University College London
MSc. Computer Science

Final year project

Simulating Interaction in Financial Markets

Author:
Vikram Bakshi

Supervisor:
Dr. Christopher Clack

September, 2015

This report is submitted as part requirement for the MSc Computer Science degree 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

As a result of regulation and large amounts of fragmentation, the United States National Market System (NMS) is unique in many ways compared to the other financial districts of the world. Mandates such as requiring the existence of a consolidated quotation system (CQS) and enforcing specific conditions in which a trade is permitted to execute (discussed in Chapter 2) are examples of features which make it particularly distinct.

The main goal of this project is to extend an existing Haskell agent based simulator such that it is better able to cope with the nuances involved in correctly simulating a system such as the United States NMS.

A major development provided by this project is the extension of the simulator to account for the delays associated with message passing between agents. Additionally the project enhances the simulator's messaging infrastructure by implementing the Financial Information eXchange (FIX) protocol. These improvements are not necessarily limited to simulating the US NMS as they facilitate the support of any type of large scale simulation with many different interacting agents.

The final contribution made by this project is the foundation of an easily extensible NMS framework. This is achieved by not only implementing the FIX protocol but by also creating an agent modeling the behaviour of the aforementioned CQS.

Contents

List of Abbreviations **v**

List of Figures **vi**

List of Algorithms **x**

List of Tables **xi**

1 Introduction **1**
1.1 Preamble 1
1.2 Motivation 2
1.3 Project Goals 4
1.4 Project Management 5
1.5 Report Outline 5

2 Background Information **6**
2.1 United States Federal Securities Legislation 6
2.2 The National Market System 6
2.3 Regulation National Market System 7
2.3.1 REG NMS Overview 7
2.3.2 Rule 603: Market Data Rules and Plans 8
2.3.3 Rule 610: Access Rule 8
2.3.4 Rule 611: Order Protection Rule 9
2.3.5 Rule 612: Sub-Penny Pricing 10
2.3.6 Protected and Automated Quotations 10
2.4 Other Regulation 10
2.5 NMS Components 11

3 The Agent Based Simulator **12**
3.1 Background and Requirements Capture 12
3.1.1 The Current Simulator and its Limitations 12
3.2 Analysis of Requirements 18

3.2.1 Simulator Extensions 18
3.2.2 Agent Wrapper Functions 18
3.3 Design and Implementation 19
3.3.1 Simulator Extensions 19
3.3.2 Decoupling Agent IDs from Wrappers 31
3.4 Testing and Validation 34
3.4.1 Simulator Extensions 34
3.4.2 The FunArg3 Runtime Argument 35

4 The Messaging Infrastructure **36**
4.1 Background and Requirements Capture 36
4.2 Analysis of Requirements 39
4.3 Design and Implementation 41
4.3.1 Exploiting Haskell's Type System 42
4.3.2 The BATS Messaging Protocol 49
4.4 Testing and Validation 53

5 Simulating the Consolidated Quotation System **55**
5.1 Background and Requirements Capture 55
5.2 Analysis of Requirements 55
5.3 Design and Implementation 57
5.4 Testing and Validation 62

6 Conclusions **64**
6.1 Evaluation 64
6.2 Conclusion 67
6.3 Further Work 68

Bibliography **69**

A Simulator History **73**

B User Manual **74**
B.1 Compilation 74
B.2 Running the Simulation 75

C System Manual **76**

C.1 Creating a New Experiment 76
C.2 Creating a New Agent 77

D Code for CQS Agent 79

E Testing and Validation 81

E.1 Simulator Extensions 81
E.1.1 The FunArg3 Runtime Argument 84
E.2 The CQS Agent 86
E.3 Message Tags 87

F Miscellaneous Code 90

F.1 Functions for Insertion into Lists 90
F.2 Generating Broadcast Subscription Lists 92
F.3 Updating the Simulator's State 93
F.4 Accessory Functions for Filtering the List of All States 98
F.4.1 Old Versus New Versions of Functions 99
F.5 Functions Related to FunArg3 100
F.6 BATS Messages and FIX Tags 103
F.6.1 BATS Messages 103
F.6.2 FIX Tags 106
F.6.3 Tag Accessory Functions 109

List of Abbreviations

BB	Best Bid.
BBO	Best Bid and Offer.
BO	Best Offer.
BYX	BATS BYX Exchange Inc.
CQS	Consolidated Quotation System.
CTA	Consolidated Tape Association.
CTS	Consolidated Tape System.
ECN	Electronic Communications Network.
FIX	Financial Information eXchange.
HFT	High Frequency Trading/Trader.
HFTs	High Frequency Traders.
IOC	Immediate or Cancel.
ISO	Intermarket Sweep Order.
ITS	Intermarket Trading System.
NBBO	National Best Bid and Offer.
NMS	National Market System.
NYSE	New York Stock Exchange.
OTC	Over the counter.
REG NMS	Regulation National Market System.
SEC	U.S. Securities and Exchange Commission.
SIP	Securities Information Processor.
SROs	Self Regulatory Organisations.
UTP	Unlisted Trading Privileges.

List of Figures

3.1 Graph representation of an example experiment. 13
3.2 The worst case scenario for a four agent experiment. 15
3.3 The initial condition, recurrence relation, and solution for $f(n)$. . . . 15
3.4 The type of an agent wrapper function. 16
3.5 The type signature and parameters of the `sim` function. 17
3.6 The hard coding of twenty agent IDs illustrating the unintuitive integer interface to refer to agents and the tight coupling to the order in which wrappers are passed to `sim`. 19
3.7 Redacted Arg_t data declaration showing the type signature for the 'max delay' argument, `FunArg1`, and `FunArg2` respectively. 20
3.8 Updated graph representation of an example experiment. 22
3.9 Adjacency matrix encoding of example experiment graph. 23
3.10 Example of the `getTimestepDelay` function, `FunArg1`, and max delay arguments. 23
3.11 Example `getAgentsSubscribedToChannel` function, and `FunArg2`. . 24
3.12 Current type synonym definition for the simulator state. 26
3.13 Current function definition for initialising an empty simulator state. . 26
3.14 Extended type synonym definition for the simulator state. 27
3.15 Updated function definition for initialising an empty simulator state. 28
3.16 Updated filtering of the relevant messages, broadcasts and routed broadcasts for an agent. 31
3.17 `FunArg3` and `AgentIdentifier_t` type signatures. 32
3.18 Redacted example call to `sim` for sixty timesteps including `FunArg3`. . 33
3.19 Usage of FunArg3 to retrieve the agent IDs of the two exchanges within the simulation. 34

4.1 Redacted Data Type Declaration for `Msg_t` and `Broadcast_t`. 36
4.2 Example illustration of the industry's usage of FIX [Vali, 2009]. . . . 38
4.3 Data declarations for `Entity` and `TagType`. Ellipses indicate the easy extensibility of the declarations. 42

4.4 Data declaration for a tag value and the tag qualifier class. 42
4.5 Potential implementation of the OrdType and TimeInForce tags. . . . 43
4.6 TagInterface type class with separate data declarations as a solution. 45
4.7 Using GHCi to query the implementation. 46
4.8 The final TagInterface type class specification. 46
4.9 Example instance declaration of the TagInterface type class for the TimeInForce tag. 48
4.10 Redacted data declaration of the BATSMessage data type which contains all of the relevant messages for a BATS exchange. 50
4.11 Data declaration showing the record for a BATSNewOrderMessage. . 50
4.12 The implementation of the Side FIX tag. 51
4.13 The setTagValue and checkAndSetTagValue functions for setting tag values. 52

5.1 Data declaration of the CQS Short Quote and CQS Quote Conditions. 58
5.2 Data declaration of the CQS NBBO record. 59
5.3 Updated versions of the Msg_t and Broadcast_t data declarations (redacted). 60
5.4 Data declaration of the internal state of the CQS agent. 60

B.1 Contents of the Main module. 74
B.2 Compiling the Experiment using the MakeFile. 75
B.3 Compiling the Experiment using a custom command. 75
B.4 Running the simulation. 75

C.1 Example code for an experiment. 76
C.2 Example code for an experiment. 77
C.3 Example extension to Agentstate_t. and data declaration of CQSStateRecord. 77
C.4 Example part code for CQS agent wrapper. 78

D.1 CQS Update BBO Function. 79
D.2 CQS Wrapper Function. 80

E.1 Results of testing the Simstate_t data structure after initialisation. . . 81
E.2 Results of testing whether compatability mode is entered and the checksums of the output files. 83

E.3 Example agent which utilises FunArg3 to get the ID of the broker they wish to communicate with. 85
E.4 Tabular representation of the validation experiment for the CQS agent. 86
E.5 Results of test 1 for message tags. 88
E.6 Set up of test 2 for message tags. 88
E.7 Results of test 2 for message tags. 88
E.8 Successfully encoding a tag. 89

F.1 insertAsHead Function 90
F.2 insertIntoInnerList Function 90
F.3 insertElementsIntoInnerList Function 91
F.4 genBCastSubscriptionLists Function 92
F.5 Function for updating the simulator state. 93
F.6 Function for updating the simulator state in compatibility mode. . . 94
F.7 Function for updating the simulator state with routed broadcasts and delay queues. 95
F.8 Function for inserting direct messages into the relevant delay queue. . 96
F.9 Function for creating and inserting routed broadcasts into the relevant delay queue. 97
F.10 Function for moving messages to the correct inner list. 98
F.11 Accessory function, providing a 'safe index' into a list. 98
F.12 Accessory function, providing 'safe' access to the head of a list. 98
F.13 Old accessory function used for retrieving the time of a simulator state 99
F.14 Updated accessory function used for retrieving the time of a simulator state 99
F.15 Old accessory function, used for retrieving the messages of an agent with the given ID. 99
F.16 Updated accessory function, used for retrieving the messages of an agent with the given ID. 99
F.17 Old accessory function, used for retrieving the broadcast messages sent to a particular channel. 99
F.18 Updated accessory function, used for retrieving the broadcast messages sent to a particular channel. 99
F.19 Newly created accessory function, used for retrieving the routed broadcasts sent to a particular channel. 100

F.20 Code for the transforming the user defined agent list from the new style to the old. 100
F.21 Code for the generating a bidirectional mapping between the agent labels and IDs. 101
F.22 Code for the getCorrespondingAgentIdentifier function and how to create a FunArg3. 102
F.23 Part 1 of the data declaration for a BATSMessage. 103
F.24 Part 2 of the data declaration for a BATSMessage. 104
F.25 Part 3 of the data declaration for a BATSMessage. 105
F.26 Part 4 of the data declaration for a BATSMessage. 106
F.27 Part 1 of the implementation of FIX tags. 107
F.28 Part 2 of the implementation of FIX tags. 108
F.29 Accessory functions for tags. 109
F.30 Accessory functions for tags part 2. 110

List of Algorithms

1 Logic flow for updating `Simstate_t` to account for routed broadcasts and delay queues. 30

2 CQS Agent Initialisation Logic Flow. 61

3 CQS Agent Logic Flow Post Initialisation. 62

List of Tables

4.1 Example redacted BATS new order message [BATS, 2015]. 39
4.2 Possible messages between BATS exchanges and its members. 49

5.1 CQS Short Quote Message Contents. 57
5.2 CQS Short NBBO Message Contents. 59

E.1 Adjacency matrix representation of the simulation graph. 83
E.2 Table showing each message which was to be sent from and to each agent with the timestep (TS) it should also be received.) 84

Chapter 1

Introduction

1.1 Preamble

"Make use of time, let not advantage slip."

— William Shakespeare

In January 1790, the United States House of Representatives was debating whether or not the federal government should assume responsibility for the domestic debt accrued by states during the American Revolutionary War [Garber, 1991]. The proposition would eventually be passed in August later that year as the Funding Act of 1790 [Congress, 1790].

Upon hearing the proposal, informed traders chartered high speed boats with the intention of reaching Georgia and purchasing debt prior to the information becoming common knowledge. By doing so, they would be able to exploit their information advantage to make use of the fact that the debt was trading, in some cases, at 10% or less of its face value [Pisani, 2014].

When it comes to trading, the groups with the fastest access time have always had an inherent advantage. Whether it be their faster access to relevant news (as was the case in 1790), or their ability to receive quotations from an exchange on a sub-second basis, being the first one to access the latest information gives you the lead in being the first one to potentially react.

It is impossible for participants in a given trading market to all have the same access time to the available information. The fact remains that if new information is to be consumed there will always be a certain individual/groups who, whether due to their closer proximity to the source or other factors, consume it first and so by definition 'time deltas' between market participants will always exist.

1.2 Motivation

Between the differing participants in the current financial markets, of particular interest is the group known as High Frequency Traders (HFTs). Although a concrete definition for HFTs does not exist, in general they can be categorised by their use of automated trading algorithms which generate incredibly high volumes of trades, as well as their use of specialised order types [Lemke and Lins, 2014].

Additionally, they have an extremely fast access time to quotations, which is made possible by the fact that they co-locate their servers within the same data centres as the exchanges they trade on [Hasbrouck and Saar, 2013].

Typically, they compete against other HFTs rather than long-term investors [Easley et al., 2011; Vuorenmaa and Wang, 2014]. Even so, in recent years, high frequency trading has evolved into a controversial subject and remains ‘besieged by accusations that it cheats slower investors’ [Geiger and Mamudi, 2014] .

A particularly famous criticism is Michael Lewis’ 2014 book, ‘*Flash Boys*’, in which Lewis makes a number of claims against the high frequency trading firms. One of the allegations made by Lewis is that HFTs utilise their faster connection speeds to accurately predict the incoming orders of an exchange. They then use this information advantage to make a profit by ordering ahead of them — a technique referred to throughout the book as ‘front running’.

However, a much more damning accusation made by Lewis is that Regulation National Market System (REG NMS), the financial regulation introduced by the U.S. Securities and Exchange Commission (SEC) in 2005, actually favours HFTs over other market members:

> *“Reg NMS was intended to create equality of opportunity in the U.S. stock market. Instead it institutionalized a more pernicious inequality. A small class of insiders with the resources to create speed were now allowed to preview the market and trade on what they had seen.”*
>
> — Michael Lewis, Flash Boys, 2014

Lewis does not stand alone with his accusations against REG NMS — other authors such as Bodek [2013], and Amuk and Saluzzi [2012] also accuse the regulation of favouring HFTs over others, and call for reforms to be made.

Unfortunately the evidence for the claims provided by Lewis et al. remains largely anecdotal, and their publications contain a significant presence of bias. Considering the severity of the accusations, it is of the utmost importance that unbiased empirical analyses are conducted to assess the extent of their validity.

However, given the gargantuan size of the United States financial system as well as the nuances involved with successfully simulating the NMS regulation and exchanges, conducting any type of analysis presents an immense challenge.

In the past, researchers have successfully utilised agent based simulation to understand the interaction between participants of complex systems. In particular, Court and Clack [2013] as well as Tommi et al. [2014] used independent agent based simulators to fruitfully model and analyse the events of the 2010 Flash Crash — where, over the course of 20 minutes, almost one trillion dollars of market value was effectively wiped out [Madison, 2013].

This project's motivation stems from the desire to lay a foundation from which empirical analysis of the United States' national market system can be conducted. The aforementioned agent based simulator used by Court and Clack [2013] will be extended so that it is able to cope with the greater number of complexities associated with modeling such an intricate system.

The project's end goal is to not only extend the agent based simulator's internal infrastructure, but to also provide a simulator agent representing an important part of the US national market system (the Consolidated Quotation System) for future simulator users to utilise.

Regardless of any accusations, with it being estimated that HFTs accounted for approximately 50% of all 2012 US stock transactions [Stafford and Massoudi, 2013], for better or for worse, high frequency trading has become a major cog in the financial machine.

1.3 Project Goals

Specifically, the primary goals of the project are to:

1. Extend a preexisting Haskell agent based simulator's messaging infrastructure by creating a mechanism for the simulator to emulate the real world delays associated with message passing between agents.
2. Also extend the simulator by implementing a relevant subset of the 'Financial Information eXchange' (FIX) protocol so that agents within the simulation can communicate with each other using the industry standard convention[1].
3. Build an initial framework encapsulating specific parts of the United States National Market System (NMS) for trading equities. The aim is to create the following:
   3.1. A simulator agent implementing the most pertinent aspects of the Consolidated Quotation System (CQS) of the NMS. The functionality to be implemented is that which relates to the portion of the CQS which:
      i. Receives the best bid and best offer (BBO) for a particular equity from each exchange in the NMS.
      ii. Collates the information and calculates the national best bid and offer (NBBO).
      iii. Disseminates the NBBO of that equity to all NMS participants as well as any CQS subscribers.

   3.2. Ensure that created CQS agent complies with the regulation mandated by the U.S. Securities and Exchange Commission. In particular, it is required to comply with Regulation NMS [SEC, 2005].

The secondary goals of the project are to:

1. Make use of a modular design so that future users of the simulator can reuse the created code and extend it as per their requirements.
2. Decouple simulator agents from having to store the integer IDs of those who they will interact with as part of their code's logic. The existing simulator method of hard coding agent IDs is not maintainable for simulations with a large number of agents.

[1] Implementing FIX also provides a flexible messaging subsystem to facilitate future simulations of complex markets containing many different exchanges with different messaging protocols.

3. Retain full backwards compatibility with the experiments which were coded prior to any simulator amendments made as part of this project.

1.4 Project Management

The project's management will be based on the Unified Process (UP) for development as described by Arlow and Neustadt in their text '*UML and the Unified Process: Practical Object-Oriented Analysis and Design*' [2002]. Although the choice of simulator language, Haskell, is not object oriented, the Unified Process is a truly paradigm independent scheme and a perfect fit for defining the workflow of the project.

In accordance with the Unified Process, the required final system is broken down into smaller, more manageable chunks which each form an 'iteration' once completed. The intention of each iteration is to generate a partially finished version of the final system along with any associated project documentation. Increments are defined as the difference between two completed iterations [Arlow and Neustadt, 2002].

Within each iteration there are five core stages:

1. Requirements – capturing what the system should do;
2. Analysis – refining and structuring the requirements;
3. Design – realizing the requirements in system architecture;
4. Implementation – building the software;
5. Testing and Validation – verifying that the implementation works as desired;

In addition to the system described above, the project workflow will make full use of the version control system Git. A new branch in the repository is to be created for each new iteration and only merged into the master branch once the testing stage is completed successfully. A merge into the master branch marks an increment in the project.

1.5 Report Outline

This report is structured such that, in addition to the background and conclusion, each major project iteration forms its own chapter. Each iteration chapter is split into the stages defined by the Unified Process.

Chapter 2

Background Information

2.1 United States Federal Securities Legislation

Following the aftermath of the Wall Street Crash in 1929 and the ensuing Great Depression, Congress in the United States passed the Securities Act of 1933, and the Securities Exchange Act of 1934. Prior to these acts the governance of securities had been a state issue rather than a federal one [Lin, 2010].

As well as being the first federal legislation to regulate the offer and sale of securities, the acts also established the Securities and Exchange Commission (SEC) — which, to this day, remains responsible for the enforcing of federal security law. Additionally, the acts introduced the requirement that the sale of securities, besides those excepted under certain conditions, be registered with the SEC [Congress, 1933, 1934].

2.2 The National Market System

A few decades after the original 1933 and 1934 Securities Acts, Congress brought about a number of amendments — with the most major being that of 1975. In particular, Congress directed the SEC to facilitate the establishment of a national market system (NMS) and a nationwide system for the clearance and settlement of securities transactions [Gillis, 1975].

As a result of the SEC's efforts to establish a centralised NMS, the Consolidated Tape Association (CTA) was formed with the objective of overseeing the dissemination of real time trade and quote data. Trade data distribution would occur through the Consolidated Tape System (CTS) — established 1976 — and quote data distribution would occur through the Consolidated Quotation System (CQS) — established 1978 [Harman, 1978].

April of 1978 saw the linking of the New York Stock Exchange (NYSE) and five other exchanges to form the Intermarket Trading System (ITS), allowing specialist members from one market to trade directly with any other ITS exchange [Cohen et al., 1985].

The establishment of the ITS, CTS and CQS, went hand in hand with the NMS objective 'to centralize all buying and selling interest so as to permit each investor the opportunity for the best possible execution of their order, regardless of where in the system it originated' [Gillis and Dreher, 1982].

2.3 Regulation National Market System

At the turn of twenty first century the SEC sought to 'modernize the regulatory structure of the U.S. equity markets' by updating and consolidating the existing rules into a single NMS regulation. In particular, their desire was to address four specific issues — market data, intermarket access, trade throughs, and sub-penny pricing [SEC, 2004].

The SEC's proposals were called 'Regulation NMS' (REG NMS) and in April 2005 they were approved for adoption. Enforcement would begin in 2007 to allow the financial ecosystem time to adjust to the new mandate [SEC, 2007].

