Individual Project Report

UNIVERSITY COLLEGE LONDON

DEPARTMENT OF COMPUTER SCIENCE

Interference and Interaction between Trading Algorithms
Effects on Market Stability

Under the supervision of:  
Dr. Christopher D. Clack

Student:  
Andrei Airimitoaie

Disclaimer: This report is submitted as part requirement for the BSc Degree in Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text.

The report may be freely copied and distributed provided the source is explicitly acknowledged.
Abstract

Currently one of the most debated topics in the world of finance, algorithmic trading (AT) was responsible for 73% of all the traded volume in the United States in 2009[30]. The rate at which these types of trading techniques have been expanding, as well as the associated risks they have with regards to the stock market’s general stability[10] necessitates a thorough analysis of AT in a controlled, risk-free environment.

This project is focused on discovering market instabilities that arise from the interactions between trading algorithms. The main trading algorithm types we will evaluate are Time-Weighted-Average-Price(TWAP), as well as a category of profit algorithms, based on a strategy known by the name of Static Order Book Imbalance. The software framework we propose is an agent-based model developed in an object-oriented language.

We will make use of the agent-based-simulation technique to discover how TWAP and “Opportunistic” algorithms can induce changes in stock prices during a certain time frame. The statistical evidence illustrated in this report shows that the simple presence of a TWAP algorithm in a market with limited liquidity can have a significant impact on the distribution of the traded prices of a stock. Likewise, the total volume on the buy side of the limit order book increases, creating a liquidity imbalance in the market. Moreover, we find a correlation between the total amount of shares traded by the TWAP algorithm and the average stock price. Finally, we introduce a profit algorithm based on Static Order Imbalance to assess its impact on the limit order book properties. Our statistical data suggests that opportunistic trading algorithms will amplify the effect created by the TWAP agent, by increasing the average traded price and accentuating the liquidity imbalance on the limit order book.
# Contents

1 Introduction ........................................... 5
   1.1 Motivation ........................................ 5
   1.2 Aims and Goals ..................................... 6
   1.3 Potential Contributions ............................ 6
   1.4 Outline of the Report ............................. 7

2 Background ............................................. 8
   2.1 Stock Markets and Electronic Order Books .............. 8
      2.1.1 Overview .................................. 8
      2.1.2 Mechanisms of a Limit Order Book ............. 8
      2.1.3 Flow of Orders ............................. 9
   2.2 Algorithmic Trading ............................... 9
      2.2.1 General Overview ........................... 10
      2.2.2 Benchmark Algorithms ....................... 10
      2.2.3 Profit Algorithms ........................... 11
   2.3 Agent-Based modelling ............................. 12
      2.3.1 Overview .................................. 12
      2.3.2 Applications in Financial Markets .......... 12
      2.3.3 Advantages of Agent-Based Simulation .......... 13
   2.4 JADE - Java Agent DEvelopment Framework ............. 14
      2.4.1 What is JADE ? ............................ 14
      2.4.2 Agent Communication ......................... 14
      2.4.3 JADE Architecture .......................... 14
      2.4.4 Agent Behaviour ............................ 16
      2.4.5 Why JADE ? ............................... 16

3 Analysis and Requirements ............................. 17
   3.1 Overview of the system ............................ 17
      3.1.1 Components of the Stock Market ............... 18
   3.2 System Requirements .............................. 20
      3.2.1 Functional Requirements ...................... 20
      3.2.2 Non-Functional Requirements .................. 22

4 Design and Implementation ............................ 24
   4.1 Design Features ................................. 24
      4.1.1 Encapsulation .............................. 24
      4.1.2 Following an Agile approach ................. 24
   4.2 UML Class Diagram ............................... 24
   4.3 Market Side Implementation ....................... 26
      4.3.1 Limit Order Book ........................... 26
      4.3.2 Brokerage Service and Market Agent .......... 26
G Code Listing

G.1 Limit Order Book ................................................. 68
G.2 Order ................................................................. 71
G.3 Market Agent ....................................................... 75
G.4 Noise Agent ......................................................... 87
G.5 Twap Agent ........................................................ 97
G.6 Opportunistic Agent .............................................. 106
G.7 Update Behaviour - Common to Noise Agent, Opportunistic Agent and Twap Agent ......................................................... 112
G.8 Initiation Agent ..................................................... 115
G.9 Simulation Agent ................................................... 115
Chapter 1

Introduction

1.1 Motivation

In today’s financial market environment, one of the fundamental pieces in gaining an edge over competitors is to have a solid technological framework on which highly advanced trading strategies can be developed and used to reduce costs and minimize market impact. In many cases, however, these automated trading tactics have an additional goal: generate significant amounts of profit in relatively short time spans. From the trader’s point of view, the more sophisticated and faster algorithms ought to yield higher “rewards”. However, one must always take into consideration the fact that these achievements will heavily rely on the status of the stock market, as traders interact in a shared environment.

The sheer complexity of the stock market and the way it functions - as one of the most complicated distributed systems in the world - has yet to be fully understood. As automated algorithms generously contribute to the overall trading activity, we are obliged to look at how their interactions would impact the general market stability.

Recent history has demonstrated that our understanding of the financial market and its instabilities is limited. For instance, on May 6, 2010, the U.S stock market indices dropped by 5 percent in the course of approximately 30 minutes, followed by an equally fast increase that brought the index back to its original price. The event is commonly referred to as the “Flash Crash” - defining a short period of abnormal volatility - and its occurrence is believed to have uncovered structural and stability deficiencies of financial markets [16]. According to Kirilenko, “representatives of individual investors, asset management companies, and market intermediaries suggested that in the current electronic marketplace, such an event could easily happen again.”[36]

Phenomena such as the one described above occur in large non-deterministic environments and, in order to explain the causes behind them, one must initially apprehend the way trading algorithms interfere with each other at a lower level. This rationale has convinced me to undertake the challenge of building a framework on which such interferences can be observed.
1.2 Aims and Goals

The main aim of this work is to get a better understanding about the dynamics behind the price formation in the stock markets, as well as instabilities that arise as a result of interferences between trading algorithms. Such a broad comprehension will be achieved by:

- Understanding the structure and mechanisms of an electronic limit order book, the flow of orders and how the state of a limit order book is analysed at any point in time.
- Getting an insight into how the liquidity in the limit order book is provided using a certain type of market players, called Noise Agents.\(^1\)
- Looking into how some of the regularly used benchmark algorithms, such as TWAP, interfere in the price formation process.
- Understanding how a particular class of “opportunistic” agents\(^2\) can interfere with TWAP algorithms and possibly cause market instabilities.

To satisfy the targets set above, the following milestones have to be completed:

- Develop a fully functional limit order-book, which can serve the most commonly used types of orders: limit (bid or ask), market (buy or sell) and cancel.
- Implement a realistic behaviour of several noise traders that will act as „liquidity providers“ for the market and also influence the stock price formation.
- Implement a benchmark TWAP algorithm.
- Implement several “opportunistic” agents that will use specific strategies to reduce trading costs and maximize profit.
- Statistically analyse different scenarios wherein the mentioned algorithms interact with each other and draw conclusions on their effects on the market stability.

1.3 Potential Contributions

1. A realistic Agent-Based framework for discovering instability issues with Algorithmic Trading Strategies, including:

- Noise Agents for providing liquidity, according to statistical data from London Stock Exchange\(^{40}\).
- New Generation TWAP algorithms and opportunistic algorithms.

2. Statistical evidence from experiments to:

- Validate the behaviour of the simulation framework.
- Show instabilities in the market, due to algorithmic trading interference.

\(^1\)According to Fischer Black[8], noise agents are the „essential missing ingredient“ in analysing the structure of financial markets. In this project, noise agents will be used to provide liquidity to the market, as well as contributing to the price formation of a stock.

\(^2\)Also known as “profit” algorithms, these strategies analyse some of the market’s indicators, such as the current best bids/offers, spread, liquidity etc. and try to maximize their profit.
The desired expectation of this project is to make use of agent-based simulation techniques to set new research paths in the area of algorithmic trading and market stability issues. At this point, the framework developed as part of this project is intended to help the existing models by offering an alternative perspective that can validate some of the findings in this area or fill in some gaps in existence research, which couldn’t be identified using classical methodologies.

1.4 Outline of the Report

This report will comprise of seven chapters.

Chapter II gives some background on three separate sections with direct relevance to this project:

- Mechanisms of stock markets and the limit order book
- Introduction to Algorithmic Trading
- Agent-based Modelling

Chapter III outlines aspects regarding system analysis, functional and non-functional requirements, as well as a high-level view of the software’s structure - domain model diagram.

Chapter IV presents the a more detailed perspective of the design (UML diagram), programming techniques, implementation of the limit order book and algorithmic traders and some of the challenges faced along the way.

Chapter V illustrates the main techniques used for testing (unit, component integration, regression and performance testing) and validating the simulation framework. A summary of the testing coverage report and the obtained results will be provided.

Chapter VI consists of experiments carried out on distinct phases of the project. The observations will follow a chronological order, showing different outcomes, as the implementation of algorithmic trading progresses.

Chapter VII gives a final overview of the project, critically evaluates the agent-based application, summarizes the main findings of the experiments and suggests some further improvements and adjustments that can be brought to the simulation framework.
Chapter 2

Background

2.1 Stock Markets and Electronic Order Books

2.1.1 Overview

In today’s fast-paced economic climate, the stock market is the core component that drives the price formation of companies’ shares by matching different market participants, such as retail investors, hedge funds, mutual funds, investment banks, day traders etc. Since the end of the 20th century, the automation of stock markets has experienced a dramatic increase, by almost eliminating “old-fashioned” floor trading and transitioning to virtual listed exchanges\(^3\). Now more than half of the world’s financial markets use limit order books to match buyers and sellers[29].

2.1.2 Mechanisms of a Limit Order Book

An order book is simply defined as an electronic collection of outstanding limit orders for a financial instrument, such as a stock[34].

Limit Orders

A limit order is an option to buy (bid) or sell (ask/offer) a certain number of shares in a stock at a specified price or better. Limit orders are organised according to their limit prices. All bids are situated bellow the best bid, whereas all offers are placed above the best offer. The difference between the bid and the ask is called the spread. In Figure 2.1\(^4\), a basic representation of a limit order book is illustrated.

We can think of a limit order book as a snapshot[49] of all buy and sell limit orders (with their respective price and volumes) that haven’t been executed. If, for instance, a buy order arrives, it is matched against a sell order from the opposite side of the book and vice versa. If the orders match, the transaction (execution) occurs. An order will only get removed from the book if its entire volume has been executed or if the issuer (trader) decides to cancel it.

Market Orders

Market orders are buy or sell orders that are placed on the market’s current best price. A sell market order for example will always be filled as long as there are buy orders on the order book. These types of orders are preferred when the issuer of the order is wants to make sure his trade gets executed immediately. However, the volume specified in the market order may

\(^3\)http://en.wikipedia.org/wiki/Stock_market#History
\(^4\)Extract from (Kane et. al., 2011)[34]
cause the transaction to be split across several limit orders on the other side of the book before the order gets filled completely.

**Cancel Orders**

Any limit order can be cancelled by its issuer, removing it completely from the order book. Moreover, limit orders that have additional constraints, such as Fill Or Kill (FOK) - execute the entire volume of the order on the first attempt or cancel it - can be automatically cancelled should the specified constraints hold.

### 2.1.3 Flow of Orders

In this project, we will consider an asynchronous simulation of the stock market, wherein orders are submitted to the limit order book on a continuous basis. The underlying message infrastructure ensures all orders are picked up by the market and placed on the order book as they arrive. Orders are sorted according to the following rules:

1. Market orders get filled in immediately. In case a market order only gets partially filled, the rest of the order is removed from the limit order book.

2. Bid orders are sorted in descendent order, placing the lowest bid on top of the book. Analogously, ask orders are sorted in ascendant order, with the highest ask on top of the book.

3. Orders are sorted by their timestamps (the time at which they were accepted into the order book), giving precedence to orders submitted at an earlier time if the prices coincide.

### 2.2 Algorithmic Trading

In a world driven by the expansion of technological enhancements, the financial sector has always incorporated these changes in its internal structure and trading is no exception. It is arguably the area that has experienced the most significant transformations in the past few years. At first, exchanges in the world allowed market participants to remotely access virtual
order books without the need for physical presence on the exchange floor. Now a new

2.2.1 General Overview

Algorithmic trading is a technique used by various players in the stock market for executing
trade orders automatically. It relies on pre-implemented mathematical models to make
transaction decisions based on the data from the markets. In essence, software programs
choose when, how and where to trade certain financial instruments with the goal of generating
profit or managing market impact and risk.

Trading algorithms will perform correlation analysis (identify similarities in the present and
historical behaviours of stocks) and make use of benchmarks such as VWAP
(Volume-Weighted Average Price) and TWAP (Time-Weighted Average Price) to measure
trade execution. By responding mechanically to market data, algorithmic strategies can have
a major influence on price movements in the stock market. The instability may arise from
amplifying responses to price changes, which don’t always correlate to the changes in
information regarding the actual underlying value of the assets.

2.2.2 Benchmark Algorithms

Today, the majority of the algorithms are implemented due to several reasons. One of them is
to provide cheaper execution costs. Another reason is to make the executions less risky and a
third one is to disguise the trades so as to be more anonymous[44]. Therefore, fundamental
investors can appeal to strategies based on benchmarks, such as VWAP (Volume-Weighted
Average Price) and TWAP (Time-Weighted Average Price). It is important to note that these
are execution algorithms, and hence, they are implemented once the trading decisions has
already been taken.

TWAP Algorithms

Definition

The Time-Weighted-Average-Price is one of the most used benchmarks in trading algorithms.
It is defined by the average price of shares over a specific period of time. Let $p_{twap}$ be the
sought average price and $\{p_1, p_2, ..., p_n\}$ the execution prices of a certain stock at the current
point in time.

$$p_{twap} = \frac{\sum_{i=1}^{n} p_i}{n}$$

Usage

The benchmark is preferred when attempting to buy or sell a share when it reaches the TWAP
or better. As an algorithm, the TWAP is used to trade a known number of shares over a set
period of time. The order is broken down into discrete time-slices and the quantities traded in
each slice are equal[1]. Often associated with the first generation of execution algorithms[45],
TWAP is seen as a passive strategy whose main intention is to lessen its impact on the market.

As firms become more sophisticated about algorithms, there has been an increasing demand in
more flexible, customized products[35]. As a result, some of the trading algorithms based on
common benchmarks such as TWAP have incorporated random "noise" in their strategies to prevent them from being "sniffed" by other algorithms.

2.2.3 Profit Algorithms

Profit algorithms now represent one of the "neater" ways to optimally control some aspects of trading and to ensure that a maximum amount of profit can be achieved, without the implicit costs of a human trader. These algorithms follow a model design that depends on various parameters mapped to chosen trading strategy.[2]. Some of these strategies can deal with:

- Minimising cost and risk.
- Statistical arbitrage: mean - reversion strategy that finds “statistical mispricing of one or more assets based on the expected value of these assets”[56].
- Pairs trading - a strategy seeking to hedge against the market[55].

Opportunistic Algorithms

Definition

Opportunistic algorithmic traders do not have a pre-defined execution strategy; instead, they utilise real-time market data to actively search for optimal times to execute trades. Some strategies use a trend-following approach (TF) based on price prediction, others employ market-making strategies to seek profit from price volatility instead of the overall price movement. A specific category of profit strategies tries to anticipate price movements based on volume/prices imbalances between buy and sell orders books - Static Order Book Imbalance[18]. We will look into more detail at the later one in the following lines.

Usage

Static Order Book Imbalance (SOBI) is used as a criterion to identify price movements in the market. This type of algorithms can be used by “risk-averse” traders that wish to exploit short-term opportunities that arise from imbalances between the two sides of the limit order book. In YEO (2009) paper[57], it is suggested that limit orders beyond best quotes contain information on prices. This is due to the relationship between the “the outstanding orders in the limit order book and stock returns in the short run, since informed traders may attempt to preempt new information in the limit order book, resulting in imbalance between buy and sell orders for a particular stock”[57]. A sharp difference between buy and sell distributions might be an indication of upcoming price movement that can be profitably acted upon. The trader analyses the volume-weighted average prices on each side of the book. We define the volume weighted average price $p_{vwap}$ as follows:

$$p_{vwap} = \frac{\sum_{i=1}^{n} p_i \times v_i}{\sum_{i=1}^{n} v_i}$$

where:

- $n$ - total number of executed orders up until the current time
- $i$ - the $i^{th}$ trade

---

5The approach follows an explicit model of market dynamics, based on linear regression to guide order placement[51].
• $p_i$ - price of trade $i$
• $v_i$ - volume of trade $i$

The SOBI mechanism perpetually computes the volume-weighted average price on each side of the book. Depending on the trader’s level of access, these prices can be computed for the top of the limit order book (ex. 5% of the volume on the side) or even for the entire order book. The agent will submit a buy or a sell order, according to some pre-set benchmarks. Let us define:

• $b_{vwap}$ - the volume-weighted average price on the buy side of the limit order book.
• $s_{vwap}$ - the volume-weighted average price on the sell side of the limit order book.
• $\theta$ - a benchmark to which the difference between $b_{vwap}$ and $s_{vwap}$ is computed.
• $\alpha$ - number of ticks away from the current best price.
• $last\ price$ - the last executed price;

The order side and price will be computed as follows:

$$order\ Side = \begin{cases} 
  \text{buy}, & b_{vwap} - s_{vwap} > \theta \\
  \text{sell}, & s_{vwap} - b_{vwap} < \theta 
\end{cases}$$

$$order\ Price = \begin{cases} 
  last\ price - \alpha, & \text{for buy orders} \\
  last\ price + \alpha, & \text{for buy orders} 
\end{cases}$$

In Chapter V, we will illustrate the detailed implementation of this algorithm.

### 2.3 Agent-Based modelling

#### 2.3.1 Overview

Agent-based modelling is a widely-spread simulation approach whereby autonomous\(^6\) agents (representing different classes of entities) perform certain tasks and interact with each other in order to re-create and analyse the complex phenomena in large-scale, non-deterministic systems. The researcher is interested in how macro phenomena are emerging from micro level behaviour among a heterogeneous set of interacting agents[32].

The Agent-Based-Model (ABM) starts as a simple representation of reality that evolves dynamically over time[26], through the low-level interactions that occur between the agents the model is comprised of. An agent-based system is essentially a collection of autonomous entities, referred to as agents. Every agent has an independent behaviour and a set of rules for making decisions depending on the situation. Complex behaviour may emerge from interactions between the agents [13, 14, 15].

#### 2.3.2 Applications in Financial Markets

The Agent-Based-Model paradigm has proven useful for many domains, ranging from biology (e.g. [6, 7]) and physics to social sciences. However, this approach can be also used in the area of financial markets - an environment in which many outcomes are determined by sophisticated strategies used by players in the stock market to get an edge over their competitors [21]. In the following lines, we will present some of the main pieces of work in this research area.

\(^6\)Autonomy property means that agents have control both over their internal state and over their own behaviour[33].
**Agent-Based Simulation of Financial Markets - A modular, continuous-time approach, Katalin Boer-Sorban**

With the goal of gaining a broader understanding of the inner-workings of financial markets, Boer-Sorban determines the aspects that drive the dynamics of the market by applying a “microstructure based approach”[9]. The proposed framework doesn’t represent a single artificial stock market, but a ”modular tool” for studying several types of markets and an arbitrary number of trading strategies. Also, the author pays special attention to common features of the stock market, such as continuous trading sessions. Although the proposed tool captures various characteristics of financial markets, it does not include the presence of algorithmic traders, which are now a major piece in the stock market puzzle.

**Agent-based Financial Markets: Matching Stylized Facts with Style, Blake LeBaron**

An advocate of the agent-based approach, LeBaron(2005)[39] states that many of the puzzling empirical results from finance that contradict the traditional theoretical models deal with problems of behavioural heterogeneity. The dynamics of heterogeneity can “only be explored in a framework that allows agent strategies to adapt and adjust over time, and more importantly, to respond to features of the aggregate population around them.” According to LeBaron(2005), the main advantage of agent-based models is that they can match features of finance in a fairly “stylized manner”, by having a complex world comprised of agents that adjust and adapt to “maximize relatively simple objectives”[39].

**Asynchronous Simulations of a Limit Order Book, Gilles Daniel**

In this thesis, Gilles(2006) presents a computer implementation of an asynchronous model of a limit order book. By developing zero-intelligence agents7, Gilles (2006) is able to replicate “a realistic order flow”[20], and qualitatively and quantitatively facts exhibited by real markets. Using non-trivial statistical properties, the framework proposed by Gilles can simulate price dynamics, starting from a micro-founded agent-based model. We will apply the statistical parameters used in Gilles’s thesis to obtain a more realistic behaviour of our noise agents and hence, produce more reliable conclusions as we will show in Chapter VI.

**Agent-Based Modeling - What I Learned From the Artificial Stock Market, Paul E. Johnson**

This article brings a more technical perspective on agent-based modelling, by highlighting some of the challenges of implementing such a framework. The proposed recommendations are based on an artificial stock market implemented by the Santa-Fe Institute and highlight the importance of using an object-oriented approach to model such frameworks.

### 2.3.3 Advantages of Agent-Based Simulation

It has been proven that common mathematical models have major flaws when it comes to successfully assessing the outcome of certain scenarios which occur in stock markets. The reason lies in the fact that these models can’t simulate a non-deterministic environment and therefore, rule out some of the real-life situations that arise on a regular basis. For instance, Boer-Sorban[9] argues that the simplistic assumptions made (investor’s homogeneous rational behaviour, price formation at equilibrium etc.) can be rejected by empirical studies. LeBaron(2002) reminds us of the “fragility of the equilibrium models that we consider in finance and macroeconomics”[38], whereas Gilles(2006) argues that agent-based models can be

---

7Zero-intelligence agents are an equivalent of noise traders in this project report.
used to verify whether the classical models of price formation “would break-down when tested in a more realistic environment”[20].

