$=9$ $=1$

Eugene

Agent-Based Market Simulator for use in Validation and System Testing of Trading Algorithms

Jakub Kozłowski
MEng Computer Science
Submission Date: 27th April 2012
Internal Supervisor: Dr. Christopher D. Clack
External Supervisor: Ian Rose-Miller, UBS

ABSTRACT

At 3:40 PM on November 14^{th}, 2007, a buggy Credit Suisse proprietary algorithm (SmartWB) sent approximately 600,000 cancel/replace messages for non existent orders that lead to a disruption of trading. Credit Suisse was fined $150,000 [33]. On May 6, 2010, the U.S. stock market has experienced a sudden price drop of 5%, followed by a rapid recovery, all in the course of about 30 minutes. Subsequent analyses concluded that the incident was triggered by a large sell program that led to a "hot-potato" effect, where more than 27,000 futures contracts were bought and sold, with the net effect of only around 200 contracts [27].

These prominent examples of software errors in Trading Algorithms highlight an urgent issue with the way these systems are tested. In order to prevent Trading Algorithms from exhibiting such behaviour, there is a need for a strategy of rigorous testing in a realistic environment. Current testing techniques (backtesting on historical data) fail to capture the dynamic nature of markets and hence do not provide an environment for effective flaw identification.

In this thesis we investigated the application of Agent-Based Simulation to testing and verification of Trading Algorithms. We demonstrated an Agent-Based Stock Market Simulator and experimentally showed its potential to discover a difference in behaviour between a correctly and incorrectly implemented VWAP Algorithm. Furthermore, as an extension, we aimed to explore discovering a difference in behaviour between two correctly implemented VWAP Algorithms, but ones that trade with different frequency, however concluded that the current implementation of the system is not capable enough to demonstrate such a subtle effect. Nevertheless, the project clearly fulfilled its goals and thus provides a useful methodological advancement over the more established ways of testing Trading Algorithms.

CONTENTS

1 INTRODUCTION 7
- 1.1 Motivation 7
- 1.2 Research Objective and Null Hypotheses 9
  - 1.2.1 Research Objective 9
  - 1.2.2 Null Hypothesis 1 10
  - 1.2.3 Null Hypothesis 2 10
- 1.3 Research methodology 10
- 1.4 Outline of the Report 11

2 BACKGROUND 12
- 2.1 Market microstructure 12
  - 2.1.1 Instruments 12
  - 2.1.2 Order Types 12
  - 2.1.3 The limit order book 13
  - 2.1.4 The matching process 13
  - 2.1.5 Transparency 13
  - 2.1.6 Electronic Communications Protocols 14
- 2.2 Agent-Based Artificial Stock Markets 14
  - 2.2.1 Agent-Based Modelling 15
  - 2.2.2 Discrete Time Artificial Stock Markets 15
  - 2.2.3 ABSTRACTE 15
  - 2.2.4 Zero-intelligence model of price formation 16
- 2.3 Trading Algorithms 16
- 2.4 VWAP Trading 17
  - 2.4.1 The Price-Volume Trading Model 17
  - 2.4.2 VWAP Trading Algorithm under the price-volume trading model 18
- 2.5 Tools and Frameworks 19
  - 2.5.1 Open Services Gateway Initiative Framework 19
  - 2.5.2 Java Agent DEvelopment Framework 19
  - 2.5.3 Simple Logging Facade for Java 19
- 2.6 Previous Work 20

3 ANALYSIS AND DESIGN 21
- 3.1 Trading System Design 21
- 3.2 Simulating a Trading System 22
- 3.3 Overall Design 22
- 3.4 Requirements Analysis 24
  - 3.4.1 Functional Requirements 24
  - 3.4.2 Non Functional Requirements 25

3.5 Use Case Analysis 25
4 IMPLEMENTATION AND TESTING 26
4.1 Overall Principles 26
4.2 jade-unit (jade-unit/) 27
4.3 Order Book (market/book/) 28
4.4 Simulation Ontology (simulation/ontology/) 28
4.5 Simulation Agent (simulation/agent/) 28
4.6 Market Ontology (market/ontology/) 29
4.7 Market Agent (market/agent/) 29
4.8 Client API (market/client/) 31
4.9 Noise Trader Agent (agent/noise/) 32
4.10 Vwap Agent (agent/vwap/) 33
4.11 Testing 33
4.11.1 Unit Testing 34
4.11.2 Integration Testing (integration/) 34
4.11.3 Continuous Integration and Test Coverage 34
5 EXPERIMENTS AND RESULTS 35
5.1 Experimental Procedure 35
5.2 Null Hypothesis 1 35
5.2.1 Experimental Results 36
5.3 Null Hypothesis 2 38
5.3.1 Experimental Results 38
5.4 Summary of Results 40
6 CONCLUSIONS AND FURTHER WORK 41
6.1 Summary and Evaluation 41
6.2 Future Work 41
6.3 Final Thoughts 43
A USE CASES 44
B SIMULATION ONTOLOGY MESSAGES 54
C MARKET ONTOLOGY MESSAGES 55
C.1 Fields (eugene.market.ontology.field) 55
C.1.1 Single valued fields 55
C.1.2 Enum fields 56
C.2 Messages 56
C.2.1 Order management messages (eugene.market.ontology.message) 56
C.2.2 Market data messages (eugene.market.ontology.message.data) 58
D LOGGING 59
D.1 Events 59
D.1.1 NEWORDER 59
D.1.2 REJECTORDER 59
D.1.3 ADDORDER 60
D.1.4 EXECUTION 60

D.2 marketData.log 60
D.3 executions.log 60
D.4 rejections.log 61
E USER AND SYSTEM MANUAL 62
E.1 Requirements 62
E.2 Build 62
E.3 Run 62
E.4 Distribution 62
E.5 Continuous Integration 63
E.6 Wiki and Issue Tracker 63
E.7 Version Control 63
E.8 Licence 63
F PROJECT COVERAGE 64

1 INTRODUCTION

1.1 MOTIVATION

The design of a software system gradually evolves over time, as the understanding of the problem domain improves and the business requirements change. The cost of change of a software system is not negligible, however it is relatively low in comparison to other Engineering Disciplines, and can be controlled by adhering to principles of good design and by following software development processes [21]. This propensity encourages an iterative approach to software development; controlled improvements, extensions and revisions of the artefacts move the system from one version to the next [7].

The key to controlled evolution of a software system is to rely on rigorous testing and validation at all levels of the architecture: from testing individual units of code (Unit Testing), through testing interfaces between components (Integration Testing), to testing a completely integrated software system (System Testing). Recent trends in software development practices even encourage the tests of a functionality to be written prior to the sourcecode [10]. It is argued that such systems should promote low-coupling, as their design needs to accommodate testing components in isolation and allow control over the dependencies, however evidence is contradictory as to the exact effects of following this practice [39].

In order to ensure the correct operation of a software system over time, all the levels of testing need to be automatic and fast, in order to encourage the developers to perform them locally, following every change (Automated Testing, Continuous Integration). Adhering to those principles reduces reliance on manual testing, prevents regression errors in existing functionality (Regression Testing) and improves release time.

These principles apply well to deterministic systems; testing comes down to verifying that a set of inputs will produce a particular set of outputs, because the operation of such a system can be approximated with a deterministic-state machine. The difficulty occurs when dealing with non-deterministic systems; testing involves verifying that a particular set of inputs will produce a correct set of outputs with a statistically significant probability.

Since the mid-80s, financial markets have been undergoing a phase of extensive automation of the way order flows are handled. The first revolution took place with the

introduction of electronic markets with an open access to the limit order book, that replaced traditional, open-outcry markets. However, since the mid-2000s, a second revolution is in the making: Trading Algorithms have been replacing human traders in the process of negotiating prices and reducing market impact. Those sophisticated real-time systems, designed to replicate certain trading strategies, have a clear advantage over human traders in the amount of information they can take into account, as well as the speed at which they are able to take positions [28].

The actions of the Trading Algorithms take an active part in shaping the market dynamics and, although constantly monitored, they sometimes contribute to *flash crashes*: brief periods of extreme market volatility. On May 6, 2010, the U.S. stock market experienced a sudden price drop of 5%, followed by a rapid recovery, all in the course of about 30 minutes. One subsequent analysis of that incident concluded that:

> [...] technological innovation is critical for market development. However, as markets change, appropriate safeguards must be implemented to keep pace with trading practices enabled by advances in technology [27].

Overall, lack of rigorous testing of Trading Algorithms and other trading systems does not only affect the P&L of the owner of the technology, but may also lead to systemic instability of the market.

Trading Algorithms are an example of a non-deterministic system that operates in a highly non-deterministic environment. They are designed to respond to the situation on the market and to have a certain expectation as to the effect their actions will have on the market.

Conventionally, Trading Algorithms are backtested on historical data by rebuilding the past order flows on the limit order book. Certain characteristics can be measured (e.g. VWAP of the algorithm) and compared to corresponding benchmarks (e.g. VWAP of the Market) to evaluate whether the algorithm is doing the right thing. This approach can give certain level of confidence as to the correct operation of the trading algorithm, however it nonetheless fails to model the reactions of other market participants to the order flows coming from the algorithm [15].

In recent years, the Agent-Based Modelling (ABM) approach has been applied to understanding the complex phenomena observed in economic and financial systems. An Agent-Based Model is a computer simulation of evolving and autonomous software decision makers (agents) that interact through a set of prescribed rules. Applying ABM models to financial markets offers the possibility of studying how behaviour of individual agents affects the overall market dynamics [11, 19]. Most widely discussed approaches to Agent-Based Artificial Stock Markets implement discrete time, call markets and employ various techniques in order to simulate traders' behaviour [25]; recently, however, the researchers have been moving towards asynchronous, continuous time simulation [11, 22].

In an attempt to supplement the strategy of backtesting Trading Algorithms on historical data and address its shortcomings, in this report we propose *Eugene*: a real-

time, continuous trading session Agent-Based Artificial Stock Market for Validation and System Testing of Trading Algorithms. We aim to demonstrate that simulations involving simple market participants (`Noise Traders`) can be used for discovering a difference in behaviour between a correctly and incorrectly implemented VWAP Algorithm. Furthermore, we explore whether the proposed implementation is capable of discovering a difference in behaviour between two correctly implemented VWAP Algorithms, but ones that trade with different frequency.

We will start by formulating the research objectives and null-hypotheses, followed by a discussion of the research methodologies.

1.2 RESEARCH OBJECTIVE AND NULL HYPOTHESES

1.2.1 Research Objective

We focus in this thesis on testing Trading Algorithms using Agent-Based Simulation and try to demonstrate that the results obtained from the simulations can be used to automatically detect a difference in behaviour resulting from simple programming errors using statistical analysis. This will allow us to clearly test whether the proposed approach could be successfully applied as a complementary tool for verification of Trading Algorithms. We decided to focus on semantic programming errors, as those are particularly difficult to discover using other approaches.

As an extension, we attempt to explore whether the proposed system is able to detect a difference in behaviour of two correctly implemented algorithms, but ones that operate with different parameters. If successful, we will demonstrate that the proposed methodology could potentially be applied to evaluating economic performance of Trading Algorithms.

In order to demonstrate the efficacy of applying this approach to verification of Trading Algorithms, we will study an implementation of a VWAP algorithm that will trade in a Simulated Stock Market that consists of Noise Traders. Given this VWAP algorithm implementation, a logical polarity error will be introduced into the implementation, which will cause the algorithm to always cross the spread, by checking the wrong side of the limit order book for a price when sending a limit order. We hypothesise that by performing statistical analysis on the distribution of errors between VWAPs achieved by both algorithms and those of the overall market, it will be possible to detect a statistically significant change in the distribution of errors. In this work we do not focus on building a classification of errors and their possible manifestations, and thus we are not able to demonstrate whether the change

in behaviour is a result of a particular error; we only set out demonstrate that the two tested setups do exhibit statistically different behaviour.