2.3.1 REG NMS Overview

As previously stated, the approval of Regulation NMS meant not only adopting new rules, but also consolidating the existing NMS mandates into a single regulation. Given this fact, it is important to note that of the twelve REG NMS rules, the preexisting ones prior to 2005 were [SEC Division of Trading and Markets, 2015]:

Rule 601: Public dissemination of trade reports;
Rule 602: Public dissemination of quotations;
Rule 604: Display of customer limit orders;
Rule 605: Disclosure of order execution information;
Rule 606: Disclosure of order routing information;
Rule 607: Customer account statements;
Rule 608: National market system plans;
Rule 609: Registration of securities information processors;

The four REG NMS mandates which were new for 2005 were Rules 603, 610, 611 and 612. They correspond to the SEC's desire to regulate market data, intermarket access, trade throughs, and sub-penny pricing respectively. The regulation also included Rule 600 which provided definitions for the terms used throughout the text.

2.3.2 Rule 603: Market Data Rules and Plans

Rule 603 updated the formulas used by the SEC for allocating revenues to the various Self Regulatory Organisations (SROs) they oversee. This was required as it had been determined that the previous formulas incentivised distortive behaviour such as wash sales[1] and trade shredding[2] [SEC, 2005, pp. 30-31]

The rule also grants market centres and their members (broker-dealers) the freedom to distribute their own data independently, with or without fees. However, it remains a requirement that market centres provide their best quotations and trades to the relevant plan processor[3] for consolidated dissemination [SEC, 2005, p. 30].

Two of the three NMS plans are managed by the Consolidated Tape Association and are concerned with the data relating to Tape A[4] and Tape B[5] listed securities. Trade and quotation data for these securities is disseminated by the CTS and CQS respectively. The third and final plan — the Unlisted Trading Privileges (UTP) plan — governs the dissemination of consolidated trade and quotation data for Tape C[6] securities [Wall Street and Tech, 2005].

2.3.3 Rule 610: Access Rule

Rule 610 defines a maximum cap on the price that any trading centre can charge for accessing a protected quote[7]. Additionally, the rule enables the usage of private linkages offered by connectivity providers to access quotations. This is in contrast to the previous mandate of requiring quotations to be accessed through a collective linkage facility such as the ITS [SEC, 2005, p. 335].

1 A wash sale is a sale of a security at a loss and a repurchase of the same or substantially identical security shortly before or after.
2 Trade shredding is the splitting of large trades into a series of 100-share trades.
3 The plan processor's role is to provide a central point for the consolidation of all trade and quotation data for securities.
4 Tape A securities are those listed on the New York Stock Exchange (NYSE).
5 Tape B securities are securities listed on a regional exchange other than NYSE or NASDAQ.
6 Tape C securities are NASDAQ listed securities.
7 See subsection 2.3.6 for the definition of a protected quote.

Furthermore, the Access Rule requires that each national securities exchange and national securities association establish, maintain, and enforce written rules relating to the locking[8] and crossing[9] of the NMS market.

The rules, among other things, must 'prohibit their members from engaging in a pattern or practice of displaying quotations that lock or cross the protected quotations of other trading centers' [SEC, 2005, p. 335].

2.3.4 Rule 611: Order Protection Rule

The Order Protection Rule promotes intermarket price protection by restricting 'trade-throughs' — the execution of trades on one venue at prices that are inferior to publicly displayed protected quotations on another venue [SEC Division of Trading and Markets, 2015].

An integral aspect of the Order Protection Rule is the National Best Bid and Best Offer of a given NMS security. REG NMS Rule 600(b)(42) defines the NBBO as the best bid and best offer 'that are calculated and disseminated on a current and continuing basis by a plan processor' [United States Federal Register, 2005]. Accordingly the NBBO for Tape A and B securities is disseminated by the CQS and for Tape C securities by the UTP quotation feed (UQDF).

Additionally, the scope of the Order Protection Rule extends only to NMS stocks as defined by Rule 600(b)(46)(47), and as a result it does not apply to the trading of options. Furthermore, Rule 611 allows for certain exceptions in which trading through the NBBO would be permitted. Of these exceptions, the most notable are for flickering quotations and intermarket sweep orders (ISOs).

The motivation behind the flickering quotations exception stems from the desire to avoid false indications of trade throughs which, in actuality, are attributable to rapidly moving quotations. A transaction is excepted if, within the last one second, the trading center displaying the quotation that would have been traded through had also displayed an inferior or equal price [SEC, 2005, p. 152].

The ISO exception allows trading centers to execute an order without regard to the NBBO. This is because the sender of the ISO is obligated to route additional ISOs

[8] A locked market occurs when the price to buy a stock is the same as the price to sell a stock.
[9] A crossed market occurs when the price to buy a stock is higher than the price to sell a stock.

in an attempt to execute against all applicable protected quotations [SEC, 2005, p. 153].

2.3.5 Rule 612: Sub-Penny Pricing

The SEC's motivation for issuing the Sub-Penny Pricing rule stemmed from their desire to prevent the practice of 'stepping-ahead' of limit orders by trivial amounts. As such, Rule 612 prohibits market participants from displaying, ranking, or accepting quotations in NMS stocks that are priced in an increment of less than $0.01, unless the price of the quotation is below $1.00 [SEC, 2005, p. 29].

2.3.6 Protected and Automated Quotations

Rule 600(b)(57)(58) defines the term 'protected quotation' to mean that the quotation is the best displayed bid/offer for a trading center, and that it is automated. Furthermore, for a trading centre to display an automated quotation it must meet the criteria defined by Rule 600(b)(3), as described below.

Firstly, it must permit incoming orders to be marked as immediate-or-cancel (IOC). Secondly it must immediately and automatically execute an order marked as IOC against the displayed quotation up to its full size. If any portion of the order is not executed it must be canceled without being routed elsewhere. Moreover, the trading centre must immediately and automatically transmit a response to the sender of the IOC order indicating the action taken.

2.4 Other Regulation

Although the scope of the regulation to be considered for this project is limited to Regulation NMS, other regulation which could potentially be considered for implementation in the future include:

- Regulation SHO — The Regulation of Short Sales.
- Regulation ATS — The Regulation of Alternative Trading Systems.
- NMS Plan To Address Extraordinary Market Volatility — Also known as the Limit Up/Limit Down Plan, this was adopted as a response to the 2010 Flash Crash.

2.5 NMS Components

A complete emulation of the United States' National Market System would include:

- Security Information Processors
  - For Tape A and Tape B securities:
    - CTS — For disseminating trade data.
    - CQS — For disseminating quote data and the NBBO.
  - For Tape C securities:
    - UTP Trade Data Feed — For disseminating trade data.
    - UTP Quote Data Feed — For disseminating quote data and the NBBO.
- Exchanges
- Alternative Trading Systems e.g. Dark Pools[10].
- Brokers
- High Frequency Traders
- Compliance with SEC regulation and plans:
  - Regulation NMS
  - Regulation SHO
  - Regulation ATS — Alternative Trading Systems
  - The Limit Up/Limit Down Plan

The component of the NMS that this project will contribute is the CQS. Independent of the NMS members above, the project will also provide the fundamental messaging system which future agents of the NMS will all use to communicate (FIX). Furthermore, the project will enhance the simulator so that it is able to accurately model the message latencies between each component in the NMS.

After the modifications to the simulator are complete, further work will be able to easily add other components listed above to the initial NMS framework provided by this project. The ultimate goal is to reach a point where the entirety of the US National Market System is replicated.

[10] Dark Pools are electronic Alternative Trading Systems where, unlike stock exchanges, orders remain dark. In other words the size and price of the orders are not revealed to other participants in the market place.

Chapter 3

The Agent Based Simulator

As previously mentioned in the introduction, this project will extend an existing Haskell agent based simulator. A short history of the simulator is provided for reference in Appendix A. This chapter is structured such that each section forms part of the stages defined by the Unified Process as described in section 1.5. The same structure is repeated in all of the implementation related chapters of the report.

3.1 Background and Requirements Capture

3.1.1 The Current Simulator and its Limitations

Graph Representation

A user of the simulator, prior to beginning their coding, is expected to establish a graph representation of their experiment. In their graph, the nodes represent agents and the edges represent the connections between agents. Two agents sharing a connection signifies that they will be communicating with each other in the simulation.

An example graph representation of an actual previous experiment conducted by Chopra [2015] is given in Figure 3.1. In the experiment, two exchanges receive orders from two brokers and a single HFT. The HFT, acting as a 'Front Runner', has servers co-located with both exchanges. Figure 3.1 is intentionally drawn to highlight a number of limitations of the simulator.

Firstly, edges between two nodes in the graph are undirected — implying that a connection between two agents represents bidirectional communication. This may not always be an accurate reflection of the intention of the user, as there could potentially be situations in which they require message passing to occur in a single direction — for example between the CQS and brokers who consume their feed.

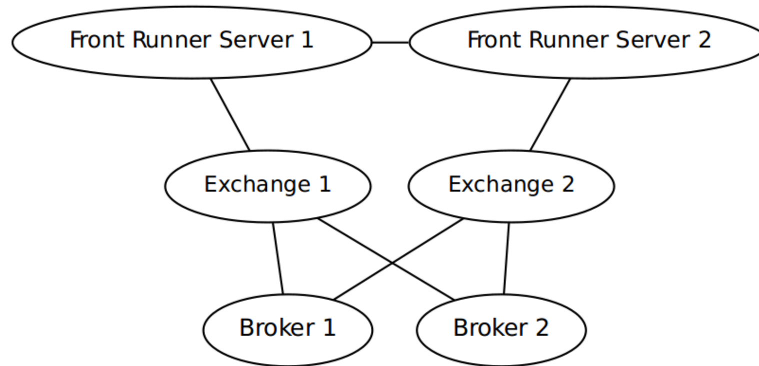

Figure 3.1: Graph representation of an example experiment.

The undirected edges correspond to a limitation of the simulator — it is completely oblivious to the connections which are supposed to exist in a given simulation. Each message contains a (from, to) tuple which serves as the height of the simulator's knowledge of the user's intention.

Since there is no way for users to indicate the connections that are present in their simulation there is no way for the simulator to know if the passing of a message between two agents is valid or not. Suppose an attempt is made to pass a message from one agent to another when, according to the user, a connection between the agents should not exist. Due to the simulator's ignorance of the invalidity of the message it has no choice but to route it to its recipient regardless. Thus the onus of coding the experiment so as to accurately reflect the graph is placed on the user, with no help in enforcing agent connections being given by the simulator itself.

Another aspect of the graph to note is that the time latencies between the agents who are exchanging messages are not attached to its edges. This is the case because the simulator does not have an internal mechanism for modeling the differing message lags between agents. In other words each agent message, regardless of its source and destination, is treated equally by the simulator and takes the same number of timesteps to be sent to its recipient. This limitation, coupled with the fact that the graph is undirected makes it extremely difficult for delays to be represented within the simulator.

Previous Workarounds

As previously stated, a concrete internal mechanism for modeling delays within the simulator does not exist. However, in the past, users have created ad-hoc solutions in an attempt to overcome this limitation for their experiments.

One of these solutions was the creation of a runtime argument called 'delay' [Court and Clack, 2013]. The presence of the argument would indicate to agents which supported it to delay the consumption of their input by the defined amount. Another solution took the form of a 'delay agent' where each delay agent would hold an internal queue retaining messages until their defined time lag expired [Chopra and Clack, 2015]. At that point, the recipient agent would be able to consume the messages.

The mechanism implemented by Court delayed all inbound messages by the same amount regardless of the sender and thus is limited in its applicability to larger experiments. Additionally, the solution only accounted for direct agent to agent messages and offered no support for delaying broadcasts[1]. Conversely, although not integral to the concept of a 'delay agent', Chopra's solution offered no support for delaying direct agent to agent messages. Instead all agents communicated solely through broadcasts even if there was only a single intended recipient. By ignoring direct agent to agent messaging the solution caused ambiguous semantics around the notion of a broadcast and thus should be avoided in the future.

In the worst case of an experiment every agent is connected to every other agent and the connections between them have asymmetric delays. In other words, passing a message in one direction between two agents does not take the same time as passing it in the other direction. An example worst case scenario for a four agent experiment is given in Figure 3.2. The messaging latencies have been chosen arbitrarily to illustrate the asymmetry between connections.

The main issue with both of the previously defined solutions — the 'delay argument' and the 'delay agent' — is that in the worst case scenario the number of arguments/agents required to successfully cater to all of the messaging latencies is equal to the number of edges in the graph.

Consider a function, $f(n)$, where n is the number of agents in a given experiment and

[1] Figure 3.4 and its surrounding discussion provides an explanation of direct messages and broadcast messages.

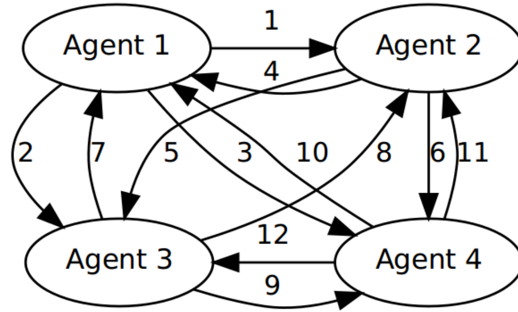

Figure 3.2: The worst case scenario for a four agent experiment.

$f(n)$ is the number of edges resulting in the worst case graph of that experiment. By inspection $f(2) = 2$, in other words when there are two agents in an experiment, in the worst case, there will be two edges in the graph. When an additional agent is added, the existing connections remain and all previous agents receive a connection to and from the new agent. This information can be utilised to establish and solve a recurrence relation as seen in Figure 3.3.

$$f(n) = \begin{cases} 2 & \text{if } n = 2 \\ f(n-1) + 2(n-1) & \text{otherwise} \end{cases}$$

$$f(n) = n^2 - n$$

Figure 3.3: The initial condition, recurrence relation, and solution for $f(n)$.

For complex simulations which could potentially involve ten or more agents this would mean having to individually attach up to $f(10) = 10^2 - 10 = 90$ delay arguments/agents to the experiment. This is unmaintainable for large simulations and thus any future solution must avoid the $O(n^2)$ complexity associated with either of the previous methods.

The Agent_t type, Direct Messages, and Broadcasts

Once the user has established a graph representation of their experiment they then determine the components of their simulation which utilise existing agents and the components which require new code. If a new agent needs to be coded the user is required to compose an 'agent wrapper'.

An agent wrapper is a function with the type of `Agent_t` as defined in Figure 3.4. By

coding wrapper functions which conform to the Agent_t type the user ensures the agent's seamless integration with the simulator.

```
type Agent_t = Agentstate_t -> [Arg_t] -> [(Int, [Msg_t], [Msg_t])] -> Int
           -> [[Msg_t]]
```

Figure 3.4: The type of an agent wrapper function.

As is seen by the type declaration in Figure 3.4, the wrapper function is expected to take four input parameters: the internal state of the agent, a list of arguments, a list of three tuples: [(simulation time, messages, broadcasts)] and the agent's identifier (which is an Int). The wrapper must return a list containing a list of messages for each simulation step ([[Msg_t]])[2].

At this point it is important to distinguish between the two types of messages an agent can send and receive — direct messages, and broadcast messages. Direct messages are those sent from an agent to another agent and as such form a one to one correspondence between sender and receiver.

In contrast, broadcast messages are sent from an agent to a broadcast channel and any agent subscribed to that channel[3] is able to receive those messages. Thus broadcast messages form a one to many correspondence between sender and receiver(s). Broadcast channels are particularly useful for modeling the ubiquitous producer/consumer relationships which exist in the real world such as subscriptions to exchanges for their latest quotation data.

Each simulation is set up such that the potentially infinite list[4] of all messages is filtered for every agent prior to the first timestep occurring. As is seen in the Agent_t type definition (Figure 3.4), when writing the code for an agent wrapper users are able to exploit the fact that their agent will only ever receive messages which are sent directly to it and broadcasts which are sent to the channels it is subscribed to.

In order for the simulator to accurately model the latencies between the sender and receiver(s) of a broadcast message it needs to determine the agents which are actually

[2] If an agent has no messages to output in a simulation step, it must either output an empty list of messages or a list containing a Hiaton (a special type to indicate the absence of any output) for that timestep.

[3] See Figure 3.5's corresponding explanation to understand how an agent's subscription to particular broadcast channels is defined by the user.

[4] Haskell's lazy evaluation semantics make it particularly easy to operate on infinite lists. The evaluation of the data structure occurs on an as needed basis rather than completely upfront.

subscribed to that broadcast channel — which it currently does not do. Apart from when initially filtering the potentially infinite all messages list (as described above), the simulator does not retain the information regarding agent subscriptions to broadcast channels. This limitation of the simulator will need to be corrected for it to successfully model message and broadcast latencies between agents.

Hard Coding of Agent IDs

Figure 3.5 shows the type signature and pattern match of the `sim` function which is called by the user to begin their simulation. As parameters, the user is expected to provide the number of timesteps they would like their simulation to run (`Int`), a list of runtime arguments (`[Arg_t]`), and a list of two tuples (`[(Agent_t, [Int])]`). The resulting output of the function call is of type `IO ()` because a 'trace' file containing the simulator state for each timestep is written to disk. This file is what is utilised by the user to conduct any analyses on the results of their simulation.

```
sim :: Int -> [Arg_t] ->          [ (Agent_t, [Int]) ]           -> IO()
sim    steps     args    listOfAgentsAndSubscribedBroadcastChannels = ...
```

Figure 3.5: The type signature and parameters of the `sim` function.

The list of runtime arguments passed to the simulator is also available to each agent in the simulation. This means that the inclusion of a runtime argument could result in an adjustment of the behaviour of an agent or the simulator itself. For example, the presence of a 'randomise' argument results in all messages for a given timestep being randomly shuffled prior to their consumption by the relevant agents.

The third argument serves a double purpose. As well as allowing the user to define the agents they would like included in their simulation, each two tuple in the list also contains a list of integers which correspond to the broadcast channels that the agent in question subscribes to throughout the simulation.

For a message to be sent directly to a particular agent, the sender is required to know that agent's integer ID. The current method employed by users is to hard code the relevant integer IDs within an agent's logic.

The integer IDs of each agent (which are assigned within the body of the `sim` function) begin from one and correspond to the order in which each wrapper is included in the list — with the zeroth ID being reserved for the simulator itself. For example, if there

were two agent wrappers included in the list passed to `sim`, the first wrapper would be assigned the ID of one and the second would be assigned the ID of two.

By hard coding the integer IDs of agents into their wrapper's logic users are exposing themselves to a major refactoring issue. Since the ID of each agent is tightly coupled to the order in which it appears in the call to `sim`, any amendment to the list could potentially result in having to refactor every hard coded integer ID within all other wrappers. This method is very inflexible for large simulations and as such it is of paramount importance that an alternative be established.

3.2 Analysis of Requirements

In order to overcome the limitations of the simulator described in the previous section, the requirements listed in the following subsections must be satisfied.

3.2.1 Simulator Extensions

The user must have the ability to:

- Specify the connections that exist and/or do not exist between agents.
- Specify the direction(s) of connections which exist between agents.
- Specify the messaging latency for each connection i.e. the number of timesteps it would take for a particular message to be sent between two agents.

The simulator must:

- Be aware of the user defined connections at all times and prevent messages from being passed between agents for whom a connection does not exist.
- Provide a mechanism such that messages and broadcasts are received by agents only after the user defined latency between the sender and recipient has passed.
- Remain backwards compatabile with previously encoded experiments.

3.2.2 Agent Wrapper Functions

The user must have the ability to:

- Define the identifiers of the other agents that wrappers will communicate with through a label rather than integer IDs.
- Provide a mapping from agent label to simulator ID for their experiment as a runtime argument.

- Have the option to utilise a preexisting function to generate a mapping between their custom defined agent labels and simulator integer IDs.

The previously mentioned example experiment (Figure 3.1) contained six agents and utilised the 'delay agent' approach for modeling the latencies between them. Although the worst case scenario for the number of delay agents is $f(6) = 6^2 - 6 = 30$, in actual fact only fourteen were required.

```
fr2tofr1delayid         = 1;  exch2tobroker1delayid = 8;   fr1exchid = 15;
fr1tofr2delayid         = 2;  fr1toexch1delayid     = 9;   fr2exchid = 16;
exch1tofr1delayid       = 3;  fr2toexch2delayid     = 10;  fr1id     = 17;
exch2tofr2delayid       = 4;  broker2toexch1delayid = 11;  fr2id     = 18;
broker1toexch1delayid = 5;  broker2toexch2delayid = 12;  broker1id = 19;
broker1toexch2delayid = 6;  exch1tobroker2delayid = 13;  broker2id = 20;
exch1tobroker1delayid = 7;  exch2tobroker2delayid = 14;
```

Figure 3.6: The hard coding of twenty agent IDs illustrating the unintuitive integer interface to refer to agents and the tight coupling to the order in which wrappers are passed to `sim`.

Figure 3.6 illustrates the hard coding which occurred for the integer IDs of each of the fourteen delay agents and the six other 'normal' agents of the experiment. Having to refer to the twenty agents via integers within each agent wrapper presents an unintuitive and confusing interface to the user. Furthermore, forcing the user to keep track of the ordering of agent wrappers is an unreasonable expectation to have.