As a result, agent-based modelling represents a more suitable approach to simulating complex environments, such as the financial markets, on the grounds that it can generate elaborate behaviours using relatively simple rules.

2.4 JADE - Java Agent DEvelopment Framework

2.4.1 What is JADE?

In a nutshell, JADE is a software platform that facilitates the development of multi-agent systems in Java. The implementation of agents is achieved through a middle-ware that uses FIPA-ACL communication language to send messages between participating entities (agents).

2.4.2 Agent Communication

JADE uses an asynchronous message-based paradigm to establish the communication between agents. This form of communication eliminates the temporal dependencies between the sender and the receiver, which means that the messages are sent even when the receiver is not “available”. Moreover, the receiver can select which message to process or discard, according to pre-set priority rules, called “Message-Templates”.

Message exchanging operates under a pre-defined format, called ACL (Agent Communication Language). A message has the following fields:

- **AID** - the unique identifier of the sender
- **Receivers** - list of the AID’s of the recipients
- **Intention** (or performative) - the most commonly used are REQUEST, INFORM, CFP (call for proposal) and REPLY.
- **Content** - the information contained by the message
- **Content Language** - the syntax used to express the content
- **Ontology** - can be a user-defined vocabulary of the symbols used in the message content.

The **Peer-to-Peer** approach the system uses allows agents to directly send messages to other agents using their AID, or to discover each other through the Yellow Pages Service8 services provided by JADE.

2.4.3 JADE Architecture

JADE Environment consists of the following components:

- **Container** - A host for one or more agents
  - **Main Container** - the default active container
  - **Normal Container** - optional container that has to register with the main container

8The Yellow Pages service offers agents the possibility of publishing their services for other agents to discover and exploit.
Note: We will stress the importance on having our trader agents and the market agent on the main container, as this minimises the latency in message sending and hence, improves the performance of the simulation.

- **Platform** - A set of containers

- **AMS** (Agent Management System) - A special agent located on the main container with the following duties:
  - Ensures each agent in the platform has a unique name.
  - Provides the ability of creating/killing agents on remote containers[11].

- **DF** (Directory Facilitator) - provides the Yellow Pages Service.

- **MTS** (Message Transport System)\(^9\) - Controls the exchange of all messages within the same platform or from remote platforms.

By having these features in place, JADE can exhibit a fully distributed system inhabited by agents representing different entities. Moreover, agents are able to transparently communicate with one another thanks to “a unique location-independent API that abstracts the underlying communication infrastructure.”[5] Figure 2.2 displays an overview of the JADE Environment[5].

---

\(^9\) Also called Agent Communication Channel - ACC
2.4.4 Agent Behaviour

An agent’s actions are carried out within behaviours. A **behaviour** is essentially a task that can be performed by the agent. It is worth specifying that once a behaviour has been defined, it can be used by any agent on the platform, should the programmer consider necessary. This is achieved by simply adding the desired behaviour to the agent when it is started or within other behaviours. There are two main categories of behaviours:

1. **Simple Behaviours** - are further separated into:
   
   - **One Shot Behaviour** - is executed only once.
   - **Cyclic Behaviour** - is executed cyclically as long as its agent is alive.
   - **Ticker Behaviour** - a cyclic behaviour, which is executed periodically.
   - **Waker Behaviour** - is executed at a specified time.
   - **Receiver Behaviour** - is executed when a message is received.

2. **Composite Behaviours**: - consist of several "sub-behaviours" which are executed following the rules below:
   
   - **Sequential Behaviour** - sub-behaviours are executed one after the other and terminate when the last sub-behaviours has ended.
   - **Parallel Behaviour** - sub-behaviours are executed in parallel and can either terminate when all, any or N sub-behaviours are done, as specified by the programmer.

An agent can execute several behaviours concurrently and the scheduling of these is cooperative, which means the programmer has to define when agents switch from one behaviour to another one.

2.4.5 Why JADE?

The idea of using a framework based on an object-oriented programming language is instantly appealing in the context of modelling agent-based simulations. The simple, yet effective implementation of agents can result in generating complex interactions. Since this project follows a similar pattern found in other simulations of financial markets - modelling micro-environments from the interaction of individual strategies - JADE offers a comprehensive set of features that grant the achievement of such a goal.
Chapter 3

Analysis and Requirements

3.1 Overview of the system

The framework presented in this report will attempt to replicate a small-scale market, where agents have access to a single limit order book of a particular stock. It is essential to state who the main players in this framework are, as well as the mechanisms behind order executions and price formations.

Figure 3.1: Overview of the Simulation Framework

The framework presented in this report will attempt to replicate a small-scale market, where agents have access to a single limit order book of a particular stock. It is essential to state who the main players in this framework are, as well as the mechanisms behind order executions and price formations.
3.1.1 Components of the Stock Market

Market Agent

We visualise the stock market as an autonomous entity in itself. As shown in Figure 3.1, the market has two main components:

1. **Passive (or Reactive) Component** - Handling the incoming order messages submitted by traders.

   The market agent acts as a liaison between traders and the order book, ensuring the requests are added in chronological order. The market can essentially serve two types of messages:
   
   - **Orders** - limit or market
   - **Cancellations** - a request from a trader to remove its order from the limit order book.

   It is important to note that the market agent doesn’t actually process the orders, but only redirects them to the Brokerage Service, which will be explained in the next subsection. Order submissions, cancellations and executions are events in the Brokerage component to which the market agent responds by sending messages to the agents involved in these actions.

2. **Active Component** - Generating the "information feed" with the latest updates to be sent to the traders.

   Concurrently with handling messages to and from the trader agents and Brokerage Service, the market agent is also responsible for keeping track of the current status of the limit order book and sending regular feeds to all the active traders. The feeds contain information such as the current best bid, best offer, last execution price, last executed volume etc.

Brokerage Service

A major component in our simulation, the Brokerage Service is responsible for analysing the content of the messages received by the market agent, as well as validating, matching and executing orders. The process flow is different depending on the type of message received:

- **Orders** - An order (limit or market) is firstly certified (ensuring it contains all the required fields - type, side, price and volume), after which it is added on the order book. A confirmation message is sent to the trader via the market agent.

- **Cancellations** - Cancellations are sent using the unique order id of the previously placed order. The Brokerage searches in the limit order book for the order identification. If a match is found, the order gets removed and a reply is sent to the trader with the help of the market agent.

Executing Orders

The Brokerage analyses the orders on top of the book. If there is a match, i.e. the bid is equal or greater than the offer, the orders get executed and executions report are sent to the involved counter parties through the market agent.
Limit Order Book

The Limit Order Book contains the actual data structures used for storing the outstanding orders. It also stores a "history" of all the executions of a simulation session.

Note: The priority rules explained in Section 2.1.3 are embedded in the internal structure of the limit order book, ensuring the orders are placed correctly, in line with their side (buy or sell), price, and arrival time.

Noise Traders

Noise Traders submit order requests to the market agent at random times. The side, order type, volume and price are each chosen from specific sets of distributions, as shown in Gilles (2006)[20]. While side, order type and volume are independent variables, the price is influenced by the current best bid/ask.

TWAP Traders

The TWAP trader operates by a set of pre-defined variables, such as the total volume to be executed, the desired time span in which the order size should be filled, the number of slices in which the order size is broken down and aggressiveness. A first order is sent to the market, after which any subsequent orders are submitted after the previous unfilled orders are cancelled.

Opportunistic Traders

Opportunistic Traders are continuously monitoring the market and try to detect favourable indicators in price movements, such as the total outstanding volume on a particular side of the limit order book and map them against their own “beliefs”.
3.2 System Requirements

The diagram exhibited in Figure 3.1 serves as a basic model, on top of which we can analyse the requirements needed for the framework implementation. The requirements of the system have been categorised in functional (what the system should do) and non-functional (how should the system perform) and assigned priorities conforming to the MoSCoW approach[4].

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>C</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUST have this</td>
<td>SHOULD have this</td>
<td>COULD have this</td>
<td>WON'T have now</td>
</tr>
</tbody>
</table>

3.2.1 Functional Requirements

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>MoSCoW</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System Components</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>The system shall contain a Limit Order Book.</td>
<td>MUST</td>
</tr>
<tr>
<td>F2</td>
<td>The system shall contain a Market Agent.</td>
<td>MUST</td>
</tr>
<tr>
<td>F3</td>
<td>The system shall contain a Brokerage Service.</td>
<td>MUST</td>
</tr>
<tr>
<td>F4</td>
<td>The system shall contain Noise Agents.</td>
<td>MUST</td>
</tr>
<tr>
<td>F5</td>
<td>The system shall contain TWAP Agents.</td>
<td>MUST</td>
</tr>
<tr>
<td>F6</td>
<td>The system shall contain Opportunistic Agents.</td>
<td>MUST</td>
</tr>
<tr>
<td>F6</td>
<td>The system shall contain a Simulation Agent.</td>
<td>MUST</td>
</tr>
<tr>
<td>F7</td>
<td>The system shall contain an Initiation Agent.</td>
<td>MUST</td>
</tr>
</tbody>
</table>

**Limit Order Book**

| F1 | The Limit Order Book shall store limit orders. | MUST |
| F2 | The Limit Order Book shall store a list of executions. | MUST |
| F3 | The Limit Order Book shall sort the limit orders by type. | MUST |
| F4 | The Limit Order Book shall sort the limit orders by price. | MUST |
| F5 | The Limit Order Book shall sort the limit orders by arrival time. | MUST |

**Market Agent**

| F | The Market Agent shall accept limit orders. | MUST |
| F | The Market Agent shall accept market orders. | MUST |
| F | The Market Agent shall accept cancellation orders. | MUST |
| F | The Market Agent shall send order confirmations to the order issuers. | MUST |
| F | The Market Agent shall send execution order updates to traders involved in the execution. | MUST |
| F | The Market Agent shall send regular market updates. | MUST |

**Market Updates**

| F | The Market Updates shall contain the current best bid. | MUST |
| F | The Market Updates shall contain the current best offer. | MUST |
| F | The Market Updates shall contain the last executed price. | MUST |
| F | The Market Updates shall contain the last executed volume. | MUST |
| F | The Market Updates shall contain the order size of the best bid. | SHOULD |
| F | The Market Updates shall contain the order size of the best volume. | SHOULD |

**Brokerage Service**

<p>| F | The Brokerage Service shall validate limit orders. | MUST |
| F | The Brokerage Service shall validate market orders. | MUST |</p>
<table>
<thead>
<tr>
<th><strong>F</strong></th>
<th>The Brokerage Service shall validate cancellation orders.</th>
<th><strong>MUST</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>F</strong></td>
<td>The Brokerage Service shall add limit orders to the limit order book.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The Brokerage Service shall execute limit orders against other limit orders.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The Brokerage Service shall execute market orders against limit orders.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The Brokerage Service shall remove orders from the limit order book.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The Brokerage Service shall change the status of an orders when it is executed or removed from the limit order book.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The Brokerage Service shall log all the submitted orders in a .csv file.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The Brokerage Service shall log all the executed orders in a .csv file.</td>
<td><strong>MUST</strong></td>
</tr>
</tbody>
</table>

**Noise Agent**

<table>
<thead>
<tr>
<th><strong>F</strong></th>
<th>The Noise Trader shall submit limit orders.</th>
<th><strong>MUST</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>F</strong></td>
<td>The Noise Trader shall submit market orders.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The Noise Trader shall send cancellation orders.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The Noise Trader shall randomly submit orders following statistical distributions.</td>
<td><strong>SHOULD</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The Noise Trader shall keep an inventory of order reports.</td>
<td><strong>SHOULD</strong></td>
</tr>
</tbody>
</table>

**TWAP Agent**

<table>
<thead>
<tr>
<th><strong>F</strong></th>
<th>The TWAP Agent shall submit limit orders.</th>
<th><strong>MUST</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>F</strong></td>
<td>The TWAP Agent shall submit market orders.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The TWAP Agent shall send cancellation orders.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The TWAP Agent shall set the total volume to be executed.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The TWAP Agent shall have a pre-set time span during which the order must be executed.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The TWAP Agent shall set a number of suborders to be sent.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The TWAP Agent shall send an order at a regular interval.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The TWAP Agent shall randomly adjust the interval at which it is sending orders.</td>
<td><strong>COULD</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The TWAP Agent shall slice the total volume into equal amounts given by the number of suborders.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The TWAP Trader shall introduce a random variance in the volume of suborders.</td>
<td><strong>COULD</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The TWAP Agent shall set the aggressiveness of the order prices.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The TWAP Agent shall keep an inventory of order reports.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The TWAP Agent shall track the current total executed volume and volume left.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The TWAP Agent shall cancel any outstanding orders if the volume left reaches 0.</td>
<td><strong>MUST</strong></td>
</tr>
</tbody>
</table>

**Opportunistic Agent**

<table>
<thead>
<tr>
<th><strong>F</strong></th>
<th>The Opportunistic Agent shall submit limit orders.</th>
<th><strong>MUST</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>F</strong></td>
<td>The Opportunistic Agent shall submit market orders.</td>
<td><strong>MUST</strong></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>The Opportunistic Agent shall send cancellation orders.</td>
<td><strong>MUST</strong></td>
</tr>
</tbody>
</table>
The Opportunistic Agent shall keep an inventory of order reports. MUST
The Opportunistic Agent shall have access to the entire limit order book. MUST
The Opportunistic Agent shall use the volume-weighted average price on each side of the limit order book to make a trading decision. MUST

Initiation Agent
The Initiation Agent shall read a list of limit orders from a preloaded .csv file. MUST
The Initiation Agent shall submit limit orders to the market agent in the order they appear in the .csv file. MUST

Simulation Agent
The Simulation Agent shall start one Market Agent. MUST
The Simulation Agent shall start an Initiation Agent. MUST
The Simulation Agent shall start several Noise Agent. MUST
The Simulation Agent shall start a TWAP Agent MUST
The Simulation Agent shall start several Opportunistic Agents. MUST
The Simulation Agent shall use a timer to start the rest of the agents after the Initiation Agent. MUST
The Simulation Agent shall kill the initiation Agent after the rest of the agents are started. MUST
The Simulation Agent shall use a timer to stop the simulation after a specified number of seconds. MUST

Order
The Order shall contain a unique order ID. MUST
The Order shall contain the trader AID. MUST
The Order shall contain the side - buy or sell. MUST
The Order shall contain the type - limit or market. MUST
The Order shall contain a price. MUST
The Order shall contain a volume (order size) MUST
The Order shall contain an entry time. MUST
The Order shall contain a status. MUST

Order Status
The Order Status shall contain the stage of the order - new, filled, partially filled, or cancelled. MUST
The Order Status shall contain the initial volume of the order. MUST
The Order Status shall contain the executed volume of the order. MUST
The Order Status shall contain the remaining volume of the order. MUST
The Order Status shall contain the last price the order was executed at. MUST

3.2.2 Non-Functional Requirements

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>MoSCoW</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>The System shall be implemented in JAVA.</td>
<td>MUST</td>
</tr>
<tr>
<td>F</td>
<td>The System shall be unite tested.</td>
<td>MUST</td>
</tr>
<tr>
<td>F</td>
<td>The System shall be component-integration tested.</td>
<td>MUST</td>
</tr>
<tr>
<td>F</td>
<td>The System shall be regression tested.</td>
<td>MUST</td>
</tr>
<tr>
<td>F</td>
<td>The System shall be validated.</td>
<td>MUST</td>
</tr>
<tr>
<td>F</td>
<td>The System shall be performance tested.</td>
<td>MUST</td>
</tr>
<tr>
<td>F</td>
<td>The System shall produce data that can be statistically analysed.</td>
<td>MUST</td>
</tr>
<tr>
<td>F</td>
<td>The System shall handle approx. 30 agents without significant performance drawbacks.</td>
<td>MUST</td>
</tr>
<tr>
<td>F</td>
<td>The System shall be FIPA compliant.</td>
<td>MUST</td>
</tr>
<tr>
<td>F</td>
<td>The agents’ message queue shall contain no more than 100000 messages at any one time.</td>
<td>MUST</td>
</tr>
</tbody>
</table>
Chapter 4

Design and Implementation

4.1 Design Features

The design of this project will follow an object-oriented paradigm, because the implementation of the main components in an artificial stock market can be more easily achieved by having them “wrapped-up” into class members. A high level of complexity is achieved by putting the emphasis on classes and exchangeable components. These allow for the implementation of more elaborated programs by combining together or reusing existing components[50].

4.1.1 Encapsulation

The code is divided into several small, cohesive, self-contained parts (classes). These parts can implement interfaces and have a set of methods, which can be called by other parts of the program to make use of their services. The internal implementation of class objects minimises the direct access to their components and hence, allows for changes to be made without having to locate dependencies with other classes. Essentially, each slice of functionality is implemented by a self-contained set of classes and objects, following the idea of “separation of concerns”[50].

4.1.2 Following an Agile approach

Given the scale of this project, the design and implementation phases have followed an incremental, iterative procedure, due to continuous changes in the system requirements. Each phase of the project started with refining the functional and non-functional requirements, then improving the existing code and finally, adding extra layers of complexity. We created Java classes for each of the main components in the high-level design of the system. To force encapsulation, we then further sliced the main classes into smaller, reusable objects. As code was written, we commented the most important sections of the code using the Javadoc Tool[48] to ensure the system design could be followed conveniently by developers who wish to bring further improvements to the framework.

4.2 UML Class Diagram
4.3 Market Side Implementation

4.3.1 Limit Order Book

The order book contains several data structures to store the outstanding bids and offers, as well as an internal “history” of all the submitted orders and execution prices.

We separated the bids and offers into two priority queues. A priority queue places elements according to a pre-set order specified at construction time according to their natural order. The Order objects that are placed in these queues implement the Comparable interface, which can be configured to adhere to the rules set by the limit order book (see Section 2.1.3). We have chosen this particular data structure for several reasons:

- It does not permit insertions of null elements or objects that do not implement the Comparable interface.
- The implementation provides $O(\log(n))$ time for insertion and polling methods, linear time for removing a specific object and constant time for retrieval methods[47].
- Supplies blocking retrieval operations, making it fully thread-safe.

As most of the operations occur on top of the order book (order retrieval, removal etc.), this data structure has proven to be the most optimal solution for hosting outstanding orders. The execution prices are stored in a `DoubleArrayList`, which permits fast retrievals of the current Time-Weighted-Average price through its `Descriptive.mean()` method[17]. Finally, an updated copy of all submitted orders is kept in an `ArrayList`.

4.3.2 Brokerage Service and Market Agent

Serving Requests

All messages sent by trader agents go through the Market Agent and get redirect to the Brokerage Service. The two main types of messages received are orders and cancellations. To automatically process these messages as they arrive, we use JADEs Ontology Server. This is a special type of cyclic behaviour that invokes a method of the form `public void serveActionMessage (Action a, ACLMessage message)` for each incoming message that encompasses a specific template.

```java
public void serveSendOrderRequest(final SendOrder sendOrder, final ACLMessage request) {
    Order order = new Order();
    //Create new instance of Order
}
```

The content of `Action a` is scanned and automatically directed to a serving method of type `serveActionMessage`.

```java
public void serveSendCancelOrderRequest(final CancelOrder cancelOrder, final ACLMessage request) {
    String cancelID = cancelOrder.getOrderID();
    String side = cancelOrder.getSide();
    //Invoke the cancel method with the values obtained above
    cancelOrder(cancelID, side);  
}
```

A `SendOrder` object contains a type (limit or market), a side (buy or sell), a price (double value with 2 decimals) and a volume. It is the brokerage service’s responsibility to create a
new object of type Order, containing the parameters set in SendOrder. In addition to these, the Order object embodies additional information:

- **Trader’s ID** - in the form of the local AID of the sender (see Section 2.4.2).
- **Unique order ID** - local variable updated incrementally after adding a new order to the order book.
- **Entry Time** - Time Stamp given by `System.currentTimeMillis()`, which returns current UTC time in milliseconds [46].
- **Order Status** - a new class object supplying additional information, such as stage (“New”, “Filled”, “Partially Filled”, “Cancelled”), volume left, volume executed, last execution price.

The Order is then validated, added on the limit order book and logged in a .csv file. Next, a new `Action` class object is created and wrapped inside the content of a new `ACLMessage`. The object contains an order acknowledgement confirmation, which is sent to the trader. The following code section illustrates a standard JADE message sending protocol.

```java
private void sendOrderConfirmation (final Order order, ACLMessage request){
    final SendOrderConfirmation confirmation = new SendOrderConfirmation();
    confirmation.setOrderID(order.getOrderID());
    confirmation.setOrderType(order.getOrderType());
    confirmation.setSide(order.getSide());
    confirmation.setPrice(order.getPrice());
    confirmation.setVolume(order.getVolume());
    confirmation.setOrderStatus(order.getStatus());
    try{
        final Action action = new Action(request.getSender(), confirmation);
        final ACLMessage replyMessage = request.createReply();
        if (order.getStatus().getStage().equalsIgnoreCase("CANCELED")) {
            replyMessage.setPerformative(ACLMessage.REJECT);
        } else {
            replyMessage.setPerformative(ACLMessage.CONFIRM);
        }
        replyMessage.setReceiver(request.getSender());
        replyMessage.setOntology(MarketOntology.getInstance().getName());
        replyMessage.setLanguage(language);
        marketAgent.getActiveAgent().fillContent(replyMessage, action);
        marketAgent.sendReplyMessage();
    } catch (Exception ex){}
}
```

**Executing Orders**

The Brokerage Service always operates on “the top” of the limit order book, trying to match and execute buy orders and sell orders. If the orders cross, they get executed and the execution price is logged in a separate .csv file. A `SendOrderReport` message is sent to both parties containing the order’s ID, side, type, stage (filled or partially filled), initial volume, executed volume, volume left, and the executed price to which a “brokerage” fee of 2% is added.
Feeding Market Information

The Market Agent incorporates a “feed behaviour”, whereby the latest updates on the limit order book are transmitted to the active traders on the platform (see Section 3.1.1). JADE provides a neat and intuitive solution for broadcasting messages by making use of AMS Service. The search() method returns an array of Agent objects that meet a specific criteria (AMSAgentDescription). The programmer can then iterate through the array, extract the local AIDs of the trader agents and add them as recipients in the message content.