Similarly, we want to see whether the system could be used to detect a change in the distribution of errors from target VWAP between two correctly implemented VWAP algorithms, but ones that divide the trading duration into a different number of equal time intervals. It is hypothesised that the VWAP algorithm that trades more often, should track the target VWAP more closely.

1.2.2 Null Hypothesis 1

> Given a correct implementation of a VWAP Algorithm, if a logical polarity error is introduced that causes the algorithm to always check the wrong price when sending a limit order, there will be no effect on the distribution of percentage errors from the target VWAP.

1.2.3 Null Hypothesis 2

> Given 2 correct implementations of a VWAP Algorithm, one that divides the trading duration into 10 equal time intervals and the second that divides the trading duration into 40 equal time intervals, there will be no effect on the distribution of percentage errors from the target VWAP.

1.3 RESEARCH METHODOLOGY

In order to design a stock market simulator, the first step is to study the relevant aspects of real stock markets, therefore we studied the literature on market microstructure of electronic, limit order book stock markets.

Similarly, in order to realistically model the behaviour of a stock market as a whole, we turned to analyse existing Agent-Based Artificial Stock Markets. We specifically focused on the types of behaviours that are modelled in order to choose an appropriate implementation for the Noise Traders.

The results of the analysis of stock markets and Agent-Based Artificial Stock Markets enabled us to design a flexible and modular framework for the simulator, that effectively mirrors the features of real stock markets. The design and implementation phase followed an iterative approach, with short (2 weeks) iterations, and work pro-

gressed on an issue-by-issue basis (features were submitted to an issue tracker and then closed when implemented).

After completing the design and implementation phases, the simulator was evaluated by performing two experiments that correspond to the two null hypotheses. By introducing the simulator and demonstrating its usefulness in discovering errors in Trading Algorithms we aimed to provide a methodological advancement that will be a useful complement to traditional ways of testing Trading Algorithms.

1.4 OUTLINE OF THE REPORT

In Section 2.1 we describe the model of the market mechanism we will use in the simulator. We describe the market architectures employed in the stock exchanges during continuous trading sessions, namely the model of a limit order book, and how it enables market participants to interact asynchronously.

In Section 2.2 we summarise several artificial stock markets from the available literature. We provide a model of zero-intelligence agents that will be implemented by the Noise Traders, that is based on the work by Gilles [22, chap. 4]. Moreover, in order to provide a background for implementing a VWAP Trading Algorithm for the experiments, we survey available sources, most prominently the work in Coggins et al. [15], Kakade et al. [26].

Chapter 3 provides an analysis of the design of the system, by presenting the main elements of a Trading System and *Eugene*'s place in the architecture. We introduce the overall design and relations between the main modules and provide a summary of requirements and use cases that the modules set out to implement.

In Chapter 4 we highlight the main principles that guided the implementation of the simulator and provide the implementation details of all modules. Lastly, the unit and integration testing strategies are presented.

Chapter 5 summarises the steps taken in order to evaluate the efficacy of the system according to the null-hypotheses presented in Section 1.2.

Finally, in Chapter 6 we summarise the key points of this work and its applicability to testing Trading Algorithms. We then review the contributions and achievements introduce the possible extensions to this project.

2 BACKGROUND

2.1 MARKET MICROSTRUCTURE

In this section we describe the model of the market mechanism we will use in the simulator. We describe the market microstructure employed in major stock exchanges during continuous trading sessions, namely the model of a limit order book, and how it enables market participants to interact asynchronously (for a study of market microstructures in main stock exchanges, see [16]).

2.1.1 Instruments

Instruments are the different types of *contracts* that can be traded on an exchange, e.g. cash equities, fixed income and derivatives. Every instrument is associated with a set of technicalities, such as the minimum tick size (minimum amount of money by which the price can change) or price variation controls (conditions for a market halt due to unexpected price volatility). In this work we focus on cash equities trading, but the methodology could be applied to trading other instruments.

2.1.2 Order Types

Market participants indicate their intentions in the form of trading instructions called orders. We will consider only two types of orders: limit and market orders. A market order specifies the instrument to trade, the quantity (order size) and the side of the trade (buy or sell). A limit order additionally specifies a limit price: the maximum (buy) or minimum (sell) price that the trader accepts for an order.

Traders can also cancel existing orders that have not been executed. Lillo and Farmer [29] estimate that on the London Stock Exchange (LSE) on-book market, up to 30% of outstanding limit orders are cancelled before execution and order cancellations play an important role in the price formation process.

2.1.3 The limit order book

A limit order book for a single instrument consists of limit orders, sorted by price and time of arrival and stored in two queues: one for bid (buy) orders and one for ask (sell) orders. At a specific time t, the order book can be described as [22]:

$$\beta_n \leqslant \ldots \leqslant \beta_2 \leqslant \beta_1 < \alpha_1 \leqslant \alpha_2 \leqslant \ldots \alpha_m$$

where β_i represent bid orders and α_j represent ask orders. The highest bid β_1 (or best bid) and lowest ask α_1 (or best ask) define the spread $\alpha_1 - \beta_1$, and are referred to as the top of the book.

2.1.4 The matching process

An incoming order goes through the matching process. If the incoming order is a market order and there are no limit orders on the opposite side of the limit order book, the order is rejected. If the incoming order is either a market ask order or a limit ask order, the matching process will proceed from the highest bid. β_1 will be executed only if the incoming order is a market order, or a limit order with a limit price lower than or equal to β_1. If that is the case, the orders match, therefore a trade will be executed at the price of β_1 and the quantity of the smaller of the two orders. If the quantity specified by β_1 is smaller than or equal to that of the incoming order, β_1 is said to be filled. If the trade leaves the incoming order with outstanding quantity, the matching process continues on the following orders in the queue, until either the incoming order is filled, or there are no more matching orders, in which case if the incoming order is a market order, the remaining quantity will be rejected, otherwise if the incoming order is a limit order, the remaining quantity will be placed at the top of the ask queue.

Similarly, if the incoming order is a market bid order, or a limit bid order, the matching process will proceed in an analogous way on the ask side.

2.1.5 Transparency

Market transparency is defined as the *"ability of market participants to observe information in the market"*[16]. It can refer to two stages in the lifetime of an order: pre-trade and post-trade transparency. Pre-trade transparency refers to the ability of other market participants to observe the limit orders entering the order book, whereas post-trade transparency refers to observing trades after they have taken place.

The extent of pre-trade transparency varies across different exchanges, but generally two levels of disclosure emerge: level 1 and level 2. Level 1 usually refers to publishing best bid/ask quotes with aggregate volumes, whereas level 2 discloses entire limit order book in real-time (often referred to as tick-by-tick, where a tick refers to a single execution), by publishing messages whenever a new limit order enters the limit order book, or an existing limit order is cancelled or executed. In case of level 2 access, the identity of the traders is hidden behind exchange-generated order IDs [16].

2.1.6 Electronic Communications Protocols

Since the mid 80-s, the financial markets have undergone a phase of extensive automation towards an electronic handling of order flows. This has led to the establishment of industry-wide messaging protocols for real-time exchange of financial information.

The de-facto standard for order management is *Financial Information eXchange (FIX) protocol* [20], that defines message types and fields that can be used for pre-trade communications and trade execution.

Similarly, exchanges usually publish real-time full depth of book quotations and execution information (level 2) through a derivative of the *ITCH* protocol (in this work we focus on the *BATS Multicast PITCH 2.X protocol* [9]). Whereas *FIX* is the de-facto standard for order management, the exchanges compete on the speed of Level 2 access and the design of the protocol can have a great impact on the overall latency, therefore the exact format differs from exchange to exchange.

2.2 AGENT-BASED ARTIFICIAL STOCK MARKETS

In this section we summarise several artificial stock markets from the available literature. We provide a model of zero-intelligence agents that will be implemented by the Noise Traders (Section 4.9), that is based on the work by Gilles [22, chap. 4]. Moreover, in order to provide a background material for implementing a VWAP Trading Algorithm for the experiments, we survey available sources, most prominently the work in Coggins et al. [15] and Kakade et al. [26].

2.2.1 Agent-Based Modelling

Agent-Based Modelling is a simulation technique concerned with designing societies of rule-based software agents that interact in particular ways, with a view of assessing the influence of individual (or groups of) agents on the system as a whole. Among the benefits of agent-based modelling is the ability to study emergent phenomena: behaviour resulting from complex interactions of many individual entities.

Agent-Based Models that attempt to explain economic processes are branded as Agent-Based Computational Economics [41]. In recent years, building Artificial Stock Markets has become a promising research area due to the fact that this methodology reflects the fundamental nature of a stock market, where the current situation is a result of the complex interaction of actions of many heterogenous investors that have various expectations and different levels of rationality.

2.2.2 Discrete Time Artificial Stock Markets

Most efforts in Artificial Stock Markets to date focus on studying discrete time simulations, i.e. at each time period t, some fraction of traders submit orders to the market, the orders are cleared, a new market price is announced, the agents update their positions and a new time period $t + 1$ starts. This style of simulation is well suited to study of *call markets* [18], where batches of orders are executed at predetermined time intervals, at prices that best match supply and demand. The ASMs employ a wide range of techniques in order to best approximate *stylised facts* (i.e. statistical properties) observed in real stock markets, with promising results. For a comparative study of discrete time Artificial Stock Markets, see Jha et al. [25] and Boer-Sorban [11].

2.2.3 ABSTRACTE

The discrete time simulation fails to reflect the nature of continuous trading sessions, that evolve in a highly asynchronous manner thanks to the limit order book microstructure. The importance of continuous time simulation was demonstrated by Boer-Sorban [11], who describes *ABSTRACTE*: Agent-Based Simulation of Trading Roles in an Asynchronous Continuous Trading Environment. *ABSTRACTE* was used to model a market with information asymmetry, where prices are set by a learning market maker; this model extends the work in [37], but evolves the simulation in continuous time. Boer-Sorban [11] showed that moving to asynchronous, continuous time simulation renders different price dynamics and concluded that the continuous nature of trading in real stock markets should be explicitly taken into account in agent-based models.

2.2.4 Zero-intelligence model of price formation

Another strong case for turning to continuous time simulation is presented by Gilles [22, chap. 4], who studied the primary role played by liquidity dynamics in the price formation mechanism. Liquidity is modelled using zero-intelligence agents trading through the asynchronous limit order book, whose order placement and cancellation process, in terms of order type, size and limit price, are designed to come as close to a realistic aggregate order flow. The simulation is implemented on top of the *Natural Asynchronous-Time Event-Lead Agent-Based Platform (NatLab)* [22, chap. 3].

The simulation consists of N Zero-intelligence agents, initially endowed with a fixed amount of cash and shares (similarly to [32]). The stock does not distribute dividends and cash does not yield interest; the price formation mechanism arises through the non-linear interactions between agents' demand and supply.

The agents place buy or sell orders with equal probability. The agents can either cancel an existing order with probability π_c, send a market order (π_m), or send a limit order ($1 - \pi_c - \pi_m$). The limit order price is either in the spread and uniformly distributed between best bid and best ask (π_{in}), or outside the spread and power law distributed (π_{out}).

The order sizes of limit orders follow a log-normal distribution, and market order sizes match the best counterpart. In order to model the well documented pattern of trading activity that is more dense at the start and end of the day [13], the agents sleep for random times that are drawn from a stretched exponential distribution, that is obtained as a mixture of two exponential distributions with different means (this work cites the influence of sleep time on price dynamics, as explored in [38]).

This simple model of zero-intelligence agents was able to reproduce realistic dynamics in terms of spread and book shape, and some stylised facts. A simplified version of zero-intelligence model [22, chap. 4] will be used to implement the Noise Traders (see Section 4.9).