Not only would the introduction of user defined labels aid in minimising this unnecessary complexity it would allow users to express intent within their code without having to know the ordering of their agent wrappers. This, in turn, would reduce the probability of the user introducing a semantic error within their logic and thus increase their productivity in the long run.

3.3 Design and Implementation

3.3.1 Simulator Extensions

MaxDelay, FunArg1, and FunArg2 as Runtime Arguments

As specified in subsection 3.2.1 the user must have the ability to describe the connections between the agents in their experiment (including their directions). Additionally

they must also be able to represent the messaging latency for each connection. Effectively, the user is required to encode their simulation graph.

Rather than imposing that the graph be represented through a specific data structure the chosen design is such that the required functionality remains loosely coupled to the implementation details. Users are required to include three runtime arguments within their simulation — a 'max delay' argument, a 'FunArg1' and a 'FunArg2'.

Figure 3.7 illustrates the type signatures for the three different runtime arguments. As is seen in Figure 3.7 the differing constructors associated with the Arg_t type all contain a value of type 'Str'. This is simply a string which is used as a key when performing look ups for arguments.

```
data Arg_t = ... | Arg (Str, Double) | FunArg1 (Str, (Int -> Int -> Int))
                 | FunArg2 (Str, (Int -> [Int])) | ...
```

Figure 3.7: Redacted Arg_t data declaration showing the type signature for the 'max delay' argument, FunArg1, and FunArg2 respectively.

An existing convention is to encode any numeric value using the Arg constructor as a double (regardless of whether it is an integer or real number). This aids in increasing the level of abstraction from 'Int' and 'Double' to simply be 'number' and thus reduces unnecessary complexity.

The simulator extensions require the user to provide a runtime argument, 'max delay', which defines the maximum possible message latency between two agents in the their experiment. Given that the value associated with the maximum delay within an experiment could only ever be an integer, following the existing convention, the max delay argument is attached to an Arg constructor as a double with the Str key of 'maxDelay'. The argument is required by the simulator for initialisation purposes as will be demonstrated later within this section.

The second required runtime argument is aptly called 'FunArg1' as its intention is to contain a function, rather than any specific value. As is seen in Figure 3.7 the function contained within FunArg1 must have the type signature of (Int -> Int -> Int). Additionally, the Str key of the argument must be defined as 'FunArg1'.

When given two Int agent IDs the function within FunArg1 should return the corresponding message latency between them with the direction of the connection being assumed to be 'from' the first agent 'to' the second. If no connection exists then the

function should instead return an error and halt the program for the user to begin debugging. Thus `FunArg1` provides a solution to the requirement that the simulator should enforce the communication topology of the graph (see subsection 3.2.1). A representative name of the function within `FunArg1` would be '`getTimestepDelay`'.

The third required runtime argument, `FunArg2`, also contains a function within it. In this case the internal function has the type signature of `(Int -> [Int])`. When given a broadcast channel ID it should return a list of IDs of the agents who are subscribed to that channel. If given an ID of a broadcast channel which does not exist the function should return an error to make the user aware of their mistake. Appropriately, the `Str` key of the argument must be defined as 'FunArg2'. Although it may seem redundant to name the key the same as the constructor for `FunArg1` and for `FunArg2` this is done intentionally to permit them to support other runtime arguments that contain functions with the corresponding type signature in the future. A representative name of the function within `FunArg2` would be '`getAgentsSubscribedToChannel`'.

The intention of `FunArg1` is to provide a mechanism for the simulator to retrieve the message latency between two given agent IDs. On the other hand, `FunArg2`'s usage is intended to provide the simulator with a method of retrieving the agent IDs of those agents subscribed to a given broadcast channel. As the simulator remains coupled only to the type of the functions passed as arguments the user remains free to utilise any data structure/methodology to encode their graph as they see fit.

MaxDelay, FunArg1, and FunArg2 Example

In order to illustrate the flexibility given to the user by requiring them to conform to specific type signatures rather than a more strict methodology consider the following. A user wishes to slightly modify the example experiment of Figure 3.1, and so draws an updated graph representation as seen in Figure 3.8.

The graph that the user is able to draw is not only much more representative of their intentions but also mirrors the new found functionality of the simulator (implementation details to follow in a later subsection). Since the graph is now directed, all of the connections between agents are clearly expressed and completely unambiguous.

Furthermore, it is extremely easy to differentiate between connections with symmetric latencies (such as 'Broker 1' — 'Exchange 1') and those with asymmetric latencies (such as 'Front Runner Server 1' — 'Front Runner Server 2'). Moreover, the user is

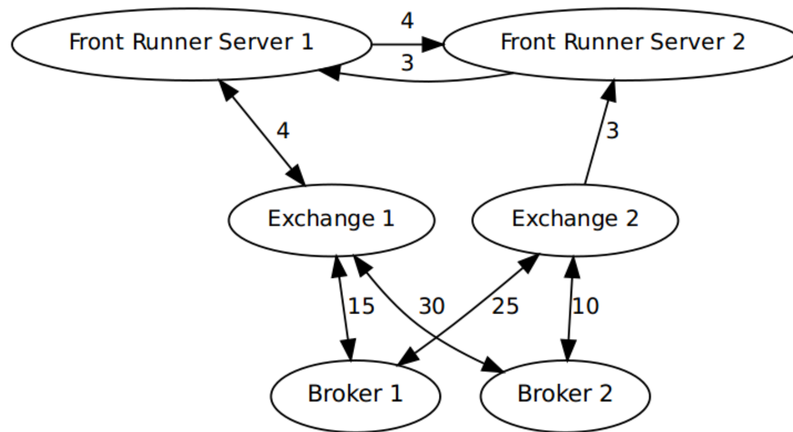

Figure 3.8: Updated graph representation of an example experiment.

able to express their intention for a given connection to be unidirectional as is the case between 'Exchange 2' and 'Front Runner Server 2'.

Once the graph topology of their experiment has been finalised, the user now has to provide the function contained within `FunArg1` — `getTimestepDelay`. The user can decide upon any data structure they feel comfortable with but in this case suppose that they decide to utilise an adjacency matrix representation of their graph implemented through an array[5].

Figure 3.9 illustrates how the user could encode the adjacency matrix of their graph utilising arrays. Firstly, they define each delay within a two dimensional list (of type `[[Maybe Int]]`) called '`delays`'. Then they utilise the `Data.Vector` module to convert the list of lists into a vector of vectors and bind the result to '`delaysAsVector`'. The `Data.Vector` module utilises an array as the underlying data structure of a `Vector` and so the user is guaranteed $O(1)$ retrieval.

Close inspection of Figure 3.9 brings about a number of considerations which are important to note. Firstly, the details surrounding all of the connections are made completely explicit — if a connection between two agents exists the latency is attached to a '`Just Int`' value and if it does not exist then a '`Nothing`' value is attached. It is important to make the distinction between a value of `Just 0` and a value of `Nothing`. In the case of the former although a connection does exist there is no latency attached to message passing, whereas in the case of the latter no connection exists at all.

[5] Alternative data structures could be a Map of Maps or another more suitable data structure from a library specialising in representing graphs.

```
delays :: [ [Maybe Int] ]
delays = [ {-0-Sim   1-FRS1   2-FRS2   3-Ex1    4-Ex2    5-Br1    6-Br2 -}
{-0-Sim -} [Just 0,  Just 0,  Just 0,  Just 0,  Just 0,  Just 0,  Just 0],
{-1-FRS1-} [Just 0, Nothing,  Just 4,  Just 4, Nothing, Nothing, Nothing],
{-2-FRS2-} [Just 0,  Just 3, Nothing, Nothing, Nothing, Nothing, Nothing],
{-3-Ex1 -} [Just 0,  Just 4, Nothing, Nothing, Nothing, Just 15, Just 30],
{-4-Ex2 -} [Just 0, Nothing,  Just 3, Nothing, Nothing, Just 25, Just 10],
{-5-Br1 -} [Just 0, Nothing, Nothing, Just 15, Just 25, Nothing, Nothing],
{-6-Br2 -} [Just 0, Nothing, Nothing, Just 30, Just 10, Nothing, Nothing]
         ]
delaysAsVector = Data.Vector.fromList (map Data.Vector.fromList delays)
```

Figure 3.9: Adjacency matrix encoding of example experiment graph.

Furthermore, the agent with the ID of zero is always defined as the simulator itself and there should be no delay between any agent and the zeroth agent[6] (this is the reason for the first column and row all having values of `Just 0`). Moreover, all values in the diagonal are `Nothing` apart from the connection between the zeroth agent and itself. This is a requirement in order to preserve backwards compatibility with the mechanism employed by the simulator in handling a `Hiaton`[7]. Once the user is content with their adjacency matrix they are required to provide the `getTimestepDelay` function and `FunArg1`, for which an example is shown in Figure 3.10.

```
1  getTimestepDelay :: Int -> Int -> Int
2  getTimestepDelay agentFrom agentTo = delay
3    where
4      agentFromDelayList  = delaysAsVector Data.Vector.! agentFrom
5      delay | agentFromDelayList Data.Vector.! agentTo == Nothing
6              = error ("getTimestepDelay: There is no delay specified"
7                ++"between "++(show agentFrom)++" and "++(show agentTo))
8            | otherwise
9              = fromJust (agentFromDelayList Data.Vector.! agentTo)
10 funArg1     = FunArg1 (Str "FunArg1", getTimestepDelay)
11 maxDelayArg = (Arg (Str "maxDelay", 30))
```

Figure 3.10: Example of the `getTimestepDelay` function, `FunArg1`, and max delay arguments.

In line four of the figure, the `Data.Vector.!` operator indexes into `delaysAsVector`

[6] Having a 'simulator agent' whose delay between other agents in both directions is zero is useful for debugging purposes.

[7] A Hiaton is an 'empty message' used by agents to indicate the absence of an output. To maintain backwards compatibility the fact that a `Hiaton` is automatically given a from and to ID of 0 necessitates that a connection exist between the simulator and itself.

as defined in Figure 3.9 to retrieve the 'from' agent's list of connections/latencies. This list is then offset by the 'to' agent ID and if a `Nothing` value is returned an error halts the program in order for the user to begin debugging. Otherwise the actual latency amount is extracted from the `Just` constructor using the `fromJust` function (line nine) which is defined as part of the `Data.Maybe` module within Haskell. The final two lines illustrate the creation of `FunArg1`, and the maximum delay argument which the user is required to include in the list of arguments passed to `sim`.

Following the construction of both the max delay and `FunArg1` arguments the user must create `FunArg2`. An example creation of `FunArg2` and the required function contained within it — `getAgentsSubscribedToChannel` is illustrated in Figure 3.11.

```
broadcastSubscriptionVectors
  = ((Data.Vector.fromList).genBCastSubscriptionLists) listOfAgents
getAgentsSubscribedToChannel :: Int -> [Int]
getAgentsSubscribedToChannel broadcastChannel
  = broadcastSubscriptionVectors Data.Vector.! broadcastChannel
funArg2 = FunArg2 (Str "FunArg2", getAgentsSubscribedToChannel)
```

Figure 3.11: Example `getAgentsSubscribedToChannel` function, and `FunArg2`.

The list of agents and their subscribed broadcast channels[8] is fed to the function `genBCastSubscriptionLists`[9]. Thus function is provided to meet the requirement specified in subsection 3.2.2 that the user have the option to utilise a preexisting function to generate a mapping between their custom defined agent labels and simulator integer IDs. The function returns a two dimensional list of integers (`[[Int]]`) where each inner list contains the IDs of those agents subscribed to the broadcast channel corresponding to that inner list's index.

This means that the definition of the `getAgentsSubscribedToChannel` simply returns the list contained at the index specified by the 'broadcastChannel' parameter. It uses the previously mentioned `Data.Vector.!` operator to return the list stored at the required offset in the vector.

Design of Simulator Extensions

The user goals of being able to specify the connections, directions, and messaging latencies for their experiment are all achieved through the use of the maxDelay, `FunArg1`

[8] See explanation of Figure 3.5 for a description of the 'list of agents'.

[9] See appendix Figure F.4 for code corresponding to the `genBCastSubscriptionLists` function.

and FunArg2 arguments. The only component which remains is the simulator implementation to comply with the specified message latencies.

For direct messages an extension to the simulator to model the latency between agents would be simple. Since direct messages form a one to one relationship from sender to receiver the simulator could simply utilise the (from, to) tuple attached to each message to look up and find the corresponding message latency (achieved through the use of the getTimestepDelay function within FunArg1). Once the latency is found the simulator could hold the message in an internal delay queue and only output it to the recipient after the lag expires.

However for broadcasts the (from, to) tuple forms a one to many relationship between the sender and receiver(s) because the 'to' in the tuple represents a broadcast channel rather than an agent ID. This begs the question that, since each connection in the one to many relationship could have a different value for the message latency, how should the simulator go about routing the broadcast to each individual subscriber?

The solution that will be adopted is for the simulator to look up and retrieve a list of agents who are subscribed to the message's broadcast channel (achieved through the use of the getAgentsSubscribedToChannel function within FunArg2). Then the broadcast would be 'exploded out' such that for each agent within the retrieved subscription list a new message, of type 'routed broadcast', is produced. As well as containing the contents of the original broadcast each 'routed broadcast' would form a one to one correspondence between the sender and receiver.

Since each routed broadcast would contain a (from, to) tuple with a one to one correspondence between sender and receiver, the simulator could use the tuple to look up the message latency and then hold the message in its internal delay queue just as it would for a direct message. As such, subscribers to broadcast channels would only ever receive routed broadcasts, with the original broadcast message being discarded after being 'exploded out'.

Extending the Simulator's Internal State and Initialisation Function

Figure 3.12 contains the current definition of a simulator state in a given timestep (Simstate_t). Figure 3.13 illustrates how the first empty simulator state is generated through the use of the sim_emptystate function. This function is only ever called once, at the beginning of a simulation, and remains private to the simulator with no

access being granted to it for the user.

```
type Simstate_t = ([[Msg_t]],    Int,       [Msg_t],         [[Msg_t]] )
--          Direct Messages, System Time, Harness Messages, Broadcasts
```

Figure 3.12: Current type synonym definition for the simulator state.

```
sim_emptystate :: [(Agent_t,[Int])] -> Int -> Simstate_t
sim_emptystate [] hbrc = ([[]], 0, [], (map f [0..hbrc]))
  where
   f x = []
sim_emptystate (a:as) hbrc = (([]:x), b, c, br)
  where
   (x,b,c,br) = sim_emptystate as hbrc
```

Figure 3.13: Current function definition for initialising an empty simulator state.

When initialising the first empty simulator state the `sim_emptystate` function takes in the user defined list of agents[10] and an integer representing the highest defined broadcast channel. The first element in the tuple of a `Simstate_t` is a list created in such a way that it contains an inner list for each agent within the simulation.

The mechanism for generating the first element in the tuple is as follows: when called, the `sim_emptystate` function recurses through each agent in the user defined agent list adding an empty list for that agent. Once the base case is reached an empty list is added to account for the zeroth agent: the simulator itself.

For example, assuming that a user defined agent list contained six agents, the number of inner lists generated for direct messages (the first element of the `Simstate_t` tuple) would be seven — six for the user defined agents, and one for the zeroth agent.

The purpose of the first element in the `Simstate_t` tuple is to act as the data structure containing direct agent to agent messages. It must comply with the constraint that, for a given `Simstate_t`, each message should reside within the inner list that corresponds to the message recipient's agent ID. For example, any message with the intended recipient of agent zero must be contained in the zeroth inner list.

The second element of the tuple corresponds to the simulator time, which for the initialisation state should always be zero.

[10] See Figure 3.5 and the surrounding explanation for a description of the user defined agent list.

The third tuple element is a list of 'harness messages'. These are messages sent directly to the zeroth agent. Since direct messages are also contained in the first element of the tuple, for a given Simstate_t, the third tuple element (harness messages) is equivalent to the head of the first (direct messages). This repetition is an intentional mechanism used to enhance the visibility of messages sent to the zeroth agent — which for most cases are for debugging purposes.

The fourth and final element in the Simstate_t tuple is the data structure containing the broadcast channels for the simulation. Once sim_emptystate reaches its base case the function f x = [] is mapped to each element of a generated list (map f [0..hbrc]). The generated list in question is the list of natural numbers from zero to the highest broadcast channel of the experiment. By mapping the aforementioned function each natural number is converted to an empty list — resulting in a list containing an inner list for each broadcast channel within the simulation.

The broadcast message data structure imposes a constraint similar to the one enforced on the first element of the Simstate_t tuple (direct messages). For a given Simstate_t, each broadcast message should reside within the inner list that corresponds to the message's defined broadcast channel. For example, any broadcast message output to channel five must be contained in the inner list with index five.

From the design detailed above it stands to reason that Simstate_t needs to be extended to account for the extensions (internal message delay queues and routed broadcasts). The extended Simstate_t can be seen in Figure 3.14.

```
type Simstate_t
  = ([[Msg_t]], Int, [Msg_t], [[Msg_t]], [[Msg_t]], [[Msg_t]])
-- Direct Messages, Timestep, Harness Messages, Broadcasts,
-- Delay Queues, Routed Broadcasts.
```

Figure 3.14: Extended type synonym definition for the simulator state.

The Simstate_t type (Figure 3.14) has been extended to include two more elements in the tuple, and the sim_emptystate function (Figure 3.15) has been extended to also take the user defined list of runtime arguments as a parameter.

```
sim_emptystate :: [(Agent_t,[Int])] -> [Arg_t] -> Int -> Simstate_t
sim_emptystate [] args hbrg
 = ([[]], 0, [], (map f [0..hbrg]), (map f [0..maxDelay]), [[]] )
  where
   f x = []
   maxDelay | (arg_lookup "maxDelay") args == EmptyArg = 2.0
            | otherwise
              = arg_extract_arg_value ((arg_lookup "maxDelay") args)
sim_emptystate (a:as) args hbrg
 = (([]:x), b, c, br, delayQueues, ([]:y))
  where
   (x,b,c,br, delayQueues, y) = sim_emptystate as args hbrg
```

Figure 3.15: Updated function definition for initialising an empty simulator state.

The logic attached to the four preexisting elements of the Simstate_t tuple within the sim_emptystate function remains unchanged — the new code in the function body relates only to the extensions.

The first new addition to the Simstate_t tuple is that of delay queues — which are generated using the exact same logic as for broadcast channels. When the base case of sim_emptystate is reached, a lookup for the max delay runtime argument is performed. If the lookup results in the return of the EmptyArg then the user has not attached a max delay argument to their experiment. In that case, in order to retain backwards compatibility, the maximum delay is arbitrarily set to 2.0[11] and the simulator enters 'compatibility mode'[12]. Otherwise the value of the Double is extracted and an inner empty list for each natural number between zero and the maximum delay is created within the delay queues data structure. Furthermore, the simulator enters a mode which supports delay queues and routed broadcasts.

The idea behind the delay queues is to only allow agents to consume the messages which reside within its zeroth inner list. Also, newly produced agent messages are placed into the inner list which corresponds to that message's associated latency. After a timestep has passed, each message is promoted such that it resides within an inner list one level closer to the zeroth, until it finally reaches the zeroth and is output. For example a message where the latency between the two agents is defined as five timesteps would be placed inner list with index five, and the next timestep

[11] See Figure 3.7 and its corresponding explanation to understand why this value is a Double.

[12] Compatibility mode is explained further within this section. Essentially the simulator operates in exactly the same way as prior to these extensions.

would be promoted to the fourth.

This simple design provides an intuitive understanding that for a given timestep the head of the delay queues corresponds to the messages to be output, and the tail corresponds to messages which need to be lagged for at least one more timestep.

The second and final extension to the `Simstate_t` tuple is the inclusion of a routed broadcasts data structure. The data structure contains the same number of empty inner lists as there are user defined agents (as well as one for the zeroth agent). The logic for producing this is exactly the same as for direct messages. The constraint put on the routed broadcasts data structure is also exactly the same as for direct messages — for a given `Simstate_t` each message should reside within the inner list that corresponds to the message recipient's agent ID.

Extending the State Updating Function

An integral part of the design of the simulator extensions is the updating of the component which takes the output messages of all agents and places them within the `Simstate_t` tuple so that they can be consumed by the intended agent. For example, suppose a direct message is sent to the agent with an ID of one. Under the current scheme, the message would be moved to the direct messages data structure within the `Simstate_t` tuple. It would be moved such that it resided within the inner list at index one — to match the message's intended recipient. At the next timestep the message would be available to agent one for consumption. This mechanism of 'gluing' agent messages to their recipients needs to be updated to account for the user defined latencies and routed broadcasts.

For the simulator to successfully send routed broadcasts and utilise its delay queues the user has to have attached the max delay, `FunArg1` and `FunArg2` arguments. Figure F.5 in the appendix illustrates how if any of the three arguments are missing the simulator enters 'compatibility mode'. In compatibility mode the simulator acts in the same was as it would have prior to the extensions — with no support for routed broadcasts/delay queues.

In the case where the three argument are present the simulator enters a mode which caters for routed broadcasts and the delay queues. The logic for the function which updates the `Simstate_t` tuple every timestep is highlighted in algorithm 1 (see display) with the actual implementation code in the Appendix (Figure F.7).