Performance Improvements

Behaviour scheduling in JADE is cooperative (see Section 2.4.4), which means the action() method of a behaviour is never interrupted for other behaviours to step in. The problem arises when the behaviour has to perform a blocking operation, which blocks the entire agent and not just itself. This has been solved by placing the Ontology Server behaviour in a dedicated thread, using the JADE’s Threaded Behaviour Factory.

```java
if (!limitOrderBook.getBuyOrders().isEmpty() && !limitOrderBook.getSellOrders().isEmpty()) {
    final Order buyOrder = limitOrderBook.getBuyOrders().peek();
    final Order sellOrder = limitOrderBook.getSellOrders().peek();
    final double executionPrice = matcher.executeAtPrice(buyOrder, sellOrder);
    if (executionPrice == -1) {
        execute(buyOrder, sellOrder, executionPrice);
        executionLogger.addExecutionPrice(executionPrice);
    }
}

public void sendMarketFeed(String sms) {
    try {
        AMSAgentDescription [] agents = null;
        SearchConstraints c = new SearchConstraints();
        c.setMaxResults(new Long(-1));
        agents = AMSService.search(myAgent, new AMSAgentDescription (), c);
        final Action action = new Action(myAgent.getAID(), sms);
        ACLMessage statusMessage = new ACLMessage(ACLMessage.INFORM);
        statusMessage.setOntology(MyOntology.getInstance().getName());
        statusMessage.setLanguage(language);
        for (int i = 0; i < agents.length; i++) {
            statusMessage.addReceiver(agents[i].getName());
        }
        myAgent.getStatusManager().fillContent(statusMessage, action);
        myAgent.send(statusMessage);
    } catch (Exception ex) {
    }
}
```
4.4 Algorithmic Traders Implementation

4.4.1 Noise Traders

Why implement noise?

“I do not believe it makes sense to create a model with information trading but no noise trading where traders have different beliefs and one trader’s beliefs are as good as any other traders beliefs. Differences in beliefs must derive uniformly from differences in information”[8]. Following Black’s beliefs, Shleifer et. al. (1990) state that the investor sentiment/ limited arbitrage approach “yields a more accurate description of financial markets than the efficient markets paradigm”[52], also adding that this model proves some empirically testable implications about asset prices. Finally, Alfarano and Lux (2007) aim to replicate some of the findings illustrated in previous papers, by testing a simplified model in which noise traders interact with more “rational agents”. Consistent with previous research, it is found that models such as the one described above can reproduce some of “the key stylized facts of financial markets”[3]: unit roots\(^{10}\), fat tails\(^{11}\) and volatility clustering\(^{12}\).

Note: We need to understand that noise agents are not a particular type of trading algorithm per se, but the they are implemented, so that they can exhibit a large number of small events, which are often the causal factor of inefficiencies in the market. Furthermore, the implemented noise agents will act as liquidity providers in the market for the TWAP and opportunistic traders.

The main challenge in programming a noise trader behaviour was its ability of exhibiting some of the emergent dynamics of price formation that mimic a realistic stock market. Gilles (2006)[20] stresses the importance of finding the “key ingredients” for the emergence of realistic market dynamics when modelling an agent-based framework. According to Gilles, some of those ingredients can be implemented by robustly recovering several properties of markets that feature a realistic micro-structure. We will apply some of the parameters used in Gilles’ order-book simulation that were found to be consistent with empirical facts derived from London Stock Exchange’s on-book market[40].

The noise agent implementation adopts the following parameters:

\[
\text{OrderSide} = \begin{cases} 
\text{buy}, & \text{probability} = 0.5 \\
\text{sell}, & \text{probability} = 0.5 
\end{cases}
\]

Let us define the probability variables:

- \(p_l\) - the trader submits a limit order.
- \(p_m\) - the trader submits a market order.
- \(p_c\) - the trader sends a cancellation order.

\(^{10}\)Unit Roots are defined as features of processes that evolve through time that can cause problems in statistical inference if it is not adequately dealt with.

\(^{11}\)A fat-tailed distribution is a probability distribution that has the property, along with heavy-tailed distributions, that they exhibit extremely large skewness or kurtosis.

\(^{12}\)Volatility clustering refers to the observation, as noted by Mandelbrot (1963), that “large changes tend to be followed by large changes, of either sign, and small changes tend to be followed by small changes.”
The type of order will be chosen as follows:

\[
\text{OrderType} = \begin{cases} 
\text{limit,} & \text{probability } = p_l \\
\text{market,} & \text{probability } = p_m \\
\text{cancellation,} & \text{probability } = p_c 
\end{cases}
\]

Next, we define:

- \( \text{price}_{\text{limit}} \) - the price at which the limit order is submitted.
- \( p_{\text{in}} \) - the probability of a limit order price to be inside the spread.
- \( p_{\text{out}} \) - the probability of a limit order price to be outside of the spread.
- \( \mathcal{U}(a,b) \) - a uniform distribution between \( a \) and \( b \).
- \( \Delta \) - the distance between the limit order price and the current best price.

\[
\text{price}_{\text{limit}} = \begin{cases} 
\mathcal{U}(\text{best Bid, best Offer}), & \text{probability } = p_{\text{in}} \\
P_l(\Delta) \sim \frac{1}{\Delta^{1+\alpha}}, & \text{probability } = p_{\text{out}} 
\end{cases}
\]

In other words, there is a \( p_{\text{in}} \) probability the limit order price will be uniformly distributed between the best bid and the best price - in the spread, or power-law distributed outside the spread with probability \( p_{\text{out}} \).

On Euronext[20], reports on the volume of orders indicates that order sizes follow a log-normal distribution for limit-orders, whereas the volume of market orders is almost always the same as the best counterpart’s. Hence we define:

- \( v(o) \) - volume of the order \( o \).
- \( \mathcal{N}(\mu, \sigma) \) - normal distribution with mean \( \mu \) and standard deviation \( \sigma \).

\[
\text{orderVolume} = \begin{cases} 
v(\text{best Bid}) \mid v(\text{best Offer})] & \text{for market orders} \\
P_l(\log v) \sim \mathcal{N}((\mu, \sigma)) & \text{for limit orders}
\end{cases}
\]

Balancing the inventory: We assume that our agents are initially “empowered” with \( S \) shares and \( C \) units of cash. Let \( W(t) \) be the wealth of an agent at time \( t \). We also define \( p(t) \) to be the latest traded price at time \( t \). According to Gilles (2006)[20], the wealth is defined as:

\[
W(t) = S(t) \times p(t) + C(t)
\]

Let us presume that our noise agents have a preference \( f(f < 1) \) for shares and \( 1 - f \) for cash. Gilles (2006) defines the current status \( S^* \) of an agent’s inventory using the following formula:

\[
S^* = f \times \frac{W(t)}{p(t)}
\]

We will introduce an extra parameter for our market orders to balance the agent’s inventory.

\[
\text{orderSignMarket} = \begin{cases} 
\text{buy,} & S < S^* \\
\text{sell,} & S > S^* 
\end{cases}
\]

“Sleeping time”: In a typical market, the trading activity is more dense at the beginning and at the end of the day[20]. Our simulation, however, captures a certain time frame in the middle of a normal day of trading. Hence, our noise agents are endowed with a random uniform distribution \( \mathcal{U}(x,y) \), where \( s_1 = 0 \) and \( s_2 = 2000 \text{ms} \).

\[
time \sim \mathcal{U}(s_1, s_2)
\]
4.4.2 TWAP Traders

TWAP Agents are firstly set-up with pre-defined parameters (see Section 3.1.1 - TWAP Traders), which are passed to an agent as arguments. The arguments are stored in a generic Object array that can be accessed by the agent’s behaviour at start-up.

TWAP traders have special inventories used for tracking the total current executed volume and the volume left to be executed. When sending a new order, the TWAP agent calculates the percentage of the initial order size that has been executed and compares it with the percentage of the time that has passed. This is calculated as the difference (in milliseconds between) the current time and the time when the first order was sent.

An ideal scenario expects a small percentage difference between the passed time and executed volume, meaning that at least 90% of the size of the previously submitted order must get filled before a new order is submitted.

However, subject to the available liquidity in the market and frequent changes in best bids or offers, the scenario described above is unlikely to materialize. Therefore, we introduced in the
TWAP’s strategy an aggressiveness factor. Let \( d \) be the difference between the elapsed time (as a percentage of the total time span in which the order is set to be executed) and the current executed volume (as a percentage of the total volume to be executed). We also define \( \text{orderPrice} \) to be the price at which the order is sent and \( \text{bestPrice} \) - the current best bid or ask. Finally, \( \text{tickSize} \), as the name suggests, is the smallest increment by which the price of a stock can move and is set at 0.01\(^{13}\). The “±” shows the amount added (best bid) or subtracted (best offer) to/from \( \text{bestPrice} \).

\[
\text{OrderPrice} = \begin{cases} 
\text{bestPrice} \pm \text{tickSize} \times 1 & 0 < d \leq 5 \\
\text{bestPrice} \pm \text{tickSize} \times 2 & 5 < d \leq 10 \\
\text{bestPrice} \pm \text{tickSize} \times 3 & 10 < d \leq 20 \\
\text{bestPrice} \pm \text{tickSize} \times 4 & 20 < d \leq 30 \\
\text{bestPrice} \pm \text{tickSize} \times 5 & d > 30 
\end{cases}
\]

In real stock markets, common implementations of TWAP algorithms include variations on the size of sub-orders to reduce detectability\cite{37}. As an additional discretion measure that prevents our TWAP algorithm to be detected by other algorithms, we deviated from the standard fixed amount of shares in each submitted order and introduced a random “noise”. Let \( \text{volume} \) be the initial number of shares in the TWAPs child limit order. We define:

- \( \text{twoPercent} = \frac{2 \times \text{orderVolume}}{100} \)
- \( \text{noise} = U(0, \text{twoPercent}) \) - a uniformly distributed value between 0% and 2% of the child order volume.

The final volume of the child order becomes:

\[ \text{volume} = \text{volume} \pm \text{noise} \]

4.4.3 Opportunistic Traders

Motivation

The format in which we set up our simulations makes it difficult for opportunistic trading strategies based on “historical data” (ex. volatility index) to perform well under a realistic scenario. A short-term approach, such as the static order book imbalance can therefore achieve better results under strict time constraints. In addition to this, order book imbalances

\[^{13}\text{The standard tick size used for orders between 50 and 99.9900, according to BATS}[41].\]
can be seen as a sign of instability in the market. YAO (2009) suggests that a causality between liquidity imbalances and market returns can be established, adding that this phenomena should be carefully studied in the light of “ever increasing importance of limit orders in security markets[57].”

Implementation

Opportunistic agents have the same level of access as noise agents. However, their strategy seeks to maximise profits by finding imbalances between the volume weighted average price of the buy side \( (b_{vwap}) \) and the volume weighted average price of the sell side \( (s_{vwap}) \) of the limit order book. We also define:

- \( \theta \) - a benchmark to which the difference between \( b_{vwap} \) and \( s_{vwap} \) is computed.
- \( \alpha \) - a number of ticks away from the current best price.
- \( last_price \) - the last executed price;

\[
orderSide = \begin{cases} 
  \text{buy}, & s_{vwap} - b_{vwap} > \theta \\
  \text{sell}, & b_{vwap} - s_{vwap} > \theta 
\end{cases}
\]

\[
orderPrice = \begin{cases} 
  \text{last}_{\text{price}} - \alpha, & \text{for buy orders} \\
  \text{last}_{\text{price}} + \alpha, & \text{for buy orders}
\end{cases}
\]

We also maintained a log-normal distribution for the order volume of limit orders.

\[
orderVolume = P_i(\log v) \sim \mathcal{N}(\mu, \sigma)
\]

We endowed our opportunistic agents with a portfolio consisting of 10000 cash units and 1000 shares and enforced additional constraints to maintain a realistic portfolio management. We defined:

- \( c \) - the current number of cash units in the portfolio
- \( s \) - the current number of shares in the portfolio
- \( p_c \) - a percentage of the initial number of cash units in the portfolio
- \( p_s \) - a percentage of the initial number of shares in the portfolio

\[
action = \begin{cases} 
  \text{buy}, & s_{vwap} - b_{vwap} > \theta \text{ and } c > p_c \\
  \text{sell}, & b_{vwap} - s_{vwap} > \theta \text{ and } s > p_s \\
  \text{sell}, & c < p_c
\end{cases}
\]

As a result, the opportunistic agent only sends buy limit orders as long as the level of cash is kept at a minimum pre-set threshold. Furthermore, the opportunistic agent will balance the current amount of cash in the portfolio by selling outstanding shares to meet the minimum set threshold.

Setting the parameters of Opportunistic Agent[18]:

<table>
<thead>
<tr>
<th>Description</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access level to the limit order book</td>
<td>( a )</td>
<td>10%</td>
</tr>
<tr>
<td>Minimum sleeping time before the opportunistic agent is activated</td>
<td>( s_1 )</td>
<td>0.00ms</td>
</tr>
</tbody>
</table>
Maximum sleeping time before the opportunistic agent is activated $s_2$ 5000.00ms
Order Volume $(\mu, \sigma)$ (4.5, 0.8) shares
Initial number of shares $s$ 10000 shares
Initial amount of cash $c$ 1000 units
Percentage of the initial number of shares $p_s$ 10%
Percentage of the initial amount of cash $p_c$ 10%
Benchmark for the difference between volume-weighted average prices on the two sides of the order book $\theta$ 0.07
Number of ticks away from the last executed price $\alpha$ 0.1

4.4.4 Common Feature of Trading Algorithms - Update Behaviour

The Update Behaviour is a task (executed in parallel with other behaviours) performed cyclically by every active trader in the stock market. The `UpdateBehaviour` class consists of three components:

- **Market Status** - an `ArrayList` of `SendMarketStatus` objects.
- **Portfolio** - an `ArrayList` of `SendOrderConfirmation` objects.

The rationale behind implementing an Update Behaviour is to separate the passive component of the noise trader (message receiving) from the actual trading algorithm wrapped inside another behaviour. This object-oriented approach minimises the possible interdependencies between classes, as well as occurrences of duplicated code.

As we have seen so far, a trader agent can receive three types of messages (contained in JADE’s action objects): `SendOrderConfirmation`, `SendMarketStatus` and `SendOrderReport`. All these messages are stored in the agent’s message queue that operates on a `FIFO` (first in, first out) basis. To dissociate between these messages and place them in the right array lists,
we specified “templates” to be used when calling the agents `receive()` method. These templates are created using the `jade.lang.acl.MessageTemplate` factory class and, in our case, match a certain `sender` and `message performative`.

```
MessageTemplate confirmMessageTemplate = new MessageTemplate.and(
        MessageTemplate.MatchPerformatives(ACLMessage.CONFIRM),
        MessageTemplate.MatchSender(new AID("market", null)));

MessageTemplate feedMessageTemplate = new MessageTemplate.and(
        MessageTemplate.MatchPerformatives(ACLMessage.INFO),
        MessageTemplate.MatchSender(new AID("market", null)));

MessageTemplate reportMessageTemplate = new MessageTemplate.and(
        MessageTemplate.MatchPerformatives(ACLMessage.CFP),
        MessageTemplate.MatchSender(new AID("market", null)));

ACLMessage confirmMessage = myAgent.receive(confirmMessageTemplate);
ACLMessage feedMessage = myAgent.receive(feedMessageTemplate);
ACLMessage reportMessage = myAgent.receive(reportMessageTemplate);
```

In the case of the `Market Status` list for instance, the agent is only interested in the very latest update. Since it can’t retrieve this information directly from the message queue (earlier message get read first), we have create a new data structure that grants access to the most recent message.

```
if (feedMessage != null)
{
    try{
        ContentManager content = myAgent.getContentManager();
        Action contentActionFeed = (Action) content.extractContent(feedMessage);
        SendMarketStatus currentStatus = (SendMarketStatus) contentActionFeed.getAction();

        // Check if the arraylist contains update and remove the oldest
        if (marketStatus.getUpdates().size() > 0)
            try{
                marketStatus.getUpdates().remove(0);
            } catch (Exception e) {
            }

        // Add a the latest market update to the array list
        marketStatus.getUpdates().add(currentStatus);

    } catch (Exception e) {
    }
}
```
Chapter 5

Testing and Validation

5.1 Unit Testing

The first phase of the testing process is unit testing - testing the behaviour of a unit of code - class or method. We started by unit testing objects comprised of primitive types (String, double, long etc.) and then we moved on to classes that instantiated objects of other classes in order to achieve a reliable component integration testing. We used JUnit 4.0[31] for its ability to exploit Java’s annotation features to identify tests in our test suites. The unit testing coverage is approximately 72%.

5.1.1 Regression Testing

The intent of this type of testing is to ensure that any changes made to the simulation framework do not introduce new faults[43]. An agile development approach requires the automation of test suites, as features are continuously being added to the system. Therefore, regular testing after subsequent changes to the program has been done.

5.2 Component Integration Testing

Testing the individual behaviour of the components of the system does not suffice, as our system is based on method calls between individual modules. Therefore, the purpose of component integration testing is to verify that the system’s dynamic behaviour is accurately modelled by the interactions between class objects.

The main challenge is that, in order to test method calls between classes, we often need to include the implementation of object types in the test, due to external dependencies. An easier solution, however, is to create “mock” objects dynamically. For this, we used Mockito 1.9.0[28]. This testing framework also provides a high-level API for specifying how the object under test should invoke the mock objects it interacts with, and how the mock objects will behave in response. Figure 5.1 exemplifies such a situation.

As shown in Section 2.4, communication in JADE is done via asynchronous message parsing. The standard Message Transport Protocol (MTP), defined by FIPA, includes the definition of a transport protocol and a standard encoding of the message envelope[5]. The main difficulty is to test whether the right messages get sent between agents. To do this, we have set up JADE with an additional tool - JAVA SNIFFER[53]. Thus, we can remotely connect to our platform (see Section 2.4.2) and visualise the message parsing between our agents (refer to Figure 5.2).
5.3 Performance Testing

We will assess the performance of the framework by considering two criteria:

1. The number of exchanged messages during the simulation.
2. The number of logged orders.
3. The number of logged traded prices.

To increase the number of exchanged messages during a fixed time span, we chose to place all our agents on the same platform, and more precisely - in the Main Container (see Section 2.4.3). According to (Cortese et. al., 2002), this renders a “better performance performance when grouping in the same container agents that need to interact a lot”[19]. Using both the JAVA SNIFFER[53], as well as JADE’s embedded sniffer agent[5], we set up the number of agents on the platform and the duration of the simulation and obtained the following results:

- Number of active agents: 13.
- Simulation duration: 400 seconds.
- **Number of exchanged messages:** approx. 9600.
- **Number of logged orders:** approx. 2200.
- **Number of traded prices:** approx. 700.

A snapshot of the results is listed in Appendix F.

**Note:** According to data from January 2010 to February 2012, on London Stock Exchange, the daily number of trades has been on average 593403[23] on approx. 1400 traded stocks. We will consider Vodafone Share Price (VOD) as a reference for a single stock traded on the limit order book during the market hours of one day. According to London South East, on 28th of March, 2012, the number of VOD traded shares after market close was 11488[22]. Hence, we can scale up our simulation time to imitate a time span of approx. 30 minutes of trading during a normal trading day.

**Hardware and software configuration:**

<table>
<thead>
<tr>
<th>Hardware and Software Configuration</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel(R) Core(TM)2 Duo CPU P8700 @2.53GHz (2CPUs)</td>
</tr>
<tr>
<td>BIOS Version</td>
<td>08.00.10</td>
</tr>
<tr>
<td>Memory</td>
<td>6144MB RAM</td>
</tr>
<tr>
<td>Operating System</td>
<td>Microsoft Windows 7 Professional 64-bit</td>
</tr>
<tr>
<td>JAVA</td>
<td>JDK 7</td>
</tr>
<tr>
<td>JADE</td>
<td>4.1.1</td>
</tr>
</tbody>
</table>

5.4 **Validation**

A critical phase in the software development life cycle, validation is used to assure the quality of the software and increases its usability and reliability[12]. Since the complexity of this framework prevents it from being exhaustively tested, we need to establish the confidence that the software is fit for its intended use. The main tasks the system must perform are illustrated in Section 3.2 - System Requirements. Therefore, before validating our framework, we verified that:

- There are no inconsistencies among requirements.
- All requirements are expressed in terms that are measurable or objectively verifiable.
- All requirements are appropriate for the intent of the system.

We analysed the content of the .csv files to ensure the orders are logged and executed according to the specifications of the limit order book (see Section 2.1.3). Furthermore, we monitored the type of orders, as well as the portfolios and inventories of our trading algorithms to check if the models stated in Section 4.4 are implemented correctly. Finally, the results we obtained from our simulations are consistent with the real stock market behaviour, as we will show in Chapter VI.

