File Menu Items

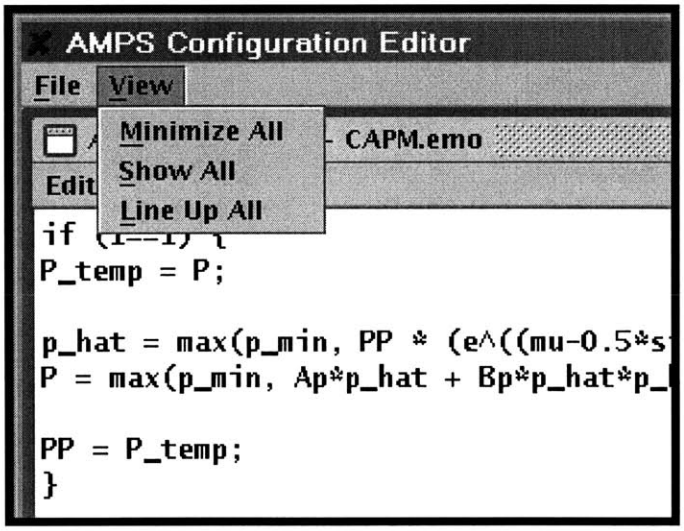
The functions of the File Menu items are summarized in the table below.

File Menu Item	*Function / Description*
New	Creates a new file in a new internal frame and brings the frame to the top.
Open	Opens an existing file in a new internal frame and brings the frame to the top. Clicking on this item launches an "Open" window that asks the user for the name and path of the file to open. The default open location is the directory in which the system configuration file *amps.ini* resides. For a screenshot of this function, see Appendix G: Configuration

	Editor Screenshots.
Save	Saves the file that is currently on top (if one or more files are open) under the same name. If the file is a new document without a name, the "Save As" window is launched (see section below on "Save As" menu item).
Save As	Saves the file that is currently on top (if one or more files are open) under a different name. Clicking on this item launches a "Save As" window that asks the user for a new filename and path. The default "Save As" location is the directory in which the original file resides. For a screenshot of this function, see Appendix G: Configuration Editor Screenshots.
Close	Closes the file and internal frame that is currently on top (if one or more files are open).
Exit	Closes Configuration Editor.

Table 5-3: File Menu Items and Functions

View Menu in the master window provides a number of frame management functions, allowing the user to organize the internal frames for better viewing. The items found under View Menu are shown as follows:



```
if (i==1) {
P_temp = P;

p_hat = max(p_min, PP * (e^((mu-0.5*s
P = max(p_min, Ap*p_hat + Bp*p_hat*p_

PP = P_temp;
}
```

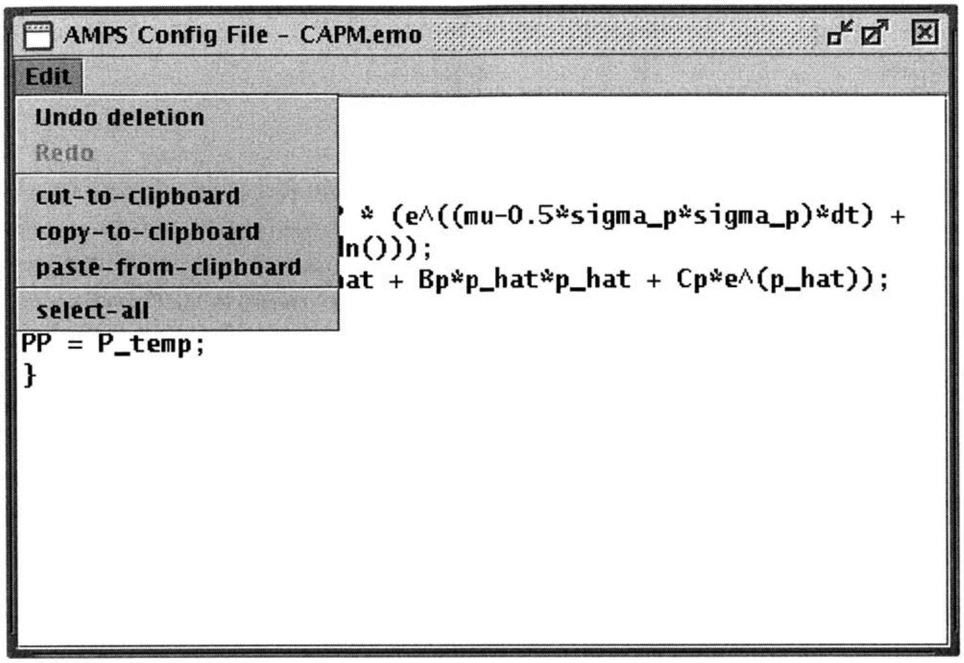
Figure 5-5: Configuration Editor: View Menu Items

View Menu items and their functions are summarized below:

View Menu Item	*Function / Description*
Minimize All	Minimizes all internal frames into icons at the bottom of the master window. This has the same effect as clicking the "Minimize" icon on each of the internal frames. For a screenshot of this function, see Appendix G: Configuration Editor Screenshots.
Show All	Restores all internal frames from minimized state back to their original size and location. For a screenshot of this function, see Appendix G: Configuration Editor Screenshots.
Line Up All	Restores all internal frames from minimized or maximized state and lines them up in a cascaded manner with all the title bars visible. For a screenshot of this function, see Appendix G: Configuration Editor Screenshots.

Table 5-4: View Menu Items and Functions

An internal frame is a sub-window within Configuration Editor that displays the contents of a file and allows the user to perform editing. It provides an Edit Menu with basic text editing functionality, and icons on the title bar for common frame management functions. In the figure below, an internal frame is shown with its Edit Menu opened.



```
 * (e^((mu-0.5*sigma_p*sigma_p)*dt) +
ln()));
at + Bp*p_hat*p_hat + Cp*e^(p_hat));

PP = P_temp;
}
```

Figure 5-6: Configuration Editor: Internal Frame and Edit Menu Items

Edit Menu items and their functions are summarized below.

Edit Menu Item	*Function / Description*
Undo	Undoes the last edit. A short description of the last edit is displayed behind the word "Undo". For example, in the figure above, the Undo menu item displays "deletion", indicating that clicking on the menu item at this time will undo the last deletion.
Redo	Redoes the last action undone by the Undo menu item. A short description of the last action undone is displayed behind the word "Redo".
Cut-to-Clipboard	Cuts any selected text and pastes to the clipboard in memory.
Copy-to-Clipboard	Copies any selected text and pastes to the clipboard in memory.
Paste-from-Clipboard	Pastes any text from the clipboard in memory to the current cursor location in the document. If there is text currently selected, the new text is pasted over the selected text.
Select-All	Selects the entire text in the document.

Table 5-5: Edit Menu Items and Functions

The frame management icons are shown and labeled in Figure 5-3, and their functions are as follows:

Frame Icon	*Function / Description*
Minimize Icon	Minimizes the internal frame into an icon at the bottom of the master window. For a screenshot of this function, see Appendix G: Configuration Editor Screenshots.
Maximize Icon	Maximizes the internal frame to fill the entire master window. For a screenshot of this function, see Appendix G: Configuration Editor Screenshots.
Close Icon	Closes the internal frame and file.

Table 5-6: Internal Frame Icons and Functions

In addition, the internal frame supports a few key bindings for the convenience of the user, some of which are provided by default through the Java components used.

Key Binding	*Function / Description*
Ctrl-a	Selects the entire text in the document.
Ctrl-x	Cuts any selected text and pastes to the clipboard in memory.
Ctrl-c	Copies any selected text and pastes to the clipboard in memory.
Ctrl-v	Pastes any text from the clipboard in memory to the current cursor location in the document. If there is text currently selected, the new text is pasted over the selected text.
Ctrl-b	Moves the cursor backward by 1 character.
Ctrl-f	Moves the cursor forward by 1 character.
Ctrl-p	Moves the cursor up one line.

Ctrl-n	Moves the cursor down one line.

Table 5-7: Internal Frame Key Bindings and Functions

5.2 Web Market Server and Client User Interface

Web Market server and client are not part of the AMPS prototype, however, because AMPS works in close conjunction with Web Market, brief descriptions of their user interfaces are provided for the convenience of the reader.

5.2.1 Client User Interface

The user interface for the Web Market client is a Java applet that allows the trading subject to participate in the Web Market simulation and the AMPS experiment using a web browser. The main features provided include monitoring real-time price movements, placing orders, tracking portfolio, and receiving market news and system messages. For a screenshot of the client user interface, see Appendix H: Web Market Client and Server Screenshots.