Input: `:: Int -> [Arg_t] -> Simstate_t -> [[Msg_t]] -> [Int]`

Time -> Runtime Arguments -> Simstate_t -> New Messages -> Random Numbers

Output: `:: Simstate_t`

(Direct Messages, Timestep, Harness Messages, Broadcasts, Delay Queues, Routed Broadcasts)

```
1  begin
2      Retrieve the functions getTimeStepDelay and
       getAgentsSubscribedToChannel from FunArg1 and FunArg2 respectively.
3      Filter the agent to agent messages and broadcast messages creating two
       separate lists.
4      Insert every direct message into the delay queues.
5      Create and insert each routed broadcast into the delay queues.
6      Take the head of the delay queues. These are the messages which have had
       their associated lag expire and need to be output.
7      Add an empty list to the end of the tail of the delay queues to maintain its
       size. This forms the final delay queues which are included in the output
       Simstate_t tuple.
8      Split the messages to be sent into two lists (by filtering) — agent to agent
       messages and routed broadcast messages.
9      Move the messages contained within the filtered lists (direct messages and
       routed broadcasts) so that they reside within the inner lists corresponding
       to their 'to' ID. Direct messages should be moved to the first element of the
       Simstate_t tuple and routed broadcasts to the sixth. This is done so that
       the messages are in position for the relevant agents to consume.
10     Any messages which are sent to agent zero are 'harness messages'. Retrieve
       the harness messages from the head of the updated Simstate_t tuple. This
       should form part of the output Simstate_t tuple.
11     if User Has Included Randomise Runtime Argument then
12         Randomise all of the messages and routed broadcasts which will be
           consumed in the next timestep.
13     end
14     else
15         Do not randomise output.
16     end
17     Output the updated Simstate_t tuple.
18 end
```

Algorithm 1: Logic flow for updating `Simstate_t` to account for routed broadcasts and delay queues.

The final piece of the extensions puzzle is to update the function responsible for the filtering of the potentially infinite list of all messages. As stated previously, this filtering is what allows users to code their wrappers without having to worry about how the agent actually receives its messages. The mechanism must be updated because, in addition to direct messages and broadcasts, each agent wrapper now has the ability to also receive routed broadcasts.

```
g agentID simState
  = ((fromIntegral (sim_gettime simState)), sim_getmymessages simState agentID,
     (concat (map (sim_getmybroadcasts simState) brcs))
     ++ (sim_getmyroutedbroadcasts simState agentID))
```

Figure 3.16: Updated filtering of the relevant messages, broadcasts and routed broadcasts for an agent.

Figure 3.16 illustrates the updated filtering function, `g` which is contained within the body of the `sim` function (see Figure 3.5). When given an agent ID and a `Simstate_t` tuple, it extracts the state's internal time, direct messages for that agent, and broadcast messages to channels the agent is subscribed to (`brcs`). The extension to the original function results in the additional extraction[13] of the routed broadcasts, and the concatenation of the returned list with the extracted broadcasts. This ensures backwards compatibility for previously encoded experiments.

Thus the simulator goal of being aware of the user defined connections and preventing invalid messages from being passed is achieved through `FunArg1`. The goal of providing a mechanism for messages to be received only after their lag has expired is achieved through `FunArg1`, `FunArg2` and the extensions to `Simstate_t` (routed broadcasts, and delay queues). Finally, after all of these extension, the simulator remains backwards compatible through the use of its explicit compatibility mode.

3.3.2 Decoupling Agent IDs from Wrappers

FunArg3

The flexibility provided by only enforcing the user to conform to a type signature rather than a specific data structure continues in the design of `FunArg3`. `FunArg3` attempts to address the problem of having to hard code agent IDs into the logic

[13] The extraction function definitions are included within the Appendix subsection F.4.1.

of agent wrappers by containing a function, getCorrespondingAgentIdentifier, within it.

```
data Arg_t
  = ... | FunArg3 (Str, (AgentIdentifier_t -> AgentIdentifier_t)) | ...
data AgentIdentifier_t =  AgentID Int | AgentLabel String
```

Figure 3.17: FunArg3 and AgentIdentifier_t type signatures.

Figure 3.17 highlights the type signature of FunArg3 by illustrating a redacted version of the Arg_t data declaration. Additionally it shows the data declaration for the AgentIdentifier_t type. An AgentIdentifier_t encapsulates the ways in which an agent of a simulation could potentially be identified. Thus referencing an agent should be possible through their integer ID (AgentID Int) or their user defined label (AgentLabel String).

The idea of the getCorrespondingAgentIdentifier function is that when given an AgentID Int it should return the AgentLabel String of the agent with the specified integer ID. On the other hand, if given an AgentLabel String it should return the AgentID Int of the agent with the corresponding label. If an invalid agent identifier is given, then the function should halt the program with an error.

The chosen design means that instead of declaring the user defined list of agents to be of the type [(Agent_t, [Int])] as is currently the case, users define it to be of the type [(String, (Agent_t, [Int]))] where the String is the label given to that particular wrapper.

The user then utilises the provided function generateAgentBimap[14] to generate a bidirectional mapping between the integer agent IDs and their labels. Thus the user becomes free from having to maintain a reference of the ordering of their agents and are only required to provide the getCorrespondingAgentIdentifier function within a FunArg3 argument.

An example implementation of the getCorrespondingAgentIdentifier function is given in Figure F.22 and could potentially be utilised by the user in order to provide the FunArg3 argument. Although users now define their agent list to be of the form [(String, (Agent_t, [Int]))], in order to maintain backwards compatibil-

[14] See Figure F.21 for corresponding code.

ity, the type of the agent list passed to `sim`[15] cannot be changed and must remain as `[(Agent_t, [Int])]`,

As illustrated in Figure 3.18 this does not present a problem, as the user can simply make use of the provided function `transformForSim`[16] when calling `sim`. The `transformForSim` function converts the user defined agent list to the required type for the `sim` function. The bidirectional mapping between agent labels and integer IDs remains within `FunArg3` for future use.

```
sim 60 runtimeArguments (transformForSim userDefinedAgentlist)
 where
  runtimeArguments = [ ... funArg3 .. ]
  funArg3 = FunArg3 (Str "FunArg3", getCorrespondingAgentIdentifier)
  userDefinedAgentList
   = [ ("Exch_1", (exchange1Wrapper, [0]    )),
       ("Exch_2", (exchange2Wrapper, [0]    )),
       ("Broker", (brokerWrapper,    [1,2]  )) ]
```

Figure 3.18: Redacted example call to `sim` for sixty timesteps including `FunArg3`.

Given that all agents have access to the runtime arguments they can utilise `FunArg3` to extract the agent IDs of those who they need to communicate with. Consider the example experiment in Figure 3.18, and suppose the broker wrapper is required to communicate with the two exchanges with labels 'Exch_1' and 'Exch_2'.

Figure 3.19 illustrates an example usage of `FunArg3` to extract the IDs of the two exchange agents dynamically rather than attaching hard coded integers within the wrapper. In addition to being relieved of having to hard code agent IDs into their wrapper logic, by using `FunArg3`, the user is also relieved of the need to maintain an ordering in their agent list. This means that if the order of agents in the list defined in Figure 3.18 was to change, the code would continue to work without the need of any refactoring. Therefore, `FunArg3` presents a major improvement in the overall usability of the simulator.

[15] `sim :: Int -> [Arg_t] -> [(Agent_t, [Int])] -> IO()`
[16] See Figure F.20 for corresponding code.

```
brokerWrapper :: Agent_t
brokerWrapper agentstate args ((time, messages, broadcasts) : rest) myid
 = ( ( [msg] ) : (brokerWrapper (Emptyagentstate) args rest myid) )
   where
    funArg3 | (arg_lookup "FunArg3" args) == EmptyArg
      = error "Your simulation has no BiMap (FunArg3)."
            | otherwise = (arg_lookup "FunArg3" args)
    getCorrespondingAgentIdentifier = arg_extract_FunArg3_value funArg3
    (AgentID exch1ID) = getCorrespondingAgentIdentifier (AgentLabel "Exch_1")
    (AgentID exch2ID) = getCorrespondingAgentIdentifier (AgentLabel "Exch_2")
    msg = ... -- Messages for exchanges can be sent using exch1ID exch2ID.
```

Figure 3.19: Usage of FunArg3 to retrieve the agent IDs of the two exchanges within the simulation.

As is seen in Figure 3.19 the goal of the user to be able to define their wrappers to communicate through the use of labels rather than integer IDs is achieved through the usage of `FunArg3`. The 'to' integer required in order to output a message is retrieved by the `getCorrespondingAgentIdentifier` function rather than hard coding the integer IDs within the logic of the wrapper.

The requirement that the user be provided with a function for generating the mapping between agent label to simulator ID is met through the `generateAgentBimap` function. Therefore all of the listed requirements for the extensions to the simulator and agent wrappers have now been met and must be validated.

3.4 Testing and Validation

3.4.1 Simulator Extensions

The extensions to the simulator have been quite extensive and so it is of the utmost importance that their validity be established. As a result a number of tests have been devised to determine the correctness of each component of the extensions.

The entire simulation depends on the appropriate initial empty state being generated and so the `sim_emptystate` function is to be tested in the following way:

- Failing to provide the 'maxDelay' runtime argument. This should simply result in the generation of an arbitrary number of delay queues. The rest of the generated `Simstate_t` should be as normal.
- Provide the 'maxDelay' runtime argument. This should result in the creation

of the correct number of delay queues. The rest of the generated `Simstate_t` should be as normal.

For the `sim_updatestate` function the following tests have been devised:

- Failing to provide all three runtime arguments (maxDelay, `FunArg1`, `FunArg2`). This should result in the simulator entering compatibility mode.
- Running a previously encoded experiment to intentionally enter compatibility mode. The trace file output should be compared to the output of the simulator prior to the extensions and there should be no difference between the files. If that is the case then this will confirm the correctness of compatibility mode.
- Encoding an example experiment which makes use of the simulator's newly created routed broadcast and delay functionality (this in turn will also be testing the three runtime arguments). The experiment should be devised such that the correct timestep each message should be received in is known in advance. Once the experiment is conducted the simulator's output should be compared to what is known to be accurate. If they match it will validate the correctness of the new delays and routed broadcasts mode. Additionally, it will confirm the correct coding of the maxDelay, `FunArg1`, `FunArg2` runtime arguments.

3.4.2 The FunArg3 Runtime Argument

For the `FunArg3` runtime argument the following test has been devised:

- Encode an example experiment whose agents all use of `FunArg3` to retrieve the IDs of the agents they will be communicating with. This will confirm the correctness of the `FunArg3` runtime argument.

All the listed tests have been carried out successfully and thus the extensions to the simulator have been verified to be correct. The associated code for the tests can be found in the appendix section E.1. This result means that future users can make full use of the extensions to the simulator whilst relying on their soundness.

Chapter 4

The Messaging Infrastructure

4.1 Background and Requirements Capture

The Current Messaging System and its Limitations

```
data Msg_t = Hiaton
 | Broadcastmessage (Int,Int) Broadcast_t | RoutedBroadcast (Int, Int) Msg_t
 | Ackmessage (Int,Int) ...                | Cancelmessage (Int,Int) ...
 | Ordermessage (Int,Int) ...              | ...
data Broadcast_t = Orderlistbroadcast ... | Tradebroadcast ... | ...
```

Figure 4.1: Redacted Data Type Declaration for `Msg_t` and `Broadcast_t`.

Messages which an agent can send or receive are encapsulated by the `Msg_t` data type. Figure 4.1 shows redacted versions of the data declarations for `Msg_t` and `Broadcast_t`, and illustrates the ease of which they can be extended. For example, in order to expand the simulator's messaging system to cater for routed broadcasts the simple addition of '`| RoutedBroadcast (Int, Int) Msg_t`' to the data declaration of `Msg_t` was all that was required. Additionally, Figure 4.1 demonstrates how every `Msg_t` (apart from a `Hiaton`) must include a tuple `(Int, Int)` within it indicating the IDs of the sender and receiver of the message.

As discussed previously, all agents within a given simulation have the type `Agent_t` and as a result their output takes the form of a list of lists of messages (`[[Msg_t]]`). Each inner list within the agent's output corresponds to their messages for a given timestep. The internal design of the simulator mandates that each agent output something for all timesteps regardless of whether their logic determines they need to issue a message or not. This constraint does not actually restrict agents in any way as for the timesteps in which it is determined that an agent has nothing to output they can simply generate an empty list or a list containing a `Hiaton`.

Prior to this project only a single exchange agent modeling the Chicago Mercantile Exchange (CME) has been created for use within simulations. Accordingly, the simulator's messaging system has continually evolved to accommodate the users of that exchange. As seen in Figure 4.1 messages have been established for order creation (`Ordermessage`), cancellation (`Cancelmessage`), acknowledgment (`Ackmessage`), the broadcasting of the exchange's limit order book (`Orderlistbroadcast`), and trade executions (`Tradebroadcast`).

Although they serve their purpose, the contents of these messages have been heavily simplified compared to those actually used in the real world. The decision to create simplified messages made perfect sense at the time since the intention was to conduct experiments involving only one exchange.

However, as of now it has become a major ambition to eventually model the entire US national market system within the simulator. This goal, for reasons discussed below, would be best served by adopting an existing messaging protocol rather than attempting to create a simplified one.

The FIX Messaging System

In the financial ecosystem message passing between two parties requires the encoding of a large amount of information. At the very least there needs to be an explicit indication of the purpose of the message — whether it is for a new order/to relay a trade execution etc.

For a given type of message a minimum amount of data would be expected to be transmitted within it. For example, among other things a new order message would be expected to contain instructions on whether the order is to buy/sell and the amount of shares involved. Additionally, the message could contain any number of optional instructions such as a discretionary amount or routing constraints.

Each exchange within any financial district defines its own distinct optional instructions for their members and thus the notion of a 'simplified' messaging system is an illusion. Supposing a 'simplified' messaging system were created it would have to be extended and amended for each additional exchange added to the simulator.

Furthermore, the danger exists that the implemented system may fail to capture an important component resulting in its entire redesign once the missing element is discovered. This is a major drawback which could potentially hinder the further

development of exchange agents within the simulator.

Therefore, rather than attempt to simplify and recreate a messaging system containing all of the relevant possible options, a wiser decision would be to model an existing protocol used throughout the industry. A major benefit to working with an industry standard messaging system is the potential reuse of the same code. The current simplified messages created for the CME (see Figure 4.1) do not follow a specific standard and thus cannot be reused for any future exchange agents. This limitation of the simulator's current messaging infrastructure would be mitigated through the adoption of a standard protocol.

The Financial Information eXchange (FIX) protocol is a messaging system used throughout the industry. Figure 4.2 illustrates the typical communication flow between institutional investors, brokerages, and liquidity sources who utilise FIX.

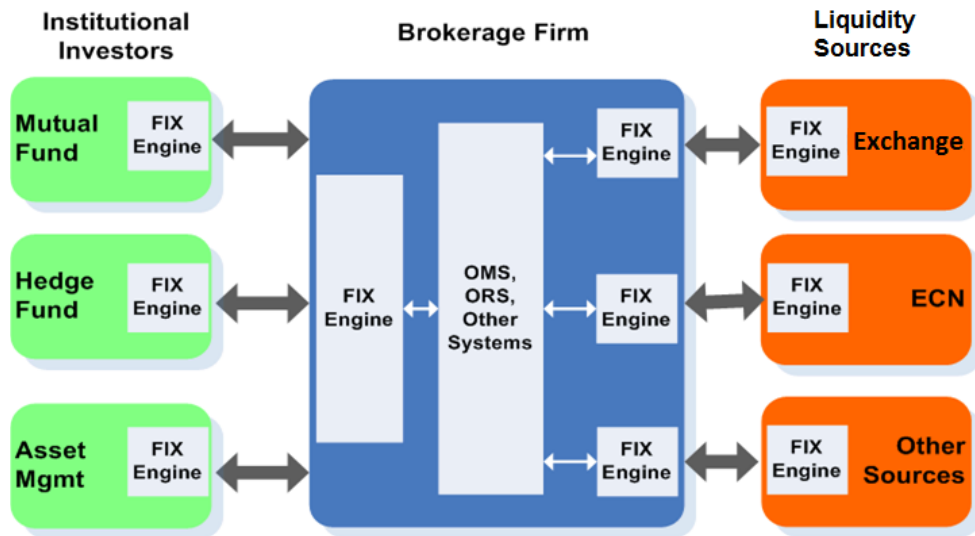

Figure 4.2: Example illustration of the industry's usage of FIX [Vali, 2009].

Each individual/group wanting to communicate uses a FIX engine to manage their network connection and encode/decode any message. All participants in the system, including brokerage systems responsible for order management (OMS) and order routing (ORS), are expected to only send/receive messages defined by the protocol.

FIX defines a number of message types such as 'New Order Single'/'Order Cancel Request' and each message type contains a number of predefined tags. Tags are simply key-value pairs where the key corresponds to the tag's integer identifier and the value takes the form of one of the defined qualifiers.

External entities may also define tags which can be used in tandem with existing ones in the FIX standard. Other institutions may choose to adopt those tags within their messages if they see fit to do so. Externally defined tags usually have IDs (keys) between 5,000 and 9,999 and can take any value specified as a valid qualifier by the adopting entity. For example, exchanges which are owned by BATS Global Markets define a tag with the ID of 9303 to allow the user to enter a BATS specific routing instruction. An illustration of the use of tags to construct a message is given in Table 4.1.

4.2 Analysis of Requirements

Tag Name	**Tag Type**	**Tag Key**	**Tag Value/Qualifier**
OrderQty	FIX	38	100
OrdType	FIX	40	Market
Side	FIX	54	Buy
RoutingInst	BATS	9303	Book Only
...	...	...	...

Table 4.1: Example redacted BATS new order message [BATS, 2015].

Table 4.1 illustrates how a FIX message is composed of a number of tags. Since tags are uniquely identified by their key it is not necessary to include their type and name in the message — they are shown in the table for clarity.

The example is a redacted new order single message to buy one hundred shares of a particular stock from a BATS exchange. The routing instruction of ‘Book Only’ indicates the order should stay within the exchange’s book and not be routed i.e. it should only be executed against orders to sell that occur on the BATS limit order book, and not forwarded on to another exchange.

An important point to make is that although FIX defines the acceptable values for a tag, certain qualifiers may be rejected by a particular entity. For example, FIX defines over twenty separate order types which could potentially form a valid qualifier for the ‘OrdType’ tag. Valid FIX ‘OrdType’ qualifiers such as ‘Forex Limit’ and ‘Forex Swap’ are not accepted on BATS exchanges as foreign exchange (Forex) order types are completely irrelevant to them.

In other words, the qualifiers which are accepted by an entity could potentially form a subset of those which are defined by FIX. This cements the requirement that any implementation of FIX must include a mechanism for defining the subset of qualifiers which are valid for each entity.

Another important consideration is the fact that there are many versions of FIX currently used in production. Therefore it will be necessary to conduct an assessment and determine the version of FIX which the simulator will implement. Newer FIX versions are, for the most part, simply supersets of the previous ones and so upgrading to a later version should not be a difficult task.

The final consideration relating to FIX is to do with the large number of tags the standard defines. FIX v.5.0 contains over one thousand tags — many of which would be irrelevant for any future simulation. For example, the size and checksum tags are completely unnecessary since the messages will remain within the simulator and not be transported over a network.

Given that the simulator's implementation of FIX only needs to include a subset of the standard it is necessary to establish a criterion for filtering out the irrelevant tags. The potential problem associated with only implementing a subset of the protocol is that an important tag could be overlooked and remain unimplemented. To combat this, the implementation must be created such that it is extremely easy for a user to understand and extend as needed.

From the above discussion the requirements for extending the simulator's messaging infrastructure to incorporate the FIX protocol become:

- Decide the FIX version which should be implemented.
- Create a criterion which determines the subset of FIX tags which should actually be implemented.
- Introduce a system for creating entity specific tags as well as FIX tags. The system must be easily extendable and present a consistent interface to the user.
- Implement the relevant FIX tags which meet the conditions of the criterion.
- Define a system for specifying the subset of tag qualifiers which are valid for each possible entity.

4.3 Design and Implementation

In order to establish which version of FIX to model within the simulator a survey of what is utilised by the main exchanges of the NMS is necessary. As of August 2015 the BATS FIX Specification [BATS, 2015] states that all four BATS owned exchanges use FIX 4.2. The NYSE FIX Gateway [NYSE, 2015], Nasdaq [NASDAQ, 2015], and the Chicago Stock Exchange [CSE, 2015] state that they support the FIX 4.0, 4.1 and 4.2 standards.

As a result of inspecting the FIX specifications of the main NMS exchanges the decision has been taken to model the protocol against version 4.2. The specification for the 4.2 standard defines over four hundred tags and so the next step is to create a criterion through which relevant ones to implement can be determined.