---

14 Operating day is from 8:00 am to 4:30 pm.
Chapter 6

Experiments

This chapter will comprise of a series of experiments that aim to discover how interferences between trading algorithms affect the dynamics of price formation and the liquidity in the limit order book.1

6.1 Hypotheses to be tested

Before we state which hypotheses we are going to test, we must define a set of variables that will be used throughout this section.

- $S$ - the name of the single stock traded on the limit order book.
- $\mu_S$ - the mean of the population distribution of traded prices of the stock $S$ before TWAP or “opportunistic” agents are introduced.
- $\mu_T$ - the mean of the population distribution of traded prices of the stock $S$ when a TWAP algorithm is introduced.
- $\mu_O$ - the mean of the population distribution of traded prices of the stock $S$ when several opportunistic algorithms are introduced.
- $L$ (Liquidity) - the total volume of all orders placed on the limit order book during the simulation.
- $\text{vol}_{\text{Traded}}$ - the total traded volume on the limit order book during the simulation.
- $\text{vol}_{\text{Twap}}$ - the total order size to be executed by the TWAP algorithm during the simulation.
- $pV$ - a pre-set percentage of the total order size to be executed by the TWAP algorithm.
- $f_{\text{Twap}}$ - the frequency at which the TWAP algorithm is submitting orders. It is defined as the time span / number of sub orders.

6.1.1 Null Hypothesis 1

*Given that a TWAP algorithm is introduced, there will be no effect on the distribution $\mu_S$ of the traded prices of the stock $S$.*

6.1.2 Null Hypothesis 2

*An increase in the total order size $\text{vol}_{\text{Twap}}$ by $pV\%$ will have no effect on the distribution $\mu_T$ of traded prices of the stock $S$.*
6.1.3 Null Hypothesis 3

A change in the frequency $f_{\text{Twap}}$ at which the TWAP algorithm is trading will have no effect on the distribution $\mu_T$ of the traded prices of stock $S$.

6.1.4 Null Hypothesis 4

Given that several opportunistic algorithms are introduced, there will be no effect on the distribution $\mu_T$ of the traded prices of the stock $S$.

6.2 Experiment Setup

Before we started the experiments, we needed to ensure that all the initial randomness in our simulations is minimised. Having a common starting point (populating the limit order book with the same set of data every time we run the simulation) ensures that the first orders submitted by our noise agents have the same reference price. This will significantly reduce the variances between the average traded prices in a sample obtained after running the same experiment several times. Hence, all of our runs start from a pre-loaded limit order book, containing 500 orders, with the following parameters:

- $orderSign \sim Bern(0.5)$ BUY or SELL → 238 buy orders and 262 sell orders.
- $orderVolume = P_i(\log v) \sim N(\mu, \sigma)$ where $\mu = 4.5$ and $\sigma = 0.8$[20].
- $orderPrice = \begin{cases} U(74.45, 74.92), & \text{for buy orders} \\ U(75.05, 75.59), & \text{for sell orders} \end{cases}$
- all orders are limit orders.
- tick size is 0.01[41].

Another justification for using a pre-populated limit order book is the need for sufficient market liquidity, before the TWAP agent is introduced. Hence, this approach was used instead of a standard Opening Auction used by the London Stock Exchange model\textsuperscript{15}. The main disadvantage of having a pre-loaded order book is the occurrence of “resistance levels”\textsuperscript{16} in the market that might affect the dynamics of price formation.

6.2.1 Setting simulation parameters

We started each of our experiments with 10 noise agents, endowed with 1000 shares and 100000 of cash units and one market agent. After each run, the following metrics were recorded:

- Mean average traded price.
- Total volume of submitted orders.
- Total volume of submitted orders on the buy side.
- Total volume of submitted orders on the sell side.
- Sell Side volume.

\textsuperscript{15}According to Lillo et. al. (2004), an opening auction is a period leading up to the start of the continuous double auction[20], wherein orders are placed but no transactions take place.

\textsuperscript{16}Resistance levels may cause periods of congestion during which the stock price fluctuates between the support (bottom) and the resistance (top) levels.
• Total volume traded.
• Number of submitted buy orders.
• Number of submitted sell orders.
• Number of trades/executions.

For each null hypothesis we tested, we ran the simulation 15 times and extracted the metrics mentioned above. A full list of results can be found in Appendix A.

6.2.2 Statistical Analysis

To statistically analyse the results of our experiments, we used the following tests:

• Kolmogorov Smirnov test (KS test)\textsuperscript{17} for testing the normality of the distributions.
• Mann - Whitney U test\textsuperscript{18} for testing the null hypotheses.

Null Hypothesis for normality: The data in the samples containing traded prices is normally distributed.

We have tested several samples of traded prices for normality, using the Kolmogorov - Smirnov test and, for each type of simulation, we found the following $p$ - values:

• Simulation with Noise Agents: $p = 0.0000174$

• Simulation with Noise Agents and Twap Agent: $p = 0.00000224$

Since $p < 0.05$, we rejected the null-hypotheses and found that our samples are not normally distributed. Hence, we used the Mann - Whitney U test.

Our first data set comprised of the average traded prices after running the simulation with 10 Noise Agents.

![Figure 6.1: Trajectory of traded prices - Noise Agents](image)

\textsuperscript{17}In the Kolmogorov Smirnov test\cite{42}, samples are standardised and compared with a standard normal distribution.

\textsuperscript{18}The Mann - Whitney Test (also called Wilcoxon rank sum test)\cite{24} is statistical hypothesis significance test that analyses the differences between two samples of independent observations.
Null Hypothesis 1: Given that a TWAP algorithm is introduced, there will be no effect on the distribution $\mu_S$ of the traded prices of the stock $S$.

To test the first null hypothesis, we introduced a TWAP Agent with these settings:

- Total order volume to be executed: 8000
- Number of child orders (suborders) to be sent: 15
- Time span in which the order should be executed: 300 seconds.

The Mann - Whitney Test on traded price distributions generated a p-value $p = 0.000727$. Therefore, with 99% confidence level, we rejected the null hypothesis 1 and concluded that the introduction of a TWAP agent does affect the distribution $\mu_S$ of the traded prices of the stock $S$.

We have also found that the introduction of the TWAP agent causes an imbalance in the liquidity provided in the limit order book. As expected, we discovered that the liquidity on the buy side of the order book is, on average, greater than the liquidity on the sell side of the order book by 17%.

Finally, we analysed the volume traded during the simulation and found an average increase of 14% after introducing the TWAP Agent.

Null Hypothesis 2: An increase in the total order size $\text{vol}_{\text{Twap}}$ by $pV\%$ will have no effect on the distribution $\mu_T$ of traded prices of the stock $S$.

Next, we increased the total order volume to be executed ($\text{vol}_{\text{Twap}}$) by 25%.

- Total order volume to be executed: 10000
- Number of child orders (suborders) to be sent: 15
- Time span in which the order should be executed: 300 seconds.

The Mann - Whitney Test on traded price distributions generated a p-value $p = 0.002693$. Hence, with 99% confidence level, we also rejected the null hypothesis 2 and concluded that an increase by 25% in the total order size traded by the Twap agent will have an affect the
distribution $\mu_T$ of traded prices.

**Null Hypothesis 3:** A change in the frequency $f_{\text{Twap}}$ at which the TWAP algorithm is trading will have no effect on the distribution $\mu_T$ of the traded prices of stock $S$.

For this experiment, we increased the frequency at which the sub-orders are sent to the market by 33.3%.

- Total order volume to be executed: 8000
- Number of child orders (suborders) to be sent: 20
- Time span in which the order should be executed: 300 seconds.

The Mann-Whitney test on traded price distributions resulted in a p-value $p = 0.4013$. As a result, we accepted this hypothesis.

**Null Hypothesis 4:** Given that several opportunistic algorithms are introduced, there will be no effect on the distribution $\mu_T$ of the traded prices of the stock $S$.

In this experiment, we ran the simulation with two opportunistic agents set at the parameters described in Section 4.4.3.

![Figure 6.3: Trajectory of traded prices - Noise Agents + Twap Agent + Opportunistic Agents](image)

The Mann-Whitney test on traded price distributions generated a p-value $p = 0.00006925$. Consequently, the null hypothesis was rejected with a 99% confidence level, meaning that the introduction of several opportunistic traders based on static order imbalance strategies will affect the price distribution of the traded stock. Furthermore, the average traded price increased from 72.244 to 72.442 (20 ticks), confirming the amplify effect caused by the interference between the TWAP algorithm and the Static Order Imbalance algorithms.

Our experiment also revealed an accentuated liquidity imbalance, as the share volume on the buy side exceeded the volume on the sell side by 24%. Finally, the introduction of opportunistic traders improved the overall volume traded by 13%.

A table showing the results obtained from the experiments can be viewed in Appendix A.
Chapter 7

Conclusions and Evaluation

7.1 Summary
This project has been concerned with analysing market instabilities due to the interaction between trading algorithms. We have developed an agent-based framework using an object-oriented language and we have used this framework to test how several types of algorithms, operating on different parameters, affect the dynamics of price formation in the market. We have also considered the consequences of trading algorithm interference on the provision of liquidity and the imbalance in the provision of liquidity on a limit order book.

7.2 Discussion
7.2.1 Project Evaluation
This section will critically evaluate the completion of the milestones set at the beginning of this project. The simulation framework has the following components/features:

- A limit order book capable of serving limit orders, market orders and cancellation orders.
- A set of noise agents with customisable parameters that are able to generate a realistic order flow and provide liquidity to the market. We need to mention, however, that populating the order book with uniformly distributed limit orders can affect the order flow to some extent, due to the generated resistance levels (refer to Section 6.2).
- A realistic implementation of a TWAP algorithm, with additional modifiable variables, such as order price aggressiveness and order volume “noise” (refer to Sections 2.2.2 and 4.4.2).
- A basic implementation of opportunistic trading algorithms using Static Order Imbalance Strategies (refer to Sections 2.2.3 and 4.4.3).

Furthermore, a statistical analysis has been undertaken in order to extract the qualitative and quantitative facts exhibited as a result of the interactions between trading algorithms.

7.2.2 Contributions
In this report, we have presented a model which is able to replicate the behaviour of noise agents, as well as TWAP trading algorithms and Opportunistic trading algorithms based on Static Order Book Imbalance Strategies. First of all, we have discovered how the statistical properties of a limit order book can be inferred from the interactions between limit orders and market orders, as they provide liquidity or remove it from the market. These orders are
submitted according to the strategies implemented by different categories of algorithmic traders. Furthermore, we have found that modifications in the parameter setting of our algorithmic trading behaviours can have a significant impact on the market’s stability. Finally, we have explored the agent-based model’s capabilities of generating complex dynamics even with relatively simple behavioural rules. Frameworks such as the one presented in this report can serve as alternatives to the “analytically tractable efficient market models”[20], because their structure can give rise to emergent properties that could not be deduced by standard theoretical models.

7.3 Further Work

7.3.1 Suggested improvements on the Limit Order Book and Market Agent

In this version, our limit order book accepts limit and market orders, as well as cancellation orders. However, Lillo et. al. (2004)[40] mention additional order types, such as stop-loss orders, “fill or kill” (FOK) and “immediate or cancel” (IOC). Furthermore, in order to reliably mimic the behaviour of real stock markets, this framework can be enhanced by an opening auction mechanism (see Section 6.2.1). This will reduce the likelihood of resistance levels occurrence and render better price formation dynamics.

7.3.2 Further development of Noise Agents

We have demonstrated that Noise Agents play a significant role in rendering a realistic order flow and also provide constant liquidity to the limit order book. The parameters we used to calibrate our model (see Section 4.4.1 - Setting the parameters) replicate empirical facts from the London Stock Exchange’s on-book market[40]. However, this research can be taken further by assessing the role of each of these parameters on the dynamics of price formation and liquidity provision[20]. Hence, a parameter can take values over a given range (the others are kept at their original value) and the impact, if any, on the order book dynamics can be analysed in the context interferences between noise agents and trading algorithms.

7.3.3 Further development of Trading Algorithms

Opportunistic Trading Algorithms

The constraints of our simulations limit the range of ”opportunistic” trading algorithms to short-term strategies, such as Static Order Book Imbalance (SOBI). Sherstov et. al. [51] suggest three possible profit trading strategies, which have been tested against the SOBI algorithm: The Trend Following19 Agent, the Market-Making20 Agent and the Reinforcement Learning Agent21. Although these strategies have been modelled to assess their performance in relation to profit, one could complement Sherstov’s findings by analysing the strategies’ impact on market stability.

High Frequency Trading Algorithms

A distinct category of algorithms, High - Frequency (HFT)[25] trading algorithms have been experiencing an increasing popularity amongst automated trading strategies. However, a full

19Trend Following is an approach whereby a model of market dynamics based on linear regression is used to guide order placement
20Unlike the trend-following strategy, the market making strategy “capitalizes on small fluctuations rather than long-term trends and is likely to produce a smaller variance in profit.”[51]
21Reinforcement learning is a machine - learning methodology for achieving long-term performance “in poorly understood and possibly non-stationary environments”[51]
picture of the potential systemic risks triggered by HFT in the stock market has not been yet established[27]. Therefore, this framework can be complemented by an effective implementation of different HFT strategies, ranging from market-making to arbitrage and even pure speculative algorithms, such as “sniffing” high-frequency traders[54]. As suggested by Kirilenko et. al. (2011)[36] and Gomber et. al. (2011)[27], high-frequency traders may exacerbate short-term extreme volatility periods, such as the May 6, 2011 Flash Crash. While Gomber et. al. (2011) argue that “empirical research on HFT is restricted by a lack of accessible and reliable data”, we suggest an agent-based approach as a potential starting point for modelling HFT strategies.

7.4 Conclusion

The advent of technology has radically transformed the nature of trading in most stock exchanges. As a result, the impact of algorithmic trading on market stability is a rich research area with important practical applications. We have built a small scale agent-based model that analyses stability features of the market in relation with algorithmic trading, and we hope to have succeeded in setting up a foundation on which improved limit order book models and innovative algorithmic trading strategies can be designed.
Bibliography


[18] Computer Science and Information - University of Pennsylvania. Static Order Book Im-

[19] Elisabetta Cortese, Filippo Quarta, and Giosue Vitaglione. Scalability and Performance of


pressure in futures markets. In Proceedings of the 9th annual conference on Genetic and


pothesis tests and multiple interpretations of decision rules. Statistics Surveys, 4:1–39,
2010.


[29] Martin D Gould, Mason A Porter, Stacy Williams, Mark Mcdonald, Daniel J Fenn, and
D Sam. Limit order books. 2009.


[33] Nicholas R Jennings. On agent-based software engineering. Artificial Intelligence,

[34] David Kane, Andrew Liu, and Khanh Nguyen. Analyzing an Electronic Limit Order Book.


[36] Andrei Kirilenko, Albert S Kyle, Mehrend Samadi, and Tugkan Tuzun. The Flash Crash :

microstructure October 2010 R esume. e. 2010.


# Appendix A

## Experiment Results

### A.1 Experiment 1

<table>
<thead>
<tr>
<th>Average Traded Price</th>
<th>Total Volume</th>
<th>Volume Traded</th>
<th>Number of submitted buy orders</th>
<th>Number of submitted sell orders</th>
<th>Number of executions</th>
<th>Volume on buy side</th>
<th>Volume on sell Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>75.0557</td>
<td>269190</td>
<td>49918</td>
<td>1032</td>
<td>1064</td>
<td>673</td>
<td>128475</td>
<td>140715</td>
</tr>
<tr>
<td>75.06403</td>
<td>268531</td>
<td>50078</td>
<td>1111</td>
<td>1067</td>
<td>611</td>
<td>126914</td>
<td>142276</td>
</tr>
<tr>
<td>75.11803</td>
<td>253968</td>
<td>39068</td>
<td>1056</td>
<td>1007</td>
<td>661</td>
<td>129271</td>
<td>124697</td>
</tr>
<tr>
<td>75.10622</td>
<td>256255</td>
<td>50019</td>
<td>1106</td>
<td>990</td>
<td>646</td>
<td>136925</td>
<td>119330</td>
</tr>
<tr>
<td>75.06265</td>
<td>260791</td>
<td>46037</td>
<td>1049</td>
<td>1071</td>
<td>670</td>
<td>133126</td>
<td>127665</td>
</tr>
<tr>
<td>75.0557</td>
<td>260334</td>
<td>44811</td>
<td>1046</td>
<td>1011</td>
<td>686</td>
<td>133061</td>
<td>127173</td>
</tr>
<tr>
<td>75.05507</td>
<td>261412</td>
<td>46558</td>
<td>1082</td>
<td>1109</td>
<td>705</td>
<td>128688</td>
<td>132744</td>
</tr>
<tr>
<td>75.03904</td>
<td>217604</td>
<td>35850</td>
<td>950</td>
<td>956</td>
<td>668</td>
<td>115967</td>
<td>101857</td>
</tr>
<tr>
<td>74.999</td>
<td>248304</td>
<td>40635</td>
<td>1154</td>
<td>1093</td>
<td>643</td>
<td>137175</td>
<td>144494</td>
</tr>
<tr>
<td>75.10148</td>
<td>271920</td>
<td>47172</td>
<td>1087</td>
<td>1093</td>
<td>643</td>
<td>137175</td>
<td>144494</td>
</tr>
<tr>
<td>75.02459</td>
<td>254647</td>
<td>41004</td>
<td>1115</td>
<td>1107</td>
<td>664</td>
<td>126222</td>
<td>128425</td>
</tr>
<tr>
<td>75.13511</td>
<td>260998</td>
<td>55127</td>
<td>1058</td>
<td>1071</td>
<td>702</td>
<td>135896</td>
<td>133902</td>
</tr>
<tr>
<td>75.00726</td>
<td>277148</td>
<td>53424</td>
<td>1127</td>
<td>1103</td>
<td>687</td>
<td>143874</td>
<td>133274</td>
</tr>
<tr>
<td>75.0288</td>
<td>263458</td>
<td>48240</td>
<td>1136</td>
<td>1092</td>
<td>707</td>
<td>134892</td>
<td>128566</td>
</tr>
<tr>
<td>75.12525</td>
<td>264150</td>
<td>53282</td>
<td>1023</td>
<td>1030</td>
<td>685</td>
<td>136230</td>
<td>127920</td>
</tr>
</tbody>
</table>
Figure A.1: Submitted order prices

Figure A.2: Submitted order volumes

Figure A.3: Submitted prices

Figure A.4: Traded Prices
## A.2 Experiment 2

<table>
<thead>
<tr>
<th>Average Traded Price</th>
<th>Total Volume</th>
<th>Volume Traded</th>
<th>Number of submitted buy orders</th>
<th>Number of submitted sell orders</th>
<th>Number of executions</th>
<th>Volume on buy side</th>
<th>Volume on sell side</th>
</tr>
</thead>
<tbody>
<tr>
<td>75.15507</td>
<td>248452</td>
<td>44650</td>
<td>957</td>
<td>846</td>
<td>647</td>
<td>131856</td>
<td>116596</td>
</tr>
<tr>
<td>75.18302</td>
<td>280747</td>
<td>55887</td>
<td>1041</td>
<td>894</td>
<td>673</td>
<td>152500</td>
<td>128247</td>
</tr>
<tr>
<td>75.24949</td>
<td>280554</td>
<td>56073</td>
<td>1132</td>
<td>1005</td>
<td>634</td>
<td>150657</td>
<td>19897</td>
</tr>
<tr>
<td>75.22778</td>
<td>278800</td>
<td>51354</td>
<td>1043</td>
<td>866</td>
<td>650</td>
<td>159486</td>
<td>198954</td>
</tr>
<tr>
<td>75.18916</td>
<td>282440</td>
<td>58274</td>
<td>1158</td>
<td>1014</td>
<td>704</td>
<td>156314</td>
<td>126126</td>
</tr>
<tr>
<td>75.2185</td>
<td>294509</td>
<td>53181</td>
<td>1083</td>
<td>966</td>
<td>669</td>
<td>158195</td>
<td>136134</td>
</tr>
<tr>
<td>75.23805</td>
<td>289501</td>
<td>53643</td>
<td>1142</td>
<td>1021</td>
<td>641</td>
<td>151149</td>
<td>138352</td>
</tr>
<tr>
<td>75.36103</td>
<td>280950</td>
<td>50214</td>
<td>1156</td>
<td>1017</td>
<td>667</td>
<td>150720</td>
<td>130230</td>
</tr>
<tr>
<td>75.19799</td>
<td>288961</td>
<td>54925</td>
<td>1064</td>
<td>981</td>
<td>678</td>
<td>152867</td>
<td>125519</td>
</tr>
<tr>
<td>75.21022</td>
<td>280862</td>
<td>54348</td>
<td>1152</td>
<td>1015</td>
<td>726</td>
<td>152984</td>
<td>127387</td>
</tr>
<tr>
<td>75.34073</td>
<td>259674</td>
<td>57141</td>
<td>1134</td>
<td>1016</td>
<td>670</td>
<td>136109</td>
<td>123565</td>
</tr>
<tr>
<td>75.22607</td>
<td>259883</td>
<td>48085</td>
<td>1042</td>
<td>908</td>
<td>663</td>
<td>140771</td>
<td>119112</td>
</tr>
<tr>
<td>75.35909</td>
<td>290463</td>
<td>52234</td>
<td>1162</td>
<td>1024</td>
<td>646</td>
<td>158669</td>
<td>131794</td>
</tr>
<tr>
<td>75.28448</td>
<td>250887</td>
<td>58745</td>
<td>1103</td>
<td>1069</td>
<td>662</td>
<td>139226</td>
<td>117961</td>
</tr>
<tr>
<td>75.31181</td>
<td>261053</td>
<td>58484</td>
<td>1122</td>
<td>1014</td>
<td>642</td>
<td>139696</td>
<td>121357</td>
</tr>
</tbody>
</table>
Figure A.5: Submitted order prices