5.2.2 Server User Interface

The Web Market server user interface is a Java application launched from the server-side, allowing the server administrator to monitor and control the market. The main features provided include opening and closing the market, monitoring the status of connected clients, and sending broadcast messages to connected clients. For a screenshot of the server user interface, see Appendix H: Web Market Client and Server Screenshots.

Chapter 6

Software Test Plan

6.1 Overview

In the AMPS project, testing is performed through a bottom-up approach – components and functions are always tested individually before they are tested in larger groups or as a whole. The strategy involves three primary levels of testing throughout the development process: unit testing, functional testing, and system testing, described in the sections below. As new code is produced, pertinent test cases at each level are performed, starting at the unit level, moving to the functional level, and eventually to the system level. The practice of ongoing regression testing[17] ensures that the existing code is not adversely impacted by the new code.

The following sections explain the different levels of testing during AMPS development. They also summarize the test cases involved to provide an overview of the scope of efforts undertaken at each level.

6.2 Unit Testing

Unit testing involves the lowest-level building blocks of the application or methods within these building blocks. It covers the functionality of all the individual components of the application in their entirety. In AMPS context, unit testing covers all the methods of the Java class files, including the correctness and timeliness of their input and output. For instance, this effort consists of validating the functionality of the rule set parser,

[17] Regression testing is designed to confirm that unchanged portions of the system still work correctly in light of new changes made during development or maintenance [7].

networking components, database queries, price computations, and basic user interface features.

Methods and functions of most AMPS components are unit-tested as they are developed. The remaining functions are tested in groups at the functional level, for simplicity and timeframe reasons.

For the main components of AMPS, test applications have been developed to automatically run the unit test scripts and test cases. The Java source files for these component-specific test applications are listed below, along with their descriptions.

Test Script / Code	***Description***
CETest.java	Launches Configuration Editor by itself to test all functionality provided by the user interface.
DBConnectorTest.java	Tests all embedded queries and methods of DB Connector.
MessageTest.java	Tests the parsing of message files and the delivery of broadcast messages, and the data structure used for messages.
ParserTest.java	Tests the library of custom functions and the JEP expression parser.
RuleSetTest.java	Tests the EmotionRuleSet and BidAskRuleSet classes, as well as the rule set parsers used by Emotion Engine and Bid/Ask Engine.
RSConnectorTest.java **RSConnectorTestServer.java**	Tests the functions and error handling of RSConnector, including RStudio network connection and local log file connection.

Table 6-1: Unit Test Scripts and Descriptions

6.3 Functional Testing

Functional testing, or black-box testing, involves the testing of business functions – the user activities defining the basic functions of the system. In AMPS context, functional testing includes the exhaustive list of tasks the research administrator can perform through Simulation UI or by modifying any of the configuration files. This type of testing is also carried out on an ongoing basis as new functionality is added.

While functional testing involves application-level features available to the user, its test cases are mostly executed manually. The test cases are outlined as follows, grouped by functions from the user perspective.

User Function	*Test Cases / Descriptions*
System Management	▪ Startup and shutdown. ▪ State transitions and invocation of processes. ▪ Simulation UI startup and shutdown. ▪ Interruption, failure, and exception handling.
Simulation and Sessions Management	▪ Simulation UI display and updating of session details and run sequence. ▪ Simulation UI display and updating of session time left, total time left, and simulation status. ▪ Simulation process control – start, stop, pause, and continue. ▪ Selection of starting session. ▪ Loading and reloading of configuration files (including rule sets and message files). ▪ Errors, warning, and information messages.
Price Pattern Generation	▪ Emotion Engine and Bid/Ask Engine initialization. ▪ Emotion and bid/ask rule set expression parsing and evaluation. ▪ Real-time updating of Market Variables, Target Price Variable, and Order Variables. ▪ Parsing error and exception handling.
Message Delivery	▪ Message Engine initialization. ▪ Correctness and timeliness of message delivery *. ▪ Delivery error and exception handling.
Database Connection Management	▪ DB Connector initialization and database connectivity. ▪ Correctness of queries and data retrieval.

	▪ Error and exception handling, including query and connection errors.
Data Logging Management	▪ RS Connector initialization. ▪ Connectivity to RStudio (manual and auto) and connection retries. ▪ Connectivity to local log file. ▪ Correctness and timeliness of data logging. ▪ Connection status display, including updates upon receiving RS Connector state change notifications. ▪ Data logging errors and exception handling.
Web Market Integration	▪ AMPS login and logout. ▪ Correctness and timeliness of order submissions. ▪ Correctness and timeliness of broadcast message (Web Market Client) *. ▪ Correctness and timeliness of order display (Web Market Client).
Database Access Scripts	▪ Command line input and data retrieval. ▪ Data storage and display. ▪ Error and exception handling, including command line and parameter errors.

* Broadcast message delivery through Web Market server has not been completed as of the writing of this thesis, and thus has only been tested to a limited extent.

Table 6-2: Functional Test Cases and Descriptions

6.4 System Testing

System testing is the highest level of testing which evaluates the functionality of the system as a whole for overall usability and performance. It is summarized as follows:

System Test	***Descriptions***
Performance Testing	Ensures that order computation and submission time are adequate for the intended purposes of AMPS.

Compatibility Testing	Verifies that integration with Web Market and RStudio is fully functional and robust.
Usability Testing	Verifies that overall user experience, including user interfaces, system configurations, and rule set scripting, is satisfactory for the research administrator and the research subject.
Recovery Testing	Ensures that software and hardware failures are handled gracefully without affecting the data store or the stability of the underlying platform.

Table 6-3: System Test Cases and Descriptions

6.5 Test Environment

The environment under which AMPS has been tested, including both hardware and software specifications, are as follows.

	AMPS Server (Administrator Machine)	***AMPS Client (Subject Machine)***
Operating System	Red Hat Linux Release 7.2	Windows 2000 Service Pack 3
Processor	4-Processor, 900 MHZ Intel Pentium III	600 MHZ Intel Pentium III
Memory	3.8 GB	384 MB
Java Version	J2SE (Java™ 2 Standard Edition) SDK Version 1.4.0_01 for Linux	J2SE (Java™ 2 Standard Edition) SDK Version 1.4.0_01 for Windows
JEP Version	JEP Version 2.24	N/A
Browser	N/A	Internet Explorer Version 6.0
Database	Oracle Version 8.1.7	N/A

Table 6-4: Test Environment Specifications

Chapter 7

Future Work

The prototype as it stands addresses some of the key initial objectives of the research on trading psychology at the MIT Lab for Financial Engineering. In future phases of the research effort, however, the use of AMPS may expand and require support for additional data inputs, usability, robustness, scalability, performance, and simulation-related features. A number of proposals for future development are discussed in the following sections.

7.1 Support for Physiological Data Inputs

Modules should be developed to support real-time physiological data feed and allow the research administrator to incorporate these new inputs into the generation of price patterns. In combination with the subject's trading behavior, real-time physiological data provide valuable alternative insights into the cognitive state of the trading subject, and can prove useful in generating effective emotion-inducing price patterns. Some of these physiological data, for example, include electrocardiogram (EKG), brain wave (EEG), skin conductance (SCR), blood volume pulse (BVP), electromyographical signals (EMG), heart rate, respiration rate, and body temperature [8].

RStudio, developed by Eric Ho at the MIT Lab for Financial Engineering, is a system for collecting and monitoring a variety of the aforementioned physiological data types, using portable biofeedback equipment [6]. One potential direction for AMPS is to incorporate the data feed collected by RStudio into the emotion generation process by representing the data using real-time rule set variables, enabling the administrator to utilize the data in target price computation.

7.2 Logging System Enhancements

In the logging scheme of the current implementation, system information concerning runtime status, events, and errors are manually piped to a local log file or displayed on screen. Future enhancements to server-side logging system are recommended to improve the efficiency and timeliness of the reporting and debugging processes. Administrators and developers should also be provided with finer control over the level of detail and relevancy of reported data.

One proposed upgrade is the migration to a channel and subscription-based logging system. In this model, log data is categorized by the level of detail and topic, and published through different channels corresponding to the different levels of detail or topics. AMPS components act as publishers that output real-time log data through the different channels depending on their data type. Components and processes, both internal and external to AMPS, can register as data subscribers for one or more channels to receive the data specific to their individual needs. Data subscribers may include, for example, a process that outputs to a database, a process that writes directly to the file system, or one that simply displays the information on screen. The main advantage of this subscription-based scheme is to allow the administrator or developer to subscribe only to the log data that is most relevant for his or her specific purposes.