A potential approach to determining the tags in which to implement could be to study each one sequentially and consider its utility to a user of the simulator. Those tags which are established to possess the greatest utility would be implemented and the irrelevant ones would not. Whereas this method has an advantage of resulting in only the most important tags being created it cannot be chosen as it would be impossible to complete within the given time constraints.

A far better method would be a two pronged incremental approach. Firstly, to only implement those tags which are needed for the most frequent operations — order creation, cancellation etc. Secondly, to create the system such that users will find it extremely easy to add future tags on an ad-hoc/as needed basis.

Although important tags may initially be absent from the implementation they are bound to eventually be captured due to the approach's incremental nature. Furthermore, this model remains completely possible to implement within the given time constraints, and as such is the method which will be adopted. Thus the relevant subset of tags to implement becomes those which are defined as part of the most common messages — order creation, modification, cancellation etc.

An important point to make is that all of the additions being made to the messaging infrastructure will not break any existing code — previous experiments and agents will continue to function as normal. However, since these would have been coded using the preceding simplified messaging system they will need to be rewritten if the

user wishes to make use of the FIX extensions.

4.3.1 Exploiting Haskell's Type System

Enforcing the validity of tag qualifiers for each entity is an onerous responsibility and as such it requires the utmost attention. Therefore as part of the design, the implemented FIX messaging protocol deliberately makes extensive use of Haskell's type system so that the compiler automatically tests the soundness of each program.

The first part of the design is to declare the entities which will accept messages containing FIX tags. Figure 4.3 shows a data declaration containing only the BYX and NYSE exchanges but this could easily be extended to accommodate any other entity such as a broker or another exchange.

Table 4.1 previously illustrated how a message could contain a tag created by an external entity such as BATS. Thus, the next step in the design is to compose a data declaration to encapsulate the different types of tags which could exist within a message. The declaration constructed, shown in Figure 4.3, currently only differentiates between `FIXTag`s and `BATSTag`s but could easily be extended as well.

```
data Entity  = BYX    | NYSE    | ...
data TagType = FIXTag | BATSTag | ...
```

Figure 4.3: Data declarations for `Entity` and `TagType`. Ellipses indicate the easy extensibility of the declarations.

Figure 4.4 illustrates the record declaration for a tag with the fields being the tag's type (`TagType`), name (`String`), ID (`Int`), and value (`TagValue a`). The declarations for the tag (`Tag a`) and its value (`TagValue a`) are intentionally polymorphic so that they can be used flexibly to contain any future qualifier. The `EmptyTagValue` is defined to allow for the creation of 'neutral' tags with the intention of being altered with a `TagValue a` in the future.

```
data Tag a = Tag { tagType :: TagType, tagName  :: String,
                   tagID   :: Int     , tagValue :: TagValue a }
data TagValue a = EmptyTagValue | TagValue a
```

Figure 4.4: Data declaration for a tag value and the tag qualifier class.

The final piece of the puzzle is to create a mechanism for defining the qualifiers/values that a tag can take and the subset of the values which each entity accepts. An ideal

solution would also present an intuitive interface to the user who, at a glance, would be able to explicitly delineate the relevant information for each entity.

The design decision taken is to capture the subsets which form the valid qualifiers of an entity through the usage of a function rather than attempting to develop a solution relying on the type system. The reasons for this choice are as follows: firstly subtyping, which is the obvious way to express this in a type system, is not idiomatic Haskell[1] and secondly any subtyping implementation would rely upon the usage of language extensions[2] which, in their current state, would provide a complicated interface to the user. Thus adopting a type system solution would act to hamper the goals of providing ease of use and extensibility.

Rather than illustrate the final structure of the actual implementation, the following discussion will first consider the alternative methods in which it could have been done. This is done intentionally as it highlights the reasoning behind the decisions which have been taken for the definite implementation.

```haskell
data OrdType = Market | Limit | ...
validOrdTypeQualifier :: Entity -> OrdType -> Bool
validOrdTypeQualifier BYX qualifier  | qualifier == Market = True
                                     | ...
validOrdTypeQualifier NYSE qualifier | ...

data TimeInForce = Day | GoodTillCancelled | ...
validTimeInForceQualifier :: Entity -> TimeInForce -> Bool
validTimeInForceQualifier BYX qualifier  | qualifier == Day = True
                                         | ...
validTimeInForceQualifier NYSE qualifier | ...
```

Figure 4.5: Potential implementation of the `OrdType` and `TimeInForce` tags.

A potential solution would be for each tag to have an individual data declaration which enumerates all of its possible qualifiers. Additionally, a function would be defined for each of the tags which captures the relevant subset of valid qualifiers for all of the entities within the implementation. Figure 4.5 provides an illustration of how this type of implementation would look for two of the FIX tags (`OrdType` and `TimeInForce`).

[1] See `http://stackoverflow.com/questions/9204158/subtypes-for-natural-language-types` for a discussion.

[2] For example, see the refinement type extension: `http://goto.ucsd.edu/~rjhala/liquid/haskell/blog/blog/2013/01/01/refinement-types-101.lhs/`

Each tag in Figure 4.5 has a data declaration which enumerates all of the possible values. For example, in the case of the `OrdType` tag the (redacted) enumeration includes the qualifiers '`Market`' and '`Limit`' and for the `TimeInForce` tag the (redacted) enumeration includes the qualifiers '`Day`' and '`GoodTillCancelled`'.

Furthermore, both tags also have a function defined which, when given an `Entity` and a tag qualifier, returns `True` if that value is valid for the given `Entity` and `False` otherwise. For example, for `BYX` the `validOrdTypeQualifier` function returns `True` for the `Market` qualifier because it is permitted on the `BYX` exchange. Although the rest of the function is redacted (as is the body of the `NYSE` pattern match) it is clear to see how it could be extended to fully capture the subset of values which are valid for each `Entity`. The `validTimeInForceQualifier` function is defined in a similar manner for the `TimeInForce` tag.

The usage of this design to implement FIX is sub optimal for a number of reasons. Firstly, the user will be required to call a separate function to check the validity of a qualifier for each tag. Although for two tags it is not a problem, having to remember/look up the differing function names for each individual tag could prove to be a rather labourious task in the long run. A design which allows the usage of a single function for all tags would be much more flexible and convenient.

Secondly, there is no way in which a user could query the implementation to find out which tags have actually been created. The ability to run such a query would be essential for users who wish to extend the messaging system. In the case of two tags existing side by side as in Figure 4.5 this is not a problem, but at any given time a user may wish to gain a high level overview of the simulator's entire FIX implementation and this cannot be done through the above methodology.

For example, if dozens of tags were implemented and spread across multiple modules[3] the user would either have to manually search each source file directly to find out if a tag exists or use a command line tool such as `grep` to attempt to find it. A design which allows the immediate listing of all implemented tags without having to resort to `grep` would act to facilitate much easier extensibility.

In the case of both of these problems a solution exists in the form of a type class. Figure 4.6 illustrates how the same two tags (`OrdType` and `TimeInForce`) could be

[3] This would actually be the case in any implementation as separate modules would be created to contain each type of tag (`FIXTags`/`BATSTags` etc.).

implemented by using a type class named `TagInterface`.

```
data OrdType = Market | Limit | ...
data TimeInForce = Day | GoodTillCancelled | ...

class TagInterface tag where
  validQualifier :: Entity -> tag -> Bool

instance TagInterface OrdType where
  validQualifier BYX qualifier    | qualifier == Market = True
                                  | ...
  validQualifier NYSE qualifier   | ...

instance TagInterface TimeInForce where
  validQualifier BYX qualifier    | qualifier == Day = True
                                  | ...
  validQualifier  NYSE qualifier | ...
```

Figure 4.6: `TagInterface` type class with separate data declarations as a solution.

In Figure 4.6 the data declarations are exactly the same as they were for the previous solution. However, there is now a type class that defines a single function, `validQualifier`, which has similar semantics to the functions created in (Figure 4.5). The difference now is that to implement each tag's function, the tag must become an instance of the `TagInterface` type class.

The previous problem of having to remember many different function names for each individual tag no longer exists. This is because each instance of the `TagInterface` type class defines the `validQualifier` function with a different tag as its parameter. This ad-hoc polymorphism allows the `validQualifier` function to be called with different types of tags as an argument and as such the goal to utilise a single function to provide the same functionality for all tags is accomplished.

The second problem of not being able to query the implementation to find out which tags have actually been created is also removed through the usage of the `TagInterface` type class. Figure 4.7 illustrates how the GHCi interpreter can be used to return all of the instances of the `TagInterface` type class by passing in the command ':info TagInterface'.

The command returns the definition of the type class and also lists all of its instances. Furthermore, it provides meta data about each instance allowing the user

```
*ghci*>:info TagInterface
class TagInterface tag where
  validQualifier :: Entity -> tag -> Bool
          -- Defined at Tags.hs:27:7
instance TagInterface OrdType -- Defined at Tags.hs:30:10
instance TagInterface TimeInForce -- Defined at Tags.hs:35:10
```

Figure 4.7: Using GHCi to query the implementation.

to immediately see the file in which each tag's instance is defined as well as the exact line number. The ability to provide this high level overview of the entire FIX implementation is extremely beneficial to users wishing to extend the system.

Although the usage of the `TagInterface` type class solves the problems associated with the previous solution there is still the potential for a separate issue. The fact that the enumeration of qualifiers for each tag is separate from each instance of the type class means that users have the ability to only provide a partial implementation for a given tag. For example a user could create the data declaration for the `OrdType` tag (`data OrdType = Market | ...`) but fail to provide its corresponding instance definition (`instance TagInterface OrdType where ...`).

Although in practice this situation may never arise a better solution would enforce the full implementation of a tag and not allow any partial declarations. This problem provides the reasoning behind the adoption of the simulator's actual implementation of FIX as it largely solves the aforementioned issues.

As it currently stands a data declaration cannot be included within a type class without the usage of a language extension. It is for this reason that the actual simulator implementation utilises the GHC 'type/data families' extension.

```
class TagInterface tag where
  data TagQualifier tag :: *
  validQualifier :: Entity -> (TagQualifier tag) -> Bool
```

Figure 4.8: The final `TagInterface` type class specification.

Figure 4.8 shows the final type class definition used for the actual implementation. The type class contains a signature for the `TagQualifier` data family and the `validQualifier` function. Similar to the way a type class allows for the overloading of functions, a data family allows for the overloading of a data type. This is what permits the type signature of the `validQualifier` function in Figure 4.8 to

now take a parameter of the type '`TagQualifier tag`' instead of the type '`tag`' as was the case in Figure 4.6.

Prior to illustrating an example instance of the type class it is important to establish the meaning of the signature (`data TagQualifier tag :: *`) attached to the `TagQualifier` data family and its associated semantics.

A type signature for a function is used in order to define its inputs and outputs. For example the signature '`f :: Int -> Int -> Either Int String`' defines how the function takes two '`Int`' parameters and returns a value which is of the type '`Either Int String`'. In the same way that functions take parameters to return a value, a type constructor takes other types as parameters to eventually produce a concrete type. For example, the '`Either a b`' type constructor (from Haskell's `Data.Either` module) takes two parameters of type '`Int`' and '`String`' in order to produce a concrete type of '`Either Int String`'.

Thus signatures can also be given for the type of types in which case they are known as 'kind' signatures. Type constructors which do not take any type parameters or have received all of the parameters they expect, such as '`Int`' or '`Either Int String`', are known as concrete types and have the kind signature of '`:: *`'. In the case of '`Either a b`' its kind signature is '`:: * -> * -> *`' with the first two stars representing the two type parameters it will take (e.g. '`Int`' and '`String`'), and the final star representing the concrete type that it will return (e.g. '`Either Int String`').

Since the `TagQualifier` data family takes a single parameter (the `tag`) its kind signature is '`:: * -> *`'. Using a type class to contain the data family declaration guarantees that all instances will provide the `tag` parameter explaining why the kind signature in Figure 4.8 is '`data TagQualifier tag :: *`'.

An important distinction to make is that having the kind of a concrete type (`:: *`) does not restrict the value that can be produced to being nullary. Instead it means that all of the type parameters (if any) have been provided to the type constructor such that it can produce a concrete type.

For example, '`Either Int String`' has been provided with its two type parameters (in this case `Int` and `String`) and so has the kind of a concrete type. Nonetheless it could produce two possible values (see the `Data.Either` module for definition of the data type), both which have non-nullary constructors e.g. '`Left 5`' or

'Right "String"'. Thus for each instance the TagQualifier data family is not limited in what values it can produce but rather is limited to taking a single tag parameter as input — as it should be.

In essence, the TagQualifier data family allows each tag which is assigned to it to be given a specialised data declaration. Although having an individual data declaration for each tag is the same as for the preceding solutions the TagQualifier data family offers the distinct advantage of being possible to include within the TagInterface type class. Thus the design of the type class as in Figure 4.8 is perfect for the simulator's implementation as it presents the requirements for a new tag explicitly to the user. Furthermore, since each tag will form part of the family and given the family's polymorphic declaration it is clear to see that it could easily be enhanced to encapsulate all of the tags defined in the FIX specification.

Moreover, by creating the system such that implementing a tag involves making it an instance of the TagInterface type class the compiler will enforce the full implementation of each tag. Thus, a situation where the possible qualifiers for a tag are defined but the subsets of values which are valid for each entity are not will never occur. Figure 4.9 demonstrates the procedure for declaring a tag (in this case the TimeInForce tag) to be an instance of the TagInterface type class.

```
instance TagInterface TimeInForce where
  data TagQualifier TimeInForce
    = Day         | GoodTillCancel    | AtTheOpening | ImmediateOrCancel
    | FillOrKill  | GoodTillCrossing  | GoodTillDate | AtTheClose
  validQualifier BYX qualifier  | qualifier == AtTheOpening = False
                                | qualifier == AtTheClose   = False
                                | otherwise                 = True
  validQualifier NYSE qualifier | ...
```

Figure 4.9: Example instance declaration of the TagInterface type class for the TimeInForce tag.

The TagQualifier data family declaration in Figure 4.9 enumerates all of the possible qualifiers for the tag and the validQualifier function defines the valid subset for each entity. In the case of the BYX exchange the function captures the fact that all of the qualifiers are valid apart from AtTheOpening and AtTheClose. The ellipses following the redacted definition of validQualifier for NYSE highlight the ease of which an additional entity could be included in the implementation.

4.3.2 The BATS Messaging Protocol

As of now the version of FIX to implement has been established (v 4.2), a system presenting an extensible and intuitive interface for the creation of tags has been created, and a mechanism for specifying the subset of tag qualifiers which are valid for each possible exchange has been determined.

Additionally a criterion for deciding which FIX tags should be implemented has been chosen — implement the tags needed for the most frequent operations such as order creation, cancellation etc. The final step to achieving all of the requirements is to actually implement those tags which meet the criterion.

Of the some dozen exchanges within the US national market system four are owned by BATS Global Markets Ltd. All four exchanges present the same message specification to their users allowing them to communicate in a consistent manor. Given that BATS exchanges cover a third of the NMS the decision has been taken to use their FIX specification as the basis from which to determine the tags to implement. This has been done to greatly increase the extensibility of the work of this project as the resulting implementation will directly provide the messaging infrastructure that future BATS exchange agents will rely upon.

Message Name	**Level**
Login Response V2	Session
Logout	Session
Server Heartbeat	Session
Replay Complete	Session
Order Acknowledgment V2	Application
Order Rejected V2	Application
Order Modified V2	Application
Order Restated V2	Application
User Modify Rejected V2	Application
Order Cancelled V2	Application
Cancel Rejected V2	Application
Order Execution V2	Application
Trade Cancel or Correct V2	Application

(a) BATS to member messages.

Message Name	**Level**
Login Request V2	Session
Logout Request	Session
Client Heartbeat	Session
New Order V2	Application
Cancel Order V2	Application
Modify Order V2	Application

(b) Member to BATS messages.

Table 4.2: Possible messages between BATS exchanges and its members.

As seen in Table 4.2 BATS messages are split into two categories — session and application. The session level protocol, among other things, includes messages for logging in/out and signaling that a connection should still exist (heartbeats) between

the client and exchange. If BATS receives no inbound data/heartbeat messages for five seconds the client is immediately logged out. The session level protocol is largely administrative overhead which is unnecessary for inclusion in the simulator.

On the other hand the application level protocol forms an integral part of the communication which would be desirable to capture within the simulator — new orders, cancellations etc. Figure 4.10 shows a redacted data declaration containing a record for each type of `BATSMessage` in the application protocol.

Each record (expressed within Figure 4.10 by {...}) represents the collection of FIX and BATS specific tags which form that particular message. To extend the simulator's messaging system those tags need to be implemented.

```
data BATSMessage
 = BATSNewOrderMessage {...}            | BATSCancelOrderRequestMessage  {...}
 | BATSModifyOrderRequestMessage {...} | BATSOrderAcknowledgmentMessage {...}
 | BATSOrderRejectedMessage {...}       | ...
```

Figure 4.10: Redacted data declaration of the `BATSMessage` data type which contains all of the relevant messages for a BATS exchange.

Figure 4.11 expands the `BATSNewOrderMessage` record as {...}) to reveal the tags which make up the message. The implementation code for the rest of the message types can be found in the appendix section F.6.

```
data BATSMessage = BATSNewOrderMessage
 { _BNO_SendingTime_FT :: Tag (TagQualifier SendingTime_FT)
 , _BNO_ClOrdID_FT     :: Tag (TagQualifier ClOrdID_FT)
 , _BNO_OrderQty_FT    :: Tag (TagQualifier OrderQty_FT)
 , _BNO_OrdType_FT     :: Tag (TagQualifier OrdType_FT)
 , _BNO_Side_FT        :: Tag (TagQualifier Side_FT)
 , _BNO_Symbol_FT      :: Tag (TagQualifier Symbol_FT)
 , _BNO_RoutingInst_BT :: Tag (TagQualifier RoutingInst_BT)
 , _BNO_ExecInst_FT    :: Tag (TagQualifier ExecInst_FT)
 , _BNO_Price_FT       :: Tag (TagQualifier Price_FT)
 , _BNO_TimeInForce_FT :: Tag (TagQualifier TimeInForce_FT)
 , _BNO_ExpireTime_FT  :: Tag (TagQualifier ExpireTime_FT)   } | ...
```

Figure 4.11: Data declaration showing the record for a BATSNewOrderMessage.

GHC automatically defines 'getter' functions for each field in the record such that the name of the function corresponds to the name of the field. For example, GHC automatically defines the functions `_BNO_SendingTime_FT`, `_ClOrdIF_FT` etc. which,

when applied to a BATSNewOrderMessage, return the corresponding tag. This presents a problem as the same tag can appear in more than one type of message.

Consider the tag 'price' which appears in the new order and modify order messages. If the field is named 'price' in both message records it will cause a name clash between the two generated GHC functions. As a result the convention to prefix each field within the record with a relevant identifier to make it unique has been adopted. In Figure 4.11 the prefix '_BNO' was chosen to represent 'BATS New Order'.

Additionally to explicitly indicate the entity which defined the tag, each is given a suffix e.g. '_FT' indicates it is a FIX tag and '_BT' indicates it is a BATS tag. This suffix does not solve any naming problem in GHC but rather is the naming convention which has been adopted within the implementation.

Although Figure 4.11 illustrated how messages are defined by combining different tags the mechanism for actually creating a tag has not yet been shown. Figure 4.12 highlights the steps required to create the Side_FT tag.

```
-- 1: Make a declaration of the tag so that it can become an instance of
--    the TagInterface.
data Side_FT
-- 2: Write the tag's instance for TagInterface class.
instance TagInterface Side_FT where
  data TagQualifier Side_FT = Side_FTVal_Buy | Side_FTVal_Sell
    deriving (Show, Eq)
  validQualifier BYX qualifier any = True
  validQualifier NYSE qualifier any = True
-- 3: Create a neutral version of the tag for users to amend as needed.
side_FT :: Tag (TagQualifier Side_FT)
side_FT = Tag { tagType = FIXTag, tagName = "Side",
                tagID = 54, tagValue = EmptyTagValue  }
```

Figure 4.12: The implementation of the Side FIX tag.

Since the underlying concept of a tag is the same for all types, showing the process of implementing a single one goes a long way in illustrating how the rest of the tags could also be constructed. Additionally, Figure 4.12 demonstrates the straightforward extensibility of the system as any user could follow the given blueprint to generate new tags in the future.

The steps to create a tag are as follows: firstly, declare a data type with the exact same

name as the tag followed by the relevant suffix[4]. Since the data families extension is being used, the declaration can take the form shown in Figure 4.12 (`data Side_FT`) i.e. with no constructors. The actual enumeration of constructors for the type will occur as part of the `TagQualifier` data family declaration.

Secondly write the instance for the tag so that it becomes part of the `TagInterface` class. By doing so the tag will also become part of the `TagQualifier` data family and the user can thus define all of the possible qualifiers allowed by FIX. In the case of the `Side_FT` tag the only possible values are 'buy' and 'sell'. However in order to enforce clarity and not introduce duplicates into the constructor namespace each option is prefixed with 'Side_FTVal' so the options become `Side_FTVal_Buy` and `Side_FTVal_Sell`. At this time there are only two entities to consider (`BYX` and `NYSE`) which both accept either option and so the `validQualifier` function simply returns `True` for both of them.