2.3 TRADING ALGORITHMS

Trading Algorithms are sophisticated real-time systems that enter trading orders and decide on aspects of the order like timing, price and quantity, without human intervention. The basic classification can be made between *benchmark* and *profit* algorithms. Benchmark algorithms are designed to be used by human traders in order to implement trading strategies in a cost-effective manner, whereas profit algorithms proactively search for profit-opportunities and enter the market without human intervention. For an overview of Trading Algorithms, see [31].

2.4 VWAP TRADING

Whenever a large institutional investor wants to take a position or liquidate a holding, the execution of the transaction is faced with price risk. Putting a large limit order onto the limit order book gives an incentive to other investors to change their prices (up or down, against the large limit order) and hence drive the cost of execution unnecessarily high.

The solution is to split the parent order into smaller child orders in order to hide the intent of the investor and keep the market impact under control. Traditionally, such task would be delegated to human traders, however following the advent of algorithmic trading, it is usually a machine that executes the order. The quality of execution is measured by comparing against an appropriate benchmark.

Among the most popular benchmarks is *VWAP* or *Volume Weighted Average Price*, that is a measure of average price achieved in the market. When used to measure the quality of execution of an algorithm, the volume weighted prices achieved by the algorithm are compared to all the other trades that occurred in the market during the period of the algorithm's activity.

A *VWAP Algorithm* attempts to buy or sell a fixed number of shares at a price that closely tracks the *VWAP* of the market. Therefore, the problem of tracking the market *VWAP* can be stated in terms of splitting the order into a series of smaller orders, whose size corresponds to the forecast intra-day volume pattern of a stock.

2.4.1 The Price-Volume Trading Model

In the *price-volume trading model* [26], the *intra-day* trading activity can be summarised by a discrete sequence of price and volume pairs (p_t, v_t), for $t = 1, \ldots, T$. Each pair represents the fact that a total volume of shares v_t was traded at a price p_t.

Assume that there is a trading algorithm A that traded during this period. Then, the market VWAP, $VWAP_m$, for an intraday trading sequence $S_m = (p_1, v_1), \ldots, (p_T, v_T)$ that excludes the trades executed by the algorithm A, is then defined as follows:

$$VWAP_m(S_m) = \frac{\left(\sum_{t=0}^{T} (p_t \times v_t)\right)}{\sum_{t=0}^{T} v_t} \tag{2.1}$$

Similarly, the *VWAP* the algorithm A, $VWAP_A$, for a sequence $S_A = (p_1, v_1), \ldots, (p_T, v_t)$ is defined as follows:

$$VWAP_A(S_A) = \frac{\left(\sum_{t=0}^{T}(p_t \times v_t)\right)}{\sum_{t=0}^{T} v_t} \tag{2.2}$$

We also define *percentage-error from target VWAP* for the algorithm A, that achieved $VWAP_A$ compared to market $VWAP_M$:

$$\Delta\%_{VWAP_M, VWAP_A} = 1 - \frac{VWAP_A}{VWAP_M} \tag{2.3}$$

2.4.2 VWAP Trading Algorithm under the price-volume trading model

Coggins et al. [15] define the following rule based approach to a *VWAP* **buy** execution for a stock S (the definition for a sell algorithm follows):

1. Divide the trading period into N equal time slots, allocating a given percentage of trade volume to each time interval. The percentage volume targets would be derived from historical trading volumes for the stock S, averaged over some past training period.
2. At each time slot, submit a limit order at the best bid, with the size that corresponds to the percentage of trade volume allocated to this slot.
3. If within x minutes, the best bid has gone up and the current order has not executed, amend the order to the best bid.
4. If by the end of the time slot, the order has not fully completed, amend it to become a market order to force completion.

2.5 TOOLS AND FRAMEWORKS

2.5.1 Open Services Gateway Initiative Framework

`OSGi` [35] is dynamic module and service system platform for Java. Application components (distributed as `bundles`) can be remotely managed without requiring a reboot of the entire container. A package management system enables developers to package public and private APIs within the same bundle, but only expose public APIs at runtime. The dynamic service system allows the components to discover the addition of new services and act accordingly.

Eugene mainly uses `OSGi` as a convenient deployment system, but also to enforce the principle of information hiding (see Section 4.1) through the use of private APIs that are hidden at runtime.

2.5.2 Java Agent DEvelopment Framework

`JADE` [24] is a software framework for developing distributed, multi-agent software systems in Java. `JADE` operates as a set of nodes which together form a platform (cluster); the nodes can be separated by a network layer, but standalone mode is also available. Each node hosts a set of `Agents` and is responsible for managing asynchronous communication between them and the other nodes. Each `Agent` (encapsulated in `jade.core.Agent` class) operates in a separate thread of control, therefore allowing independent, preemptive behaviour. `Agents` in `JADE` are not units of behaviour; they are only responsible for scheduling the different many behaviours that an `Agent` can have and exposing a set of primitives for sending and receiving messages (message queues). The actual behaviours are encapsulated in `jade.core.behaviours.Behaviour` class and its many subclasses, designed as building blocks for forming more sophisticated behaviours.

`JADE` exposes an `OSGi` service, therefore each `JADE` node can be run inside an `OSGi` container. However, we only use `OSGi` at deployment time, not during testing; that is because no part of *Eugene* directly depends on `OSGi` and due to overall difficulty of testing code inside an `OSGi` container.

2.5.3 Simple Logging Facade for Java

`SLF4J` [40] is an abstraction for various logging frameworks that allows the user to plug in the desired implementation at deployment time. Combined with `LOGBack` [30]

implementation, it is a very powerful and versatile logging framework. Among the most useful features is the ability to add *Markers* to a log entry to indicate the type of the event being logged and redirect events to different files based on the *Markers*.

2.6 PREVIOUS WORK

Previous work referenced in Section 2.2 focuses on realistically simulating Stock Market activity. However, we are not aware of any application of Agent-Based Modelling to test Trading Algorithms for existence of programming errors and incorrect behaviour, as a way to complement backtesting.

3 ANALYSIS AND DESIGN

3.1 TRADING SYSTEM DESIGN

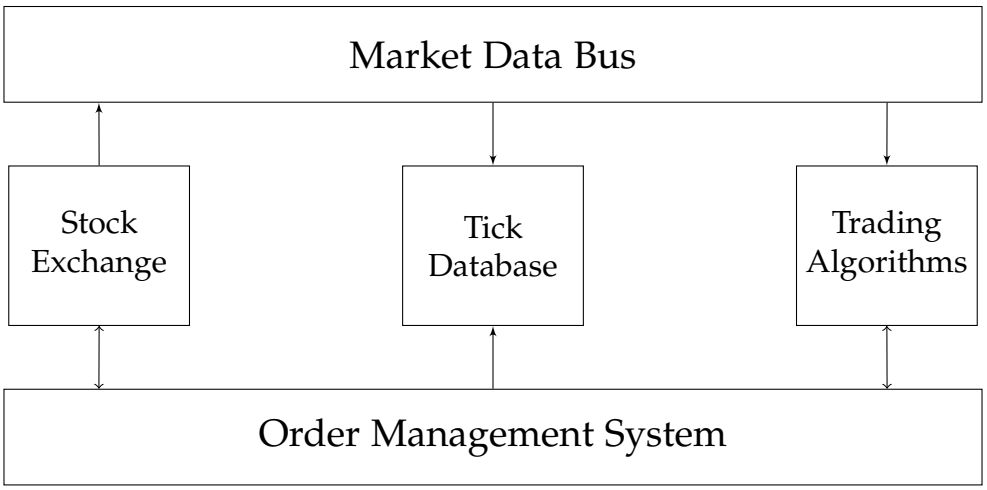

Figure 3.1: Trading System Architecture

Figure 3.1 shows a high-level design and the direction of the flow of messages in a Trading System [12]. Various market participants submit orders to the Stock Exchange in order to trade. Different stock exchanges publish Level 2 data feed messages in a variety of formats (Section 2.1.6). In order to deal with this complexity, investment banks have developed internal Market Data Buses that subscribe to feeds published by different stock exchanges in order to republish them in a standardised way to various internal systems, e.g. Trading Algorithms.

Similarly, in order to allow Automatic Order Routing between different stock exchanges and enhance other back-office roles, investment banks develop Order Management Systems that register with various stock exchanges in order to provide a standardised way of submitting orders for various internal systems, e.g. Trading Algorithms.

Historical data is stored in Tick Databases that subscribe to the Market Data Bus in order to perform statistical analysis on the behaviour of different stocks. The result of the statistical analysis is a set of parameters which describe various aspects of stock

behaviour. The parameters are published to various internal systems, e.g. Trading Algorithms.

Trading Algorithms are highly sophisticated and parameterised systems which accept parent orders and execute them using different strategies (Section 2.3). Trading Algorithms register with the Market Data Bus in order to react to the current situation on the market. They continuously compare the behaviour of stocks with their historical behaviour (using data and analysis from the Tick Database) in order to minimise market impact of the different strategies.

3.2 SIMULATING A TRADING SYSTEM

Trading Algorithms respond to messages received from the Stock Exchange (via the Market Data Bus) and send orders to the Stock Exchange (via the Order Management System). They expect the orders to have some influence on the situation on the Stock Exchange, based on the analysis of past behaviour received from the Tick Database. In our implementation we are going to explicitly omit the issue of past behaviour analysis and the Tick Database, as it is beyond the scope of this report[1].

Therefore, in order to simulate a Trading System for testing Trading Algorithms, two systems need to be put in place:

- Order Matching Engine that will accept orders on to a limit order book, match orders and send executions/market data back to traders, as described in Section 2.1.
- Market behaviour simulator to generate a realistic order flow, as described in Section 2.2.

3.3 OVERALL DESIGN

Figure 3.2 shows the high-level design and messages exchanged between different agents. All messages explicitly carry the information about the agent that originates the message.

The role of the Order Matching Engine is handled by the `Market Agent` that accepts all messages and passes the responsibility of processing the messages to the

1 The Tick Database is not explicitly modelled, however the VWAP algorithm (see Section 4.10) will be parameterised and provided with expected volume.

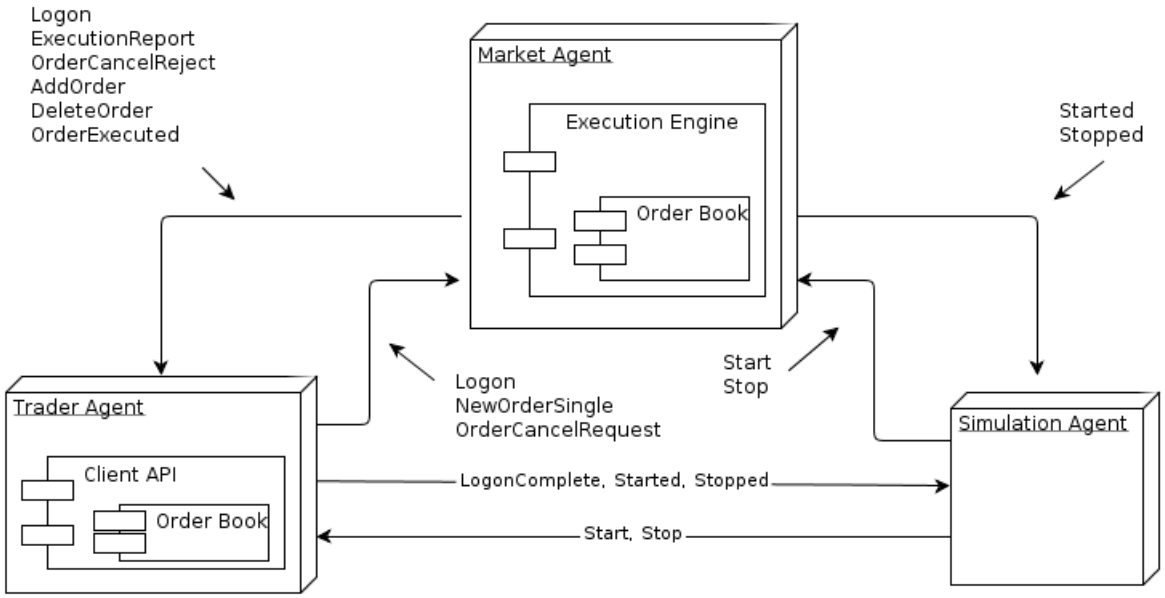

Figure 3.2: Overall Design. The arrows refer to the names and flow of messages.