7.3 Improved Exception and Error Handling

Java exceptions and other runtime errors should be handled in a more discrete and structured manner, and incorporated into the logging scheme to facilitate system monitoring and improve overall robustness. Currently, only the most critical errors are handled at the application level or reported to the administrator through user interface events and messages. The majority of Java exceptions are reported individually as they occur, and the error messages are delivered to standard output (screen display by default). The loose reporting scheme may lead to difficulties in debugging, as system architecture expands and becomes more complex. Building more structure in the error handling process will improve fault tolerance and facilitate error tracking.

7.4 Additional Logic Constructs and Rule Set Syntax

The rule set scripting language in the current prototype only supports if-then statements. As the usage of AMPS grows, additional logic constructs can simplify the development of rule sets and make the scripting process more intuitive. Some additional constructs that are good candidates to append to the existing semantics include if-then-else statements, for-loops, and while-loops.

7.5 Support for Multiple Concurrent Trading Subjects

Enhancements should be made to enable multiple trading subjects to participate in an AMPS experiment simultaneously, in order to add extra dimensions to the control and observations of the experiment. The functionality should allow different emotion rule sets for multiple emotion targets during each trading session. It should also allow the administrator to set up some form of prioritization of emotion targets, in case of price pattern conflicts. The data feedback mechanism, which retrieves the trading subject's real-time behavior data, must also be upgraded to obtain the behavior data for multiple traders.

Chapter 8

Discussion

AMPS is a prototype simulation system designed to facilitate research on trading psychology by providing tools to systematically and consistently exert emotional influence on the trading subjects through controlled price movements. While the subject trades under the elicited emotional states, real-time trader behavior data is fed back to the system to generate subsequent price patterns. The design enables emotions and emotion-inducing price patterns to be generated relative to the trader's behavior patterns, and thus in theory, relative to the current cognitive state underlying these behavior patterns. This allows the researchers to define and study emotions in terms of observable behavior patterns in the context of securities trading, making them more easily quantifiable and approachable.

Although the development and testing process fell short of the initial timeframe estimate, the project reached a satisfactory closure towards the end and the main design objectives have been met. The disciplined, continuous testing and debugging throughout the development phase have made a deciding difference in uncovering deeply rooted problems that might have been costly to fix at later stages during the project. The source code, configuration files, and scripts have been adequately documented within the files, and usage instructions have been prepared to ensure smooth knowledge transfer and usage training.

The major complexities during the development process include the simulation process control and integration with Web Market server. Because the simulation process flow is driven by a number of underlying thread components and concurrent processes, the design must be performed carefully to manage the state transitions correctly. Web Market integration was more troublesome than expected because part of the server source code was not available[18] during the integration process, leading to unanticipated delays in testing and debugging.

As of the writing of this thesis, AMPS has been fully installed and tested on-site at the MIT Lab for Financial Engineering. However, live simulation trials have been carried out only to a limited extent. As additional trials are performed, user feedback should be utilized to fine-tune emotion rule sets and bid/ask rule sets as well as to identify significant system limitations for future improvement. User interfaces should also be maintained on an ongoing basis and upgraded as experiment needs evolve.

[18] Web Market has been undergoing major source code revisions during the timeframe of AMPS development, and a compilable version of the code base has not been available.

Appendix A: AMPS Custom Functions

The list of AMPS custom functions is provided in this section, along with descriptions and explanations of their parameters and usage. These functions have been developed specifically for price computations in AMPS, and they can be used in emotion rule sets and bid/ask rule sets the same way JEP functions are used.

Symbol	Description	Sample Usage	Evaluation Result
expRand()	Returns an exponentially distributed random number.	expRand()	*<Random Number>*
max(value1, value2...)	Returns the max of all input values.	max(1,2,3)	3

round_to_tick(x, p, dp, type)
 	
where:
x = bid/ask price
p = actual traded price
dp = minimum tick
type = bid/ask type (1=bid, 0=ask)
	Rounds the given bid/ask price to the nearest tick such that ask is always greater than bid, and returns the rounded bid/ask price.	round_to_tick(9.9, 10, 0.25, 1)	9.75

round_lot(x,dv)
 	
where:
x = volume
dv = minimum volume increment
	Rounds the given volume to the nearest larger round lot, and returns the rounded volume.	round_lot(8.5, 2)	10

randn()	Returns a normally distributed random number.	randn()	*<Random Number>*
vecNew()	Returns a new and empty vector.	vecNew()	[ ] *
vecSize(vector)	Returns the current size of the given vector.	vecSize([1 1 2])	3
vecAdd(vector,	Adds the given value to	vecAdd([1 2 3],	[1 2 3

value)	the given vector, increasing its size by 1, and returns the resulting vector.	4)	4]
vecAddToSize(vector, size_limit, value)	Adds the given value to a vector and remove the oldest value(s) until the new size is less than or equal to the given size limit, and returns the resulting vector.	vecAddToSize([1 2 3], 3, 4)	[2 3 4]
vecSum(vector)	Returns the sum of all values in the given vector.	vecSum([1 2 3])	6
vecAvg(vector)	Returns the average of all values in the given vector.	vecAvg([1 2 3])	2

* The notation '**[*value1 value2 value3*]**' denotes a vector with values shown between the square brackets. Values are listed from left to right based on the order they have been added to the vector.

Table A-1: List of AMPS Custom Functions

Appendix B: JEP Operators, Functions, and Constants

The JEP operators, functions, and constants available for use in AMPS rule sets are described in this section. The information presented below is based on JEP (Java Mathematical Expression Parser) Version 2.24 documentation [3]. The list of functions has been truncated to include only those that are relevant and applicable for AMPS's purposes.

Operators

Symbol	Description	Sample Usage *	Evaluation Result *
^	Power	2^3	8
!	Boolean Not	!(3>2)	*False*
+X, -X	Unary Plus, Unary Minus	-(1+1)	-2
%	Modulus	3 % 2	1
/	Division	3 / 2	1.5
*	Multiplication	3 * 2	6
+, -	Addition, Subtraction	3 + 2	5
<=, >=	Less or Equal, More or Equal	3 >= 2	*True*
<, >	Less Than, Greater Than	3 > 2	*True*
!=, ==	Not Equal, Equal	3 == 2	*False*
&&	Boolean And	(3==3) && (3==2)	*False*
||	Boolean Or	(3==3) || (3==2)	*True*

* Sample usage and evaluation result are not part of the JEP documentation. The two columns are added here as a supplement to describe their exact usage in AMPS rule sets, and the evaluation results are based on the author's test cases.

Table B-1: List of JEP Operators

Functions

Symbol	Description
sin()	Sine
cos()	Cosine
tan()	Tangent
asin()	Arc Sine
acos()	Arc Cosine
atan()	Arc Tangent
sinh()	Hyperbolic Sine
cosh()	Hyperbolic Cosine
tanh()	Hyperbolic Tangent
asinh()	Inverse Hyperbolic Sine
acosh()	Inverse Hyperbolic Cosine
atanh()	Inverse Hyperbolic Tangent
ln()	Natural Logarithm
log()	Logarithm Base 10
angle()	Angle
abs()	Absolute Value / Magnitude
rand()	Random Number (Between 0 and 1)
mod()	Modulus
sqrt()	Square Root
sum()	Sum

Table B-2: List of JEP Functions

Constants

Symbol	Description	Value
pi	Pi	3.141592653589793
e	Natural Logarithmic Base	2.718281828459045

* Constants are not covered in the JEP documentation. The values and decimal precision provided are based on the author's test cases.

Table B-3: List of JEP Constants

Appendix C: List of AMPS Configuration and Log Files

The comprehensive list of configuration files and log files used by AMPS are provided below.

Filename	***Location***	***Description***
amps.ini	<*CWD*>/config/amps.ini *	System configuration file.
<emotion name>.emo ***<emotion name>.ini***	*<CWD><relative emotion directory path specified in amps.ini>*	Emotion rule set file and corresponding variable initialization file.
<bid/ask name>.baa ***<bid/ask name>.ini***	*<CWD><relative bid/ask directory path specified in amps.ini>*	Bid/Ask rule set file and corresponding variable initialization file.
<message name>.msg	*<CWD><relative message directory specified in amps.ini>*	Message file.
<RS log file name>	*<CWD><relative RS log file path specified in amps.ini>*	RS Connector local log file.

* '<*CWD*>' denotes the "current working directory" of the user, or the directory from which the administrator started AMPS. This is dynamically obtained during runtime through Java Runtime's system property '**`user.dir`**'.

Table C-1: List of AMPS Configuration and Log Files

Appendix D: System Configuration File and Parameters

This section discusses the usage of the AMPS system configuration file, *amps.ini*, and its parameters in detail. *amps.ini* contains configuration parameters for sessions, RStudio Connector, and Database Connector, as well as paths to other configuration files. This master configuration file is loaded at startup from its default path to initialize the system.