The third and final step as illustrated in Figure 4.12 is to actually create the FIX tag. Each of the fields of the tag record are defined and the value is left as the `EmptyTagValue`. This is intentionally done to provide users with a 'neutral' tag without a qualifier which they can amend as needed. The function `checkAndSetTagValue` as described in Figure 4.13 provides users with a mechanism for setting the value to a qualifier that is accepted on their choice of exchange.

```
setTagValue :: Tag a -> a -> Tag a
setTagValue tag value = tag { tagValue = TagValue value }

checkAndSetTagValue :: Entity -> Tag (TagQualifier tag)
                    -> TagQualifier tag -> Tag (TagQualifier tag)
checkAndSetTagValue entity tag tagValue
    | validQualifier entity tagValue = setTagValue tag tagValue
    | otherwise = error ("Your tagValue of "++(show tagValue)++" is invalid "
                  ++ " for the entity: "++(show entity))
```

Figure 4.13: The `setTagValue` and `checkAndSetTagValue` functions for setting tag values.

Rather than expect the user to manipulate a tag's record directly to amend its value, the function `checkAndSetTagValue` (Figure 4.13) is provided. When given an entity, a current version of the tag, and the desired qualifier the function calls `validQualifier`

[4] The suffix is an indicator of the entity which defined the tag e.g. '_FT' is used to indicate it is a FIX tag, '_BT' is used to indicate it is a BATS tag etc.

to determine whether the value given is valid for that entity or not. If the value is valid then the tag is amended to contain it and returned. If not, an error halts the program to alert the user of their incorrect intention.

This highlights the importance of third step in Figure 4.12 — the neutral tag is intended to be a predefined resource the user can provide to the `checkAndSetTagValue` function to set the tag's value. For example, suppose the user wanted to set the value of the `Side_FT` tag so that it reflected their intention to buy on the BYX exchange. In order to do this they would call the function with the parameters like so: '`checkAndSetTagValue BYX side_FT Side_FTVal_Buy`'. The important thing to note is that the neutral tag definition, `side_FT`, is being used to produce the new tag.

As a result of implementing the FIX tags which reflect the most frequently used actions (new order/modify etc.) a pertinent subset of the FIX protocol has now been implemented. Figure 4.12 also illustrated how the chosen design means that the messaging infrastructure can easily be extended by a user to introduce new tags on an as needed basis.

Thus all of the requirements of the extensions to the simulator's messaging infrastructure have now been met. Additionally, as a result of implementing the FIX tags used by BATS the messaging system for four potential exchange agents has been created — an achievement above the initial goals of this project.

4.4 Testing and Validation

As the FIX simulator extensions make heavy use of Haskell's type system certain parts of each experiment will be tested and validated to be correct automatically by GHC during compilation/runtime. In particular there are a number of typical situations which a user of the simulator could encounter that, if left undetected, would cause bugs within their experiment.

1. Attempting to use the `checkAndSetTagValue` function to set the qualifier to a value that is defined by FIX but not valid for the exchange they require.
2. Attempting to set the value of a specific FIX tag to a qualifier which is actually only valid for a different tag.

As a result of testing each of the above scenarios it was determined that the first error would cause the user to be alerted of their mistake by an exception being thrown at runtime. The second error would be accurately detected at compile time forcing the user to immediately amend their miscalculation before successful compilation would be possible. The fact that the most common errors are detected and then relayed to the user is an indication that the system has been built in a robust manner. The full test results can be found in the Appendix section E.3.

As well as testing for the potential presence of errors it is also important to determine whether the constraints set by the type system are too rigid. Users being unable to accurately code their intentions due to stringent constraints would completely defeat the purpose of the simulator's extended messaging system. In addition to testing the above the Appendix section E.3 also contains the results of successfully setting the values of tags.

Chapter 5

Simulating the Consolidated Quotation System

5.1 Background and Requirements Capture

The background and importance of the CQS has already been established in chapter 2 which described its integral role in the consolidation of quotations from all NMS exchanges. Additionally, the chapter detailed how the CQS was responsible for the dissemination of the National Best Bid and Offer for each NMS security.

The NBBO is a key aspect of Regulation NMS as it is mandated that trade executions of NMS securities occur at its price or better (Order Protection Rule). Thus, the CQS is being implemented as the first major component of the simulator's model of the NMS as future agents (exchanges/brokers) will be depending on its presence.

The Consolidated Tape Association oversees the operations of the CQS and issues two separate messaging specifications — one for input [CTA, 2015a] and the other for output [CTA, 2015b] — which must be analysed prior to any agent implementation. In particular, the CQS agent needs to send and receive messages with the same structure as the messages which are sent and received in the real world.

5.2 Analysis of Requirements

From the discussion above and the background covered in chapter 2 the following actions must be taken to provide the CQS agent implementation:

- Create a data structure which replicates the same structure of the message within the input specification used by exchanges to communicate their BBO.
- Create a data structure which replicates the same structure of the message

within the output specification used by the CQS to broadcast the NBBO.

- Extend the simulator's current messaging system such that it incorporates the two aforementioned message types.
- Encode the logic of the CQS agent such that it is able to:
  1. Receive the BBO for a particular equity from each exchange.
  2. Collate the information and calculate the NBBO.
  3. Disseminate the NBBO of that equity to all NMS participants as well as any CQS subscribers.

The CQS input specification defines two categories of message — 'A' and 'C'. Category A messages are used, among other things, for administrative purposes and for NMS participants to relay their latest quote information. Category C messages are used for 'control' purposes such as indicating the start of the trading day and testing line integrity. Category A messages which serve the requirement of communicating quotation data are of particular importance to the CQS agent.

Two distinct types of messages are defined by the CQS which, depending on the circumstance, can be used by participants to disclose their latest quotation data — 'long quote' messages and 'short quote' messages. For participants to issue a short rather than long quote a number of conditions surrounding the quotation must be met. The most important conditions relate to how the market should be a normal auction market and the quotation should have a regular way settlement[1].

In the context of the simulator's CQS agent, the purpose of the quote message is solely for the participant to confer their best bid and offer to the CQS — which is possible through both the long and short quote message types. Capturing the complexity associated with the long quote (e.g. of non-regular settlements) is unnecessary as it is beyond the intended scope of the simulator. For this reason it has been decided to adopt the short quote message format for the implementation.

Each input message, in addition to its main body, contains a header with a number of predefined fields. As the majority of these fields are made redundant by existing features of the simulator they will not be implemented. For example, the header field indicating the type of message is unnecessary as Haskell's type system will provide an expressive way of capturing that information anyway.

[1] A regular way settlement means that the will settle on the third business day after the trade date. This is the norm for most securities.

Of the fields within the header there are two which are pertinent to the implementation. The first is simply the identifier of the participant who is issuing the message. The second is intended to 'denote the time where the quote bid price and/or the offer price for a security is designated with an Exchange's Matching Engine Publication timestamp' [CTA, 2015a] — in other words it is a timestamp from the sender.

The 'short' and 'long' message types are also defined by CQS in their output specification for disseminating the NBBO. The previous choice of adopting the short rather than long quote avoids capturing the unnecessary complexity associated with the latter type of message. This line of reasoning remains valid when considering the CQS output and as such the implementation will also adopt the short message type for disseminating the NBBO.

5.3 Design and Implementation

Message Header	Bid Numerator Price
Security Symbol	Bid Size
Price Denominator Indicator	Offer Whole Price
Quote Condition	Offer Numerator Price
Bid Whole Price	Offer Size

Table 5.1: CQS Short Quote Message Contents.

Table 5.1 lists the fields contained within a CQS short quote message. As previously stated, the only relevant fields within the header are the exchange's timestamp and identifier. The Security Symbol field is used to identify the security for which the quote relates to e.g. the 'AAPL' symbol refers to shares in Apple. The quote condition field is used to inform the CQS whether the quote is (in)eligible for inclusion in the NBBO calculation. If the quote is ineligible the field defines codes which can be included within it to indicate the reason it is ineligible.

The prices for the best bid and offer are encoded through three fields — the whole price, numerator price and price denominator indicator. The whole price is used to represent the whole dollar portion of the quote whereas the numerator and denominator are used to represent the decimal portion. For example, a whole price of $15, numerator of 67 and denominator of 100 would represent the price of $15.67. For the sake of simplicity, this method for encoding price will not be mimicked in the

simulator's implementation and instead a Double will be used to contain the price in one field.

When transmitting a CQS short quote there are three possible situations a participant could find themselves in. Firstly, the participant could have no bid or offer on their book at all. Secondly, they could only have a best bid but no offer (or vice versa) and finally they could have both a best bid and best offer.

The CQS's mechanism to identify each situation is to use a combination of zeros within the price/size fields to denote the absence of a bid/offer. The design chosen for the simulator does not adopt this method and will instead utilise Haskell's type system to provide a more expressive solution.

Figure 5.1 contains the data declaration for the CQS short quote and illustrates how each of the situations is depicted. The EmptyBBO can be used by participants to represent no best bid or offer existing in their order book and the ExchangeBBO can be used to indicate either of the other two situations. This flexibility associated with the ExchangeBBO is achieved through the usage of Haskell's 'Maybe a' type.

```
data CQSShortQuoteRecord
 = EmptyBBO    { _BBOTimestamp :: Int, _BBOExchange :: Int,
                 _BBOSymbol :: String, _quoteCondition :: CQSQuoteConditions}
 | ExchangeBBO { _BBOTimestamp :: Int, _BBOExchange :: Int,
                 _BBOSymbol :: String, _quoteCondition :: CQSQuoteConditions,
                 _BBPrice :: Maybe Double, _BBSize :: Maybe Int,
                 _BOPrice :: Maybe Double, _BOSize :: Maybe Int }
data CQSQuoteConditions = EligibleQuote | IneligibleQuote
```

Figure 5.1: Data declaration of the CQS Short Quote and CQS Quote Conditions.

The 'Maybe a' type caters for the representation of an absence of a best bid/offer through its Nothing constructor. Alternatively, to indicate a value is present the participant can simply wrap it within a Just constructor. This system is much more expressive than using zeros to indicate the absence of a field as it allows the participant to encode their intent in an explicit manner.

Regardless of whether the CQS short quote is an EmptyBBO or an ExchangeBBO its condition must be indicated by the sender. The CQSQuoteConditions data declaration is also shown in Figure 5.1 and illustrates how the participant can use the value of EligibleQuote or IneligibleQuote to inform the CQS whether the quote

should be included in its calculation of the NBBO or not. In the case of an ineligible quote, the actual field defined by FIX allows the message sender to attach a reason. This is not currently implemented within the simulator but the `CQSQuoteConditions` declaration could be amended to accommodate it if it is desired in the future. Apart from the this, the absent fields in the `EmptyBBO` and the adoption of `Doubles` to represent price, the `CQSShortQuoteRecord` is a direct translation of the Table 5.1 specification.

Best Bid Participant ID	Best Offer Participant ID
Best Bid Price Denominator Indicator	Best Offer Price Denominator Indicator
Best Short Bid Price	Best Short Offer Price
Best Bid Size in Units of Trade	Best Offer Size in Units of Trade

Table 5.2: CQS Short NBBO Message Contents.

Table 5.2 illustrates the fields within the CQS Short NBBO message. A major difference between the NBBO and quote (see Table 5.1) messages is that instead of having three fields to represent price the NBBO message has only two. In the NBBO message price is determined by a combination of the 'price' and 'denominator indicator' fields. For example, if the value of the price field is 10211 and the denominator is 100, the actual price is $10211/100 = \$102.11$. However, this difference will not be present within the simulator implementation as it has been determined that using `Doubles` to store the value of the price allows for a uniform representation.

The previously stated three situations which could occur for a quote message can also occur for an NBBO message and therefore the existing convention has been adopted again as shown in Figure 5.2. The `EmptyNBBO` is used to indicate the situation when there are no national bids/offers on any exchange and the `NBBO` to indicate otherwise. Additionally, the '`Maybe a`' type is used within the `NBBO` record's fields to indicate the absence of a value.

```
data CQSNBBORecord
 = EmptyNBBO { _NBBOTimestamp :: Int, _Symbol :: String }
 | NBBO { _NBBOTimestamp :: Int, _Symbol :: String,
   _NBBTimestamp :: Maybe Int, _NBB :: Maybe Double, _NBBSize :: Maybe Int,
   _NBBExchange  :: Maybe Int,
   _NBOTimestamp :: Maybe Int, _NBO :: Maybe Double, _NBOSize :: Maybe Int,
   _NBOExchange  :: Maybe Int }
```

Figure 5.2: Data declaration of the CQS NBBO record.

This decision to adopt the same conventions for the quote and NBBO messages and utilise `Double`s to store price solves the problems associated with presenting simulator users with an inconsistent interface. As a result of this, users will be spared from any potential ambiguity within the simulator and thus will be more productive in the long run. As with the short quote message (Figure 5.1) the implementation of the CQS NBBO message is a direct translation of the Table 5.2 specification.

Now that the fields of the simulator's versions of the short quote and NBBO messages have been defined the `Msg_t` and `Broadcast_t` data declarations need to be updated. Figure 5.3 shows the additional message definitions which have been added to the two data declarations. A `CQSShortQuoteMessage` is used by an exchange to communicate its latest best bid and best offer (quote) to the CQS and a `CQSNBBOBroadcast` is used by the CQS to output the latest NBBO.

```
data Msg_t = ... | CQSShortQuoteMessage (Int, Int) CQSShortQuoteRecord | ...
data Broadcast_t = ... | CQSNBBOBroadcast CQSNBBORecord | ...
```

Figure 5.3: Updated versions of the `Msg_t` and `Broadcast_t` data declarations (redacted).

The final implementation detail is the actual creation of the CQS agent responsible for receiving `CQSShortQuoteMessage`s from NMS participants, calculating the NBBO and disseminating it through a `CQSNBBOBroadcast`. An important point to consider is that all of the simulator's previous experiments as well as the vast majority of those which will be conducted in the future concern only a single security.

It is for this reason that the CQS agent has been designed to assume all of the `CQSShortQuoteMessage`s it receives are for the same security. This does not actually restrict the user, as if their experiment involves more than one they can simply set it up such that there is one CQS agent for each security.

The internal state kept by a CQS agent is shown in Figure 5.4. The first field of the record is the latest NBBO generated by the CQS (`CQSNBBORecord`). The second field is a map from an agent's integer ID to the latest short quote which they sent to the CQS (`Map.Map Int CQSShortQuoteRecord`).

```
data CQSStateRecord = CQSStateRecord
    { _NBBO :: CQSNBBORecord, _BBOs :: Map.Map Int CQSShortQuoteRecord}
```

Figure 5.4: Data declaration of the internal state of the CQS agent.

The usage of a map as the data structure to store the latest short quotes from each CQS participant is particularly convenient for a number of reasons. Firstly, prior to the start of any simulation, it allows the CQS agent to be completely decoupled from having to know anything about the agents which it will be interacting with.

When it receives a `CQSShortQuoteMessage` the CQS agent can simply use the (from, to) tuple attached within it to obtain the ID of the exchange in question and then further use the obtained ID as the key for the map. Since a given simulation could involve any number of exchanges and a map can dynamically increase/shrink in size the CQS agent will be able to accommodate any potential experiment.

Additionally the map data structure in question (defined in Haskell's `Data.Map` module) is implemented using size balanced binary trees whose complexity for lookups, insertions and updates is guaranteed to be $O(log(n))$. This makes it particularly suited for simulations which may include a large number of agents compared to other data structures whose complexity would be $O(n)$ or worse.

Algorithm 2 (see diagram) illustrates the steps involved in the initialisation of the CQS agent. The initialisation would only cocur for the first timestep in the simulation and Algorithm 3 (see diagram) contains the logic flow for all other timesteps. The actual code for the agent wrapper and associated functions can be found in Appendix D.

begin

1. Create an empty NBBO for the security which the agent is tracking.
2. Create an empty map with key being of type `Int` (for agent integer IDs) to type `CQSShortQuoteRecord` (their latest short quote).
3. Initialise the first `CQSStateRecord` such that it contains the created empty NBBO and empty map from line 1 and line 2.
4. Broadcast the empty NBBO created in line 1 to the channel specified for use by the CQS. This will indicate to CQS participants that the CQS is ready to receive their `CQSShortQuoteMessage`s.
5. Recursively call the CQS agent wrapper with the state changed from being empty to containing the record defined in line 3.

end

Algorithm 2: CQS Agent Initialisation Logic Flow.

With the implementation of the CQS agent being complete all of the requirements set out earlier in the chapter have been met. The input message to communicate participant BBOs (`CQSShortQuoteMessage`) and the output message to broadcast the

begin
1. Extract the current NBBO and BBO map from the current `CQSStateRecord`.
2. **if** *Any received message is not a `CQSShortQuoteMessage`* **then**
   Throw an exception to halt the program and alert the user that they have an error within their logic as the CQS should only ever receive `CQSShortQuoteMessages`.
   end
3. Pass all received `CQSShortQuoteMessages` and the current BBO map to the `updateBBOsLogic` function. This function will return a new map with each of the relevant BBOs (short quotes) in the map updated.
4. Generate an NBBO from the updated map in line 3 and call it 'latestNBBO'.
5. **if** *`currentNBBO` == `latestNBBO`* **then**
   Output a `CQSNBBOBroadcast` to the broadcast channel reserved for the CQS.
   end
6. Update the internal state of the CQS Agent to contain the latest NBBO and updated BBO map.
7. Recursively call the CQS agent wrapper with the updated state.

end

Algorithm 3: CQS Agent Logic Flow Post Initialisation.

NBBO (`CQSNBBOBroadcast`) now both have concrete implementations. Additionally the `Msg_t` and `Broadcast_t` data declarations have been expanded such that simulator agents are capable of sending and receiving both types of message. Finally, the logic of the CQS agent has been encoded such that it is able to receive the BBOs from each exchange, collate the information, calculate the NBBO and disseminate it to any relevant subscriber.

5.4 Testing and Validation

In order to validate the logic encoded within the CQS agent an experiment was created. The experiment was designed such that three separate exchange agents send `CQSShortQuoteMessages` to a single CQS agent over an extended period of time.

All of the messages were predefined so the actual NBBO for each given timestep was known in advance. This was then compared with the output produced by the CQS agent to validate its accuracy and in all cases the CQS agent performed as expected.

Testing has resulted in the confirmation of the correctness of the CQS agent and as such it is safe to include in any future projects. Details surrounding the testing of the CQS agent can be found in the Appendix section E.2

Chapter 6

Conclusions

6.1 Evaluation

Through the course of this project a number of extensions to the existing Haskell agent based simulator have been designed and implemented. As a result, all of the objectives of the project defined in section 1.3 have now been fully met.

The following section lists all of the goals of the project and highlights how they have been achieved within the simulator. The primary objectives are covered first, followed by the secondary ones.

1. Extend a preexisting Haskell agent based simulator's messaging infrastructure by creating a mechanism for the simulator to emulate the real world delays associated with message passing between agents.

By using the latency definitions provided by the user in `FunArg1` the simulator can now hold direct messages within an internal delay queue and allow agents to only consume them once the associated message lag has expired. Additionally when a broadcast message is sent the simulator can use `FunArg2` to look up the list of agents that should receive it. Subsequently, it can explode the broadcast message out into a 'routed broadcast' for each agent. Since each routed broadcast has a one to one correspondence from sender to recipient the simulator can simply hold it within the delay queue just like for a direct message.

Thus through the use of the `FunArg1`, and `FunArg2` runtime arguments the objective has been achieved for both direct messages and broadcasts . An additional achievement beyond the scope of the initial objective is the fact that the simulator is now also completely aware of the graph topology of the experiment. `FunArg1` facilitates this awareness by enforcing that the simulator only route messages to agents for which a valid connection exists.

2. Extend the simulator by implementing a relevant subset of the 'Financial Information eXchange' (FIX) protocol so that agents within the simulation can communicate with each other using the industry standard convention.

The simulator's implementation of FIX mandates that each tag must become part of the `TagInterface` type class. The `TagQualifier` data family and `validQualifier` function are part of the class definition and as a result any implementation of a tag must also include definitions for them. A major advantage associated with this design is that the requirements of extending the system to include a new tag are made clear to any future user. Furthermore, the type class definition will enforce that a partial implementation of a tag is not possible within the simulator.

The instance declaration of a tag joining the `TagQualifier` data family is used as a mechanism for encapsulating all of the potential values which could be used as a qualifier for the tag. Additionally, the `validQualifier` function captures the complexity associated with the fact that the receiver of the FIX message may only accept a limited number of the qualifiers defined for the tag. The criterion chosen to determine the relevant subset of the FIX protocol was to simply implement those tags which are used in the most frequent operations.

Thus through the defined criterion and a combination of the `TagQualifier` data family, `validQualifier` function and `TagInterface` type class the objective of implementing a relevant subset of the FIX protocol has been met. Additionally, by implementing only the tags used in the most frequent operations, the infrastructure surrounding the BATS messaging system was also created. By also creating the BATS messaging system the achieved implementation is above the scope of the original objective.