`Execution Engine`[2]. Similarly, all market events (accepting a new order, executing orders, cancelling orders) are also disseminated by the `Market Agent` to `Trader Agents`. `Trader Agents` are simply the agents that simulate the participants in the market process. All orders submitted to the `Market Agent` need to carry a unique `ClOrdID` generated by the `Trader Agent` that sends the order. Whenever a market event is sent to the `Trader Agents` other than the original `Trader Agent` that submitted the order, unique `OrderIDs` generated by the `Market Agent` are sent instead of the `ClOrdIDs` (similarly all executions carry a unique `TradeID`, see Section 2.1.5).

The simulation lifecycle is controlled by the `Simulation Agent` that is responsible for starting and stopping the simulation, in particular starting the `Market Agent` and `Trader Agents`, and letting the `Trader Agents` know the identity of the `Market Agent`. Additionally, the `Simulation Agent` can optionally send initial orders to the `Order Book`, before any `Trader Agent` starts sending orders. The initial orders are sent in order to stabilise the order book.

In order to simplify the low-level details of dealing with the `Market Agent` and the `Simulation Agent`, the `Trader Agents` use the `Client API` that provides a simple way to send and receive messages. In particular, the `Client API` handles the generation of unique `ClOrdIDs` and provides several services like rebuilding the limit order book from the messages received from the `Market Agent`, so that `Trader Agents` can track the current prices and the overall situation on the market.

2 `Execution Engine` is part of the `Market Agent`, therefore there are no actual messages exchanged, just method calls, see Section 4.7 for details

3.4 REQUIREMENTS ANALYSIS

Having analysed available literature on the design of asynchronous, limit order book markets in Section 2.1, as well as existing Agent-Based ASMs in Section 2.2, we will now list the requirements that the trading simulator needs to satisfy. Requirements are prioritised according to the MoSCoW approach (M - Must, S - Should, C - Could and W - Would).

3.4.1 Functional Requirements

ID	Requirement	Priority
	Market Agent	
F01	The Execution Engine shall match orders in the Order Book	M
F02	The Market Agent shall accept limit/market orders and order cancellations.	M
F03	The Market Agent shall provide *level 2* market data access with anonymous OrderIDs to Trader Agents.	M
F04	The Market Agent shall report order executions and status changes back to the original Trader Agent.	M
F05	The Market Agent shall log all market events to a file.	M
F06	The Market Agent shall publish performance statistics at runtime.	C
	Simulation Agent	
F07	The Simulation Agent shall start the Market Agent and Trader Agents.	M
F08	The Simulation Agent shall synchronise the start of a simulation.	M
F09	The Simulation Agent shall stop a simulation after a specified amount of time.	M
F10	The Simulation Agent shall send initial orders before starting the simulation.	M
	Client API	
F11	The Client API shall handle the low-level interaction with the Market Agent and Simulation Agent and provide callback methods for receiving messages.	M

3.4.2 Non Functional Requirements

ID	Requirement	Priority
No1	The System shall maintain $\geqslant$ 70% unit test coverage.	M
No2	The System shall be integration tested.	M
No3	The System shall be highly modular and decoupled.	M
No4	The System shall be very well javadoced.	M
No5	The System shall be implemented on top of the JADE Framework.	M
No6	The System shall closely approximate message types and formats of industry standard messaging formats.	C

3.5 USE CASE ANALYSIS

The use cases are listed in Appendix A. Their application to Integration testing is explained in Section 4.11.2. For details of interaction between the `Market Agent` and `Execution Engine` (especially how the `Execution Engine` communicates with the `Trader Agents`), see Section 4.7, as the description in the use cases is simplified.

4 IMPLEMENTATION AND TESTING

In this chapter we will describe the implementation of the system, first by stating the overall principles in Section 4.1, to then proceed to introduce the main modules, by focusing on most important parts of the implementation in Sections 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9 and 4.10. Finally, in Section 4.11 we will elaborate on both unit and integration testing strategies.

4.1 OVERALL PRINCIPLES

In this section we will describe the overall programming and design principles that have informed the implementation of the system in order to refer to them in the later sections that describe the particular modules of the system. The principles will be described in no particular order, as their scope usually cuts through different architectural levels. Adherence to these principles allows for modular and decoupled implementation (requirement *No3*).

DESIGN BY CONTRACT, ASSERTIONS AND FAIL FAST Input is checked for correctness and the contract for the method or constructor parameters is clearly documented. To that end, `com.google.common.base.Preconditions` class from `guava-libraries`[23] is used extensively, and corresponding tests that verify whether the method or constructor enforces the contract are always present. Post-conditions and invariants are verified in a similar way. Strict adherence to these principles allows for much safer code that is guaranteed to be used correctly. From the point of view of the programmer, the code is much easier to reason about and is guaranteed to fail fast instead of continuing to work incorrectly, whenever an implementation error is present.

SEPARATION OF CONCERNS, PROGRAMMING TO AN INTERFACE AND INFORMATION HIDING The code is separated into many modules, so that responsibility of each part of the code is very narrowly defined. The APIs between different modules are defined in terms of interfaces, and specific implementations are hidden behind factories and private packages (Java does not have private packages, therefore we use the capabilities of the *OSGi* container instead to enforce this principle, see Section 2.5.1). This allows for low coupling between components, that leads to a robust implementation.

IMMUTABLE OBJECTS An object is considered immutable if its state cannot be changed after it is constructed. Reliance on immutable objects is considered a sound strategy for creating simple code. Immutable objects have an additional advantage of inherent *thread-safety*: since there is no state that can be changed, they can be safely shared by multiple threads, without interference. The code relies on the use of the Java `final` keyword extensively to delegate the responsibility of enforcing immutability to the compiler.

4.2 JADE-UNIT (JADE-UNIT/)

In order to fulfil requirements *No1* and *No2*, there needs to be a way to create and tear down an arbitrary number of `JADE` nodes, potentially within a single JVM. Such capability allows for reuse of established testing tools and frameworks.

There exists a `Test Suite Framework` available through the official `JADE` website, however maintenance of two entirely different testing environments seemed like an unnecessary complication.

Unfortunately, *JADE* contains a messaging module[1] that opens a socket and renders creating multiple `JADE` nodes impossible, especially when tests are run in parallel. That is because `JADE` would attempt to open the same socket multiple times, and there was no simple way to randomise the choice of sockets to avoid clashes. Also, side effects in tests are considered bad testing practice.

Luckily, the `JADE` architecture is modular and hence the implementation of the messaging module can be replaced. The `jade-unit` extension implemented for this project provides a no-op messaging module that simply ignores any messages directed to it. This allows the integration tests to reuse the entire testing architecture for tests that need to run inside a `JADE` node. It is expected that the patch file with `jade-unit` extension will find its way to the `JADE` maintainers, in due time.

The `jade-unit` extension does not have an adverse effect on the quality of testing, because the messaging module replaced is used only for cross-node communication (i.e. to send messages to `Agents` located in different nodes), and `JADE` is designed to handle this communication transparently (i.e. `Agents` are not directly affected by their node location, or the node location of other `Agents`).

1 To communicate with other nodes.

4.3 ORDER BOOK (MARKET/BOOK/)

This module implements the data structure behind a limit order book and maintains the status of the limit orders. Due to this separation of concerns, the module is used both by the `Market Agent` (Section 4.7) and `Client API` (Section 4.8) modules (see Section 3.3 for an explanation why the `Client API` needs to maintain a limit order book), thus contributing to requirement *No3*.

Following the principles outlined in Section 4.1, an interface is defined in `eugene.market.book.OrderBook` that specifies the methods for inserting new orders, executing and cancelling existing orders, and inspecting the orders at the top of the book. There are two implementations: `eugene.market.book.impl.DefaultOrderBook` and a delegate `eugene.market.book.impl.ReadOnlyOrderBook`, both hidden in a private package behind the `eugene.market.book.OrderBooks` factory.

Two classes represent an order: `Order` and `OrderStatus` (both located in the `eugene.market.book` package). The `Order` class maintains static information about an order (see Section 2.1.2), whereas the `OrderStatus` class tracks the execution of an order. Both classes are immutable (see Section 4.1). The mutable state, i.e. the mapping from the current `OrderStatus` to `Order`, is maintained by the `OrderBook` implementations internally.

4.4 SIMULATION ONTOLOGY (SIMULATION/ONTOLOGY/)

This module defines message types necessary in order to implement use case *UC1* and *UC9*. All `Simulation Ontology` message types are listed in Appendix B.

4.5 SIMULATION AGENT (SIMULATION/AGENT/)

The `Simulation Agent` implements use cases *UC1* and *UC9*. It starts the `Market Agent`, awaits for messages that synchronise the start of `Trader Agents`, sends the initial orders and starts and stops the simulation.

When starting the `Trader Agents`, `Simulation Agent` passes an implementation of `eugene.simulation.agent.Simulation` that defines methods for obtaining the address of the `Market Agent`, the `Simulation Agent` and the instrument definition (name, tick size and default price).

4.6 MARKET ONTOLOGY (MARKET/ONTOLOGY/)

This module defines message types necessary in order to implement use cases *UC2-UC8*. All `Market Ontology` message types and fields are listed in Appendix C.

In order to fulfil requirement *No6*, message names and fields, including their documentation, are modelled after a subset of the *Financial Information eXchange (FIX) protocol* [20] for order management messages and *BATS Multicast PITCH 2.X protocol* [9] for *Level 2* market data feed (Section 2.1.6). Both order management and market data feed messages in the `Market Ontology` use the same format (i.e. both message types extend `eugene.market.ontology.Message` class) and, where possible, message fields from *BATS Multicast PITCH 2.X protocol* are substituted with message fields from *Financial Information eXchange (FIX) protocol*.

The actual implementation of `Market Ontology` is loosely based on the *QuickFIX/J* message classes [36], but simplified accordingly. Porting *QuickFIX/J* messages to `JADE` was attempted, however quickly abandoned as very time-consuming and without a guarantee of success. Implementing a simplified version of the *QuickFIX/J* messages turned out to be much less involved. The task could, however, be attempted in the future, in which case the decision to keep the architecture similar presents itself as a clear advantage.

In order to correctly deal with prices and explicitly manage the problem of rounding to the nearest tick, *QuickFIX/J* messages use the `java.math.BigDecimal` class. However, `JADE` does not natively support serialising `java.math.BigDecimal` fields, therefore using those required implementing a new codec and thus a greater understanding of low-level details of `JADE`'s serialisation mechanism. Luckily, the problems were anticipated and appropriately planned for.

4.7 MARKET AGENT (MARKET/AGENT/)

The `Market Agent` plays the role of the Stock Exchange with a central limit order book. The implementation is very clearly split between the messaging subsystem and the part that maintains the order book, executes orders etc., in order to maintain clear separation of concerns and improve ease of unit-testing.

The messaging subsystem implementation is located in the `eugene.market.esma.impl.behaviours` package and consists of implementations of `jade.core.behaviours.Behaviour` class (see Section 2.5). The `OrderServer` accepts messages to create new orders, cancel existing orders and handle the logon, i.e. messages from the `Market Ontology` (Section 4.6). Similarly, the `SimulationOntologyServer` deals with messages from the `Simulation Ontology` (Section 4.4). Furthermore, the

`MarketDataServer` deals with sending out trade confirmations to counter parties and market data events (by inspecting the `MarketDataEngine`, explained below). In order to satisfy requirement *F05*, the `MarketDataServer` logs the events (using *SLF4J*, see Section 2.5). The different log files and the format of the logs are explained in Appendix D.