By design, AMPS looks for the configuration file under a path relative to the current working directory, or the directory from which the administrator started AMPS. Specifically, it loads the configuration file under the following path:

`<current working directory>/config/amps.ini`

A sample *amps.ini* is provided below:

```
################################################################
# Main configuration file for AMPS
################################################################

################################################################
# Filenames and path of config files
# Note: These are relative to user's current working directory
# (ie. directory where "java" command is run)
################################################################
EMOTION_FOLDER_PATH = /config/emotions
BIDASK_FOLDER_PATH = /config/bidask
MESSAGE_FOLDER_PATH = /config/messages
RS_LOGFILE_PATH = /tmp/rstudio.log

################################################################
# Sessions Config
# - NUM_SESSIONS: total number of sessions (number of session lines
below)
# - SESSION1, SESSION2... : List of sessions indicating type
(trading/break),
#   length in seconds, emotion file, bid/ask generation file, and
#   target user name.
################################################################
NUM_SESSIONS=7
SESSION1=trading 10 trading_default.msg CAPM.emo default.baa lwang
SESSION2=trading 30 trading_default.msg pos_affect_20s.emo
default.baa lwang
SESSION3=break 10 break10sec.msg
SESSION4=trading 300 trading300sec.msg distress_20s.emo high_vol.baa
lwang
SESSION5=break 10 break10sec.msg
SESSION6=trading 60 trading60sec.msg arousal_20s.emo high_freq.baa
lwang
SESSION7=break 10 break_final.msg

################################################################
# RStudio Config
# - For connection to RStudio and local log file
################################################################
RS_HOSTNAME=18.193.0.69
RS_PORT=7777
RS_RETRY_DELAY=5
RS_AUTO_CONNECT=0
RS_LOG_TO_FILE=1
RS_LOG_TO_RS=0

################################################################
# DB Config
# - For connection to RST Market database (Oracle)
################################################################
DB_URL = jdbc:oracle:thin:@wang:1521:OMKT
DB_USERNAME = dummy_user
DB_PASSWORD = dummy
```

Figure D-1: Sample AMPS System Configuration File: *amps.ini*

Each non-empty line is interpreted as a comment if it starts with the pound sign ('**#**'). Otherwise, it is loaded as a parameter with name and value separated by an equal ('=') sign. All spaces and tabs are ignored.

The configuration parameters found in *amps.ini* are outlined as follows:

Component	*Parameter*	*Description*
General	EMOTION_FOLDER_PATH	Path of folder containing emotion rule set files, relative to the current working directory. Example: */config/emotions*
	BIDASK_FOLDER_PATH	Path of folder containing bid/ask rule set files, relative to the current working directory. Example: */config/bidask*
	MESSAGE_FOLDER_PATH	Path of folder containing message files, relative to the current working directory. Example: */config/messages*
Simulation UI	NUM_SESSIONS *	Total number of sessions in the simulation (number of session declaration lines in *amps.ini*).
	SESSION<*number*> **	Session declaration parameters for session <*number*>. The <*number*> indicates the session's order in the run sequence, and parameter values are separated by space.
RS Connector	RS_HOSTNAME	IP or hostname of RStudio server.
	RS_PORT	Port number of Rstudio server.
	RS_RETRY_DELAY	Number of seconds to wait before retrying to connect to RStudio after a failed attempt.
	RS_AUTO_CONNECT	Setting for whether to automatically connect to RStudio at startup. 1=Yes and 0=No.
	RS_LOG_TO_FILE	Setting for whether to log the signal to the local file when RS Connector is asked to deliver a signal. 1=Yes and 0=No.
	RS_LOG_TO_RS	Setting for whether to deliver the signal to RStudio when RS Connector is asked to deliver a signal. 1=Yes and 0=No.
	RS_LOGFILE_PATH	Path of the local log file, relative to the current working directory. Example: */tmp/rstudio.log*
DB Connector	DB_URL	Database URL of the form: jdbc:<*subprotocol*>:<*subname*>
	DB_USERNAME	Database username to be used.

DB_PASSWORD	Database password to be used.

Table D-1: List of System Configuration Parameters

* NUM_SESSIONS specifies the number of SESSION declarations there are. For example, if NUM_SESSION=3, there should be exactly 3 SESSION declarations: SESSION1, SESSION2, and SESSION3.

** SESSION<*number*> specifies a simulation session. The declaration should be in the following form:

```
SESSION<number> = <session param1> <session param2> . . .
```

The parameters are separated by one or more spaces.

Each session declaration requires a minimum of 3 parameters. In the required order of appearance in the configuration file, from left to right, they are:

1. Session Type
2. Time Length (in number of seconds)
3. Message File (file containing the broadcast messages)

For break sessions, only the minimal set of 3 parameters is required, and the value of the first parameter, or session type, should be "break".

For example:

```
SESSION1 = break 10 break10sec.msg
```

For trading sessions, a total of 6 parameters are required – 3 additional parameters on top of the 3 required for all sessions. The value of the first parameter, or session type, should be "trading". In the required order of appearance in the configuration file, from left to right, the three additional parameters are:

1. Emotion File (file containing the emotion rule set)
2. Bid/Ask File (file containing the bid/ask rule set)

3. Target User (Web Market username of the emotion rule target)

For example:

```
    SESSION2 = trading 300 trading_default.msg CAPM.emo default.baa
lwang
```

Appendix E: Sample Emotion and Bid/Ask Rule Sets

Complete samples of emotion and bid/ask rule sets and their initialization files are provided below. The examples shown in the main body of this document are taken from these files.

Sample Emotion Rule Set

```
if (LTYPE==1) {
PP_temp = PP*0.98;
}
if (LTYPE==2) {
PP_temp = PP*1.02;
}
if ((LTYPE==3) && (LPRICE < PP)) {
PP_temp = PP*1.01;
}
if ((LTYPE==4) && (LPRICE > PP)) {
PP_temp = PP*0.99;
}
if (1==1) {
PP = P;
p_hat = max(p_min, PP_temp * (e^((mu-0.5*sigma_p*sigma_p)*dt) +
sigma_p*sqrt(dt)*randn()));
P = max(p_min, Ap*p_hat + Bp*p_hat*p_hat + Cp*e^(p_hat));
}
```

Figure E-1: Sample Emotion Rule Set: *sample_emotion.emo*

Sample Emotion Rule Set Initialization File

```
##############################################################
# Variable init file for sample_emotion.emo
# Recommended:
# - Variables used in the emotion file above should be declared
#   here with an initial value.
# - If a variable's initial value involves other variables,
#   make sure the variables used are defined above the line.
# - The following system variables should be initialized here:
#   P
#################################################################

# Price
P=95

# Price of previous order
PP=P
```

Figure E-2: Sample Emotion Rule Set Initialization File: *sample_emotion.ini*

Sample Bid/Ask Rule Set

```
if (1==1) {
SP = S;
BIDP = BID;
ASKP = ASK;
V_vec = vecAddToSize(V_vec, num_vlags, V);
i = i+1;
t = i*dt;
R = 0;
}
if (vecAvg(V_vec) > 0) {
inv_volume = 1/vecAvg(V_vec);
S = max(dp, SP * (As*dp + Bs*inv_volume + sigma_s*sqrt(dt))*randn());
}
if (vecAvg(V_vec) <= 0) {
S = max(dp, SP * (As*dp + sigma_s*sqrt(dt))*randn());
}
if (t >= rt) {
R = 1;
V = round_lot(Av*abs(randn()),dv);
rt_increment = 1;
}
if (!(t >= rt)) {
R = 0;
V = 0;
}
if ((t >= at) && (t >= bt)) {
p_cur = P;
BID = round_to_tick(p_cur-0.5*S,p_cur,dp,1);
bt_increment = 1;
}
if ((!((t >= at) && (t >= bt))) && (R > 0)) {
BID = round_to_tick(p_cur-0.5*S,p_cur,dp,1);
}
if ((!((t >= at) && (t >= bt))) && (!(R > 0))) {
BID = BIDP;
}
if ((t >= at) && (t >= bt)) {
p_cur = P;
ASK = round_to_tick(p_cur-0.5*S,p_cur,dp,0);
at_increment = 1;
}
if ((!((t >= at) && (t >= bt))) && (R > 0)) {
ASK = round_to_tick(p_cur-0.5*S,p_cur,dp,0);
}
if ((!((t >= at) && (t >= bt))) && (!(R > 0))) {
ASK = ASKP;
}
if (BID >= ASK) {
rand_var = randn();
}
if ((BID >= ASK) && (rand_var >= 0)) {
BID = ASK - S;
}
if ((BID >= ASK) && (!(rand_var >= 0))) {
ASK = BID + S;
}
if (rt_increment==1) {
rt = rt + expRand(1/lambda_r);
rt_increment=0;
}
if (at_increment==1) {
at = at + expRand(1/lambda_p);
at_increment=0;
}
if (bt_increment==1) {
bt = bt + expRand(1/lambda_p);
bt_increment=0;
}
```