3. Build an initial framework encapsulating specific parts of the US National Market System. The aim is to create the following:
   3.1. A simulator agent implementing the most pertinent aspects of the Consolidated Quotation System (CQS) of the NMS. The functionality to be implemented is that which relates to the portion of the CQS which:
      i. Receives the BBO for a particular equity from each NMS exchange.
      ii. Collates the information and calculates the NBBO.
      iii. Disseminates the NBBO to all NMS participants and any CQS subscribers.

3.2. Ensure that created CQS agent complies with the regulation mandated by the U.S. Securities and Exchange Commission. In particular, it is required to comply with Regulation NMS [SEC, 2005].

The implemented CQS agent is able to receive the best bid and offer from each exchange in the form of a `CQSShortQuoteMessage`. Additionally, the logic responsible for the calculation of the NBBO has been encoded as a direct translation of the description provided in the CQS output specification. Once calculated, the CQS agent is able to disseminate the NBBO in the form of a `CQSNBBOBroadcast`.

Since the logic of the agent has been taken directly from the specification and the messages it sends/receives mimic their real world counterparts the implemented agent complies with REG NMS. Thus the goal of creating an initial framework of the US NMS and making it compliant with Regulation NMS has been achieved.

Each of the secondary goals of the project were also achieved:

1. Make use of a modular design so that future users of the simulator can reuse the created code and extend it as per their requirements.

Each of the additions to the simulator have been encoded such that they can be extended and easily made use of in the future. In the case of the delay queues and routed broadcasts, the modular design means that the created data structures are completely contained within the 'Sim' module.

Furthermore, for the FIX messaging system the `TagInterface` type class explicitly defines what is expected of any extension to the current system. Additionally, all extensions relating to FIX are contained within a separate directory of the project. Within the directory there is a distinct module for FIX tags, BATS tags, BATS messages, and data structures common to the entire FIX system — thus highlighting the modular design of the system. Finally, since the CQS agent has been encoded such that is has the type `Agent_t` it can be reused in any future user experiment.

Thus the goal of making use of a modular design for the purpose of code reuse has been achieved.

2. Decouple simulator agents from having to store the integer IDs of those who they will interact with as part of their code's logic.

Users are now able to create labels to refer to each agent within their simulation. They

are provided with the function `generateAgentBimap` which automatically constructs a bidirectional mapping between the agent labels and integer IDs. Additionally they are provided with the function `getCorrespondingAgentIdentifier` which they can include within the `FunArg3` runtime argument for their experiment.

By using the `getCorrespondingAgentIdentifier` function users can create wrappers where their agents refer to others through the usage of labels rather than integer IDs. Thus the objective of decoupling agents from having to store the integer IDs of those they will be interacting with has been achieved.

3. Retain full backwards compatibility with the experiments which were coded prior to any simulator amendments made as part of this project.

Depending on whether the user attaches the relevant runtime arguments the simulator will either enter 'compatibility mode' or a mode which supports the routed broadcast and delay queue extensions. It has been validated that compatibility mode produces the exact same output for previous experiments as was produced prior to the simulator extensions. Thus the goal of retaining full backwards compatibility with the existing experiments and code base has been achieved.

6.2 Conclusion

This project has resulted in the successful implementation of numerous extensions to the Haskell agent based simulator. These extensions provide users with a major improvement to the infrastructure surrounding the internals of the simulator and its messaging system.

Future users are now able to directly encode their experiment graph and thus be relieved from the onus of manually enforcing the validity of agent connections. Furthermore, they no longer have to create sub-standard mechanisms for modeling the messaging latencies between the agents of their experiment. Instead they can simply use the validated system which now exists within the simulator and focus their time on the actual problem at hand.

Additionally, users no longer have to hard code integer identifiers within the logic of their wrapper functions. They can now refer to each agent through a label which they define — not only providing them with a much more intuitive interface but also increasing the ease of which they can conduct refactoring.

By implementing the FIX messaging protocol the ecosystem surrounding the simulator has become much more extensible. Any future agents will be presented with a consistent method for communication which they can choose to develop further according to their needs. Moreover, FIX is used throughout the Americas, UK, Europe, Asia and the Pacific Rim meaning the applicability of the implemented system would still exist should the user decide to conduct an experiment set in another region.

The CQS agent provides the initial underpinning of the model of the United States National Market System and since its logic has been verified to be correct it is ready for inclusion in the experiments of users.

In conclusion, the extensions implemented in this project provide a much more robust infrastructure for users to conduct their future research. It is hoped that the simulator will be used fruitfully to further understand the complex interactions which occur between participants of financial markets. In particular, the simulator is primed to investigate the claims of abuse made against high frequency traders within the United States National Market System.

6.3 Further Work

Future users of the simulator are well suited to conduct further work to either extend the contributions of this project or utilise them to conduct empirical analyses.

Potential extensions to this project could be to contribute any of the components yet to implemented within the simulator's NMS framework (see section 2.5). The NMS model could be enhanced to include an agent for the NYSE stock exchange or a broker, for example. Since the messaging system (FIX) used by the members of the NMS is already in place these extensions would integrate seamlessly with the work of this project.

Further research and empirical analyses could also be conducted to determine the extend of the validity of the claims made against high frequency trading by those such as Lewis [2014], Bodek [2013], and Amuk and Saluzzi [2012]. Additionally the simulator is now well suited for conducting experiments ranging from understanding the complex interactions between agents within the financial markets to determining the affect that latency plays in the successful execution of trading strategies.

Bibliography

Arlow, J. and I. Neustadt

2002. Uml and the unified process. *Practical Object-Oriented Analysis and Design. Boston, MA: AddisonYWesley.*

Arnuk, S. and J. Saluzzi

2012. *Broken Markets: How High Frequency Trading and Predatory Practices on Wall Street are Destroying Investor Confidence and Your Portfolio.* Financial Times/ Prentice Hall.

BATS

2015. Bats fix specification. [Online; accessed 24-August-2015; `http://cdn.batstrading.com/resources/membership/BATS_US_EQUITIES_FIX_SPECIFICATION.pdf`].

Bodek, H.

2013. *The Problem of HFT: Collected Writings on High Frequency Trading and Stock Market Structure Reform.* CreateSpace Independent Publishing Platform.

Chopra, A. and C. D. Clack

2015. *Modelling Front-Running in High-Frequency Trading).* University College London.

Cohen, K., R. Conroy, and S. Maier

1985. Market making and the changing structure of the securities industry.

Congress, U. S.

1790. United states statutes at large/volume 1/1st congress/2nd session/chapter 34. [Online; accessed 14-August-2015; `https://en.wikisource.org/w/index.php?title=United_States_Statutes_at_Large/Volume_1/1st_Congress/2nd_Session/Chapter_34&oldid=4426116`].

Congress, U. S.

1933. Securities act. [Online; accessed 16-August-2015; `https://www.law.cornell.edu/uscode/text/15/chapter-2A/subchapter-I`].

Congress, U. S.
1934. Securities exchange act. [Online; accessed 16-August-2015; https://www.law.cornell.edu/uscode/text/15/chapter-2B].

Court, E. and C. D. Clack
2013. *The instability of market-making algorithms: An agent-based simulation in Miranda*. University College London.

CSE
2015. Chicago stock exchange fix specification. [Online; accessed 24-August-2015; http://www.chx.com/_literature_119720/CHX_FIX_Interface_Specification].

CTA
2015a. Cqs input specification revision 24 080715. Technical report, Consolidated Tape Association.

CTA
2015b. Cqs output specification revision 60 080715. Technical report, Consolidated Tape Association.

Easley, D., M. Lopez de Prado, and M. O'Hara
2011. The microstructure of the 'flash crash': Flow toxicity, liquidity crashes and the probability of informed trading. *The Journal of Portfolio Management*, 37(2):118–128.

Garber, P. M.
1991. Alexander hamilton's market based debt reduction plan. Working Paper 3597, National Bureau of Economic Research.

Geiger, K. and S. Mamudi
2014. Hft firm fined $1 million for manipulating nasdaq". *Bloomberg*. [Online; accessed 15-August-2015; http://www.bloomberg.com/news/articles/2014-10-16/athena-to-pay-1-million-in-sec-hft-manipulation-case].

Gillis, J. G.
1975. Securities law and regulation: Securities acts amendments of 1975. *Financial Analysts Journal*, Pp. 12–15.

Gillis, J. G. and R. G. Dreher

1982. Securities law and regulation: National market system. *Financial Analysts Journal*, 38(5):pp. 13–15.

Harman, W. R.

1978. The evolution of the national market system—an overview. *The Business Lawyer*, 33(4):pp. 2275–2301.

Hasbrouck, J. and G. Saar

2013. Low-latency trading. *Journal of Financial Markets*, 16(4):646 – 679. High-Frequency Trading.

Lemke, T. P. and G. T. Lins

2014. *Soft Dollars and Other Trading Activities*. LegalWorks.

Lewis, M.

2014. *Flash boys: a Wall Street revolt*. WW Norton & Company.

Lin, T. C.

2010. Behavioral framework for securities risk, a. *Seattle UL Rev.*, 34:325.

Liu, Y. and C. D. Clack

2014. *Re-engineering agent based simulator with functional language*. University College London.

Madison, G.

2013. A look back at the flash crash of 2010: When will it happen again? *Money Morning*.

NASDAQ

2015. Nasdaq fix specification. [Online; accessed 24-August-2015; `http://nasdaqtrader.com/content/technicalsupport/specifications/TradingProducts/inet_fix_sb.pdf`].

NYSE

2015. Nyse fix gateway. [Online; accessed 24-August-2015; `https://www.nyse.com/publicdocs/nyse/markets/nyse/FIX_Specification_and_API.pdf`].

Pisani, B.

2014. Plundered by harpies. *Financial History*.

SEC
2004. Sec to publish regulation nms for public comment.

SEC
2005. Regulation nms: Final rules and amendments to joint industry plans. Technical Report 34-51808, Securities and Exchange Commission.

SEC
2007. Final rule; extension of compliance dates. Technical Report 34-55160, Securities and Exchange Commission.

SEC Division of Trading and Markets
2015. Memorandum re: Rule 611 of regulation nms.

Stafford, P. and A. Massoudi
2013. Glitches spur us share trading debate. *Financial Times*.

United States Federal Register
2005. Regulation nms - regulation of the national market system. *United States Federal Register*, 70(124):37620 – 37632.

Vali, K.
2009. Fix protocol training. `http://www.ksvali.com/2009/02/fix-protocol-training/`.

Vuorenmaa, T. A. and L. Wang
2014. An agent-based model of the flash crash of may 6, 2010, with policy implications. *Available at SSRN 2336772*.

Wall Street and Tech
2005. Regulation national market system synopsis.

Appendix A

Simulator History

This project extends a preexisting agent based simulator initially created using the functional language Miranda in 2011 by Christopher Clack of University College London.

Some time later, Court and Clack [2013] enhanced the simulator further in order to model and analyse the events of the 2010 Flash Crash. Although Miranda as a language choice allowed for an elegant programming style its interpreted nature limited the simulator's performance and resulted in an extremely slow rate of computation.

As a result the choice was made to port the simulator to Haskell — which would enable the usage of the Glasgow Haskell Compiler (GHC) to generate optimised and efficient native code. The porting was completed by Liu and Clack in 2014, with all preexisting agents also being translated to Haskell [Liu and Clack, 2014].

The port was largely successful but introduced a small number of subtle bugs within the simulator. Chopra and Clack's [2015] work removed these bugs and also contributed new agents for future use. An important aim of this project is to extend the work of Chopra and Clack by further improving upon the simulator's internal infrastructure.

Appendix B

User Manual

B.1 Compilation

The project's root directory contains a module 'Main' (filename Main.hs) which itself contains the declaration of the `main` function.

```
main :: IO()
{- Add/comment out your experiment, as needed -}
-- main = testCrossMarket
-- main = testmeHeteroPairs
-- main = testfrontrunning
-- main = funArgsMaxDelayAndAgentsExample
main = cqsTests
```

Figure B.1: Contents of the Main module.

Figure B.1 shows the contents of the main module. The user must make sure that `main` calls the function which represents their experiment. In the case of Figure B.1 `main` calls 'cqsTests' and so that is the simulation which will be run. Other experiments such as 'testCrossMarket' and 'testfrontrunning' are commented out but remain in the module for easy access/switching. Once the user has made sure that `main` points to the correct experiment they must compile the simulator's source code into a binary. To do so there are two methods — using the provided make file (recommended) or entering the compilation command directly (not recommended).

Regardless of the method used, the following instructions expect the Haskell Platform to be installed on a UNIX like machine.

1. Using the Makefile (recommended).

In the project's root directory is a file called 'Makefile'. This already has the relevant command options pre-loaded for an easy compilation process. The user can simply

enter ‘make’ into the terminal whilst in the root directory of the project. This will cause all object and header files to appear in the directory ‘CompilationOutput’ and a single binary called ‘Main’ to appear in the project’s root directory. Example output from this procedure is shown in Figure B.2.

```
market-sim(master) > make
ghc Main.hs -o Main -odir CompilationOutput -hidir CompilationOutput
[36 of 36] Compiling Main             ( Main.hs, CompilationOutput/Main.o )
Linking Main ...
```

Figure B.2: Compiling the Experiment using the MakeFile.

2. Compiling by providing your own command (not recommended).

Alternatively the user can provide their own command and options to the GHC compiler. An example command is shown in Figure B.3 which will cause the same outcome as using the makefile. A user experienced with compiling programs using GHC could enter a custom command if they so desire but it is recommended to make use of the Makefile provided.

```
ghc Main.hs -o Main -odir CompilationOutput -hidir CompilationOutput
```

Figure B.3: Compiling the Experiment using a custom command.

B.2 Running the Simulation

After choosing the experiment that the main function points to and using the makefile to compile the project the user can now run their simulation. As long as the user has execution permissions within the project’s root directory the simulation can be run by entering ‘./Main’ at the terminal as shown in Figure B.4.

```
market-sim(master) > ./Main
```

Figure B.4: Running the simulation.

Depending on which experiment is being run there may or may not be anything output in the terminal after the call to ‘./Main’. Regardless, the results of the simulation will be output as a plain text ‘trace’ file in the directory corresponding to what was defined in the experiment. The directory will itself be contained within the ‘messyoutputfolder’ (in the root directory). This ‘trace’ file can be analysed by the user to understand the results of their experiment.

Appendix C

System Manual

C.1 Creating a New Experiment

The convention when creating a new experiment is to bind it to a function defined in the module 'TestHarness'. Figure C.1 shows the associated code for the `testfrontrunning` simulation and serves as an example of how the user should set up their experiment.

```
testfrontrunning :: IO ()
testfrontrunning
  = do
    createDirectoryIfMissing True ("messyoutputfolder/"++thefilename)
    sim 100 args agents
     where
      args = [(Arg (Str "Calm", 1)), (Arg (Str "Randomise", 1)),
                  (Arg (Str fthefilename, 9989793425))]
      fthefilename = "messyoutputfolder/" ++ thefilename ++ "/" ++ thefilename
      thefilename = "Frontrunning"
      agents = delays ++ exchanges ++ frontrunners ++ brokers ++ traders
      delays = (rep (length delay_list) (delaywrapper, [0..16]))
      exchanges = [(nice_mime_wrapper, [0..16]), (nice_mime_wrapper, [0..16])]
      traders = [(mytraderwrapper, [0..16])]
      brokers = [(brokerwrapper, [0..16]), (brokerwrapper, [0..16])]
      frontrunners = [(frwrapper, [0..16]), (frwrapper, [0..16])]
```

Figure C.1: Example code for an experiment.

The call to sim must be accompanied by the number of timesteps you wish to run the simulation for, a list of runtime arguments, and a list of agents to include in the simulation (`sim 100 args agents`). If the user wishes to include the new features developed in this project they should also include the max delay and `FunArg1` runtime arguments within the list arguments passed to sim as described in Chapter 3. The

wrappers included in the simulation could exist already or the user can choose to create their own (as shown in Section C.2).

C.2 Creating a New Agent

In order to create a new wrapper the user must write a function which is of the type `Agent_t` as shown in Figure C.2.

```
type Agent_t = Agentstate_t -> [Arg_t] -> [(Int, [Msg_t], [Msg_t])]
               -> Int -> [[Msg_t]]
-- Agentstate, Runtime Arguments, [ (time, messages, broadcasts)], ID, Output.
```

Figure C.2: Example code for an experiment.

The type `Agentstate_t` is used to encapsulate the internal state of all agents. Thus, if the user requires their new agent to hold an internal state (which is most likely the case) they need to extend the declaration of `Agentstate_t` to reflect this.

Figure C.3 gives an example on how the `Agentstate_t` data declaration was extended to account for the state of the CQS agent (`CQSState CQSStateRecord`). The declaration for `CQSStateRecord` is also shown in Figure C.3.

```
data Agentstate_t
 = ... | Emptyagentstate | CQSState CQSStateRecord | ...

data CQSStateRecord
 = CQSStateRecord { _NBBO :: CQSNBBORecord,
                    _BBOs :: Map.Map Int CQSShortQuoteRecord }
```

Figure C.3: Example extension to `Agentstate_t.` and data declaration of `CQSStateRecord`.

Once the `Agentstate_t` data declaration has been extended to include the internal state of the new agent the actual code for the agent needs to be written. The convention here is to create a separate module which will contain the agent's wrapper function and any associated logic.

The first time the agent wrapper is called in the simulation it will be passed the `Emptyagentstate` as the `Agentstate_t` parameter. This is done intentionally to allow the agent to initialise its state.

```
cqsWrapper :: Agent_t
cqsWrapper (Emptyagentstate) args ((time, messages, broadcasts) : restOfSimulatorStates) myid
    = (([nbboBroadcast]) : (cqsWrapper (CQSState cqsStateRecord) args restOfSimulatorStates myid))
      where
       {-
          Insert the logic associated with the initialisation of the agent's state here.
          Note how the wrapper's recursive call to itself passes in the
          initialised state. In the case of the CQS it also has to output
          an nbboBroadcast when being initialised but that may not be the case
          for other agents.
       -}

cqsWrapper (CQSState currentCQSStateRecord) args ((time, messages, broadcasts) : restOfSimulatorStates) myid
    = (([nbboBroadcast]) : (cqsWrapper (CQSState updatedCQSStateRecord) args restOfSimulatorStates myid))
      where
       {- Insert associated logic for the rest of the timesteps in the simulation here.
          In the case of the CQS this logic must deal with any messages it receives,
          calculate the new NBBO, and output it as an nbboBroadcast.
       -}
```

Figure C.4: Example part code for CQS agent wrapper.

Figure C.4 shows the beginning stages of creating a new agent wrapper by illustrating how it was done for the CQS agent. Notice how the wrapper pattern matches against each of the four parameters shown in Figure C.2 and outputs a `[[Msg_t]]` as also shown in the figure. The logic for both states of the agent (initialisation and normal running) must be placed in the where block for each pattern match.

Once the user's custom agent has been created it can be included within their simulation by adding it to the list of agents passed to the `sim` function (see Figure C.1).

Appendix D

Code for CQS Agent

The CQS agent has too much associated code to include in the Appendix. The full code associated with the CQS agent can be found in the module 'CQS' within the root directory of the project.

```haskell
updateBBOsLogic :: [Msg_t] -> Map.Map Int CQSShortQuoteRecord -> Map.Map Int CQSShortQuoteRecord
updateBBOsLogic     []       finalBBOsMap          = finalBBOsMap
updateBBOsLogic (msg:rest)  intermediateBBOsMap    = updateBBOsLogic rest updatedBBOsMap
  where
    (CQSShortQuoteMessage (from, to) msgBBO) = msg
    currentBBO = Map.lookup from intermediateBBOsMap
    currentBBOTimestamp | currentBBO == Nothing = Nothing
                        | otherwise             = Just (_BBOTimestamp (fromJust currentBBO))
    msgBBOTimestamp     = Just (_BBOTimestamp msgBBO)
    updatedBBOsMap      | currentBBOTimestamp == Nothing
                           = Map.insert from msgBBO intermediateBBOsMap
                        | (fromJust msgBBOTimestamp) > (fromJust currentBBOTimestamp)
                           = Map.insert from msgBBO intermediateBBOsMap
                        | otherwise
                           = intermediateBBOsMap
```

Figure D.1: CQS Update BBO Function.

```haskell
cqsWrapper :: Agent_t
cqsWrapper (Emptyagentstate) args ((time, messages, broadcasts) : restOfSimulatorStates) myid
    = ( ( [nbboBroadcast] ) : (cqsWrapper (CQSState cqsStateRecord) args restOfSimulatorStates myid) )
      where
        initialNBBO = EmptyNBBO { _NBBOTimestamp = time, _Symbol = symbol }
        initialBBOs :: Map.Map Int CQSShortQuoteRecord
        initialBBOs = Map.empty
        cqsStateRecord =  CQSStateRecord {_NBBO = initialNBBO, _BBOs = initialBBOs  }
        nbboBroadcast = generateNBBOBroadcast time myid initialNBBO
cqsWrapper (CQSState currentCQSStateRecord) args ((time, messages, broadcasts) : restOfSimulatorStates) myid
    = ( ( [nbboBroadcast] ) : (cqsWrapper (CQSState updatedCQSStateRecord) args restOfSimulatorStates myid) )
      where
        currentNBBO = _NBBO currentCQSStateRecord
        currentBBOs = _BBOs currentCQSStateRecord
        messagesAreCQSShortQuoteMessage = foldr (&&) True (map msg_isCQSShortQuote messages)
        updatedBBOs | messagesAreCQSShortQuoteMessage = updateBBOsLogic messages currentBBOs
                    | otherwise = error "updateBBOsLogic: You have sent a non CQSShortQuoteMessage to a CQS agent."
        cleanNBBO  = generateCleanNBBO currentNBBO
        latestNBBO = generateNBBO updatedBBOs cleanNBBO
        nbboBroadcast | currentNBBO == latestNBBO = Hiaton
                      | otherwise                 = generateNBBOBroadcast time myid latestNBBO
        updatedCQSStateRecord = currentCQSStateRecord { _NBBO = latestNBBO, _BBOs = updatedBBOs }
```

Figure D.2: CQS Wrapper Function.