The second part of the `Market Agent` is located in the `eugene.market.esma.impl.execution` package. The `ExecutionEngine` deals with managing the order book, accepting new orders, cancelling existing orders and matching orders. All the various parts of the `ExecutionEngine` are refactored to separate classes for ease of testing: `MatchingEngine` implements the matching algorithm (Section 2.1.4) and `InsertionValidator` checks whether a market order can be accepted or should be rejected (Section 2.1.4). Every change to the limit order book (inserting a new order, cancelling an existing order and executing an order) triggers an event that is recorded in the `MarketDataEngine` (`eugene.market.esma.impl.execution.data` package). The separation of the order characteristics (price, size, type in `Order`) from the current execution status (`OrderStatus`) in the `Order Book` module, greatly simplifies the implementation of the `MarketDataEngine`: the events can refer to a snapshot of a status of an order and this snapshot can be accessed at any time, as long as a reference to the appropriate `OrderStatus` object is maintained.

The points of synchronisation between the messaging subsystem and the execution subsystem are maintained in two classes: the `Repository` (`eugene.market.esma.impl` package) and the `MarketDataEngine`. The `Repository` maintains the mapping between current active orders and the owner traders; whenever an order is accepted by the `OrderServer` it is recorded in the `Repository`. Similarly, the `MarketDataServer` retrieves events from the `MarketDataEngine` and sends them out.

In order to achieve high incoming order rate to handle the required number of `Trader Agents` (see Chapter 5), the `OrderServer` and the `SimulationOntologyServer` operate in a different thread than the `MarketDataServer`. Therefore, both the `Repository` and the `MarketDataEngine` need to be thread-safe (but not the rest of the classes, as they are not shared by different threads). Due to a low number of synchronisation points and clear separation between mutable and immutable state of the order book (Section 4.3), going from single-threaded to multi-threaded design of the `Market Agent` was relatively straightforward.

In order to satisfy requirement *F06*, various parts of the `Market Agent` publish performance statistics at runtime, using the `Metrics` library [14]. Time to reply to messages and their rate, size of the incoming and outgoing queues, are all published to a `JMX` bean, that can be read using, e.g. `JConsole` [34] that is bundled with `Java JDK`. Using those statistics we determined that the `Market Agent` can easily keep up with the required number of `Trader Agents` (see Chapter 5).

4.8 CLIENT API (MARKET/CLIENT/)

The `Client API` fulfils requirement *F11*. The primary purpose is to handle the intricacies of the simulation startup, provide a single gateway responsible for receiving messages and routing them to the user code, and provide simple method calls that construct `JADE` messages from the `Market Ontology` objects.

Following the principles outlined in Section 4.1, an interface is defined in `eugene.market.client.Session` that specifies the methods for sending and extracting the `Market Ontology` objects from `JADE` messages. The implementation is located in `eugene.market.client.impl.SessionImpl`, however no client code uses this class directly. In order to connect to a `Market Agent` defined by the current `Simulation` (Section 4.5), the `Trader Agent` will obtain a `Behaviour` from the `eugene.market.client.Sessions` factory that will establish the connection. The actual implementation is located in `eugene.market.client.impl.SessionInitiator`, in a private package.

In order to establish a `Session`, the `Trader Agent` needs to provide an implementation of the `eugene.market.client.Application` interface, that will receive notifications of all messages that will be sent and received by the `Session`. If the `Trader Agent` requires several objects listening to messages, those can be routed using an implementation of `eugene.market.client.ProxyApplication`, obtained from the `eugene.market.client.Applications` factory. The `ProxyApplication` will route messages to other `Application` instances given to it (`Application` instances can be added and removed dynamically).

One use case, when the `Trader Agent` requires messages to be routed to several objects, is when there is a need to build the order book, as well as calculate statistics about the behaviour of the instrument traded. In the interest of separation of concerns, these two roles should be implemented by different classes, and indeed such separation is possible thanks to the `ProxyApplication`.

In the interest of code reuse, an instance of an `Application` that will build the order book can be obtained from the `Applications` factory. The `proxy` pattern allows for ease of composing different `Application` instances to create complex dependency graphs: indeed an example is `eugene.market.client.TopOfBookApplication` that requires an instance of `OrderBook` that is updated from messages and exposes a simple API for tracking the last price at the top of either side of the book.

The last functionality worth mentioning is that `Session` tracks the status of orders submitted: user code can register an instance of `eugene.market.client.OrderReferenceListener` to receive updates about a single order. Similarly to `Application`, order updates can be routed if more than one instance of `OrderReferenceListener` needs to be notified

(`eugene.market.client.OrderReferenceListenerProxy` that can be obtained from `eugene.market.client.OrderReferenceListeners` factory).

4.9 NOISE TRADER AGENT (AGENT/NOISE/)

The `Noise Trader Agent` implements the *zero-intelligence model of price formation* (see Section 2.2.4), with the following simplifications:

- The `Noise Trader Agent` does not track neither its cash nor inventory position. This decision was made in order to simplify the implementation.
- The sleeping times come from a uniform distribution $[1, s_{max}]$, as the library code used for generating numbers from required distributions did not have the functionality to generate numbers from a mixture of two exponential distributions, and no suitable replacement library was found. Moreover, preliminary results showed that the `Noise Trader Agent` activity was irregular enough for the experiments.

The remaining parameters and distributions come from the original paper by Gilles [22] and are summarised in Table 4.1. The `Noise Trader Agent` uses `commons-math` library [8] to generate random numbers from required distributions. Also, the `Noise Trader Agent` uses the `Client API` (see Section 4.8) to rebuild the `Order Book` in order to track current prices, through the `TopOfBookApplication` (see Section 4.3) .

Description	Parameter	Value
Prob. of cancelling an order	π_c	0.5
Prob. of market order	π_m	0.15
Prob. limit order in spread	π_{in}	0.35
Limit price outside spread	α	0.3
Order size	(μ, σ)	(4.5, 0.8) shares
Sleeping time	s_{max}	3000 ms

Table 4.1: Summary of `Noise Trader Agent` parameters.

4.10 VWAP AGENT (AGENT/VWAP/)

The `Vwap Agent` is an implementation of the algorithm described in Section 2.4.2. There are two implementations available, to facilitate the experiment for *Null Hypothesis 1* (see Section 1.2.2), both hidden behind the `eugene.agent.vwap.VwapAgents` factory. The `Vwap Agent` implementations accept an implementation of the `eugene.agent.vwap.VwapExecution` interface (an instance of which can be obtained from the `eugene.agent.vwap.VwapExecutions` factory), that specifies the quantity to trade, the side of the trade and the percentage volume targets for each time slot. Internally, using the `eugene.agent.vwap.impl.VwapStatus` class hidden behind a private package, the duration of the simulation is divided into equal time slots (encapsulated by `eugene.agent.vwap.impl.VwapBucket` class) and the `Vwap Agent` starts trading. As described in Section 2.4.2, the `Vwap Agent` submits a limit order at the appropriate price (depending on the implementation, see Section 1.2.2) and the order is monitored for 3 seconds. If after 3 seconds, the best price has moved and the current order is not filled, the current order is cancelled and a new limit order is submitted. When the execution is in the last 2 seconds of the current time slot and the current limit order is not filled, the current limit order is cancelled, and a market order is submitted for the remaining quantity.

The `Vwap Agent` uses the `Client API` (see Section 4.8) to rebuild the `Order Book` (see Section 4.3) in order to track the current prices, as well as the `OrderReferenceListener` functionality in order to track the status of the current order.

4.11 TESTING

The requirements *No1* and *No2* set out to ensure that the implementation is thoroughly tested. Extensive testing gives confidence in the implementation and allows for safe refactoring of the code, when the design needs to be revisited. Additionally, especially unit-tests, serve as documentation for the code, i.e. in order to explore the usage of a particular class it is very useful to inspect the unit-tests. Moreover, integration tests serve a verification purpose, ensuring that the use cases (see Appendix A) are correctly implemented.

4.11.1 Unit Testing

In order to fulfil requirement *No1*, the system is extensively unit-tested. Modular architecture and programming to an interface allow for ease of testing (see requirement *No3* and Section 4.1). Any dependencies can be injected through *mocking* [5, 6], and therefore tests always test the smallest possible unit of code.

Tests are written using *TestNG* [17] testing framework, *hamcrest* [3] for writing powerful and readable assertions and *Mockito* [5] for mocking. Whenever the code needs to use `JADE` classes, we use *PowerMock* [6] to mock them. That is because `JADE` classes do not usually implement interfaces and often have methods marked `final`; mocking frameworks like *Mockito* usually use the *Dynamic Proxy API* [2] and thus cannot deal with methods and classes marked as `final`. *Powermock* uses a custom classloader and bytecode manipulation in order to enable mocking of such classes. Because it needs to recompile the classes that are mocked, it is much slower; therefore its use is limited only to cases when it is absolutely necessary. Additionally, apart from mocking `JADE` classes, the code is also tested inside a `JADE` node, using `jade-unit` (See Section 4.2).

4.11.2 Integration Testing (integration/)

Integration tests are located in a separate module, as they take more time to run than unit-tests, and therefore do not run after every local change (they do, however, run every time a change is pushed to the main repository, see Section 4.11.3). Integration tests fulfil requirement *No2* and serve a verification purpose to ensure that use cases are correctly implemented (see Appendix A). Integration tests bootstrap the *JADE* node (using `jade-unit`, see Section 4.2), use the *Simulation Agent* (Section 4.5) to bootstrap the test agents and the *Market Agent* (Section 4.7), and use the *Client API* in order to send messages. The tests then assert whether appropriate responses are sent back. This way all parts of the architecture are tested and the correctness of the overall behaviour is verified.

4.11.3 Continuous Integration and Test Coverage

Every push to the main repository triggers a build in *Jenkins* [4] (see Appendix E). The code is clean built, all the unit and integration tests are run and instrumented by *Cobertura* [1] to calculate test coverage. The requirement *No1* is fulfilled, as the branch test coverage is `76%`. Moreover, the true branch test coverage is likely much closer to `90%`; that is because *Cobertura* cannot instrument external jars, therefore integration tests (see Section 4.11.2) do not contribute to the final test coverage. See Appendix F for Project Coverage report.

5 EXPERIMENTS AND RESULTS

5.1 EXPERIMENTAL PROCEDURE

Before running the experiments, we have performed several test runs with `Noise Trader Agents` only and a set of randomly generated initial orders (requirement *F10*), that were approximating a log-normal price/volume shape of a stable limit order book, with most of the volume near the top of the book. This approach was introduced in order to control the initial period of the simulation, when the limit order book needs to accumulate enough liquidity to become stable. However, we discovered that those initial orders acted as resistance points, as they were not owned by any of the `Trader Agents`, and thus could not be canceled (Section 2.2.4). Therefore, we reverted to only using the `default price` (Section 4.5) as the starting price and let the price dynamics arise from the interaction of all the agents.

All experimental setups consisted of several `Noise Trader Agents` and one appropriate `Vwap Agent`, and were run for a fixed time period. After each run of an experiment, the `executions.log` (see Appendix D) file was parsed using a `Mathematica` [42] script in order to calculate the $VWAP_{Market}$ and $VWAP_{Algorithm}$, and from those derive the *percentage difference from the target VWAP* (Section 2.4.1). Afterwards, the results of all runs were analysed using another `Mathematica` script, to determine the distribution of results and compare the results of two setups using an appropriate `Student T-Test`. All results are quoted with $\alpha = .05$ confidence level.