Figure E-3: Sample Bid/Ask Rule Set: *sample_bidask.baa*

Sample Bid/Ask Rule Set Initialization File

```
################################################################
```

```
# Variable initialization file for sample_bidask.baa
# Recommended:
# - Variables used in the bid/ask file above should be declared
#   here with an initial value.
# - If a variable's initial value involves other variables,
#   make sure the variables used are defined above the line.
# - The following system variables should be initialized here:
#   BID, ASK, V, R
################################################################

#######################
# Simulation parameters
#######################
# Time period to simulate
Tmax = 600

# min time step (tick)
dt = 1

# min price tick
dp = 0.25

#######################
# price parameters
#######################
# current price in simulation
p_cur = 0

# price volatility
sigma_p = 0.005

# linear trend coeff
Ap = 1.0

# quadratic trend coeff
Bp = 0

# exponential trend coeff
Cp = 0

# minimum price ever
p_min = 2

# expected return
mu = 0.0005
```

```
#######################
# spread parameters
#######################
# spread
S = dp
SP = dp

# maximum possible value of spread
max_s = 15

# price tick coeff
As = 8

# inverse volume coeff
Bs = 0

# spread volatility
sigma_s = 0.5

#######################
# bid and ask parameters
#######################
# bid
BID = 0
BIDP = BID

# ask
ASK = 0
ASKP = 0

# temporal for bid
bt = 0

# temporal for ask
at = 0

# rate in the poisson process
lambda_p = 0.7

#######################
# trade and volume parameters
#######################
# volume
V=20
V_vec = vecNew()
```

```
# flag - trade or not
R = 0

# temporal for trades
rt = 0

# rate for trades
lambda_r = 0.8

# number of lags when the volume is used in calculations
num_vlags = 10

# scaling factor for volume
Av = 50

# round lot
dv = 10

#######################
# simulate bid and ask initial "arrival"
#######################
# bid temporal structure
bt = bt + expRand(1/lambda_p)

# ask temporal structure
at = at + expRand(1/lambda_p)

# trade temporal structure
rt = rt + expRand(1/lambda_r)

#######################
# Others
#######################
# current time
t = 0

# iteration counter
i = 1
```

Figure E-4: Sample Bid/Ask Rule Set Initialization File: *sample_bidask.ini*

Appendix F: AMPS System Requirements

	AMPS Server *(Administrator Machine)*	*AMPS Client* *(Subject Machine)*
Operating System	Unix / Linux (Tested platform includes: Red Hat Linux Release 7.2)	Windows 2000 / XP OR Unix / Linux (Major Distributions)
Processor	500 MHZ Intel Pentium or above	500 MHZ Intel Pentium or above
Memory	256 MB (Required) 512 MB (Recommended)	128 MB (Required) 256 MB (Recommended)
Disk Space	20 MB (Not including software requirements)	5 MB (Not including software requirements)
Installed Software	J2SE (Java™ 2 Standard Edition) SDK Version 1.4 or above for Unix / Linux * JEP (Java Mathematical Expression Parser) Version 2.24 or above * Web Market system and database	Internet Explorer Version 6.0 or above * (or equivalent browser including Netscape 6.0 or above) J2SE (Java™ 2 Standard Edition) SDK Version 1.3 or above for Windows *
Others	CLASSPATH environment variable must include path to: 1. JDBC Driver for Web Market database 2. JEP jar file 3. Web Market directory ("afm" directory) See ** for example.	

* Download website for some of the aforementioned software requirements are provided below:

J2SE SDK: http://java.sun.com/j2se/1.4.1/download.html
Internet Explorer: http://www.microsoft.com/downloads/
JEP: http://www.singularsys.com/jep/index.html

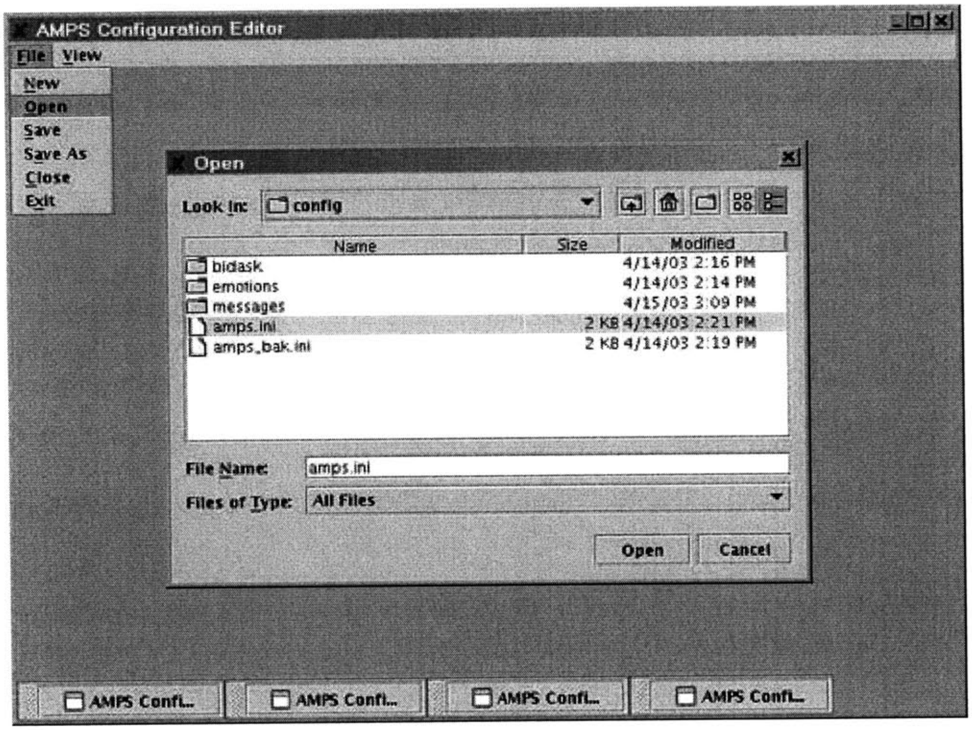
** A sample CLASSPATH is provided below:

```
CLASSPATH=/home/lwang/afm:/u2/ora8i/m01/app/oracle/product/8.1.
7/jdbc/lib/classes111.zip:/home/lwang/afm/rst/amps/lib/jep-
2.24.jar
```

Table F-1: AMPS System Requirements

Appendix G: Configuration Editor Screenshots

File Menu – Open



Look In: config

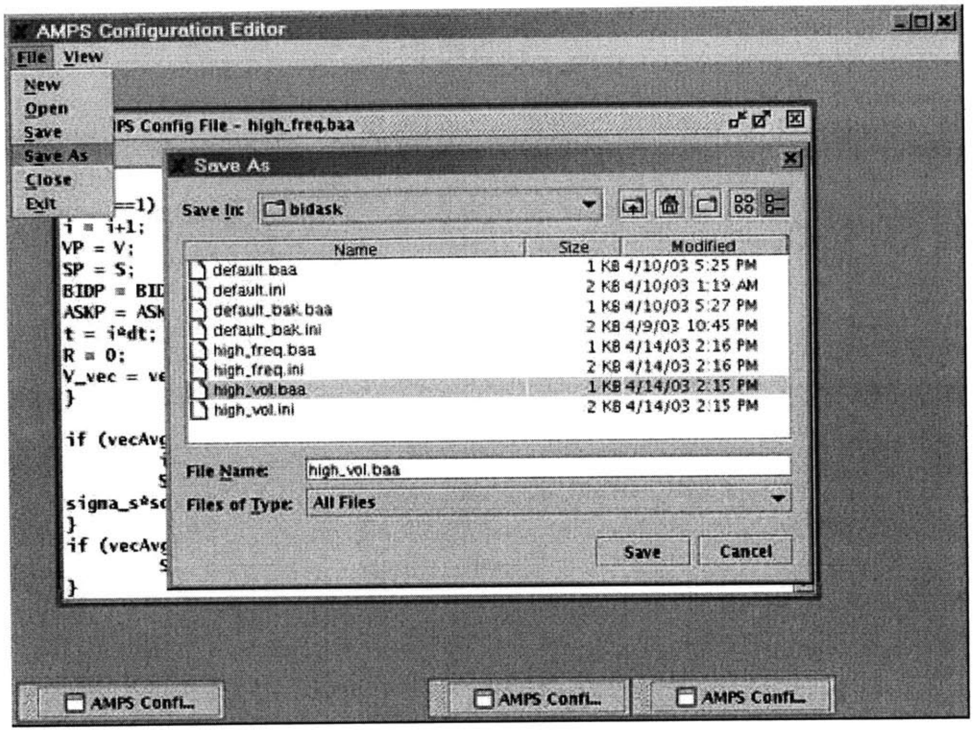
Name	Size	Modified
bidask		4/14/03 2:16 PM
emotions		4/14/03 2:14 PM
messages		4/15/03 3:09 PM
amps.ini	2 KB	4/14/03 2:21 PM
amps_bak.ini	2 KB	4/14/03 2:19 PM