Figure A.6: Submitted order volumes

Figure A.7: Submitted prices

Figure A.8: Traded Prices
### A.3 Experiment 3

EXPRESSMENT 3 (400 seconds) - 10 NOISE AGENT + TWAP AGENT (total order size: 10000 shares, time span: 300 seconds, number of sub-orders: 15)

<table>
<thead>
<tr>
<th>Average Traded Price</th>
<th>Total Volume</th>
<th>Volume Traded</th>
<th>Number of submitted buy orders</th>
<th>Number of submitted sell orders</th>
<th>Number of executions</th>
<th>Volume on buy side</th>
<th>Volume on sell side</th>
</tr>
</thead>
<tbody>
<tr>
<td>75.3896</td>
<td>274387</td>
<td>51752</td>
<td>1174</td>
<td>1017</td>
<td>670</td>
<td>146286</td>
<td>128101</td>
</tr>
<tr>
<td>75.36688</td>
<td>257310</td>
<td>54603</td>
<td>1344</td>
<td>917</td>
<td>710</td>
<td>139771</td>
<td>117539</td>
</tr>
<tr>
<td>75.31886</td>
<td>269214</td>
<td>49988</td>
<td>1185</td>
<td>939</td>
<td>678</td>
<td>151198</td>
<td>117816</td>
</tr>
<tr>
<td>75.3523</td>
<td>281750</td>
<td>55113</td>
<td>1160</td>
<td>980</td>
<td>645</td>
<td>149805</td>
<td>131945</td>
</tr>
<tr>
<td>75.35488</td>
<td>280319</td>
<td>54108</td>
<td>1220</td>
<td>963</td>
<td>706</td>
<td>156694</td>
<td>133625</td>
</tr>
<tr>
<td>75.34789</td>
<td>288650</td>
<td>57397</td>
<td>1186</td>
<td>992</td>
<td>687</td>
<td>161096</td>
<td>127554</td>
</tr>
<tr>
<td>75.34486</td>
<td>268649</td>
<td>59019</td>
<td>1105</td>
<td>905</td>
<td>670</td>
<td>151686</td>
<td>116963</td>
</tr>
<tr>
<td>75.33503</td>
<td>267327</td>
<td>51991</td>
<td>1182</td>
<td>963</td>
<td>721</td>
<td>149840</td>
<td>117487</td>
</tr>
<tr>
<td>75.36722</td>
<td>283674</td>
<td>54925</td>
<td>1169</td>
<td>988</td>
<td>677</td>
<td>157424</td>
<td>126250</td>
</tr>
<tr>
<td>75.30511</td>
<td>269951</td>
<td>59584</td>
<td>1195</td>
<td>951</td>
<td>714</td>
<td>148386</td>
<td>121565</td>
</tr>
<tr>
<td>75.36912</td>
<td>289070</td>
<td>57779</td>
<td>1241</td>
<td>996</td>
<td>712</td>
<td>157839</td>
<td>131231</td>
</tr>
<tr>
<td>75.39701</td>
<td>237597</td>
<td>51548</td>
<td>1081</td>
<td>861</td>
<td>707</td>
<td>132144</td>
<td>105453</td>
</tr>
<tr>
<td>75.41974</td>
<td>297295</td>
<td>54762</td>
<td>1134</td>
<td>986</td>
<td>682</td>
<td>161529</td>
<td>135766</td>
</tr>
<tr>
<td>74.34894</td>
<td>291231</td>
<td>56433</td>
<td>1167</td>
<td>988</td>
<td>660</td>
<td>160105</td>
<td>131126</td>
</tr>
<tr>
<td>75.31058</td>
<td>292503</td>
<td>57261</td>
<td>1174</td>
<td>936</td>
<td>721</td>
<td>165034</td>
<td>127469</td>
</tr>
</tbody>
</table>
Figure A.9: Submitted order prices

Figure A.10: Submitted order volumes

Figure A.11: Submitted prices

Figure A.12: Traded Prices
## A.4 Experiment 4

<table>
<thead>
<tr>
<th>Average Traded Price</th>
<th>Total Volume</th>
<th>Volume Traded</th>
<th>Number of submitted buy orders</th>
<th>Number of submitted sell orders</th>
<th>Number of executions</th>
<th>Volume on buy side</th>
<th>Volume on sell Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>75.25816</td>
<td>286408</td>
<td>54969</td>
<td>1121</td>
<td>947</td>
<td>681</td>
<td>160828</td>
<td>125580</td>
</tr>
<tr>
<td>75.16466</td>
<td>253254</td>
<td>51633</td>
<td>911</td>
<td>842</td>
<td>654</td>
<td>133330</td>
<td>119924</td>
</tr>
<tr>
<td>74.32072</td>
<td>284220</td>
<td>57953</td>
<td>1180</td>
<td>1012</td>
<td>668</td>
<td>157768</td>
<td>126452</td>
</tr>
<tr>
<td>75.24722</td>
<td>282884</td>
<td>54835</td>
<td>1099</td>
<td>997</td>
<td>699</td>
<td>155961</td>
<td>126923</td>
</tr>
<tr>
<td>75.27773</td>
<td>279444</td>
<td>54530</td>
<td>1171</td>
<td>1001</td>
<td>783</td>
<td>152430</td>
<td>126014</td>
</tr>
<tr>
<td>75.26616</td>
<td>262345</td>
<td>50382</td>
<td>1134</td>
<td>1056</td>
<td>667</td>
<td>138433</td>
<td>123912</td>
</tr>
<tr>
<td>75.24165</td>
<td>249437</td>
<td>47969</td>
<td>1077</td>
<td>939</td>
<td>644</td>
<td>136848</td>
<td>112589</td>
</tr>
<tr>
<td>75.261829</td>
<td>283264</td>
<td>54954</td>
<td>1110</td>
<td>965</td>
<td>777</td>
<td>157819</td>
<td>125445</td>
</tr>
<tr>
<td>75.38689</td>
<td>266124</td>
<td>57123</td>
<td>1153</td>
<td>1000</td>
<td>645</td>
<td>144614</td>
<td>121710</td>
</tr>
<tr>
<td>75.28121</td>
<td>274007</td>
<td>59883</td>
<td>1080</td>
<td>1029</td>
<td>686</td>
<td>151682</td>
<td>122325</td>
</tr>
<tr>
<td>75.27381</td>
<td>274641</td>
<td>50246</td>
<td>1130</td>
<td>1043</td>
<td>636</td>
<td>145883</td>
<td>128758</td>
</tr>
<tr>
<td>75.271</td>
<td>273440</td>
<td>57508</td>
<td>1183</td>
<td>988</td>
<td>701</td>
<td>150955</td>
<td>122485</td>
</tr>
<tr>
<td>75.28511</td>
<td>271429</td>
<td>54915</td>
<td>1062</td>
<td>909</td>
<td>666</td>
<td>154253</td>
<td>137176</td>
</tr>
<tr>
<td>75.27366</td>
<td>279588</td>
<td>55271</td>
<td>1189</td>
<td>99</td>
<td>706</td>
<td>157527</td>
<td>132361</td>
</tr>
<tr>
<td>75.40071</td>
<td>244590</td>
<td>56530</td>
<td>936</td>
<td>833</td>
<td>635</td>
<td>133060</td>
<td>111530</td>
</tr>
</tbody>
</table>

EXPERIMENT 4 (400 seconds) - 10 NOISE AGENT + TWAP AGENT (total order size: 8000 shares, time span: 300 seconds, number of sub-orders: 20)
Figure A.13: Submitted order prices

Figure A.14: Submitted order volumes

Figure A.15: Submitted prices

Figure A.16: Traded Prices
### A.5 Experiment 5

**EXPERIMENT 5 (400 seconds) - 10 NOISE AGENTS + TWAP AGENT (total order size: 8000 shares, time span: 300 seconds, number of sub-orders: 15) + 2 OPPORTUNISTIC AGENTS**

<table>
<thead>
<tr>
<th>Average Traded Price</th>
<th>Total Volume</th>
<th>Volume Traded</th>
<th>Number of submitted buy orders</th>
<th>Number of submitted sell orders</th>
<th>Number of executions</th>
<th>Volume on buy side</th>
<th>Volume on sell Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>75.45284</td>
<td>308692</td>
<td>58165</td>
<td>1225</td>
<td>1013</td>
<td>765</td>
<td>167102</td>
<td>141590</td>
</tr>
<tr>
<td>75.36413</td>
<td>327292</td>
<td>68419</td>
<td>1270</td>
<td>1026</td>
<td>745</td>
<td>182689</td>
<td>148640</td>
</tr>
<tr>
<td>75.40398</td>
<td>295107</td>
<td>57094</td>
<td>1221</td>
<td>1007</td>
<td>754</td>
<td>163714</td>
<td>131593</td>
</tr>
<tr>
<td>75.57749</td>
<td>302652</td>
<td>60929</td>
<td>1214</td>
<td>1055</td>
<td>736</td>
<td>168349</td>
<td>134103</td>
</tr>
<tr>
<td>75.54437</td>
<td>284879</td>
<td>54930</td>
<td>1161</td>
<td>1013</td>
<td>767</td>
<td>156181</td>
<td>128698</td>
</tr>
<tr>
<td>75.45563</td>
<td>305992</td>
<td>59016</td>
<td>1256</td>
<td>923</td>
<td>747</td>
<td>160774</td>
<td>139818</td>
</tr>
<tr>
<td>75.54163</td>
<td>313921</td>
<td>59257</td>
<td>1208</td>
<td>1073</td>
<td>740</td>
<td>172763</td>
<td>141158</td>
</tr>
<tr>
<td>75.54153</td>
<td>321663</td>
<td>62531</td>
<td>1219</td>
<td>1083</td>
<td>751</td>
<td>171425</td>
<td>150238</td>
</tr>
<tr>
<td>75.42279</td>
<td>312242</td>
<td>62520</td>
<td>1231</td>
<td>1046</td>
<td>761</td>
<td>175611</td>
<td>138631</td>
</tr>
<tr>
<td>75.44263</td>
<td>296191</td>
<td>58724</td>
<td>1210</td>
<td>963</td>
<td>735</td>
<td>172120</td>
<td>124071</td>
</tr>
<tr>
<td>75.51606</td>
<td>316638</td>
<td>62227</td>
<td>1261</td>
<td>1041</td>
<td>770</td>
<td>179252</td>
<td>137146</td>
</tr>
<tr>
<td>75.44244</td>
<td>326823</td>
<td>61229</td>
<td>1255</td>
<td>1099</td>
<td>739</td>
<td>175525</td>
<td>151298</td>
</tr>
<tr>
<td>75.52444</td>
<td>277730</td>
<td>51788</td>
<td>1173</td>
<td>921</td>
<td>674</td>
<td>156428</td>
<td>121302</td>
</tr>
<tr>
<td>75.44522</td>
<td>317829</td>
<td>67614</td>
<td>1258</td>
<td>1045</td>
<td>772</td>
<td>178178</td>
<td>139651</td>
</tr>
<tr>
<td>75.42102</td>
<td>368837</td>
<td>79057</td>
<td>1246</td>
<td>1095</td>
<td>806</td>
<td>206340</td>
<td>162197</td>
</tr>
</tbody>
</table>
Figure A.17: Submitted order prices

Figure A.18: Submitted order volumes

Figure A.19: Submitted prices

Figure A.20: Traded Prices
Appendix B

Gant Chart
Appendix C

Project Plan

<table>
<thead>
<tr>
<th>ID</th>
<th>Milestone</th>
<th>Start Date</th>
<th>End Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initial Requirements Analysis and literature review</td>
<td>10/10/2011</td>
<td>31/10/2011</td>
</tr>
<tr>
<td>2</td>
<td>First Sprint</td>
<td>01/11/2011</td>
<td>27/11/2011</td>
</tr>
<tr>
<td>2.1</td>
<td>Analysis:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. Research on Volume-Weighted Average Price and Time-Weighted Average Price trading algorithms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>Design:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. Create a graphical representation of a deadlock</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Set up the architecture for representing a limit order book</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>Implementation:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. Simulate a deadlock situation between several agents</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Second Sprint</td>
<td>28/11/2011</td>
<td>17/01/2012</td>
</tr>
<tr>
<td>3.1</td>
<td>Analysis:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. Refine requirements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>Design:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. Set up a refined architecture to support the simulation of several stock markets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Implementation:</td>
<td>02/12/2011</td>
<td>04/12/2011</td>
</tr>
<tr>
<td>3.4</td>
<td>Testing &amp; Validation: find and fix errors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Statistical Analysis: Initial analysis reports</td>
<td>01/01/2012</td>
<td>17/01/2012</td>
</tr>
<tr>
<td>4</td>
<td>Interim Report</td>
<td>18/01/2012</td>
<td>25/01/2012</td>
</tr>
<tr>
<td>5</td>
<td>Third Sprint</td>
<td>26/01/2012</td>
<td>05/03/2012</td>
</tr>
<tr>
<td>5.1</td>
<td>Analysis:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. Refine requirements</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. (Optional) Research on genetic algorithms/ genetic programming</td>
<td>26/01/2012</td>
<td>01/02/2012</td>
</tr>
<tr>
<td>5.2</td>
<td>Design:</td>
<td>02/02/2012</td>
<td>04/02/2012</td>
</tr>
<tr>
<td>5.3</td>
<td>Implementation:</td>
<td>05/02/2012</td>
<td>19/02/2012</td>
</tr>
<tr>
<td>5.4</td>
<td>Testing &amp; Validation: find and fix errors</td>
<td>20/02/2012</td>
<td>25/02/2012</td>
</tr>
<tr>
<td>5.5</td>
<td>Statistical analysis: Analysis report II</td>
<td>26/02/2012</td>
<td>05/03/2012</td>
</tr>
<tr>
<td>6</td>
<td>Final Sprint</td>
<td>06/03/2012</td>
<td>23/03/2012</td>
</tr>
<tr>
<td>6.1</td>
<td>Analysis:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. Refine requirements</td>
<td>06/03/2012</td>
<td>07/03/2012</td>
</tr>
<tr>
<td>6.2</td>
<td>Design: Set up final architecture</td>
<td>08/03/2012</td>
<td>09/03/2012</td>
</tr>
<tr>
<td>6.3</td>
<td>Implementation:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. General code review</td>
<td>10/03/2012</td>
<td>19/03/2012</td>
</tr>
<tr>
<td></td>
<td>2. (Optional) Add intelligent behaviours based on genetic algorithms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Testing &amp; Validation: general debugging and refactoring</td>
<td>20/03/2012</td>
<td>21/03/2012</td>
</tr>
<tr>
<td>6.5</td>
<td>Statistical analysis: Final Analysis Report</td>
<td>22/03/2012</td>
<td>23/03/2012</td>
</tr>
<tr>
<td>7</td>
<td>Final Report</td>
<td>24/03/2012</td>
<td>24/04/2012</td>
</tr>
</tbody>
</table>
Appendix D

System Manual and Contents of CD

Computer System Requirements:
- Windows XP / Vista / 7
- 1 GB RAM Minimum, 2 GB RAM recommended
- Hard Disk: 300 MB

Software Requirements:
- JAVA SE
- Java Development Kit (JDK) version 5, 6 or 7

The CD contains the following:
- src folder containing the source code files
- build folder containing the .class files
- dist folder containing an executable .jar file to run the program.
- Excel files containing:
  - the list of initial values used to populate the limit order book
  - a file containing the results of experiments run with different parameters

Setup:
- Copy the initialValues.xls file into the current working directory.
Appendix E

User Manual

To run the system, the following steps need to be undertaken:

- Run the Agent-Based-Model.jar file, located in the dist folder on the CD.
- Click Agent Platforms - Right click on Main Container - Start New Agent

- On agent name, write a generic name: Eg. Agent Starter.
• On class name, search in class path and select `abms.SimulationAgent`. Click **OK**. Leave the **Arguments** and **Owner** sections blank.

• The window will disappear after 400 seconds.

• The system will create two files: `history.csv` - contains the details of trades and `log.csv` - contains the details of submitted orders.

• The files can be then viewed and analysed using a statistical tool such as Excel or R.
Appendix F

Test Reports

F.1 CPU Usage during Simulation

F.2 Messages Exchanged during Simulation
Appendix G

Code Listing

G.1 Limit Order Book

package abms;

import cern.colt.list.DoubleArrayList;
import cern.jet.stat.Descriptive;
import java.util.Queue;
import java.util.concurrent.PriorityBlockingQueue;

/**
 * Title: Limit Order Book
 * Description: This class implements a limit order book.
 * Version: 1.5
 * Last date modified: 28/02/2012
 * @author Andrei
 */

public class LimitOrderBook {

    private final PriorityBlockingQueue<Order> buyOrders;
    private final PriorityBlockingQueue<Order> sellOrders;
    private final DoubleArrayList executions;

    public LimitOrderBook(){
        this(new PriorityBlockingQueue<Order>(), new PriorityBlockingQueue<Order>,
    }

    /**
     * A constructor that instantiates a new LimitOrderBook object
     * @param buyOrders – PriorityBlockingQueue<Order> of buy order
     * @param sellOrders – PriorityBlockingQueue<Order> of sell orders
     * @param executions – DoubleArrayList containing the latest traded prices
     */

    public LimitOrderBook(final Queue<Order> buyOrders, 
                            final Queue<Order> sellOrders, final DoubleArrayList executions, 

```java
this.buyOrders = new PriorityBlockingQueue<Order>(buyOrders);
this.sellOrders = new PriorityBlockingQueue<Order>(sellOrders);
this.executions = new DoubleArrayList();
```

```java
/**
 * This method will add a new Order object to the relevant Priority Queue in
 * the limit order book
 * @param order – Order object created by the Market Agent
 * @return ok – shows whether the order has been successfully added to the
 * order book
 */
public boolean addOrder(Order order){
    boolean ok = true;
    if (order.getSide().equalsIgnoreCase("BUY")){
        buyOrders.add(order);
    }
    else if (order.getSide().equalsIgnoreCase("SELL")){
        sellOrders.add(order);
    }
    else ok = false;
    return ok;
}
```

```java
/**
 * Gets the array list of traded price
 * @return DoubleArrayList executions
 */
public DoubleArrayList getExecutions(){
    return executions;
}
```

```java
/**
 * Gets the mean of all traded price
 * @return Descriptive.mean(executions)
 */
public double getExecutionsMean(){
    return Descriptive.mean(executions);
}
```

```java
/**
 * Gets the last traded price
 * @return Double price
 */
public double getLastExecution(){
```
```java
    return executions.get(executions.size() - 1);
}

/**
 * Gets the buy orders
 * @return buyOrder
 */
public PriorityBlockingQueue<Order> getBuyOrders() {
    return buyOrders;
}

/**
 * Gets the sell orders
 * @return
 */
public PriorityBlockingQueue<Order> getSellOrders() {
    return sellOrders;
}

/**
 * Gets the Volume-Weighted Average Price on the buy side of the order book
 * @return Double vWap
 */
public double getVwapBuySide() {
    double vWap = 0;
    double volume = 0;
    Object[] orders = buyOrders.toArray();
    for (int i = 0; i <= (buyOrders.size() - 1) / 10; i++) {
        Order order = (Order) orders[i];
        vWap += order.getPrice() * order.getStatus().getVolumeLeft();
        volume += order.getStatus().getVolumeLeft();
    }
    vWap = vWap / volume;
    return vWap;
}

/**
 * Gets the Volume-Weighted Average Price on the sell side of the order book
 * @return Double vWap
 */
public double getVwapSellSide() {
    double vWap = 0;
```
double volume = 0;
Object[] orders = sellOrders.toArray();

for (int i = 0; i <= (sellOrders.size() - 1)/10; i++){
    Order order = (Order) orders[i];
    vWap += order.getPrice() * order.getStatus().getVolumeLeft();
    volume += order.getStatus().getVolumeLeft();
}

vWap = vWap/volume;

return vWap;
}

G.2 Order

package abms;
import jade.core.AID;

/** Title: Order
 * Description: This class implements an order.
 * Version: 1.4
 * Last date modified: 28/03/2012
 * @author Andrei
 */

class Order implements Comparable<Order>{
    private AID traderID;
    private String orderID;
    private String side;
    private String orderType;
    private double price;
    private long volume;
    private long entryTime;
    private OrderStatus status;

    /**
     * @return volume
     */
    public long getVolume(){
        return volume;
    }

    /**
     * @return order ID
     */

}
public String getOrderID(){
    return orderID;
}

/**
 * Get the trader’s ID, i.e. the local AID
 * @return traderID
 */
public AID getTraderID(){
    return traderID;
}

/**
 * Sets the trader ID of the order to @param traderID
 */
public void setTraderID(AID traderID){
    this.traderID = traderID;
}

/**
 * Sets the order ID to @param orderID
 */
public void setOrderID(String orderID){
    this.orderID = orderID;
}

/**
 * Sets the number of shares in the volume to @param volume
 */
public void setVolume(long volume){
    this.volume = volume;
}

/**
 * Gets the type of order, which can be LIMIT OR MARKET
 * @return orderType
 */
public String getOrderType(){
    return orderType;
}

/**
 * Sets the order type to @param orderType
 */
public void setOrderType(String orderType){
    this.orderType = orderType;
}