5.2 NULL HYPOTHESIS 1

In order to test the `Null Hypothesis 1` (see Section 1.2.2), we have setup an experiment with `Noise Trader Agents` (see Section 4.9) and `Vwap Agents` (see Section 4.10). The data was gathered in two experimental setups, the first with a correct VWAP implementation (A), and the second with an implementation that contains a logical polarity error (B).

To control the experimental variables, we kept the remaining parameters constant throughout the two experimental setups, i.e. the simulation length, VWAP volume,

VWAP targets, the number of `Noise Trader Agents`, tick size and default price. Table 5.1 summarises the parameters.

Both setups were run 18 times and for each run the result was the *percentage difference from the target VWAP* (see Section 2.4.1). We performed an appropriate statistical test to determine whether the samples from the two setups are statistically different, in order to reject or confirm the `Null Hypothesis 1`.

Parameter	**Value**
Simulation length	6 minutes
VWAP Volume	8000
VWAP Targets (12)	4%, 12%, 7%, 9%, 8%, 8%, 8%, 8%, 8%, 8%, 8%, 12%
Number of `Noise Traders`	20
Tick size	0.001
Default Price	100.000

Table 5.1: Summary of parameters for `Null Hypothesis 1`.

5.2.1 Experimental Results

Table 5.2 summarises the results. The `Kolmogorov-Smirnov` test for normal distribution concluded that the observed errors from the target VWAP for setup A and B were both normally distributed (setup A: $p = .08$; setup B: $p = .59$), therefore a parametric `T-Test` was conducted. The results showed that the difference between the setups was significant, $t(17) = 2.44, p = .02$, therefore the `Null Hypothesis 1` was rejected.

Hence, the analysis showed that given a correct implementation of a VWAP Algorithm, if a logical polarity error is introduced that causes the algorithm to always check the wrong price when sending a limit order, there was a statistically significant effect on the distribution of percentage errors from the target VWAP.

$\Delta\%_{VWAP_M,VWAP_A}$	$\Delta\%_{VWAP_M,VWAP_B}$
0.000909961	−0.00122571
0.00438393	−0.00576884
−0.000867167	0.00213636
0.00114055	−0.00596133
−0.00104553	0.00230666
0.000489347	−0.000478133
−0.000615812	0.00162665
0.000931747	0.000996034
0.000170407	−0.00230045
−0.00168723	−0.00172148
−0.00145683	−0.00240636
−0.00187975	−0.00108177
0.00276645	−0.00294432
−0.00152978	−0.000635886
−0.00104757	−0.00714921
0.0012759	0.0000479285
0.0068051	−0.00267876
0.0000606508	−0.000568892

Table 5.2: Summary of results for `Null Hypothesis 1`.

5.3 NULL HYPOTHESIS 2

In order to test the `Null Hypothesis 2` (see Section 1.2.3), we setup a similar experiment to Section 5.2. The data was gathered in two experimental setups, both involving a correctly implemented VWAP algorithm. In the first setup, however, the VWAP algorithm trades according to 10 equal targets (A), and the second according to 40 equal targets (B).

To control the experimental variables, we kept the remaining parameters constant throughout the two experimental setups, i.e. the simulation length, VWAP volume, the number of `Noise Traders`, tick size and default price. Table 5.3 summarises the parameters. We gathered the same experimental data as in Section 1.2.2.

Parameter	**Value**
Simulation length	6 minutes
VWAP Volume	20000
Number of `Noise Traders`	20
Tick size	0.001
Default Price	100.000

Table 5.3: Summary of parameters for `Null Hypothesis 2`.

5.3.1 Experimental Results

Table 5.4 summarises the results. The `Kolmogorov-Smirnov` test for normal distribution concluded that the observed errors from the target VWAP for setup A and B were both normally distributed (setup A: $p = .23$; setup B: $p = .12$), therefore a parametric `T-Test` test was conducted. The results showed that the difference between the setups was not significant, $t(17) = 1.04, p = .31$, therefore the `Null Hypothesis 2` was not rejected.

Hence, the analysis showed that given 2 correct implementations of a VWAP Algorithm, one that divides the trading duration into 10 equal time intervals and the second that divides the trading duration into 40 equal time intervals, there was no effect on the distribution of percentage errors from the target VWAP.

$\Delta\%_{VWAP_M,VWAP_A}$	$\Delta\%_{VWAP_M,VWAP_B}$
0.00277385	0.000791962
0.00213915	0.00105847
0.00140453	0.00090313
−0.000168954	0.00142062
0.000982506	0.00113082
0.00196194	0.00211302
0.00450798	0.00199672
0.00287382	−0.000372864
0.00161029	0.000886285
0.00103771	0.00122396
0.00206086	0.000306701
−0.00253224	0.00132408
−0.000945702	−0.000667823
−0.00173609	−0.0000434588
0.000099006	0.00186177
0.00131022	0.0010309
0.0083891	0.000822814
0.00301218	0.00170759

Table 5.4: Summary of results for `Null Hypothesis 2`.

5.4 SUMMARY OF RESULTS

The experiment in Section 5.2 has rejected the *Null Hypothesis 1* (Section 1.2.2). Therefore, by using the statistical analysis, given that we know there is an error in the implementation, we can detect a significant difference in the behaviour in setup B when compared to the benchmark implementation in setup A. This is a positive result, as it confirms that the initial verification of the simulator and its goals were achieved.

On the other hand, the experiment in Section 5.3 failed to reject the *Null Hypothesis 2* (Section 1.2.3). This means that the difference in behaviour between setup A and setup B was not statistically significant. We have performed a number experiments for this hypothesis, varying the parameters like the length of the simulation or the number of targets. Increasing the length of the simulations seemed to improve the results, but not enough to account for a statistically significant difference. We think the explanation for this result is that the effect is more subtle than this particular experimental setup is able to detect. This might be due to the fact that the simulations are still relatively short, therefore the difference in the frequency of trading between the two setups does not have a detectable effect; however, if the simulations were run for longer, the effect could be more significant. The only trend that we have been able to observe is that as the algorithm trades more often, the errors have much less variance. Hence, we think that using a different metric for this particular hypothesis would be beneficial.

6 CONCLUSIONS AND FURTHER WORK

6.1 SUMMARY AND EVALUATION

In this thesis we focused on testing a VWAP Trading Algorithm using Agent-Based Simulation to demonstrate that the results obtained from the simulation can be used to automatically detect simple programming errors using statistical analysis.

In particular, we demonstrated the efficacy of this approach, by comparing a correctly implemented VWAP algorithm to a VWAP algorithm with a logical polarity error introduced into the implementation. We hypothesised that by performing statistical analysis on the distribution of errors between VWAPs achieved by both algorithms and those of the overall market, it would be possible to detect the error with high level of confidence. Indeed, the analysis showed that the simple measurements performed were enough to detect a statistically significant difference between the two implementations.

In the second experiment, we tried to detect a difference in error distribution between two correctly implemented VWAP algorithms, where one traded more aggressively than the other. The same analysis showed no statistically significant effect and we believe that that the effect was too subtle to be detected by the particular experimental setup.

Nevertheless, the project has clearly achieved its goals and is a proof of concept that the technique could be developed further in order to be used to complement the existing testing strategies. All requirements were satisfied and use cases correctly implemented (as verified by integration tests). The simulator was excellently engineered, with great focus on adhering to principles of good design and thorough testing.

6.2 FUTURE WORK

The technique presented was positively evaluated and we think it could be developed further into a full-featured tool for testing and verification of Trading Algorithms.

First extension would be to see whether the experiments could be fully reproducible, by saving the random seeds used by `Noise Traders`. This would allow for a

regression testing approach that could continuously validate the implementation of a Trading Algorithm, as it goes through stages of development, to ensure consistent quality.

Secondly, in this work we have approached the question of whether, given that we know there is an error in the implementation, we can detect a significant difference in the behaviour, when compared to the benchmark implementation. We did not, however, try to determine the type of the programming error that we are dealing with, or try to differentiate between different types of programming errors. In order to take this work further, it would be advantageous start building a classification of errors, and their possible manifestations, in order to come up with a procedure that can, given some set of measurements, determine whether a previously unseen algorithm has a particular type of error, with a confidence measure attached to it. Given this classification, we would proceed to evaluate its accuracy and breadth of application.

Thirdly, in this work we have specifically focused on a VWAP algorithm, therefore it would be interesting to see how to generalise this work to other trading algorithms, most importantly profit algorithms (Section 2.3). Our second hypothesis aimed to approach this question, however the current setup was not powerful enough to discover a difference.

Next, the implementation of the VWAP algorithm chosen for evaluation was deliberately simple. An investigation into implementing more sophisticated trading algorithms, and similarly simulating other traders, would provide for a more realistic setup.

Next, we think that in a production setup, the experiments should be run on a cluster of machines in order to complete in a reasonable time to provide instant feedback to developers. The current method of manually running the experiments and collecting results in order to perform the statistical analysis is not automated enough.

Lastly, all the simulations that were run exhibit relatively unstable behaviour at the start of the simulation. We set out to mitigate this issue through sending initial orders (Section 4.5), but we had to abandon this approach, see Section 5.1. We believe that this issue could be addressed by explicitly simulating opening *call auctions* [16] to build up liquidity and establish the opening price. Similarly, as the previous paragraph suggest, implementing more types of agents to complement the `Noise Trader Agents`, should also help stabilise the order book.

6.3 FINAL THOUGHTS

To summarise, we believe that the technique presented in this thesis has a potential to be developed into a useful tool for testing and verification of Trading Algorithms. We managed to demonstrate that a simple experimental setup can already provide useful feedback as to the difference in behaviour between a correctly and incorrectly implemented algorithm. Unfortunately, the extension proposed in Section 1.2 was not achieved, but still provided with useful insight as to how to improve the simulator. Overall, we believe that the project clearly fulfilled its goals and thus provides a useful methodological advancement over the more established ways of testing Trading Algorithms.

A USE CASES

UC1: Start the simulation

Primary Actors	Simulation Agent
Secondary Actors	Market Agent, Trader Agents
Description	The Simulation Agent starts the simulation.
Pre-conditions	
Flow of Events	1. The Simulation Agent starts the Market Agent. 2. The Market Agent starts and sends a Started message back to the Simulation Agent. 3. The Simulation Agent receives the Started message. 4. The Simulation Agent starts the Trader Agents and passes the identity of the Market Agent. 5. The Trader Agents start and log on with the Market Agent. 6. The Trader Agents send a LogonComplete message back to the Simulation Agent. 7. The Simulation Agent receives the LogonComplete messages. 8. After receiving all the Logon Complete messages, the Simulation Agent sends initial limit orders to the Market Agent. 9. After receiving acknowledgements for all the initial limit orders, the Simulation Agent sends Start messages to the Trader Agents. 10. The Trader Agents send Started messages back to the Simulation Agent. 11. The Simulation Agent receives Started messages. 12. After receiving all Started messages, the Simulation Agent sleeps until the end of the simulation.
Post-conditions	The Simulation Agent sleeps.
Alternative Flows	None.

UC2: Logon

Primary Actors	Trader Agent
Secondary Actors	Market Agent
Description	The Trader Agent sends a Logon message to the Market Agent.
Pre-conditions	
Flow of Events	1. The Trader Agent sends a Logon message including the Trader Agent ID to the Market Agent 2. The Market Agent receives the message and stores the Trader Agent ID. 3. The Market Agent sends an acknowledgement message back to the Trader Agent. 4. The Trader Agent receives the acknowledgement message.
Post-conditions	The Trader Agent ID has been saved by the Market Agent.
Alternative Flows	None.