File Name: amps.ini

Files of Type: All Files

Figure G-1: Configuration Editor: File Menu - Open

File Menu – Save As



```
      ==1)
i = i+1;
VP = V;
SP = S;
BIDP = BID
ASKP = ASK
t = i*dt;
R = 0;
V_vec = ve
}

if (vecAvg
          i
          S
sigma_s*sc
}
if (vecAvg
          S
}
```

Save In: bidask

Name	Size	Modified
default.baa	1 KB	4/10/03 5:25 PM
default.ini	2 KB	4/10/03 1:19 AM
default_bak.baa	1 KB	4/10/03 5:27 PM
default_bak.ini	2 KB	4/9/03 10:45 PM
high_freq.baa	1 KB	4/14/03 2:16 PM
high_freq.ini	2 KB	4/14/03 2:16 PM
high_vol.baa	1 KB	4/14/03 2:15 PM
high_vol.ini	2 KB	4/14/03 2:15 PM

File Name: high_vol.baa

Files of Type: All Files

Figure G-2: Configuration Editor: File Menu – Save As

View Menu – Minimize All

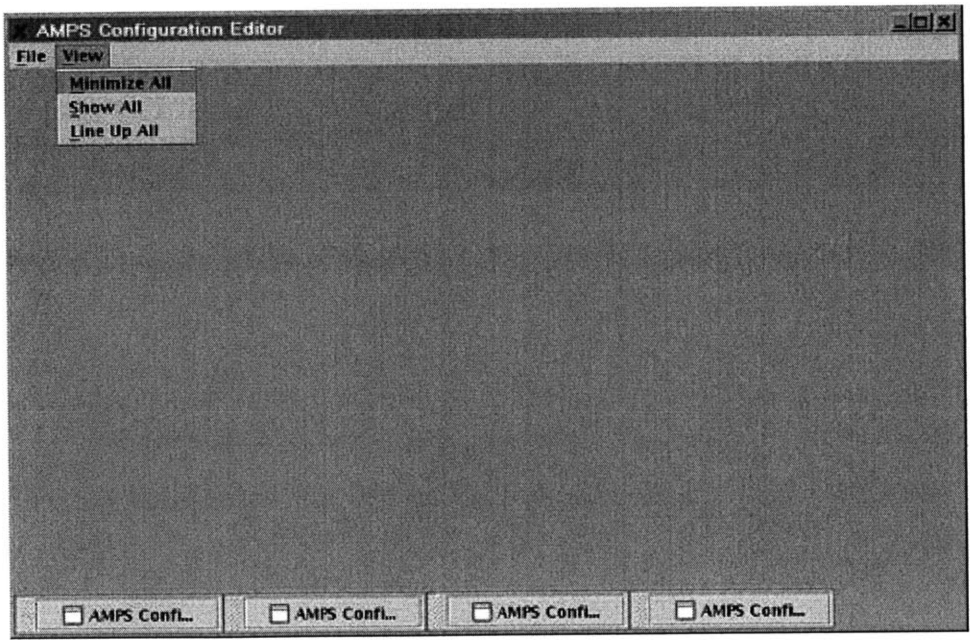
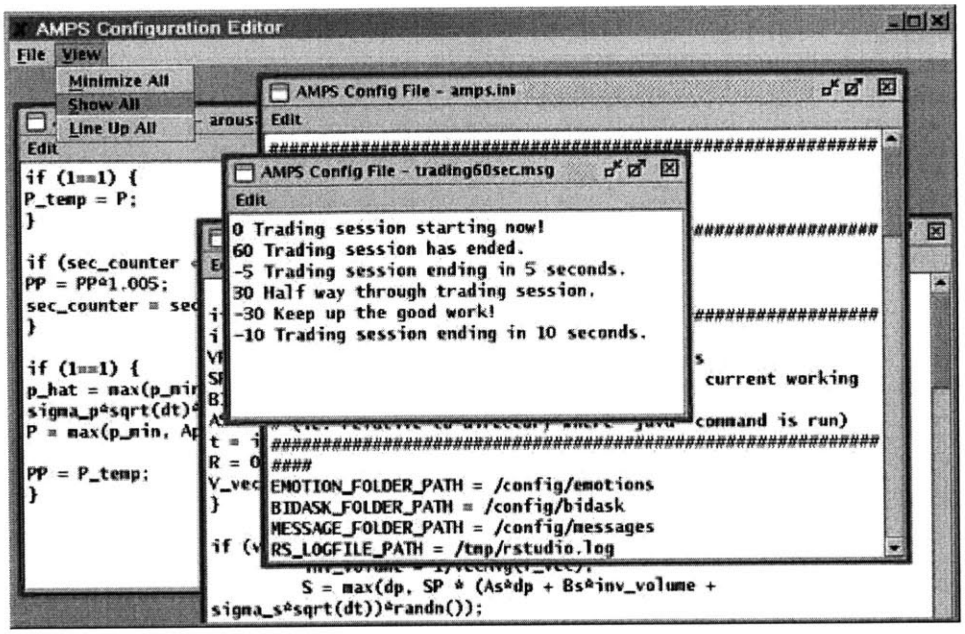


Figure G-3: Configuration Editor: View Menu – Minimize All

View Menu – Show All


AMPS Configuration Editor
File
View
Minimize All
Show All
Line Up All
arous
AMPS Config File - amps.ini
Edit
AMPS Config File - trading60sec.msg
Edit
Edit

```
if (1==1) {
P_temp = P;
}

if (sec_counter
PP = PP*1.005;
sec_counter = sec
}

if (1==1) {
p_hat = max(p_mir
sigma_p*sqrt(dt)*
P = max(p_min, Ap

PP = P_temp;
}
```

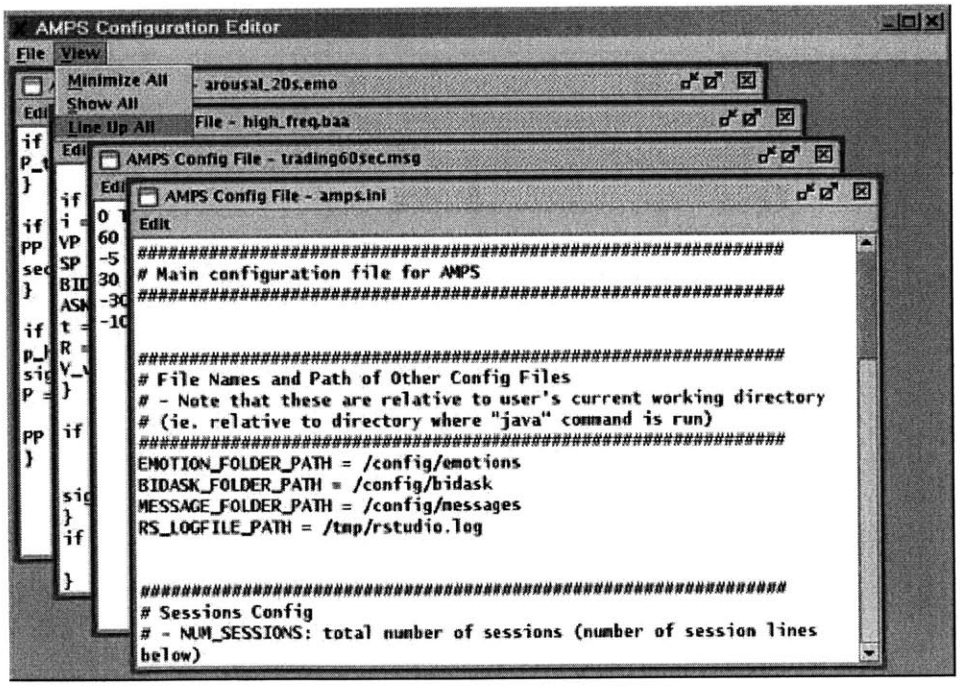
```
0 Trading session starting now!
60 Trading session has ended.
-5 Trading session ending in 5 seconds.
30 Half way through trading session.
-30 Keep up the good work!
-10 Trading session ending in 10 seconds.
```

```
################################################################
                                            ##################
                                            ##################
                                            s
                                            current working
                                            command is run)
################################################################
####
EMOTION_FOLDER_PATH = /config/emotions
BIDASK_FOLDER_PATH = /config/bidask
MESSAGE_FOLDER_PATH = /config/messages
RS_LOGFILE_PATH = /tmp/rstudio.log
```

```
        S = max(dp, SP * (As*dp + Bs*inv_volume +
signa_s*sqrt(dt))*randn());
```