/**
 * Gets the price of the order
 * @return price

/**
 * public double getPrice() {
 *   return price;
 * }
/**
 * Sets the price of the order to @param price
 */
public void setPrice(double price) {
   this.price = price;
}
/**
 * Gets the side of the order, which can be BUY or SELL
 * @return side
 */
public String getSide() {
   return side;
}
/**
 * Sets the buy/sell side of the order to @param side
 */
public void setSide(String side) {
   this.side = side;
}
/**
 *Gets the EntryTime of the order which is System.currentTimeMillis()
 * @return entryTime
 */
public long getEntryTime() {
   return entryTime;
}
/**
 * Sets the entryTime of the order to @param entryTime
 */
public void setEntryTime(Long entryTime) {
   this.entryTime = entryTime;
}
/**
 * Gets the current status of the order
 * @return status
 */
public OrderStatus getStatus() {
   return status;
}
/**
 * Sets a new @param status of the order
 */
public void setStatus(OrderStatus status){
  this.status = status;
}

/**
 * Executes an order for a price of @param price
 * and a volume of @param volume
 * This method will subtract a certain number of shares from the initial
 * order at the execution price.
 * If the order has no shares left, its status is changed to Filled.
 */
public void execute(double price, long volume){
    long volLeft = getStatus().getVolumeLeft();
    status.setVolumeLeft(volLeft - volume);
    status.setVolumeExecuted(volume);
    status.setLastExecutedPrice(price);
    if (status.getVolumeLeft() == 0){
        status.setStage("Filled");
    }
    else{
        status.setStage("Partially filled");
    }
}

/**
 * Specifies whether an order has any shares left
 */
public boolean isFilled(){
    return this.getStatus().getVolumeLeft() == 0;
}

/**
 * The comparable iterator that places orders in the priority queue
 * @param o
 * @return 1,0,-1
 */
@Override
public int compareTo(Order o) {
    if (this == o) return 0;

    // Market Order has higher precedence than limit order
    if (getOrderType().equalsIgnoreCase("Limit") && o.getOrderType().equalsIgnoreCase("Market")
        if (getOrderType().equalsIgnoreCase("Market") && o.getOrderType().equalsIgnoreCase("Limit")

    // Earlier entry time for market orders has higher precedence

74
if (getOrderType().equalsIgnoreCase("Market") && o.getOrderType().equalsIgnoreCase("Market")) {
    if (getEntryTime() < o.getEntryTime()){
        return -1;
    }
    else if (getEntryTime() > o.getEntryTime()){
        return 1;
    }
    else return 0;
}

// For buy limit orders, the highest bid is on top of the book
if (getPrice() < o.getPrice()){
    if (getSide().equalsIgnoreCase("Buy")) return 1;
    else if (getSide().equalsIgnoreCase("Sell")) return -1;
}

// For sell limit orders, the lowest offer is on top of the book
if (getPrice() > o.getPrice()){
    if (getSide().equalsIgnoreCase("Buy")) return -1;
    else if (getSide().equalsIgnoreCase("Sell")) return 1;
}

return getEntryTime() < o.getEntryTime() ? -1 : (getEntryTime() > o.getEntryTime() ? 1 : 0);


G.3 Market Agent

Setup

package abms;

import jade.content.lang.sl.SLCodec;
import jade.core.Agent;
import jade.core.behaviours.OntologyServer;
import jade.core.behaviours.ThreadedBehaviourFactory;
import jade.domain.FIPANames.ContentLanguage;
import jade.lang.acl.ACLMessage;
import java.util.ArrayList;

/** Title : Trader Agent
 * Description: This class implements a market agent. The market agent will get
 * order request from the trader agent and process it through the Brokerage Service.
 * The market agent will also have a Feed Behaviour, which sends the latest
 * market updates to the currently registered agents.
 * Version: 1.3
 * Last date modified: 02/03/2012
 * @author Andrei
 */
public class MarketAgent extends Agent {
    private final LimitOrderBook limitOrderBook;

    public MarketAgent () {
        limitOrderBook = new LimitOrderBook();
    }

    @Override
    protected void setup () {
        System.out.println("Market-Agent. " + getAID().getName() + " has been created");

        // Register language and ontology
        getContentManager().registerLanguage(new SLCodec(), ContentLanguage.FIPA);
        getContentManager().registerOntology(MarketOntology.getInstance());

        // Set the main behaviour of the Market Agent
        final ThreadedBehaviourFactory factory = new ThreadedBehaviourFactory();
        addBehaviour(new FeedBehaviour(this, limitOrderBook));
        addBehaviour(factory.wrap(new OntologyServer(this, MarketOntology.getInstance(),
            ACLMessage.REQUEST, new Brokerage(this, limitOrderBook, new Matcher(), new OrderLogger(),
                new ExecutionLogger(new ArrayList<>()));
    }
}

FeedBehaviour

package abms;

import jade.content.onto.basic.Action;
import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
import jade.domain.AMSService;
import jade.domain.FIPAAgentManagement.AMSAgentDescription;
import jade.domain.FIPAAgentManagement.SearchConstraints;
import jade.domain.FIPANames.ContentLanguage;
import jade.lang.acl.ACLMessage;

/** Title : Feed Behaviour  
 * Description: This class creates a SendMarketStatus Object to be sent to all 
 * the registered noise agents on the platform 
 * Version: 1.4 
 * Last date modified: 29/03/2012 
 * @author Andrei

76
public class FeedBehaviour extends CyclicBehaviour {

    private final String language = ContentLanguage.FIPA_SL;
    private final LimitOrderBook limitOrderBook;

    public FeedBehaviour(final Agent agent, final LimitOrderBook limitOrderBook) {
        super(agent);
        this.limitOrderBook = limitOrderBook;
    }

    @Override
    public void action() {
        SendMarketStatus sms = new SendMarketStatus();
        if (!limitOrderBook.getBuyOrders().isEmpty()) {
            sms.setBestBid(limitOrderBook.getBuyOrders().peek().getPrice());
            sms.setBestBidVolume(limitOrderBook.getBuyOrders().peek().getVolume());
        }
        if (!limitOrderBook.getSellOrders().isEmpty()) {
            sms.setBestOffer(limitOrderBook.getSellOrders().peek().getPrice());
            sms.setBestOfferVolume(limitOrderBook.getSellOrders().peek().getVolume());
        }
        if (!limitOrderBook.getExecutions().isEmpty()) {
            sms.setLastExecutionPrice(limitOrderBook.getLastExecution());
        }
        if (!limitOrderBook.getBuyOrders().isEmpty()) {
            sms.setBuySideVwap(limitOrderBook.getVwapBuySide());
        }
        if (!limitOrderBook.getSellOrders().isEmpty()) {
            sms.setSellSideVwap(limitOrderBook.getVwapSellSide());
        }
        sendMarketFeed(sms);
    }

    /**
     * Sends the SendMarketStatus object to the registered agents
     * @param sms - SendMarketStatus object
     */
    public void sendMarketFeed(SendMarketStatus sms) {
        try {
            // Create a new array for storing the current active agents
        }
    }
}
AMSAgentDescription [] agents = null;
SearchConstraints c = new SearchConstraints();
c.setMaxResults ( new Long(-1) );
agents = AMSService.search( myAgent, new AMSAgentDescription (), c )

final Action action = new Action(myAgent.getAID(), sms);
ACLMessage statusMessage = new ACLMessage(ACLMessage.INFORM);
statusMessage.setOntology(MarketOntology.getInstance().getName());
statusMessage.setLanguage(language);

for (int i = 0; i < agents.length; i++){
    statusMessage.addReceiver(agents[i].getName());
}

myAgent.getContentManager().fillContent(statusMessage, action);
myAgent.send(statusMessage);

} catch (Exception ex){}

}

Brokerage

package abms;

import jade.content.lang.Codec.CodecException;
import jade.content.onto.OntologyException;
import jade.content.onto.basic.Action;
import jade.domain.FIPANames.ContentLanguage;
import jade.lang.acl.ACLMessage;
import java.util.logging.Level;
import java.util.logging.Logger;

/** Title : Brokerage
 * Description: This class will contain a broker agent and an instance of the
 * The broker agent receives a "REQUEST" message (i.e, an order) from a trader
 * to place that order on the limit order book and send a report back to the agent.
 * on top of the order book are matched and executed.
 * Version: 1.4
 * Last date modified: 20/03/2012
 * @author Andrei
 */

public class Brokerage{

    private final MarketAgent marketAgent;
    private final LimitOrderBook limitOrderBook;
    private final OrderLogger orderLogger;
    private final ExecutionLogger executionLogger;
}
private static long orderIDCounter = 0;
private final String language = ContentLanguage.FIPA_SL;
private final Matcher matcher;

/**
 * This constructor initiates a new instance of a Brokerage
 * @param marketAgent
 * @param limitOrderBook
 * @param matcher is the instance of a class used for matching two orders at the top of the book
 * @param orderLogger is a .csv file which records all the submitted {@link Orders}
 * @param executionLogger
 */
public Brokerage(final MarketAgent marketAgent, final LimitOrderBook limitOrderBook, final Matcher matcher, final OrderLogger orderLogger, final ExecutionLogger executionLogger) {
    this.marketAgent = marketAgent;
    this.limitOrderBook = limitOrderBook;
    this.matcher = matcher;
    this.orderLogger = orderLogger;
    this.executionLogger = executionLogger;
    orderLogger.createOrderLogger();
    executionLogger.createExecutionLogger();
}

/**
 * This method searches for the order ID in the limit order book and removes it
 * @param cancelOrder - object containing the Order Id and the Order Side
 * @param request - ACL Message matching the REQUEST Performative
 */
public void serveCancelOrderRequest(final CancelOrder cancelOrder, final ACLMessage request) {
    //Get the orderID to be cancelled and the order side
    String cancelID = cancelOrder.getOrderID();
    String side = cancelOrder.getSide();

    //Invoke the cancel method with the values obtained above
    cancelOrder(cancelID, side);
}

public void cancelOrder(String cancelID, String side) {
    if (side.equalsIgnoreCase("BUY")) {
        Object[] orders = limitOrderBook.getBuyOrders().toArray();
        for (Object order : orders) {
            Order o = (Order) order;
            if (o.getOrderID().equalsIgnoreCase(cancelID)) {
                try {
                    limitOrderBook.getBuyOrders().remove(o);
                } catch (Exception ex) { System.err.println(“Could not cancel the order!”); }
            }
        }
    }
}

79
else{
    Object[] orders = limitOrderBook.getSellOrders().toArray();
    for (Object order : orders){
        Order o = (Order) order;
        if (o.getOrderID().equalsIgnoreCase(cancelID)){
            try{
                limitOrderBook.getSellOrders().remove(o);
            } catch (Exception ex) {System.err.println("Could not cancel the order!");
        }
    }
}

/**
 * This method validates and adds orders to the limit order book
 * @param sentOrder – an object containing the side, price, type and volume
 * of the submitted order
 * @param request – ACL Message matching the REQUEST Performative
 */

public void serveSendOrderRequest(final SendOrder sentOrder, final ACLMessage request){
    // Create new instance of Order
    Order order = new Order();

    // Set entry time
    order.setEntryTime(System.currentTimeMillis());

    // Set a public ID for the order
    orderIDCounter++;
    order.setOrderID(Long.toString(orderIDCounter));

    // Set the trader ID for the order
    order.setTraderID(request.getSender());

    // Set the type of the order: Market or Limit
    if (sentOrder.getOrderType() != null)
        order.setOrderType(sentOrder.getOrderType());

    // Set the price of the order
    if (sentOrder.getPrice() > 0){
        order.setPrice(sentOrder.getPrice());
    }

    // Set the side: BUY or SELL
    if (sentOrder.getSide() != null)
        order.setSide(sentOrder.getSide());

    // Set the initial status parameters of the order;

}
String stage = "NEW";
long initialVolume = sentOrder.getVolume();
long volumeExecuted = 0;
long volumeLeft = initialVolume;
double lastExecutedPrice = 0;
OrderStatus newStatus = new OrderStatus(stage, initialVolume, volumeExecuted, volumeLeft, lastExecutedPrice);

// Assign the status to the new order
order.setStatus(newStatus);

// Set the volume of the new order
order.setVolume(sentOrder.getVolume());

// Try to place the order on the Limit Order Book
if (order != null){
    boolean wasAccepted = limitOrderBook.addOrder(order);
    if (wasAccepted){
        orderLogger.addOrder(order);
        sendOrderConfirmation(order, request);

        executeOrders();
    }
}
else {
    order.getStatus().setStage("Canceled");
    sendOrderConfirmation(order, request);
}
}

/**
 * This method sends a confirmation to the agent that submitted an order
 * @param order — newly created Order object, which was placed on the limit book
 * @param request — ACL Message matching the REQUEST Performative
 */

private void sendOrderConfirmation (final Order order, ACLMessage request){

    final SendOrderConfirmation confirmation = new SendOrderConfirmation();
    confirmation.setOrderID(order.getOrderID());
    confirmation.setOrderType(order.getOrderType());
    confirmation.setSide(order.getSide());
    confirmation.setPrice(order.getPrice());
    confirmation.setVolume(order.getVolume());
    confirmation.setOrderStatus(order.getStatus());

    try{
        
    }catch (RemoteException ex) {
        Logger.getLogger(OrderBook.class.getName()).log(Level.SEVERE, null, ex);
    }
}
final Action action = new Action(request.getSender(), confirmation);
final ACLMessage replyMessage = request.createReply();

if (order.getStatus().getStage().equalsIgnoreCase("CANCELED")){
    replyMessage.setPerformative(ACLMessage.REJECT);
}
else{
    replyMessage.setPerformative(ACLMessage.CONFIRM);
}

replyMessage.setPerformative(ACLMessage.CONFIRM);
replyMessage.setOntology(MarketOntology.getInstance().getName());
replyMessage.setLanguage(language);
marketAgent.getContentManager().fillContent(replyMessage, action);
marketAgent.send(replyMessage);

} catch (Exception ex){}

/**
 * This method polls the Order objects on top of the limit order book and
 * checks whether they match or not
 */

public void executeOrders(){

if (!(limitOrderBook.getBuyOrders().isEmpty()) && !limitOrderBook.getSellOrders().isEmpty()){
    final Order buyOrder = limitOrderBook.getBuyOrders().peek();
    final Order sellOrder = limitOrderBook.getSellOrders().peek();
    final double executionPrice = matcher.areMatchedAtPrice(buyOrder, sellOrder);

    if (!executionPrice == -1)){
        executionLogger.addExecutionPrice(executionPrice);
        execute(buyOrder, sellOrder, executionPrice);
    }
    else{
        if (buyOrder.getOrderType().equalsIgnoreCase("Market")){
            limitOrderBook.getBuyOrders().remove(buyOrder);
        }
        if (sellOrder.getOrderType().equalsIgnoreCase("Market")){
            limitOrderBook.getSellOrders().remove(sellOrder);
        }
    }
}
}
public void execute(final Order buyOrder, final Order sellOrder, final Double price)
{
    final long volume;
    if (buyOrder.getStatus().getVolumeLeft() >= sellOrder.getStatus().getVolumeLeft())
    {
        volume = sellOrder.getStatus().getVolumeLeft();
    }
    else{
        volume = buyOrder.getStatus().getVolumeLeft();
    }
    buyOrder.execute(price, volume);
    sellOrder.execute(price, volume);
    executionLogger.addOrder(buyOrder);
    executionLogger.addOrder(sellOrder);
    limitOrderBook.getExecutions().add(price);
    checkIfFilled(buyOrder, sellOrder);

    try {
        sendOrderReports(buyOrder, sellOrder);
    } catch (CodecException ex) {
        Logger.getLogger(Brokerage.class.getName()).log(Level.SEVERE, null,
    } catch (OntologyException ex) {
        Logger.getLogger(Brokerage.class.getName()).log(Level.SEVERE, null,
    }
}

public void checkIfFilled(final Order buyOrder, final Order sellOrder){
    if (buyOrder.getStatus().getVolumeLeft() == 0 || buyOrder.getOrderType() == 0 ||
        removeOrder(buyOrder);
    }
    if (sellOrder.getStatus().getVolumeLeft() == 0 || sellOrder.getOrderType() == 0 ||

removeOrder(sellOrder);
}

/**
 * This method removes an order from the limit order book
 * @param order – Order object to be deleted from the order book
 */

public void removeOrder(Order order){
    if (order.getSide().equalsIgnoreCase("BUY")){
        if (!limitOrderBook.getBuyOrders().isEmpty()){  
            try{
                limitOrderBook.getBuyOrders().remove(order);

            } catch (Exception ex){}
        }
    }

    else if (order.getSide().equalsIgnoreCase("SELL")){
        if (!limitOrderBook.getSellOrders().isEmpty()){  
            try{
                limitOrderBook.getSellOrders().remove(order);

            } catch (Exception ex){}
        }
    }
}

/**
 * This method sends the execution reports to the agents involved in the trade
 * @param buyOrder – Order object on the buy side of the order book
 * @param sellOrder – Order object on the sell side of the order book
 * @throws jade.content.lang.Codec.CodecException
 * @throws OntologyException
 */

public void sendOrderReports(final Order buyOrder, final Order sellOrder) throws CodecException, OntologyException
{

    // Create the report to be sent to the trader who placed the buy order.
    final SendOrderReport reportToBuyTrader = new SendOrderReport();

    // Create the report to be sent to the trader who placed the sell order.
    final SendOrderReport reportToSellTrader = new SendOrderReport();

    // Check whether the buy order has been filled.
    if (buyOrder.isFilled()){  
        reportToBuyTrader.setCurrentStage("Filled");

    }
}
else {
    reportToBuyTrader.setCurrentStage("Partially Filled");
}

// Check whether the sell order has been filled
if (sellOrder.isFilled()){
    reportToSellTrader.setCurrentStage("Filled");
} else{
    reportToSellTrader.setCurrentStage("Partially Filled");
}

// Set the orders side
reportToBuyTrader.setOrderSide(buyOrder.getSide());
reportToSellTrader.setOrderSide(sellOrder.getSide());

// Set the unique order ID to be sent to the traders
reportToBuyTrader.setUniqueOrderID(buyOrder.getOrderID());
reportToSellTrader.setUniqueOrderID(sellOrder.getOrderID());

// Set the last executed volume of the order and volume left
reportToBuyTrader.setVolumeExecuted(buyOrder.getStatus().getVolumeExecuted());
reportToSellTrader.setVolumeExecuted(sellOrder.getStatus().getVolumeExecuted());
reportToBuyTrader.setVolumeLeft(buyOrder.getStatus().getVolumeLeft());
reportToSellTrader.setVolumeLeft(sellOrder.getStatus().getVolumeLeft());

// Set the last prices which the orders were executed at + transaction Costs
double executionCost = buyOrder.getStatus().getLastExecutedPrice() + (2 * buyOrder.getStatus().getLastExecutedPrice()) / 100;
reportToBuyTrader.setPriceExecutedAt(buyOrder.getStatus().getLastExecutedPrice() + executionCost);
reportToSellTrader.setPriceExecutedAt(sellOrder.getStatus().getLastExecutedPrice() + executionCost);

// Send the reports
try{
    final Action sendToBuyTraderAction = new Action(marketAgent.getAID(), reportToBuyTrader);
    final Action sendToSellTraderAction = new Action(marketAgent.getAID(), reportToSellTrader);

    final ACLMessage replyToBuyTrader = new ACLMessage(ACLMessage.CFP);
    final ACLMessage replyToSellTrader = new ACLMessage(ACLMessage.CFP);

    replyToBuyTrader.addReceiver(buyOrder.getTraderID());
    replyToSellTrader.addReceiver(sellOrder.getTraderID());

    replyToBuyTrader.setOntology(MarketOntology.getInstance().getName());
    replyToSellTrader.setOntology(MarketOntology.getInstance().getName());

    replyToBuyTrader.setLanguage(language);
    replyToSellTrader.setLanguage(language);

    marketAgent.getContentManager().fillContent(replyToBuyTrader, sendToBuyTraderAction);
    marketAgent.getContentManager().fillContent(replyToSellTrader, sendToSellTraderAction);
}
marketAgent.send(replyToBuyTrader);

marketAgent.getContentManager().fillContent(replyToSellTrader, sendToSellTraderAction);
}
}

catch (Exception ex){}

}

Matcher

package abms;

/**
 * Title : Matcher
 * Description: This class matches two orders from the top of the order book
 * Version: 1.4
 * Last date modified: 28/02/2012
 * @author Andrei
 */

public class Matcher {

    /**
     * This method matches orders on the top of the limit order book according to
     * price, entry time, and order type
     * @param buyOrder – Order object on the buy side of the order book
     * @param sellOrder – Order object on the sell side of the order book
     * @return Double price
     */
    public double areMatchedAtPrice(Order buyOrder, Order sellOrder) {

        // Try to match if both orders are limit orders
        if (buyOrder.getOrderType().equalsIgnoreCase("Limit") &&
            sellOrder.getOrderType().equalsIgnoreCase("Limit")){

            if ((buyOrder.getPrice() >= sellOrder.getPrice())){
                if (buyOrder.getEntryTime() >= sellOrder.getEntryTime()){
                    return buyOrder.getPrice();
                }
            } else if (buyOrder.getEntryTime() < sellOrder.getEntryTime()){
                return sellOrder.getPrice();
            }
        }

        else if (buyOrder.getPrice() < sellOrder.getPrice()){
            return -1;
        }

    }

}
// Try to match if one order is a market order and the other one is a limit order
else{