UC3: Send Limit Order

Primary Actors	Trader Agent
Secondary Actors	Market Agent, Execution Engine
Description	The Trader Agent sends a NewOrderSingle message indicating a limit order to the Market Agent.
Pre-conditions	The Trader Agent has logged on with the Market Agent and received the Start message from the Simulation Agent.
Flow of Events	1. The Trader Agent sends a NewOrderSingle message message with a unique ClOrdID to the Market Agent. 2. The Market Agent receives the message and forwards the limit order to the Execution Engine. 3. The Execution Engine processes the NewOrderSingle message and acknowledges a new limit order with the Market Agent. 4. The Market Agent sends an ExecutionReport acknowledgement message back to the Trader Agent. 5. The Trader Agent receives the ExecutionReport message.
Post-conditions	The Execution Engine processed a new limit order.
Alternative Flows	None

UC4: Send Market Order

Primary Actors	Trader Agent
Secondary Actors	Market Agent, Execution Engine
Description	The Trader Agent sends a NewOrderSingle message indicating a market order to the Market Agent.
Pre-conditions	The Trader Agent has logged on with the Market Agent and received the Start message from the Simulation Agent.
Flow of Events	1. The Trader Agent sends a NewOrderSingle message message with a unique ClOrdID to the Market Agent. 2. The Market Agent receives the message and forwards the market order to the Execution Engine. 3. The Execution Engine processes the NewOrderSingle message and acknowledges a new market order with the Market Agent. 4. The Market Agent sends an ExecutionReport acknowledgement message back to the Trader Agent. 5. The Trader Agent receives the ExecutionReport message.
Post-conditions	The Execution Engine processed a new market order.
Alternative Flows	The OrderBook is empty on the side of the NewOrderSingle message, therefore the Execution Engine rejects the new market order and the Market Agent sends an ExecutionReport rejection message back to the Trader Agent.

UC5: Cancel Order

Primary Actors	Trader Agent
Secondary Actors	Market Agent, Execution Engine
Description	The Trader Agent sends an OrderCancelRequest message with the ClOrdID of the order to cancel to the Market Agent.
Pre-conditions	The Trader Agent has logged on with the Market Agent and received the Start message from the Simulation Agent.
Flow of Events	1. The Trader Agent sends an OrderCancelRequest message to the Market Agent. 2. The Market Agent receives the message, checks if the order exists and forwards the OrderCancelRequest to the Execution Engine. 3. The Execution Engine processes the OrderCancelRequest message and acknowledges the cancellation with the Market Agent. 4. The Market Agent sends an ExecutionReport cancellation message with the quantity that was cancelled back to the Trader Agent. 5. The Trader Agent receives the ExecutionReport message.
Post-conditions	The Execution Engine processed the cancellation.
Alternative Flows	The ClOrdID referred to in the OrderCancelRequest does not exist, therefore the Market Agent sends an OrderCancelReject message back to the Trader Agent.

UC6: Process a NewOrderSingle message indicating a limit order

Primary Actors	Execution Engine
Secondary Actors	Trader Agents, Market Agent
Description	The Execution Engine processes a NewOrderSingle message indicating a limit order.
Pre-conditions	The Execution Engine accepted a NewOrderSingle message indicating a limit order.
Flow of Events	1. The Execution Engine checks if there are matching limit orders on the opposite side of the Order Book. 2. The Execution Engine executes all matching limit orders and sends ExecutionReport messages to the counterparties and OrderExecuted messages to all Trader Agents. 3. If the limit order has not been filled, the Execution Engine puts the limit order into the Order Book and sends AddOrder messages to all Trader Agents. 4. The Execution Engine acknowledges the new limit order with the Market Agent.
Post-conditions	The matching limit orders have been executed, appropriate messages have been sent and any remaining quantity has been put into the Order Book.
Alternative Flows	None.

UC7: Process a NewOrderSingle message indicating a market order

Primary Actors	Execution Engine
Secondary Actors	Trader Agents, Market Agent
Description	The Execution Engine processes a NewOrderSingle message indicating a market order.
Pre-conditions	The Execution Engine accepted a NewOrderSingle message indicating a market order.
Flow of Events	1. The Execution Engine checks if there are matching limit orders on the opposite side of the Order Book. 2. The Execution Engine executes all matching limit orders and sends ExecutionReport messages to the counterparties and OrderExecuted messages for the limit orders to all Trader Agents. 3. If the market order has not been filled, the Execution Engine sends an ExecutionReport cancellation message for the remaining quantity back to the Trader Agent. 4. The Execution Engine acknowledges the new market order with the Market Agent.
Post-conditions	The market order and matching limit orders have been executed, appropriate messages have been sent and any remaining quantity has been canceled.
Alternative Flows	The opposite side of the Order Book is empty, therefore the Execution Engine rejects the new market order.

UC8: Process an OrderCancelRequest message

Primary Actors	Execution Engine
Secondary Actors	Trader Agents, Market Agent
Description	The Execution Engine processes an OrderCancelRequest message.
Pre-conditions	The Execution Engine accepted an OrderCancelRequest message.
Flow of Events	1. The Execution Engine removes the limit order from the Order Book. 2. The Execution Engine sends a DeleteOrder message to all Trader Agents.
Post-conditions	The limit order has been canceled.
Alternative Flows	None.

UC9: Stop the simulation

Primary Actors	Simulation Agent
Secondary Actors	Market Agent, Trader Agents
Description	The Simulation Agent stops the simulation.
Pre-conditions	The simulation is running and the time limit for the simulation has been reached.
Flow of Events	1. The Simulation Agent wakes up. 2. The Simulation Agent sends a Stop message to the Market Agent and the Trader Agents. 3. The Market Agent and the Trader Agents stop and send Stopped messages back to the Simulation Agent. 4. The Simulation Agent receives the Stopped messages. 5. After receiving all the Stopped messages, the Simulation Agent stops.
Post-conditions	The simulation is stopped.
Alternative Flows	None.

B SIMULATION ONTOLOGY MESSAGES

Message	Description	Fields [Type]
LogonComplete	Sent by `Trader Agents` to indicate that the `Logon` procedure has been completed and the `Trader Agent` is ready to receive messages from the `Market Agent`.	
Start	Sent by the `Simulation Agent` to a `Trader Agent` in order to activate the `Trader Agent`.	`startTime[Date]`, `stopTime[Date]`
Started	Sent by either the `Market Agent` or the `Trader Agents` in reply to a `Start` message.	
Stop	Sent by the `Simulation Agent` to indicate that either: • `Trader Agent`'s trading time has ended and the `Trader Agent` should cease sending messages to the `Market Agent` and terminate. • The simulation has ended and therefore the recipient should terminate.	
Stopped	Sent by `Trader Agents` in reply to `Stop` message.	

Table B.1: Simulation Ontology messages.

C MARKET ONTOLOGY MESSAGES

C.1 FIELDS (EUGENE.MARKET.ONTOLOGY.FIELD)

C.1.1 Single valued fields

Tag	Field	Description	Type
6	AvgPx	Average price of all fills of an order.	BigDecimal
11	ClOrdID	Unique identifier for an order as assigned by the Agent originating the trade. Uniqueness must be guaranteed within a single trading day.	String
14	CumQty	Total quantity (e.g. number of shares) filled.	Long
31	LastPx	Price of execution.	BigDecimal
32	LastQty	Quantity executed.	Long
151	LeavesQty	Quantity open for further execution. If the `OrdStatus` is `OrdStatus#CANCELLED` or `OrdStatus#REJECTED` (in which case the order is no longer active) then `LeavesQty` could be 0, otherwise LeavesQty = OrderQty − CumQty.	Long
37	OrderID	Unique identifier for Order as assigned by the Market. Uniqueness must be guaranteed within a single trading day.	String
38	OrderQty	Quantity ordered.	Long
44	Price	Price per unit of quantity.	BigDecimal
55	Symbol	Ticker symbol. Common, "human understood" representation of the security.	String
17	TradeID	The unique ID assigned to the execution by the Market Agent.	String

Table C.1: `Market Ontology` single valued fields.

C.1.2 Enum fields

Tag	Field	Description	Values
150	ExecType	Describes the specific `ExecutionReport`'s status.	NEW, CANCELED, REJECTED, TRADE
39	OrdStatus	Identifies current status of order.	NEW, PARTIALLY_FILLED, FILLED, CANCELLED, REJECTED
40	OrdType	Order type.	MARKET, LIMIT
1409	SessionStatus	Session status at time of `Logon`. Field is intended to be used when the `Logon` is sent as an acknowledgement from acceptor of the `Logon` message.	SESSION_ACTIVE
54	Side	Side of order.	BUY, SELL

Table C.2: `Market Ontology` enum fields.

C.2 MESSAGES

C.2.1 Order management messages (eugene.market.ontology.message)

Type	Message	Description	Fields [optional]
8	ExecutionReport	The `ExecutionReport` message is used to: • Confirm the receipt of an order. • Confirm changes to an existing order (i.e. accept cancel request). • Relay fill information on working orders. • Reject orders.	`ExecType`, `Symbol`, `Side`, `LeavesQty`, `CumQty`, `AvgPx[Yes]`, `OrdStatus`, `OrderID`, `ClOrdID`, `LastPx[Yes]`, `LastQty[Yes]`
A	Logon	First message sent by the Agent to the Market in order to register with the Market. • As a result, the Agent will receive data feed updates. • If logon was successful, the Market will send back the same `Logon` message with `Logon#SessionStatus` equal to `SessionStatus#SESSION_ACTIVE`, otherwise this field will be `null`.	`Symbol`, `SessionStatus[Yes]`
D	NewOrderSingle	Used by agents wishing to submit securities orders to the Market for execution.	`ClOrdID`, `Symbol`, `Side`, `OrderQty`, `OrdType`, `Price[Yes]`
9	OrderCancelReject	Issued by the Market upon receipt of a `OrderCancelRequest` message which cannot be honored.	`OrderID`, `ClOrdID`, `OrdStatus`
F	OrderCancelRequest	Requests the cancellation of all of the remaining quantity of an existing order.	`ClOrdID`, `Symbol`, `Side`, `OrderQty`

Table C.3: Order management messages.

C.2.2 Market data messages (eugene.market.ontology.message.data)

Type	Message	Description	Fields [optional]
BP0x21	AddOrder	Represents a newly accepted visible order on the order book.	`OrderID`, `Symbol`, `Side`, `OrderQty`, `Price`
BP0x29	DeleteOrder	Sent whenever an open order is completely cancelled. The `OrderID` refers to the `OrderID` of the original `AddOrder` message.	`OrderID`
BP0x23	OrderExecuted	Sent when a visible order on the order book is executed in whole or in part at a price. The `OrderID` refers to the `OrderID` of the original `AddOrder` message.	`OrderID`, `LastQty`, `LeavesQty`, `LastPx`, `TradeID`

Table C.4: Market data messages.

D LOGGING

This section describes the format of the log files. Most fields refer to the message fields, as defined in Appendix C. Otherwise, `timestamp` is a `UNIX timestamp` obtained by calling `System.currentTimeMillis()` when the event occured, and `TraderID` is the name of the `JADE` Agent that sent the original order. The logs are in *comma-separated value* format.

D.1 EVENTS

D.1.1 NEWORDER

Indicates that an order was accepted by the `Market Agent`, but it did not yet go through the matching process. If the `OrdType` is `MARKET`, the `Price` is `0`.

Format

```
timestamp,TraderID,OrderID,ClOrdID,OrdType,Side,OrderQty,Price
```

D.1.2 REJECTORDER

Indicates that a market order was rejected, because there was no liquidity on the opposite side of the limit order book.

Format

```
timestamp,TraderID,OrderID,ClOrdID,OrdType,Side,OrderQty,Price
```

D.1.3 ADDORDER

Indicates that a limit order entered the limit order book. The `OrderQty` is not necessarily equal to the `OrderQty` in the original `NEWORDER` event; the order could have been partially executed before entering the limit order book.