Figure G-4: Configuration Editor: View Menu – Show All

View Menu – Line Up All


AMPS Configuration Editor
File
View
Minimize All
Show All
Line Up All
- arousal_20s.emo
File - high_freq.baa
AMPS Config File - trading60sec.msg
AMPS Config File - amps.ini
Edit

```
################################################################
# Main configuration file for AMPS
################################################################


################################################################
# File Names and Path of Other Config Files
# - Note that these are relative to user's current working directory
# (ie. relative to directory where "java" command is run)
################################################################
EMOTION_FOLDER_PATH = /config/emotions
BIDASK_FOLDER_PATH = /config/bidask
MESSAGE_FOLDER_PATH = /config/messages
RS_LOGFILE_PATH = /tmp/rstudio.log


################################################################
# Sessions Config
# - NUM_SESSIONS: total number of sessions (number of session lines
below)
```

Figure G-5: Configuration Editor: View Menu – Line Up All

Internal Frame – Minimize Icon

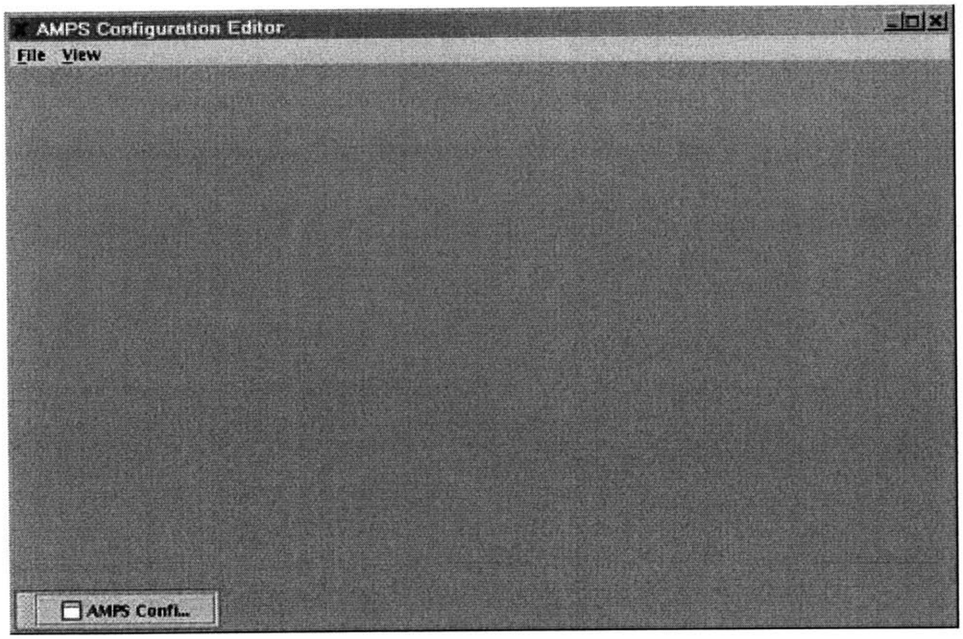
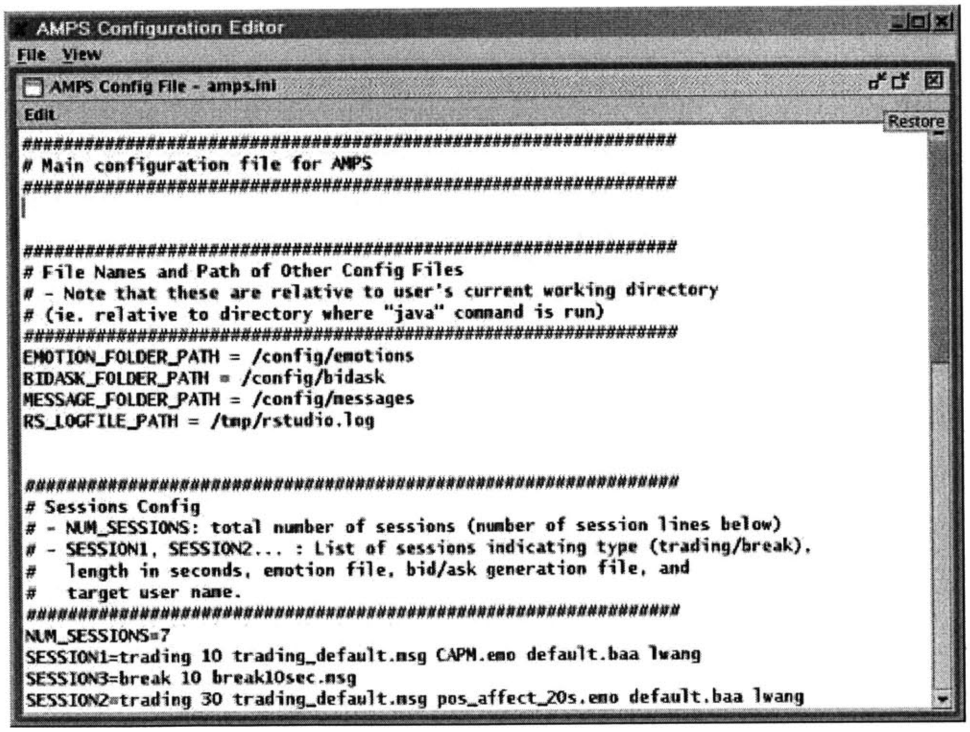


Figure G-6: Configuration Editor: Internal Frame – Minimize Icon

Internal Frame – Maximize Icon



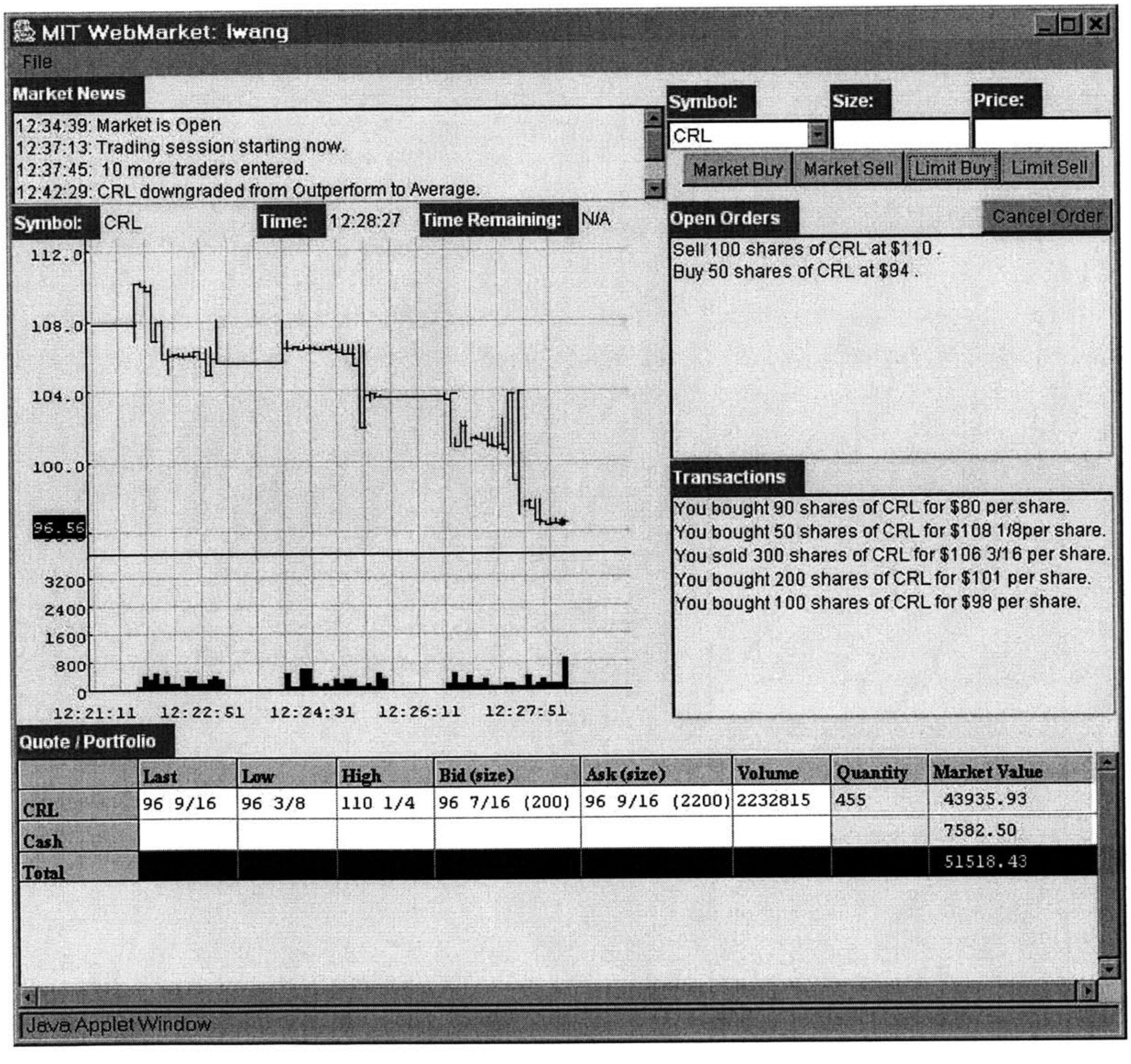
```
####################################################################
# Main configuration file for AMPS
####################################################################


####################################################################
# File Names and Path of Other Config Files
# - Note that these are relative to user's current working directory
# (ie. relative to directory where "java" command is run)
####################################################################
EMOTION_FOLDER_PATH = /config/emotions
BIDASK_FOLDER_PATH = /config/bidask
MESSAGE_FOLDER_PATH = /config/messages
RS_LOGFILE_PATH = /tmp/rstudio.log


####################################################################
# Sessions Config
# - NUM_SESSIONS: total number of sessions (number of session lines below)
# - SESSION1, SESSION2... : List of sessions indicating type (trading/break),
#   length in seconds, emotion file, bid/ask generation file, and
#   target user name.
####################################################################
NUM_SESSIONS=7
SESSION1=trading 10 trading_default.msg CAPM.emo default.baa lwang
SESSION3=break 10 break10sec.msg
SESSION2=trading 30 trading_default.msg pos_affect_20s.emo default.baa lwang
```