// BID AGAINST SELL
if (buyOrder.getOrderType().equalsIgnoreCase("Limit") &&
    sellOrder.getOrderType().equalsIgnoreCase("Market")){

    return buyOrder.getPrice();
}

// OFFER AGAINST BUY
else if (buyOrder.getOrderType().equalsIgnoreCase("Market") &&
    sellOrder.getOrderType().equalsIgnoreCase("Limit")){

    return sellOrder.getPrice();
}
}

// If no match, return -1
return -1;
}

G.4 Noise Agent

Setup

package abms;

import jade.content.lang.sl.SLCCodec;
import jade.core.Agent;
import jade.domain.FIPANames.ContentLanguage;
import java.util.Random;

/** Title: Trader Agent
 * Description: This class implements a trader agent that uses the NoiseBehaviour
 * for submitting an order.
 * Version: 1.4
 * Last date modified: 01/04/2012
 * @author Andrei
 */

public class NoiseAgent extends Agent{

    private final MarketStatus marketStatus;
    private final Portfolio portfolio;
private final OrderReport orderReport;
private final StandardInventory inventory;
private long shares;
private double cash;

public NoiseAgent(){
    marketStatus = new MarketStatus();
    portfolio = new Portfolio();
    orderReport = new OrderReport();
    inventory = new StandardInventory(shares, cash);
}

@Override
protected void setup(){
    System.out.println("NoiseAgent"+getAID().getName()+"has been created");

    // Register language and ontology
    getContentManager().registerLanguage(new SLCodec(), ContentLanguage.FIPA);
    getContentManager().registerOntology(MarketOntology.getInstance());

    // Set the main behaviour of the Noise Trader
    addBehaviour(new NoiseBehaviour(this, new Random(), marketStatus, portfolio, orderReport, inventory));
    addBehaviour(new UpdateBehaviour(this, marketStatus, orderReport, portfolio));
}

Noise Behaviour

package abms;

import jade.content.onto.basic.Action;
import jade.core.AID;
import jade.core.Agent;
import jade.core.behaviours.TickerBehaviour;
import jade.domain.FIPANames.ContentLanguage;
import jade.lang.acl.ACLMessage;
import java.text.DecimalFormat;
import java.util.Random;
import org.apache.commons.math3.distribution.LogNormalDistribution;
import org.apache.commons.math3.distribution.UniformRealDistribution;
import org.apache.commons.math3.distribution.ExponentialDistribution;

/** Title: NoiseBehaviour */
public class NoiseBehaviour extends TickerBehaviour{

    private final String language = ContentLanguage.FIPA_SL;
    private static final int SLEEP_PERIOD = 2000;
    private final Random randomGenerator;
    private final MarketStatus marketStatus;
    private final OrderReport orderReport;
    private final Portfolio portfolio;
    private final StandardInventory inventory;
    private final LogNormalDistribution volumeRange = new LogNormalDistribution(4.5, 0.8);
    private final ExponentialDistribution priceRangeOutOfSpread = new ExponentialDistribution(0.3);

    public NoiseBehaviour(final Agent agent, final Random randomGenerator, final MarketStatus marketStatus, final Portfolio portfolio, final OrderReport orderReport, final StandardInventory inventory)
    {
        super(agent, randomGenerator.nextInt(SLEEP_PERIOD)+1);
        this.randomGenerator = randomGenerator;
        this.marketStatus = marketStatus;
        this.orderReport = orderReport;
        this.portfolio = portfolio;
        this.inventory = inventory;
        inventory.setShares(1000);
        inventory.setCash(100000);
    }

    private double getTwoDecimal(double number)
    {
        DecimalFormat df = new DecimalFormat("#.##");
        return Double.parseDouble(df.format(number));
    }

    /**
     * This constructor instantiates a new NoiseBehaviour object
     * @param agent - JADE Agent to which the behaviour is added
     * @param randomGenerator - a Generator used for Normal Distributions
     * @param marketStatus - MarketStatus object containing the market updates
     * @param portfolio - Portfolio object containing confirmed orders
     * @param orderReport - Order object containing reports of executed orders
     * @param inventory - Inventory object containing information about the current status of cash and shares owned by the agent
     */
}
* This method will receive the last market status containing the best bid and the best offer.

```java
private void getMessage()
{
    while (!orderReport.getOrderReports().isEmpty()){
        try{
            updateInventory(orderReport.getOrderReports().get(orderReport.getOrderReports().size() - 1));
            updateInventory(orderReport.getOrderReports().remove(orderReport.getOrderReports().size() - 1));
        } catch (Exception ex){}
    }
    if (!marketStatus.getUpdates().isEmpty()){
       .setAction();
    }
}
```

/**
 * This method sets the type of action to be performed by the agent: cancellation order, limit order or market order
 */

```java
private void setAction()
{
    SendMarketStatus currentStatus = !marketStatus.getUpdates().isEmpty() ? marketStatus.getUpdates().get(marketStatus.getUpdates().size() - 1) : null;
    int actionProbability = randomGenerator.nextInt(100);
    // There is a 50% probability the noise trader will submit a cancel order
    if ((actionProbability >= 0) && (actionProbability < 50)){
        sendCancelOrders();
    }
    // There is a 15% probability the noise trader will submit a market order
    else if ((actionProbability >= 50) && (actionProbability < 65)){
        setMarketOrder(currentStatus);
    }
    // There is a 35% probability the noise trader will submit a limit order
    else if ((actionProbability >= 65) && (actionProbability < 100)){
        setLimitOrder(currentStatus);
    }
}
```

/**
 * This method will set a new order after taking into consideration the @param currentStatus, which gives the current best bid and best offer
 */

```java
private void setLimitOrder(SendMarketStatus currentStatus) {
```
double bestBid = currentStatus.getBestBid();
double bestOffer = currentStatus.getBestOffer();
double lastExecutionPrice = currentStatus.getLastExecutionPrice();

// If there are no bids, take the last executed price as a reference
if (bestBid == 0){
    if (lastExecutionPrice != 0){
        bestBid = lastExecutionPrice;
    } else{
        bestBid = 74.99;
    }
}

// If there are no offers, take the last executed price as a reference
if (bestOffer == 0){
    if (lastExecutionPrice != 0){
        bestOffer = lastExecutionPrice;
    } else{
        bestOffer = 75.00;
    }
}

final int inOrOutOfSpread = randomGenerator.nextInt(100);

// There is a 35% chance the order will be inside the spread
if (inOrOutOfSpread >= 0 && inOrOutOfSpread < 35){
    if (bestBid == 0 || bestOffer == 0){
        setLimitOrderAtBest(bestBid, bestOffer);
    } else{
        setLimitOrderInSpread(bestBid, bestOffer);
    }
}

// There is a 65% chance the order will be outside the spread
else if (inOrOutOfSpread >= 35 && inOrOutOfSpread < 100)
    setLimitOrderOutOfSpread(bestBid, bestOffer);

/**
 * This method will set a limit order at the current best price
 * @param bestBid – best price on the buy side of the order book
 * @param bestOffer – best price on the sell side of the order book
 */
private void setLimitOrderAtBest(double bestBid, double bestOffer){
SendOrder so = new SendOrder();
so.setOrderType("Limit");

// Draws a log normally distributed order volume from the volume range
long volume = (long) volumeRange.sample();
so.setVolume(volume);

final int sideChoice = randomGenerator.nextInt(2);

if (sideChoice == 0)
{
    so.setSide("BUY");
    so.setPrice(bestBid);
}
else
{
    so.setSide("SELL");
    so.setPrice(bestOffer);
}

sendOrderToMarket(so);

/**
 * This method will set a limit order outside of the spread
 * @param bestBid  -  best price on the buy side of the order book
 * @param bestOffer  -  best price on the sell side of the order book
 */
private void setLimitOrderOutOfSpread(double bestBid, double bestOffer) {

    SendOrder so = new SendOrder();
    so.setOrderType("Limit");

    // Draws a log normally distributed order volume from the volume range
    long volume = (long) volumeRange.sample();
    so.setVolume(volume);

    final int sideChoice = randomGenerator.nextInt(2);

    // There is a 50% probability the noise trader submits a buy order
    if (sideChoice == 0)
    {
        so.setSide("BUY");

        // Subtracts an exponentially distributed value from the best bid
        double price = getTwoDecimal(bestBid - getTwoDecimal(priceRangeOutOfSpread.sample()));
        so.setPrice(price);
    }

    // There is a 50% probability the noise trader submits a buy order
}
else{

    so.setSide("SELL");

    // Subtracts an exponentially distributed value from the best offer
    double price = getTwoDecimal(bestOffer + getTwoDecimal(priceRangeOutOfSpread.sample()));
    so.setPrice(price);
}

sendOrderToMarket(so);

}/**
 * This method sets a limit order inside the spread
 * @param bestBid - best price on the buy side of the order book
 * @param bestOffer - best price on the sell side of the order book
 */

private void setLimitOrderInSpread(double bestBid, double bestOffer){

    SendOrder so = new SendOrder();
    so.setOrderType("Limit");

    // Draws a log normally distributed value from the volume range
    long volume = (long) volumeRange.sample();
    so.setVolume(volume);

    final int sideChoice = randomGenerator.nextInt(2);

    // There is a 50% probability the noise trader submits a buy order
    if (sideChoice == 0){
        so.setSide("BUY");
        double spread = getTwoDecimal(bestOffer - bestBid);

        // If there is no spread, submit an order at the best bid
        if (spread <= 0){
            so.setPrice(getTwoDecimal(bestBid));
        }
    }

    else{

        /* Draws a uniformly distributed value between 0 and 1
           * This value is then multiplied by the spread and added to the
           * best bid.
         */

        UniformRealDistribution priceRangeInSpread = new UniformRealDistr
double price = getTwoDecimal(priceRangeInSpread.sample());
so.setPrice(price);
}

// There is a 50% probability the noise trader submits a sell order
else{
    so.setSide("SELL");
double spread = getTwoDecimal(bestOffer - bestBid);
    // If there is no spread, submit an order at the best offer
    if (spread <= 0){
        so.setPrice(getTwoDecimal(bestOffer));
    }
    else{
        /* Draws a uniformly distributed value between 0 and 1
        * This value is then multiplied by the spread and subtracted
        * from the best offer.
        */
        UniformRealDistribution priceRangeInSpread = new UniformRealDistribution(bestBid, bestOffer);
        double price = getTwoDecimal(priceRangeInSpread.sample());
        so.setPrice(price);
    }
}

sendOrderToMarket(so);
}

/**
 * This method sets a new Market order
 * @param currentStatus - SendMarketStatus object
 */
private void setMarketOrder(SendMarketStatus currentStatus){
    SendOrder so = new SendOrder();
    so.setOrderType("Market");

    final double lastExecutedPrice = currentStatus.getLastExecutionPrice();
    final double currentShares = balanceInventory(inventory.getCash(), inventory.getShares());
    int sideChoice = randomGenerator.nextInt(2);
if (currentShares > inventory.getShares()) {
    sideChoice = 0;
}
else {
    sideChoice = 1;
}

// 0 = Order is on the buy side
if (sideChoice == 0) {
    so.setSide("BUY");
    if (currentStatus.getBestOffer() != 0) {
        so.setPrice(currentStatus.getBestOffer());
        so.setVolume(currentStatus.getBestOfferVolume());
        sendOrderToMarket(so);
    }
}

// 1 = Order is on the sell side
else if (sideChoice == 1) {
    so.setSide("SELL");
    if (currentStatus.getBestBid() != 0) {
        so.setPrice(currentStatus.getBestBid());
        so.setVolume(currentStatus.getBestBidVolume());
        sendOrderToMarket(so);
    }
}

/**
 * Sends a new @param sendOrder to the Market Agent
 * *
 */
private void sendOrderToMarket(SendOrder sendOrder) {
    try {
        final Action action = new Action(new AID("market", AID.ISLOCALNAME),
                                          ACLMessage.REQUEST);
        final ACLMessage request = new ACLMessage(ALMessage.REQUEST);
        request.addReceiver(new AID("market", AID.ISLOCALNAME));
        request.setOntology(MarketOntology.getInstance().getName());
        request.setLanguage(language);
        myAgent.getContentManager().fillContent(request, action);
        myAgent.send(request);
    }
    catch (Exception ex) {}
private void sendCancelOrders() {
    try {
        CancelOrder cancel = new CancelOrder();

        // Get and remove the oldest order by order ID
        if (!portfolio.getOrderConfirmations().isEmpty()) {
            cancel.setOrderID(portfolio.getOrderConfirmations().get(0).getOrderID());
            cancel.setSide(portfolio.getOrderConfirmations().get(0).getSide());
            portfolio.getOrderConfirmations().remove(0);
        }

        // Send the cancel order to the market
        if (!cancel.getOrderID().equalsIgnoreCase(null) && !cancel.getSide().equalsIgnoreCase(null)) {
            final Action cancelAction = new Action(new AID("market", AID.ISLOCALNAME), cancel);
            final ACLMessage cancelRequest = new ACLMessage(ACLMessage.REQUEST);
            cancelRequest.addReceiver(new AID("market", AID.ISLOCALNAME));
            cancelRequest.setOntology(MarketOntology.getInstance().getName());
            cancelRequest.setLanguage(language);
            myAgent.getContentManager().fillContent(cancelRequest, cancelAction);
            myAgent.send(cancelRequest);
        }
    } catch (Exception ex) {} 
}

@Override
protected void onTick() {
    getMessage();
    super.reset(randomGenerator.nextInt(SLEEP_PERIOD)+1);
}

/**
 * This method adds the latest SendOrderReport object to the Order Report
 * @param orderReport - OrderReport object
 */
private void updateInventory(SendOrderReport orderReport) {
    if (orderReport.getOrderSide().equalsIgnoreCase("BUY")) {
        final long volumeExecuted = inventory.get Shares() + orderReport.getVolumeExecuted();
        inventory.setShares(volumeExecuted);
        final double cashRemaining = inventory.getCash() - orderReport.getPriceExecutedAt();
        inventory.setCash(cashRemaining);
    }
    else {
        final long volumeExecuted = inventory.getShares() - orderReport.getVolumeExecuted();
        inventory.setShares(volumeExecuted);
        final double cashRemaining = inventory.getCash() + orderReport.getPriceExecutedAt();
        inventory.setCash(cashRemaining);
    }
}

/**
 * This method balances the noise agent's inventory of cash and shares
 * @param cash - current cash
 * @param shares - current amount of shares
 * @param lastExecutedPrice - last known traded price on the order book
 * @return currentStatus
 */
private double balanceInventory(double cash, long shares, double lastExecutedPrice) {
    final double sharePreference = 0.5D;
    final double currentWealth = shares * lastExecutedPrice + cash;
    final double currentStatus = (sharePreference * currentWealth) / lastExecutedPrice;
    return currentStatus;
}

G.5 Twap Agent

Setup

package abms;

import jade.content.lang.sl.SLCodec;
import jade.core.Agent;
import jade.domain.FIPANames.ContentLanguage;

/** Title: TwapAgent
 * Description: This class implements a twap trader with the specified parameters
 * Version: 1.4
 * Last date modified: 01/04/2012
 * @author Andrei */

public class TwapAgent extends Agent{

    private final MarketStatus marketStatus;
    private final OrderReport orderReport;
    private final Portfolio portfolio;
    private final TwapInventory inventory;
    private long volumeExecuted;
    private long volumeLeft;

    public TwapAgent(){
        marketStatus = new MarketStatus();
        orderReport = new OrderReport();
        portfolio = new Portfolio();
        inventory = new TwapInventory(volumeExecuted,volumeLeft);
    }

    @Override
    protected void setup(){
        System.out.println("Twap-Agent:"+getAID().getName()+"has been created");
        // Register language and ontology
        getContentManager().registerLanguage(new SLCodec(), ContentLanguage.FIPA_SL);
        getContentManager().registerOntology(MarketOntology.getInstance());

        /* Get the agent's arguments, i.e the numbers of shares to buy (volume),
        * the time span, the sleep period and the number of orders to be
        * submitted in this time span.
        */

        Object [] arguments = getArguments();
        String orderSide = (String) arguments[0];
        final long volume = Long.parseLong((String) arguments[1]);
        final int timeSpan = Integer.parseInt((String) arguments[2]);
        final int timeSlices = Integer.parseInt((String) arguments[3]);
        final long startTime = System.currentTimeMillis();
        final long sleepPeriod = (timeSpan/timeSlices) * 1000;

        // Set the main behaviour of the Trader Agent
        addBehaviour(new TwapBehaviour(this, orderSide, volume, startTime, timeSpan,
                timeSlices, sleepPeriod, marketStatus, portfolio, orderReport, inventory));
        addBehaviour(new UpdateBehaviour(this, marketStatus, orderReport, portfolio));
    }
}
package abms;

import jade.content.onto.basic.Action;
import jade.core.AID;
import jade.core.Agent;
import jade.core.behaviours.TickerBehaviour;
import jade.domain.FIPANames.ContentLanguage;
import jade.lang.acl.ACLMessage;
import java.text.DecimalFormat;
import java.util.Random;
import org.apache.commons.math3.distribution.UniformIntegerDistribution;

/** Title: TwapBehaviour
 * Description: This class implements a TWAP strategy to be used by the TWAP agent
 * Version: 1.7
 * Last date modified: 01/04/2012
 * @author Andrei
 */

public class TwapBehaviour extends TickerBehaviour {

    private final String language = ContentLanguage.FIPA_SL;
    private final String orderSide;
    private final Long totalVolume;
    private final long startTime;
    private final int timeSpan;
    private final int numberOfSubOrders;
    private final long sleepPeriod;
    private final TwapInventory inventory;
    private static int ordersLeft;
    private final MarketStatus marketStatus;
    private final OrderReport orderReport;
    private final Portfolio portfolio;

    /**
     * Creates a new TWAP Behaviour
     * @param agent
     * @param orderSide specifies whether it is a buy or sell order
     * @param volume is the total volume of the order
     * @param startTime is the time at which the TWAP Trader was initiated
     * @param timeSpan is the time period during which the TWAP trader must execute
     * @param numberOfSubOrders is the fixed number of orders to be sent
     */

    public TwapBehaviour(final Agent agent, final String orderSide, final long totalVolume, final long startTime, final int timeSpan, final int numberOfSubOrders, final long sleepPeriod, final TwapInventory inventory, final MarketStatus marketStatus, final OrderReport orderReport, final Portfolio portfolio) {
        super(agent, totalVolume, startTime, timeSpan, numberOfSubOrders, sleepPeriod);
        this.language = language;
        this.orderSide = orderSide;
        this.totalVolume = totalVolume;
        this.startTime = startTime;
        this.timeSpan = timeSpan;
        this.numberOfSubOrders = numberOfSubOrders;
        this.sleepPeriod = sleepPeriod;
        this.inventory = inventory;
        this.ordersLeft = 0;
        this.marketStatus = marketStatus;
        this.orderReport = orderReport;
        this.portfolio = portfolio;
    }

    public void performAction() {
        super.performAction();
        // Perform the TWAP behaviour logic here
    }
}

99
final long startTime, final int timeSpan, final int numberOfSubOrders,
final MarketStatus marketStatus, final Portfolio portfolio, final OrderReport orderReport, final TwapInventory inventory){

super(agent, sleepPeriod);
this.orderSide = orderSide;
this.totalVolume = totalVolume;
this.startTime = startTime;
this.timeSpan = timeSpan;
this.numberOfSubOrders = numberOfSubOrders;
this.sleepPeriod = sleepPeriod;
ordersLeft = numberOfSubOrders;
this.marketStatus = marketStatus;
this.portfolio = portfolio;
this.orderReport = orderReport;
this.inventory = inventory;
}

/**
 * @param number – the number to be converted in two decimal format
 * @return the number in two decimal format
 */
public double getTwoDecimal(double number){
    DecimalFormat df = new DecimalFormat("#.##");
    return Double.parseDouble(df.format(number));
}