Format

```
timestamp,OrderID,Side,OrderQty,Price
```

D.1.4 EXECUTION

Indicates that a pair of orders have been executed. The orders must have already been referred to in a previous `NEWORDER` event, however fields are repeated for ease of data extraction. `I` refers to fields that concern the incoming order, i.e. either market or limit order that has not yet entered the limit order book, and `L` refers to the resting limit order, that has already entered the limit ordered book.

Format

```
timestamp,ITraderID,IOrderID,ISide,LTraderID,LOrderID,LSide,Quantity,Price
```

D.2 marketdata.log

Contains a log of all events; the events are disambiguated by putting `[EVENTNAME],` in front of the formatted log entry,
e.g. `"[ADDORDER],timestamp,OrderID,Side,OrderQty,Price"`.

D.3 executions.log

Contains a filtered log of only `EXECUTION` events.

D.4 rejections.log

Contains a filtered log of only `REJECTORDER` events.

E USER AND SYSTEM MANUAL

E.1 REQUIREMENTS

- JDK 1.6
- Maven3
- Git

E.2 BUILD

To compile and run all the tests, in the project root directory run:

```
$ mvn clean install
```

E.3 RUN

The experiments are located in the `experiment/` module: `vwap-error` and `vwap-no-error`.

```
$ mvn pax:run
```

E.4 DISTRIBUTION

In order to run a `*.zip` file with a standalone experiment that can be run without `Maven3`, execute:

```
$ mvn clean install assembly:single
```

This will create a `*.zip` file in the `target/` subdirectory, which can be distributed. In order to run the experiment, unpack the `*.zip` file and use the bash script located in the `bin/` directory:

```
$ sh start.sh
```

E.5 CONTINUOUS INTEGRATION

http://abfm.cs.ucl.ac.uk/jenkins/job/eugene/

E.6 WIKI AND ISSUE TRACKER

http://abfm.cs.ucl.ac.uk/redmine/projects/eugene

E.7 VERSION CONTROL

http://github.com/jkozlowski/eugene

E.8 LICENCE

F PROJECT COVERAGE

Project Coverage summary

Name	Classes	Conditionals	Files	Lines	Methods	Packages
Cobertura Coverage Report	100% 156/156	76% 570/746	100% 110/110	83% 2399/2902	92% 750/811	100% 25/25

Coverage Breakdown by Package

Name	Classes	Conditionals	Files	Lines	Methods
eugene.market.agent	100% 1/1	100% 2/2	100% 1/1	100% 19/19	100% 2/2
eugene.market.agent.impl	100% 6/6	77% 27/35	100% 3/3	97% 115/119	95% 20/21
eugene.market.agent.impl.behaviours	100% 5/5	19% 3/16	100% 3/3	41% 95/230	62% 18/29
eugene.market.agent.impl.execution	100% 6/6	79% 44/56	100% 4/4	95% 118/124	97% 31/32
eugene.market.agent.impl.execution.data	100% 7/7	N/A	100% 2/2	100% 67/67	100% 28/28
eugene.market.book	100% 3/3	82% 64/78	100% 3/3	95% 108/114	100% 29/29
eugene.market.book.impl	100% 2/2	94% 15/16	100% 2/2	100% 67/67	100% 24/24
eugene.market.client	100% 10/10	100% 6/6	100% 10/10	86% 63/73	78% 36/46
eugene.market.client.impl	100% 32/32	90% 130/144	100% 14/14	93% 464/498	94% 140/149
eugene.market.esma	100% 1/1	100% 2/2	100% 1/1	100% 19/19	100% 2/2
eugene.market.esma.impl	100% 6/6	77% 27/35	100% 3/3	97% 115/119	95% 20/21
eugene.market.esma.impl.behaviours	100% 3/3	19% 3/16	100% 3/3	38% 80/208	52% 12/23
eugene.market.esma.impl.execution	100% 6/6	79% 44/56	100% 4/4	95% 118/124	97% 31/32
eugene.market.esma.impl.execution.data	100% 7/7	N/A	100% 2/2	100% 67/67	100% 28/28
eugene.market.ontology	100% 3/3	89% 32/36	100% 3/3	95% 80/84	100% 20/20
eugene.market.ontology.field	100% 16/16	N/A	100% 16/16	97% 118/122	96% 86/90
eugene.market.ontology.field.enums	100% 5/5	100% 32/32	100% 5/5	100% 87/87	100% 33/33
eugene.market.ontology.internal	100% 2/2	48% 47/98	100% 2/2	53% 100/188	78% 14/18
eugene.market.ontology.message	100% 5/5	N/A	100% 5/5	94% 83/88	92% 57/62
eugene.market.ontology.message.data	100% 3/3	N/A	100% 3/3	92% 36/39	89% 25/28
eugene.simulation.agent	100% 3/3	83% 5/6	100% 2/2	100% 40/40	100% 7/7
eugene.simulation.agent.impl	100% 13/13	74% 62/84	100% 8/8	79% 236/300	100% 48/48
eugene.simulation.ontology	100% 6/6	83% 15/18	100% 6/6	96% 44/46	100% 16/16
eugene.utils	100% 1/1	100% 2/2	100% 1/1	100% 9/9	100% 2/2
eugene.utils.annotation	N/A	N/A	N/A	N/A	N/A
eugene.utils.behaviour	100% 4/4	100% 8/8	100% 4/4	100% 51/51	100% 21/21

Figure F.1: Cobertura Project Coverage summary

BIBLIOGRAPHY

[1] Cobertura. URL http://cobertura.sourceforge.net/.

[2] Dynamic Proxy API. URL http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/Proxy.html.

[3] Hamcrest - library of matchers for building test expressions. URL http://code.google.com/p/hamcrest/.

[4] Jenkins - an extendable open source continuous integration server. URL http://jenkins-ci.org/.

[5] mockito - simpler & better mocking. URL http://code.google.com/p/mockito/.

[6] PowerMock is a Java framework that allows you to unit test code normally regarded as untestable. URL http://code.google.com/p/powermock/.

[7] Agile Alliance. Agile Manifesto. URL http://agilemanifesto.org/iso/en/.

[8] Apache Software Foundation. Commons Math: The Apache Commons Mathematics Library. URL http://commons.apache.org/math/.

[9] BATS Trading Limited. BATS Chi-X Europe Multicast PITCH Specification, Version 4.8, March 2012. URL http://fixprotocol.org/.

[10] K. Beck. Aim, fire. *Software, IEEE*, 18(5):87–89, September/October 2001. ISSN 0740-7459. doi: 10.1109/52.951502.

[11] Katalin Boer-Sorban. *Agent-Based Simulation Agent-Based Simulation of Financial Markets*. PhD thesis.

[12] Cisco Systems, Inc. Cloud Computing for Financial Markets. Technical report, 2011. URL http://www.cisco.com/web/strategy/docs/finance/cloud_wp_c112D518876.pdf.

[13] Peter K Clark. A Subordinated Stochastic Process Model with Finite Variance for Speculative Prices. *Econometrica*, 41(1):135–55, January 1973. URL http://ideas.repec.org/a/ecm/emetrp/v41y1973i1p135-55.html.

[14] Coda Hale, Yammer Inc. Metrics - Capturing JVM and application-level metrics. So you know what's going on. URL http://metrics.codahale.com/.

[15] Richard Coggins, Marcus Lim, and Kevin Lo. Algorithmic Trade Execution and Market Impact. *Exchange Organizational Behavior Teaching Journal*, pages 518–547, 2006.

[16] Carole Comerton-Forde and James Rydge. A Review of Stock Market Microstructure. *SSRN eLibrary*, 2004. doi: 10.2139/ssrn.710801.

[17] Cic Beust. TestNG. URL http://testng.org/.

[18] Nicholas Economides and Robert A Schwartz. Electronic Call Market Trading. *Journal of Portfolio Management*, 21(3):10–18, 1995. URL http://papers.ssrn.com/sol3/papers.cfm?abstract_id=6525.

[19] J Doyne Farmer and Duncan Foley. The economy needs agent-based modelling. *Nature*, 460(7256):685–6, August 2009. ISSN 1476-4687. doi: 10.1038/460685a.

[20] FIX Protocol Limited (FPL). Financial Information eXchange Protocol. URL http://fixprotocol.org/.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[22] Daniel Gilles. *Asynchronous Simulations of a Limit Order Book*. PhD thesis, 2006.

[23] Google Inc. guava-libraries. URL http://code.google.com/p/guava-libraries/.

[24] JADE Governing Board organization. Java Agent DEvelopment Framework. URL http://jade.tilab.com/.

[25] Ashutosh Jha, Jitesh Mohan, Amrita Vishwa Vidyapeetham, and Tamil Nadu. A survey of Call Market (Discrete) Agent Based Artificial Stock Markets. *International Journal on Computer Science and Engineering*, 02(09):3025–3032, 2010.

[26] Sham M. Kakade, Michael Kearns, Yishay Mansour, and Luis E. Ortiz. Competitive algorithms for VWAP and limit order trading. In *Proceedings of the 5th ACM conference on Electronic commerce*, EC '04, pages 189–198, New York, NY, USA, 2004. ACM. ISBN 1-58113-771-0. doi: 10.1145/988772.988801. URL http://doi.acm.org/10.1145/988772.988801.

[27] Andrei A. Kirilenko, Albert S. Kyle, Mehrdad Samadi, and Tugkan Tuzun. The Flash Crash: The Impact of High Frequency Trading on an Electronic Market. *Traders*, January 2011.

[28] Marc Lenglet. The 'Algo revolution': rising machines and the coding of practices.

[29] Fabrizio Lillo and J. Doyne Farmer. The Long Memory of the Efficient Market. *Studies in Nonlinear Dynamics & Econometrics*, 8(3):1, 2004.

[30] LOGBack. LOGBack - The Generic, Reliable Fast & Flexible Logging Framework. URL http://logback.qos.ch/.

[31] MacKenzie, D. How to Make Money in Microseconds. 22:16–18, 2011.

[32] Marco Raberto and Silvano Cincotti and Sergio M. Focardi and Michele Marchesi. Agent-based simulation of a financial market. *Physica A*, (299):319–327.

[33] New York Stock Exchange LLC. Hearing Board Decision 09-NYSE-24, November 2009. URL http://www.nyse.com/pdfs/09-NYSE-24.pdf.

[34] Oracle Corporation. JConsole. URL http://docs.oracle.com/javase/6/docs/technotes/guides/management/jconsole.html.

[35] OSGi Alliance. OSGi™ - The Dynamic Module System for Java™. URL http://www.osgi.org/Main/HomePage.

[36] quickfixj.org. QuickFIX/J - 100% Java Open Source FIX Engine. URL http://quickfixj.org.

[37] Sanmay Das. *Dealers, Insiders and Bandits: Learning and its Effects on Market Outcomes*. PhD thesis, June 2006.

[38] Enrico Scalas, Rudolf Gorenflo, Hugh Luckock, Francesco Mainardi, Maurizio Mantelli, and Marco Raberto. Anomalous waiting times in high-frequency financial data. *Quantitative Finance*, 4:695, 2004. URL http://www.citebase.org/abstract?id=oai:arXiv.org:physics/0505210.

[39] Maria Siniaalto and Pekka Abrahamsson. A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage. *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 275–284, September 2007. doi: 10.1109/ESEM.2007.35.

[40] SLF4J. Simple Logging Facade for Java (SLF4J). URL http://www.slf4j.org/.

[41] Tesfatsion, Leigh. Agent-Based Computational Economics: A Constructive Approach to Economic Theory. In Leigh Tesfatsion and Kenneth L. Judd, editor, *Handbook of Computational Economics*, volume 2 of *Handbook of Computational Economics*, chapter 16, pages 831–880. Elsevier, 2006. URL http://ideas.repec.org/h/eee/hecchp/2-16.html.

[42] Wolfram. Wolfram Mathematica 8. URL http://www.wolfram.com/mathematica/.