Figure G-7: Configuration Editor: Internal Frame – Maximize Icon

Appendix H: Web Market Client and Server Screenshots

Screenshots of the Web Market client and server user interfaces are provided below.

Web Market Client User Interface



12:34:39: Market is Open
12:37:13: Trading session starting now.
12:37:45: 10 more traders entered.
12:42:29: CRL downgraded from Outperform to Average.

Symbol: CRL

Symbol: CRL Time: 12:28:27 Time Remaining: N/A

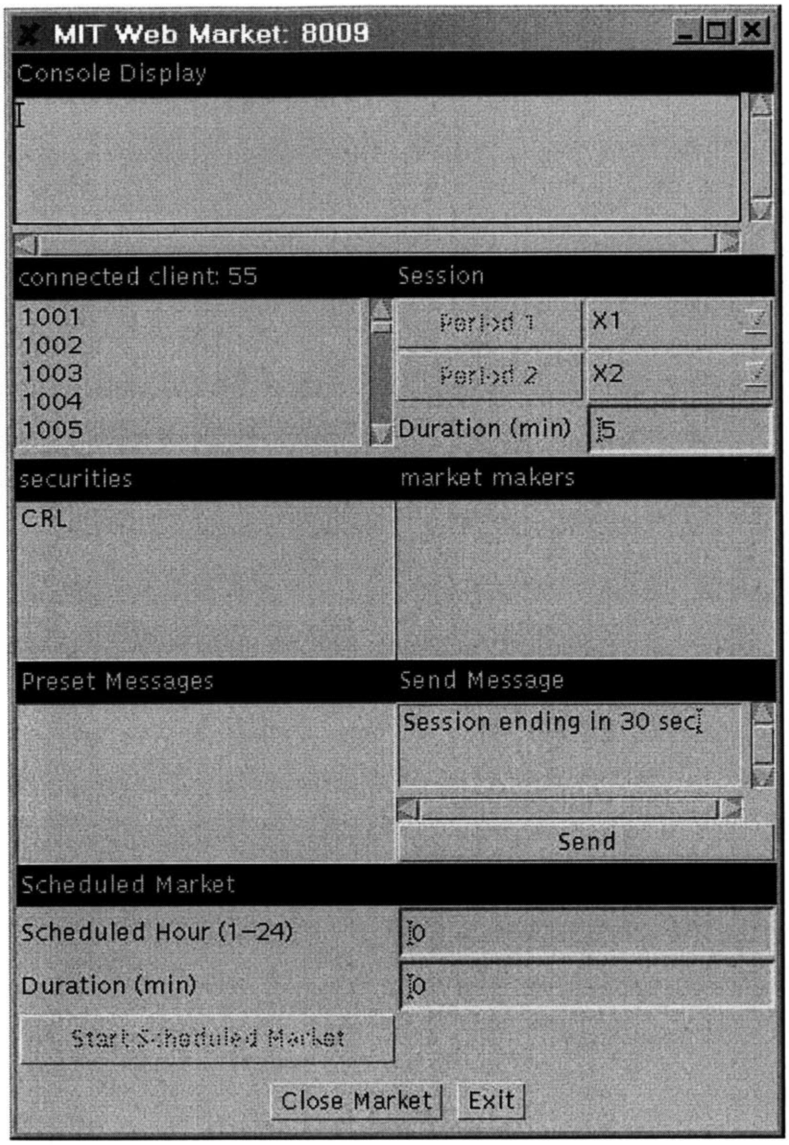
Sell 100 shares of CRL at $110 .
Buy 50 shares of CRL at $94 .

You bought 90 shares of CRL for $80 per share.
You bought 50 shares of CRL for $108 1/8per share.
You sold 300 shares of CRL for $106 3/16 per share.
You bought 200 shares of CRL for $101 per share.
You bought 100 shares of CRL for $98 per share.

	Last	Low	High	Bid (size)	Ask (size)	Volume	Quantity	Market Value
CRL	96 9/16	96 3/8	110 1/4	96 7/16 (200)	96 9/16 (2200)	2232815	455	43935.93
Cash								7582.50
Total								51518.43

Figure H-1: Web Market Client User Interface

<u>**Web Market Server User Interface**</u>



MIT Web Market: 8009

connected client: 55

1001
1002
1003
1004
1005

Session

Period 1: X1

Period 2: X2

Duration (min): 5

securities: CRL

Send Message: Session ending in 30 sec

Scheduled Market

Scheduled Hour (1–24): 0

Duration (min): 0

Figure H-2: Web Market Server User Interface

References

[1] Bosman, R. and van Winden, F., "Global Risk, Effort, and Emotions in an Investment Experiment," working paper, University of Amsterdam, 2001.

[2] Elster, J., "Emotions and Economic Theory," Journal of Economic Literature, 36, pp 47-74, 1998.

[3] Funk, N., "JEP – Java Math Expression Parser," [Online document], 2000, [cited 2003 Apr 10], Available HTTP: http://www.singularsys.com/jep/doc/index.html

[4] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA: Addison-Wesley, 1995.

[5] Haim, M., "Risk-Taking, Frame Effects, and Affect," Organizational Behavior and Human Decision Processes, 57, pp 38-58, 1994.

[6] Ho, E., "A Real-time System for Processing, Sharing, and Display of Physiology Data," diss., Cambridge: Massachusetts Institute of Technology, 2003.

[7] Lewis, W.E., Software Testing and Continuous Quality Improvement. Boca Raton, FL: CRC Press, 2000.

[8] Lo, A.W. and Repin, D.V., "The Psychophysiology of Real-Time Financial Risk Processing," Journal of Cognitive Neuroscience, 14, pp 323-339, 2002.

[9] Raghunathan, R. and Pham, M.T., "All Negative Moods are Not Equal: Motivational Influences of Anxiety and Sadness on Decision Making," Organizational Behavior and Human Decision Processes, 79, pp 56-77, 1999.

[10] Royce, W., Software Project Management: A Unified Framework. Boston, MA: Addison-Wesley, 1998.

[11] Steenbarger, B.N., The Psychology of Trading: Tools and Techniques for Minding the Markets. New York: John Wiley & Sons, 2002.

[12] Sun Microsystems, Inc., "Java Remote Method Invocation," [Online document], 2003, [cited 2003 Apr 10], Available HTTP: http://java.sun.com/products/jdk/rmi/