/**
 * This method will receive the last market status containing the best bid
 * and the best offer
 */
public void getMessage(){
    if (!orderReport.getOrderReports().isEmpty()){
        try{
            updateInventory();
        }
        catch (Exception ex){}
    }

    if (!marketStatus.getUpdates().isEmpty()){
        try{
            setOrder(marketStatus.getUpdates().get(marketStatus.getUpdates().size() - 1));
        }
        catch (Exception ex){}
    }
}
/**
public void setOrder(SendMarketStatus currentStatus)
{
    double bestBid = getTwoDecimal(currentStatus.getBestBid());
    double bestOffer = getTwoDecimal(currentStatus.getBestOffer());

    if (bestBid == 0)
    {
        bestBid = orderReport.getOrderReports().get(orderReport.getOrderReports().size() - 1).getPriceExecutedAt();
    }

    if (bestOffer == 0)
    {
        bestOffer = orderReport.getOrderReports().get(orderReport.getOrderReports().size() - 1).getPriceExecutedAt();
    }

    final long timePassed = System.currentTimeMillis() - startTime;
    final long timeLeft = (startTime + 1000 * timeSpan) - System.currentTimeMillis();

    if (ordersLeft == numberOfSubOrders)
    {
        ordersLeft --;
        if (orderSide.equalsIgnoreCase("BUY")){
            setFirstOrder(getTwoDecimal(bestBid));
        }
        else{
            setFirstOrder(getTwoDecimal(bestOffer));
        }
    }
    else if ((ordersLeft > 0 && ordersLeft != numberOfSubOrders && inventory.getVolumeLeft() > 0))
    {
        System.err.println(ordersLeft);
        ordersLeft --;
        SendOrder sendOrder = new SendOrder();

        double volumePercentage = ((double)inventory.getVolumeExecuted()) / (double)totalVolume;

        double timePassedPercentage = ((double)timePassed / (double)(timePassed + timeLeft)) * 100;

        double difference = timePassedPercentage - volumePercentage

        System.out.println(timePassedPercentage+""," +volumePercentage+""," +difference);
        sendOrder.setOrderType(" Limit");
    }
sendOrder.setSide(orderSide);

if (difference >= 0 && difference <= 2){
    if (orderSide.equalsIgnoreCase("BUY")){
        sendOrder.setPrice(getTwoDecimal(bestBid) + 0.01);
    } else{
        sendOrder.setPrice(getTwoDecimal(bestOffer) - 0.01);
    }
}

else if (difference > 2 && difference <= 5){
    if (orderSide.equalsIgnoreCase("BUY")){
        sendOrder.setPrice(getTwoDecimal(bestBid) + 0.01);
    } else{
        sendOrder.setPrice(getTwoDecimal(bestOffer) - 0.01);
    }
}

else if (difference > 5 && difference <= 10){
    if (orderSide.equalsIgnoreCase("BUY")){
        sendOrder.setPrice(getTwoDecimal(bestBid) + 0.02);
    } else{
        sendOrder.setPrice(getTwoDecimal(bestOffer) - 0.02);
    }
}

else if (difference > 10 && difference <= 20){
    if (orderSide.equalsIgnoreCase("BUY")){
        sendOrder.setPrice(getTwoDecimal(bestBid) + 0.03);
    } else{
        sendOrder.setPrice(getTwoDecimal(bestOffer) - 0.03);
    }
}

else if (difference > 20 && difference <= 30){

}
if (orderSide.equalsIgnoreCase("BUY")) {
    sendOrder.setPrice(getTwoDecimal(bestBid) + 0.04);
} else {
    sendOrder.setPrice(getTwoDecimal(bestOffer) - 0.04);
}

else if (difference > 30) {
    if (orderSide.equalsIgnoreCase("BUY")) {
        sendOrder.setPrice(getTwoDecimal(bestBid) + 0.05);
    } else {
        sendOrder.setPrice(getTwoDecimal(bestOffer) - 0.05);
    }
}

// Cancel the previously submitted order
sendCancelOrders();

final long orderVolume = totalVolume / numberOfSubOrders;
final int twoPercent = (int) ((2 * orderVolume) / 100);
UniformIntegerDistribution volumeNoiseRange = new UniformIntegerDistribution(0, twoPercent);
int volumeNoise = volumeNoiseRange.sample();

// Randomly choose to add or subtract the noise from the volume
Random generator = new Random(2);
final int plusOrMinus = generator.nextInt(2);
if (plusOrMinus == 0) {
    sendOrder.setVolume(orderVolume - volumeNoise);
} else {
    sendOrder.setVolume(orderVolume + volumeNoise);
}

// Send the new order to market
sendOrderToMarket(sendOrder);

/**
 * If the twap trader runs out of time and the @volumeLeft is more than
 * 20% of the total @volume, the trader will submit a market @Order
else if (ordersLeft == 0 && inventory.getVolumeLeft() > (5*totalVolume/100)) {
    sendCancelOrders();
    SendOrder so = new SendOrder();
    so.setOrderType("market");
    if (orderSide.equalsIgnoreCase("BUY")) {
        so.setPrice(getTwoDecimal(bestBid) + 0.07);
    } else {
        so.setPrice(getTwoDecimal(bestOffer) - 0.07);
    }
    so.setVolume(inventory.getVolumeLeft() / 4);
    so.setSide(orderSide);
    sendOrderToMarket(so);
}

else if (inventory.getVolumeLeft() <= (1*totalVolume/100)) {
    sendCancelOrders();
}

/**
 * The Twap Trader updates its portfolio every time it receives
 * an @param orderReport
 */

public void updateInventory() {
    while (!orderReport.getOrderReports().isEmpty()) {
        SendOrderReport report = orderReport.getOrderReports().get(orderReport.getOrderReports().size() - 1);
        final long volumeExecuted = inventory.getVolumeExecuted() + report.getVolumeExecuted();
        inventory.setVolumeExecuted(volumeExecuted);

        final long volumeLeft = inventory.getVolumeLeft() - report.getVolumeExecuted();
        inventory.setVolumeLeft(volumeLeft);

        orderReport.getOrderReports().remove(orderReport.getOrderReports().size() - 1);
    }
}

/**
 * Adds a new order confirmation to the portfolio
 */
* @param orderConfirmation - OrderConfirmation object

```java
public void addOrderToPortfolio(SendOrderConfirmation orderConfirmation)
{
    portfolio.getOrderConfirmations().add(orderConfirmation);
}
```

/**
 * Submits a new @param sendOrder to the Market Agent
 */

```java
public void sendOrderToMarket(SendOrder sendOrder)
{
    try {
        final Action action = new Action(new AID("market", AID.ISLOCALNAME),
                                          new ACLMessage(ACLMessage.REQUEST);
        request.addReceiver(new AID("market", AID.ISLOCALNAME));
        request.setOntology(MarketOntology.getInstance().getName());
        request.setLanguage(language);
        myAgent.getContentManager().fillContent(request, action);
        myAgent.send(request);
        System.out.println(sendOrder.getPrice());
    } catch (Exception ex){}
}
```

```java
/**
 * This method will retrieve the oldest order from the portfolio
 * and send it to the market
 */

private void sendCancelOrders()
{
    try{
        CancelOrder cancel = new CancelOrder();

        // Get and remove the oldest order by order ID
        cancel.setOrderID(portfolio.getOrderConfirmations().get(0).getOrderID());
        cancel.setSide(portfolio.getOrderConfirmations().get(0).getSide());
        portfolio.getOrderConfirmations().remove(0);

        if (cancel.getOrderID().equalsIgnoreCase(null) && !cancel.getSide().equalsIgnoreCase(null))
        {
            final Action cancelAction = new Action(new AID("market", AID.ISLOCALNAME),
                                      new ACLMessage(ACLMessage.REQUEST);
            cancelRequest.addReceiver(new AID("market", AID.ISLOCALNAME));
            cancelRequest.setOntology(MarketOntology.getInstance().getName());
            cancelRequest.setLanguage(language);
            myAgent.getContentManager().fillContent(cancelRequest, cancelAction);
            myAgent.send(cancelRequest);
        }
    }
```
catch (Exception ex){}

/**
 * Sets a first order to be submitted at the current best @param price
 */

private void setFirstOrder(double price) {
    inventory.setVolumeExecuted(0);
    inventory.setVolumeLeft(totalVolume);

    SendOrder so = new SendOrder();
    so.setOrderType("Limit");
    so.setSide(orderSide);

    if (orderSide.equalsIgnoreCase("BUY")){
        if (price != 0){
            so.setPrice(getTwoDecimal(price));
        }
        else{
            so.setPrice(marketStatus.getUpdates().get(marketStatus.getUpdates().size() - 1).getLastExecutionPrice());
        }
    }

    else if (orderSide.equalsIgnoreCase("SELL")){
        if (price != 0){
            so.setPrice(getTwoDecimal(price));
        }
        else{
            so.setPrice(marketStatus.getUpdates().get(marketStatus.getUpdates().size() - 1).getLastExecutionPrice());
        }
    }

    so.setVolume(totalVolume / numberOfSubOrders);
    sendOrderToMarket(so);
}

@Override
protected void onTick() {
    getMessage();
    super.reset(sleepPeriod);
}

G.6 Opportunistic Agent

Setup
package abms;

import jade.content.lang.sl.SLCodec;
import jade.core.Agent;
import jade.domain.FIPANames.ContentLanguage;
import java.util.Random;

/** Title: OpportunisticAgent
 * Description: This class implements an opportunistic trader.
 * Version: 1.3
 * Last date modified: 01/04/2012
 * @author Andrei
 */

public class OpportunisticAgent extends Agent{

    private final MarketStatus marketStatus;
    private final Portfolio portfolio;
    private final OrderReport orderReport;
    private final StandardInventory inventory;
    private long shares;
    private double cash;

    public OpportunisticAgent(){
        marketStatus = new MarketStatus();
        portfolio = new Portfolio();
        orderReport = new OrderReport();
        inventory = new StandardInventory(shares, cash);
    }

    @Override
    protected void setup(){
        System.out.println("OpportunisticAgent"+getAID().getName()+"has been created");
        // Register language and ontology
        getContentManager().registerLanguage(new SLCodec(), ContentLanguage.FIPA_SL);
        getContentManager().registerOntology(MarketOntology.getInstance());

        // Set the main behaviour of the Noise Trader
        addBehaviour(new OpportunisticBehaviour(this, new Random(), marketStatus, portfolio, orderReport, inventory));
        addBehaviour(new UpdateBehaviour(this, marketStatus, orderReport, portfolio));
    }
}
Opportunistic Behaviour

package abms;

import jade.content.onto.basic.Action;
import jade.core.AID;
import jade.core.Agent;
import jade.core.behaviours.TickerBehaviour;
import jade.domain.FIPANames.ContentLanguage;
import jade.lang.acl.ACLMessage;
import java.util.Random;
import org.apache.commons.math3.distribution.LogNormalDistribution;

/**
 * Title: OpportunisticBehaviour
 * Description: This class implements a Static Order Imbalance Strategy to be used by the opportunistic trader
 * Version: 1.3
 * Last date modified: 01/04/2012
 * @author Andrei
 */

public class OpportunisticBehaviour extends TickerBehaviour{

    private final String language = ContentLanguage.FIPA_SL;
    private final Random randomGenerator;
    private static final int SLEEP_PERIOD = 5000;
    private static final double THETA = 0.07D;
    private static final double ALPHA = 0.01D;
    private final LogNormalDistribution volumeRange = new LogNormalDistribution(4.5, 0.8);
    private final MarketStatus marketStatus;
    private final OrderReport orderReport;
    private final Portfolio portfolio;
    private final StandardInventory inventory;

    /**
     * This constructor instantiates a new OpportunisticBehaviour object
     * @param agent - JADE Agent to which the behaviour is added
     * @param randomGenerator - a Generator used for Normal Distributions
     * @param marketStatus - MarketStatus object containing the market updates
     * @param portfolio - Portfolio object containing confirmed orders
     * @param orderReport - Order object containing reports of executed orders
     * @param inventory - Inventory object containing information about the current status of cash and shares owned by the agent
     */
    public OpportunisticBehaviour(final Agent agent, final Random randomGenerator, final MarketStatus marketStatus, final Portfolio portfolio, final OrderReport orderReport, final StandardInventory inventory){
        super(agent, randomGenerator.nextInt(SLEEP_PERIOD)+1);
        this.randomGenerator = randomGenerator;
        this.marketStatus = marketStatus;
        this.orderReport = orderReport;
    }
}
this.portfolio = portfolio;
this.inventory = inventory;
inventory.setShares(1000);
inventory.setCash(10000);
}

/**
 * This method will receive the last market status containing the best bid
 * and the best offer
 */

private void getMessage() {

    while (!orderReport.getOrderReports().isEmpty()) {
        try {
            updateInventory(orderReport.getOrderReports().get(orderReport.getOrderReports().size() - 1));
            updateInventory(orderReport.getOrderReports().remove(orderReport.getOrderReports().size() - 1));
        }
        catch (Exception ex) {
        }
    }

    if (!marketStatus.getUpdates().isEmpty()) {
       .setAction();
    }
}

/**
 * This method sets the type of action to be performed by the agent: cancellation
 * order, limit order or market order
 */

private void setAction() {

    SendMarketStatus currentStatus = !marketStatus.getUpdates().isEmpty() ?
    SendMarketStatus.currentStatus = !marketStatus.getUpdates().isEmpty() ?
    double bestBid = currentStatus.getLastExecutionPrice();
    double bestOffer = currentStatus.getLastExecutionPrice();
    double lastExecutedPrice = currentStatus.getLastExecutionPrice();
    double buySideVwap = currentStatus.getBuySideVwap();
    double sellSideVwap = currentStatus.getSellSideVwap();
    System.err.println(buySideVwap + " , " + sellSideVwap);
    sendCancelOrders();
    setLimitOrder(lastExecutedPrice, buySideVwap, sellSideVwap, bestBid, bestOffer);
}

/**
 * This method sets a limit order
 * @param lastExecutedPrice - the last traded price on the order book
 * @param buySideVwap - volume-weighted average price on the buy side of the
 * order book
 */
private void setLimitOrder(double lastExecutedPrice, double buySideVwap, double sellSideVwap, double bestBid, double bestOffer) {
    if ((buySideVwap != 0) && (sellSideVwap != 0)) {
        SendOrder sendOrder = new SendOrder();
        sendOrder.setOrderType("limit");
        long volume = (long) volumeRange.sample();
        sendOrder.setVolume(volume);
        if ((buySideVwap - sellSideVwap) >= THETA && inventory.getShares() > 1000) {
            sendOrder.setSide("SELL");
            sendOrder.setPrice(lastExecutedPrice - ALPHA);
            sendOrderToMarket(sendOrder);
        } else if ((sellSideVwap - buySideVwap) > THETA && inventory.getCash() > 500) {
            sendOrder.setSide("BUY");
            sendOrder.setPrice(lastExecutedPrice + ALPHA);
            sendOrderToMarket(sendOrder);
        } else if (inventory.getCash() < 1000) {
            sendOrder.setSide("SELL");
            sendOrder.setPrice(bestOffer);
            sendOrderToMarket(sendOrder);
        }
    }
}

/**
 * This method adds the latest SendOrderReport object to the Order Report array list
 * @param orderReport - OrderReport object
 */
private void updateInventory(SendOrderReport orderReport) {
    if (orderReport.getOrderSide().equalsIgnoreCase("BUY")) {
        // Code to update inventory
    }
}
final long volumeExecuted = inventory.getShares() + orderReport.getVolumeExecuted();

final double cashRemaining = inventory.getCash() - orderReport.getPriceExecutedAt();

} else {

final long volumeExecuted = inventory.getShares() - orderReport.getVolumeExecuted();

final double cashRemaining = inventory.getCash() + orderReport.getPriceExecutedAt();

}

/**
 * Sends a new @param sendOrder to the Market Agent
 *
 */
private void sendOrderToMarket(SendOrder sendOrder) {

try {

final Action action = new Action(new AID("market", AID.ISLOCALNAME),
final ACLMessage request = new ACLMessage(ACLMessage.REQUEST);
request.addReceiver(new AID("market", AID.ISLOCALNAME));
request.setOntology(MarketOntology.getInstance().getName());
request.setLanguage(language);
myAgent.getContentManager().fillContent(request, action);
myAgent.send(request);

} catch (Exception ex) {} 

} } /*This method will retrieve the oldest order from the portfolio
* and send it to the market
*/
private void sendCancelOrders() {

try {

CancelOrder cancel = new CancelOrder();

// Get and remove the oldest order by order ID
cancel.setOrderID(portfolio.getOrderConfirmations().get(0).getOrderID());

}
cancel.setSide(portfolio.getOrderConfirmations().get(0).getSide());

portfolio.getOrderConfirmations().remove(0);

// Send the cancel order to the market
if (!cancel.getOrderID().equalsIgnoreCase(null) && !cancel.getSide().equalsIgnoreCase(null)) {

    final Action cancelAction = new Action(new AID("market", AID.ISLOCALNAME), cancel);
    final ACLMessage cancelRequest = new ACLMessage(ACLMessage.REQUEST);
    cancelRequest.addReceiver(new AID("market", AID.ISLOCALNAME));
    cancelRequest.setOntology(MarketOntology.getInstance().getName());
    cancelRequest.setLanguage(language);
    myAgent.getContentManager().fillContent(cancelRequest, cancelAction);
    myAgent.send(cancelRequest);
}

} catch (Exception ex) {}
private final OrderReport orderReport;
private final Portfolio portfolio;

public UpdateBehaviour(final Agent agent, final MarketStatus marketStatus, final OrderReport orderReport, final Portfolio portfolio) {
    super(agent);
    this.marketStatus = marketStatus;
    this.orderReport = orderReport;
    this.portfolio = portfolio;
}

/**
 * Get the latest market status
 * @return marketStatus - contains: bestBid: price and volume,
 * bestOffer: price and volume,
 * lastExecutionPrice
 * /
public MarketStatus getMarketStatus() {
    return marketStatus;
}

/**
 * Get the list of order reports
 * @return orderReport
 * /
public OrderReport getOrderReport() {
    return orderReport;
}

/**
 * Get the portfolio
 * @return portfolio
 * /
public Portfolio getPortfolio() {
    return portfolio;
}

@Override
public void action() {

    MessageTemplate confirmMessageTemplate = MessageTemplate.and
        (MessageTemplate.MatchPerformative(ACLMessage.CONFIRM),
         MessageTemplate.MatchSender(new AID("market", AID.ISLOCALNAME)));

    MessageTemplate feedMessageTemplate = MessageTemplate.and
        (MessageTemplate.MatchPerformative(ACLMessage.INFORM),
         MessageTemplate.MatchSender(new AID("market", AID.ISLOCALNAME)));

    MessageTemplate reportMessageTemplate = MessageTemplate.and
        (MessageTemplate.MatchPerformative(ACLMessage.CFP),
         MessageTemplate.MatchSender(new AID("market", AID.ISLOCALNAME)));
}
ACLMessage confirmMessage = myAgent.receive(confirmMessageTemplate);
ACLMessage feedMessage = myAgent.receive(feedMessageTemplate);
ACLMessage reportMessage = myAgent.receive(reportMessageTemplate);

if (feedMessage != null) {
    try {
        ContentManager content = myAgent.getContentManager();
        Action contentActionFeed = (Action) content.extractContent(feedMessage);
        SendMarketStatus currentStatus = (SendMarketStatus) contentActionFeed.getAction();

        // Check if the arraylist contains updates and remove the oldest
        if (marketStatus.getUpdates().size() > 0) {
            try {
                marketStatus.getUpdates().remove(0);
            } catch (Exception ex) {}
        }
        // Add a the latest market update to the array list
        marketStatus.getUpdates().add(currentStatus);
    } catch (Exception ex) {}}

if (reportMessage != null) {
    try {
        ContentManager content = myAgent.getContentManager();
        Action contentReport = (Action) content.extractContent(reportMessage);
        SendOrderReport latestReport = (SendOrderReport) contentReport.getAction();
        orderReport.getOrderReports().add(latestReport);

        if (latestReport.getVolumeLeft() == 0){
            removeOrderFromPortfolio(latestReport.getUniqueOrderID());
        }
    } catch (Exception ex) {}}

if (confirmMessage != null) {
    try {

114
ContentManager content = myAgent.getContentManager();
Action contentActionConfirm = (Action) content.extractContent(confirmMessage);
SendOrderConfirmation orderConfirmation = (SendOrderConfirmation) contentActionConfirm.getAction();
portfolio.getOrderConfirmations().add(orderConfirmation);
}

catch (Exception ex){}
}

/**
 * Removes an order from the portfolio
 * @param uniqueOrderID - String containing the id of the order to be removed
 */
private void removeOrderFromPortfolio(String uniqueOrderID) {
    for (SendOrderConfirmation order : portfolio.getOrderConfirmations()){
        if (order.getOrderID().equalsIgnoreCase(uniqueOrderID)){
            portfolio.getOrderConfirmations().remove(order);
        }
    }
}

G.8 Initiation Agent

Initiation Behaviour

G.9 Simulation Agent

Setup

package abms;

import jade.core.Agent;
import jade.wrapper.AgentController;
import jade.wrapper.AgentContainer;

/** Title: SimulationAgent
 * Description: This class will start up a market agent, an initiation agent, as well as noise agents, twap agents and opportunist agents
 * Version: 1.7
 * Last date modified: 01/04/2012
 * @author Andrei
 */

public class SimulationAgent extends Agent {


@Override
protected void setup()
{
    // Setting up the arguments of the TWAP Agents
    Object[] twapArgs1 = new Object[4];
twapArgs1[0] = "BUY";
twapArgs1[1] = "10000";
twapArgs1[2] = "300";
twapArgs1[3] = "15";

    AgentContainer container = getContainerController();

    try{
        boolean initiationAgentCanStart = false;
        boolean tradersCanStart = false;
        boolean simulationFinished = false;

        // Start the market agent
        AgentController marketAgent = container.createNewAgent("market", "abms.MarketAgent");
        marketAgent.start();

        Timer initiationAgentTimer = new Timer(2, System.currentTimeMillis());
        while (!initiationAgentCanStart)
        {
            initiationAgentCanStart = initiationAgentTimer.countDown();
        }

        // Start the initiation agent
        AgentController initiationAgent = container.createNewAgent("Initiator", "abms.InitiationAgent");
        initiationAgent.start();

        Timer noiseTraderTimer = new Timer(15, System.currentTimeMillis());
        while (!tradersCanStart)
        {
            tradersCanStart = noiseTraderTimer.countDown();
        }

        initiationAgent.kill();

        // Start several noise agents
        for(int i=1; i <= 10; i++)
        {
            String agentNumber = Integer.toString(i);
            AgentController noiseAgent = container.createNewAgent("NoiseAgent " + agentNumber, "abms.NoiseTrader");
            noiseAgent.start();
        }

        // Start several opportunistic agents
        for(int i=1; i <= 2; i++)
        {
            String agentNumber = Integer.toString(i);
            AgentController opportunisticAgent = container.createNewAgent("OpportunisticAgent " + agentNumber, "abms.OpportunisticTrader");
            opportunisticAgent.start();
        }
    }
}
// Start a Twap agent
AgentController twapAgent1 = container.createNewAgent("TwapTrader1", twapArgs1);
twapAgent1.start();

Timer timer = new Timer(360, System.currentTimeMillis());
while (!simulationFinished){
simulationFinished = timer.countDown();
}

// Stop the simulation after the timer expires
System.exit(0);

catch (Exception e){}