Appendix E

Testing and Validation

E.1 Simulator Extensions

This section lists the results of each of the tests described previously in section 3.4.

- Failing to provide the 'maxDelay' runtime argument. This should simply result in the generation of an arbitrary number of delay queues.
- Provide the 'maxDelay' runtime argument. This should result in the creation of the correct number of delay queues.

In order to determine the initial state that was actually being produced the trace function from the `Debug.Trace` message was used to output the data structure to the terminal when an experiment was run.

```
*ghci*> testfrontrunningNoMaxDelay
startsimstate:
([[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]
,0,[]
,[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]
,[[],[],[]]
,[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]
)
*ghci*> testfrontrunningWithMaxDelay
startsimstate:
([[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]
,0,[]
,[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]
,[[],[],[],[],[],[],[],[],[]]
,[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]
)
```

Figure E.1: Results of testing the Simstate_t data structure after initialisation.

Figure E.1 shows the initial `Simstate_t` tuple being generated for two experiments 'testfrontrunningNoMaxDelay' and 'testfrontrunningWithMaxDelay'. Both experiments had twenty one agent wrappers and a maximum broadcast channel of sixteen. The latter contained a max delay argument set to eight.

The output `Simstate_t` tuple shown in Figure E.1 is correct for both cases — when the max delay argument is and is not provided. When it is not provided the fifth tuple element contains three delay queues which is the arbitrary number it is supposed to. When the max delay argument is set to eight the testfrontrunningWithMaxDelay experiment returns a tuple with nine delay queues — as it should. The rest of the `Simstate_t` tuple is produced as normal in both tests. This test determined that the correct behaviour was occurring.

For the `sim_updatestate` function the following tests were established:

- Failing to provide all three runtime arguments (maxDelay, `FunArg1`, `FunArg2`). This should result in the simulator entering compatibility mode.
- Running a previously encoded experiment to intentionally enter compatibility mode. The trace file output should be compared to the output of the simulator prior to the extensions and there should be no difference between the files. If that is the case then this will confirm the correctness of compatibility mode.
- Encoding an example experiment which makes use of the simulator's newly created routed broadcast and delay functionality (this in turn will also be testing the three runtime arguments). The experiment should be devised such that the correct timestep each message should be received in is known in advance. Once the experiment is conducted the simulator's output should be compared to what is known to be accurate. If accurate the this will validate the correctness of the new delays and routed broadcasts mode. Additionally, it will confirm the correct coding of the maxDelay, `FunArg1`, `FunArg2` runtime arguments.

In order to see if compatibility mode is entered when either of required arguments are not present the trace function was once again used. In order to see whether the output files were exactly the same the UNIX program 'cksum' was used to produce a check sum for both files. The check sums were then compared.

```
*ghci*> testfrontrunning
Entering Compatability Mode
src > cksum Frontrunning-trace
1144769722 298493 Frontrunning-trace
src > cksum Frontrunning-traceOriginal
1144769722 298493 Frontrunning-traceOriginal
```

Figure E.2: Results of testing whether compatability mode is entered and the checksums of the output files.

As seen in Figure E.2 compatibility mode is correctly entered and the output files are exactly the same. In order to test the validity of the delay mechanism the delays for each agent in a test experiment were defined in an adjacency matrix (see Table E.1). In the figure the numbers represent the delay from sending a message from the agent on the left to the agent above and a Nothing indicates no connection between the agents.

	Agent 0	**Agent 1**	**Agent 2**	**Agent 3**	**Agent 4**	**Agent 5**
Agent 0	0	0	0	0	0	0
Agent 1	0	Nothing	1	2	3	4
Agent 2	0	1	Nothing	1	2	3
Agent 3	0	2	50	Nothing	1	2
Agent 4	0	3	2	1	Nothing	1
Agent 5	0	4	3	2	1	Nothing

Table E.1: Adjacency matrix representation of the simulation graph.

Additional relevant information for the experiment is the list of agents who are subscribed to each broadcast channel. Agent 1 is subscribed to channel 0, agents 2, 3, are subscribed to channel 1, and agents 3, 4, 5 are subscribed to channel 2.

Table E.2 shows the sender and receiver for every message which will be sent in the experiment. Additionally the type of message, either ‘message’ which is a direct message or ‘broadcast’, is shown. For direct messages the time the message is sent and the time it should be received is shown. For broadcast messages there is a row for the initial sending of the message (where the sender and the channel are populated) and a row for each agent which is subscribed to that channel (where the recipient and the time step it should be received are populated). The accompanying code for the experiment can be found electronically in the ExampleExperiment module within the

root directory of the project. It is not included in the appendix as it is too large to include in the report.

Type	Sender	Recipient	Channel	TS Sent	TS Received
Message	1	5		1	6
Message	2	4		2	5
Message	3	1		3	54
Message	4	2		4	7
Message	5	1		5	10
Broadcast	5		0	4	
Broadcast		1		4	9
Broadcast	4		1	9	
Broadcast		2			12
Broadcast		3			11
Broadcast	1		2	0	
Broadcast		3			3
Broadcast		4			4
Broadcast		5			5
Message	2	4		12	15
Message	1	5		10	15

Table E.2: Table showing each message which was to be sent from and to each agent with the timestep (TS) it should also be received.)

The experiment was run and the output compared with what was expected. They were exactly the same and as a result the delay and routed broadcast mechanisms of the simulator have been tested and verified to be accurate.

E.1.1 The FunArg3 Runtime Argument

For the `FunArg3` runtime argument the following test have been devised:

- Encode an example experiment whose agents all use of `FunArg3` to retrieve the IDs of the agents they will be communicating with. This will confirm the correctness of the `FunArg3` runtime argument.

The example experiment used to verify the mechanism for the delays and routed broadcasts (see Table E.1, Table E.2) was actually fully encoded using FunArg3 to retrieve the agent details. Figure E.3 shows the code for one of the agents (agent 1) who sends a broadcast to the second channel at time 0 and a message to the broker agent (agent 5) at time 1. The figure illustrates how the function within FunArg3 is extracted for the agent to retrieve the ID of broker (agent 5) it wishes to communicate with.

```
exampleWrapper1 :: Agent_t
exampleWrapper1 (Emptyagentstate) args ((time, msgs, broadcasts) : restOfStates) id
  = ( ( [msg] ) : (exampleWrapper1 (Emptyagentstate) args restOfStates id) )
    where
    msg | time == 0 = Broadcastmessage (id,2)
                      (Strbroadcast (Str "Broadcast from (1, 2). Sent at time 0."))
        | time == 1 = Debugmessage (id, broker1ID) "This message was sent at time 1"
                       ++ "from/to ("++(show 1)++",5). The delay is of "
                       ++  " 4 t.s. so it should arrive at time 6."
        | otherwise = Hiaton
    funArg3 | (arg_lookup "FunArg3" args) == EmptyArg = error "No FunArg3"
            | otherwise                               = (arg_lookup "FunArg3" args)
    getCorrespondingAgentIdentifier = arg_extract_FunArg3_value funArg3
    -- Use thee function to return the corresponding agentID.
    (AgentID broker1ID) = getCorrespondingAgentIdentifier (AgentLabel "Broker1")
```

Figure E.3: Example agent which utilises FunArg3 to get the ID of the broker they wish to communicate with.

E.2 The CQS Agent

		Timestep										
		0	1	2	3	4	5	6	7	8	9	10
Exchange 1	bidPrice	Nothing	391.01	391.02	391.00	391.04	391.05	391.07	Nothing	391.05	391.10	391.05
	bidSize	Nothing	100	200	200	400	500	600	Nothing	800	900	1,000
	offerPrice	Nothing	391.02	391.03	391.04	391.05	391.07	391.08	391.12	391.12	391.10	Nothing
	offerSize	Nothing	500	100	100	400	500	600	700	800	900	Nothing
Exchange 2	bidPrice	Nothing	391.01	391.02	391.02	391.03	391.04	391.05	391.08	391.07	391.08	391.11
	bidSize	Nothing	200	200	200	400	500	600	700	800	900	1,000
	offerPrice	Nothing	Nothing	391.03	391.03	391.05	391.06	391.08	391.09	391.11	Nothing	Nothing
	offerSize	Nothing	Nothing	100	100	400	500	1,500	700	800	Nothing	Nothing
Exchange 3	bidPrice	Nothing	Nothing	391.00	389.99	391.04	391.06	391.07	391.07	391.09	391.09	391.09
	bidSize	Nothing	Nothing	100	200	300	400	500	600	700	800	900
	offerPrice	Nothing	391.10	391.10	391.10	391.10	391.10	391.08	391.10	391.10	391.10	391.10
	offerSize	Nothing	1,100	1,100	1,100	1,100	1,100	1,100	1,100	1,100	1,100	900

Actual NBBO	Exchange	Should produce an Empty NBBO	2	2	2	1	3	1	2	3	1	2
	bidPrice		391.01	391.02	391.02	391.04	391.06	391.07	391.08	391.09	391.1	391.11
	bidSize		200	200	200	400	400	600	700	700	900	1,000
	Exchange		1	2	2	2	2	2	2	3	3	3
	offerPrice		391.02	391.03	391.03	391.05	391.06	391.08	391.09	391.1	391.1	391.1
	offerSize		500	100	100	400	500	1,500	700	1,100	1,100	900

Figure E.4: Tabular representation of the validation experiment for the CQS agent.

Figure E.4 is a tabular representation of the experiment previously mentioned in the section 5.4 of the report carried out to validate the correctness of the CQS agent. The experiment contains three agents whose corresponding rows are labeled clearly in the upper table. Each column represents a specific timestep of the experiment and in total there are 10.

For each exchange there is a row for all of the following: best bid price (bidPrice), best bid size (bidSize), best offer price (offerPrice), and best offer size (offerSize). Thus for all three exchanges the best bids and offers are defined precisely for all given timesteps.

The distinct table at the bottom is the actual NBBO which the CQS agent should generate if it is behaving correctly. So for example in the first timestep no exchange has any bids or offers (all values are Nothing) and so the CQS should output an Empty NBBO. In the next timestep the NBBO should be made from the best bid of exchange 2 and the best offer from exchange 1 etc. Best bids are for each timestep are highlighted in green and best offers in blue.

The corresponding code for the validation is too large to be included in the Appendix but is available in the module called 'CQSTests' in the root directory of the project. The test runs as expected and the CQS agent produces the values which it should so it is ready for inclusion in other projects.

E.3 Message Tags

The results of testing each of the following scenarios can be found in this section.

1. Attempting to use the `checkAndSetTagValue` function to set the qualifier to a value that is defined by FIX but not valid for the exchange that is required.
    - In this case the test was to attempt to set the value of the time in force tag as 'At the Opening' for the BYX exchange. Although 'At the Opening' is an acceptable FIX qualifier for the tag it is not accepted by BYX.
2. Attempting to set the value of a specific FIX tag to a qualifier which is actually only valid for a different tag.
    - In this case the value of the time in force tag is being attempted to be changed to the qualifier 'Market'. This is an error because 'Market' is a valid qualifier for the 'OrdType' tag not the time in force one.

The first test was devised to simulate a user's actions when wanting to change the value of a time in force tag to 'At the Opening' for a message to BYX. The user is unaware of the fact that that particular qualifier, although a valid FIX value, is not valid for BYX. As a result when the function is actually called the user is greeted with a runtime exception. Figure E.5 illustrates the result of the test.

```
messageTagsTest1 = checkAndSetTagValue BYX timeInForce_FT
                      TimeInForce_FTVal_OPG
-- After the function is called the user is greeted with:
Exception: Your tagValue of TimeInForce_FTVal_OPG
is invalid for the exchange: BYX
```

Figure E.5: Results of test 1 for message tags.

```
-- Create a tag with the required type but place an incorrect
-- value inside of it.
messageTagsTest2 :: Tag (TagQualifier TimeInForce_FT)
messageTagsTest2 = Tag
  { tagType  = FIXTag, tagName  = "TimeInForce", tagID = 59,
    tagValue = TagValue OrdType_FTVal_Market }
```

Figure E.6: Set up of test 2 for message tags.

The second test simulates a user trying to create a tag with a completely invalid value. The value `OrdType_FTVal_Market` is intended to be used with a tag of the type `:: Tag (TagQualifier OrdTyep_FT)` rather than a time in force tag as is being done. Figure E.7 shows the compil time error received as a result.

```
 Couldn't match type 'OrdType_FT' with 'TimeInForce_FT'
    Expected type: TagQualifier TimeInForce_FT
      Actual type: TagQualifier OrdType_FT
```

Figure E.7: Results of test 2 for message tags.

The third and final test is to see whether the `checkAndSetTagValue` function is able to cope with the constraints set by the type system. The test checks to see whether when given the correct parameters the function returns the newly created tag. It simulates the user wanting to create a time in force tag intended to be sent to BYX with the qualifier 'fill or kill'. Figure E.8 illustrates the test's success.

```
-- Set up the test.
messageTagsTest3 = checkAndSetTagValue BYX timeInForce_FT TimeInForce_FTVal_FOK
-- The tag can be called successfully as demonstrated in this call
-- to GHCI (the interpreter).
ghci> messageTagsTest3
ghci> Tag {tagType = FIXTag, tagName = "TimeInForce", tagID = 59,
     tagValue = TagValue TimeInForce_FTVal_FOK}
```

Figure E.8: Successfully encoding a tag.

Appendix F

Miscellaneous Code

The following appendix contains a large part of the relevant code referenced within the text. Code not in the Appendix can be found through the electronic submission.

F.1 Functions for Insertion into Lists

```
-- Inserts an element at the front of the list.
insertAsHead :: a -> [a] -> [a]
insertAsHead element  list = (element : list )
```

Figure F.1: insertAsHead Function

```
-- For lists of lists i.e. [ [a] ]. This inserts an element into the inner
-- list specified by the given index, and returns the newly created list.
insertIntoInnerList :: Int -> a  -> [[a]] -> [[a]]
insertIntoInnerList  any  element   []   = error "insertIntoInnerList: too large of an index."
insertIntoInnerList   0   element (x:xs) = ( (insertAsHead element x) : xs)
insertIntoInnerList index element (x:xs) = (x : (insertIntoInnerList (index-1) element xs))
```

Figure F.2: insertIntoInnerList Function

```haskell
-- Inserts a list of elements [a], one by one into the inner list
-- specified by the index, to create a final [ [a] ].
insertElementsIntoInnerList :: Int ->  [a] -> [[a]] -> [[a]]
insertElementsIntoInnerList index       []          finalList        = finalList
insertElementsIntoInnerList index (element:rest) intermediateList
  = insertElementsIntoInnerList index (rest) (insertIntoInnerList index element intermediateList)
```

Figure F.3: insertElementsIntoInnerList Function

F.2 Generating Broadcast Subscription Lists

```
genBCastSubscriptionLists :: [( String, (Agent_t, [Int]) ) ] -> [ [Int] ]
genBCastSubscriptionLists agentList = finalBCastSubscriptionLists
  where
    maxBCastGroup               = maximum (concat (map snd (map snd agentList)) ++[0])
    emptyBCastSubscriptionLists = map (\ x -> []) [0..maxBCastGroup]
    agentsWithIDs               = (zip [1..] agentList)
    finalBCastSubscriptionLists = addIDsToBCastSubscriptionLists agentsWithIDs emptyBCastSubscriptionLists
      where
        -- Iterate over every agent in the agentList, delegating the addition of their ID to the
        -- relevant subscription lists to addSubsForSingleAgent.
        addIDsToBCastSubscriptionLists        []            finalSubscriptionLists = finalSubscriptionLists
        addIDsToBCastSubscriptionLists (agent:restOfAgents)        subLists
             = addIDsToBCastSubscriptionLists restOfAgents (addAgentSubscriptions agent subLists)
          where
            -- Iterate over every broadcast channel the agent is subscribed to, adding their ID
            -- to each channel's subscription list.
            addAgentSubscriptions (agentID, (label, (wrapper, []))) subLists = subLists
            addAgentSubscriptions (agentID, (label, (wrapper, (subscribedChannel:rest)))) subLists
              = addAgentSubscriptions (agentID, (label, (wrapper, rest))) (insertIntoInnerList
                 subscribedChannel agentID subLists)
```

Figure F.4: genBCastSubscriptionLists Function

F.3 Updating the Simulator's State

```haskell
sim_updatestate :: Int -> [Arg_t] -> Simstate_t -> [[Msg_t]] -> [Int] -> Simstate_t
sim_updatestate time args cleanState agentMessages myrands
  -- The 'maxDelay', 'funArg1', and 'funArg2' runtime arguments are a set. If you have one
  --  you need to have the other two. If you are missing one one or more then the simulator cannot
  -- function with delays and routed broadcasts. As a result, the compatability mode branch will be taken.
  | arg_lookup "maxDelay" args == EmptyArg
    = (sim_updatestate_compatability_mode time args cleanState agentMessages myrands)
  | arg_lookup "FunArg1" args == EmptyArg
    = (sim_updatestate_compatability_mode time args cleanState agentMessages myrands)
  | arg_lookup "FunArg2" args == EmptyArg
    = (sim_updatestate_compatability_mode time args cleanState agentMessages myrands)
  | otherwise
    = (sim_updatestate_with_delayQs_and_routed_broadcasts time args cleanState agentMessages myrands)
```

Figure F.5: Function for updating the simulator state.

```
-- This is the sim_updatestate function as it was prior to extending Simstate_t to
-- include delay queues and routed broadcasts. It is called under 'compatability mode'
-- for backwards compatability of previous experiments.
sim_updatestate_compatability_mode :: Int -> [Arg_t] -> Simstate_t -> [[Msg_t]] -> [Int] -> Simstate_t
sim_updatestate_compatability_mode t args (m, b, c, br, dq, rb) [] myrands
 = (newm, t, safehd newm "sim_updatestate_compatability_mode", reverse br2, dq, rb)
  where
   newm = reverse (map reverse m2)
   m2 = if randomise == True
        then ((map (randomWrap myrands) (take ((length m) - 1) m)) ++ (drop ((length m) - 1) m))
        else m
   br2 = if randomise == True
         then (map (randomWrap myrands) br)
         else br
   randomise = if (arg_findval "Randomise" args) /= (-1)
               then True
               else False

sim_updatestate_compatability_mode t args (m, b, c, br, dq, rb) (x:xs) myrands
 = sim_updatestate_compatability_mode t args (fm, b, c, fcasts, dq, rb) xs (drop 72 myrands)
   where
     --updatemsgs takes messages from agents and puts them in msgqueues for agents (id 0 = for sim)
     updatemsgs 0 (y:ys) =[(filter ((==0).msg_getid) (filter ((not).msg_isbroadcast) x)) ++ y]
     updatemsgs n (y:ys) =((filter ((==n).msg_getid) (filter ((not).msg_isbroadcast) x)) ++ y):(updatemsgs (n-1) ys)
     updatecasts 0 (y:ys) =[(filter ((==0).msg_getid) (filter msg_isbroadcast x)) ++ y]
     updatecasts n (y:ys) =((filter ((==n).msg_getid) (filter msg_isbroadcast x)) ++ y): (updatecasts (n-1) ys)
     newm =  (updatemsgs ((length m) - 1) ( m))
     newcasts = (updatecasts ((length br) - 1) ( br))
     (fm, fcasts) = (newm, newcasts)
```

Figure F.6: Function for updating the simulator state in compatibility mode.

```
sim_updatestate_with_delayQs_and_routed_broadcasts :: Int-> [Arg_t]-> Simstate_t-> [[Msg_t]]-> [Int]-> Simstate_t
sim_updatestate_with_delayQs_and_routed_broadcasts time args
   (cleanAToAMsgs, t, harnessMessages, cleanBroadcastChannels, delayQs, cleanRBLists) agentMessages myrands
 = (finalAToAMessages, time, finalHarnessMessages, finalBroadcasts, finalDelayQs, finalRBroadcasts)
   where
    getTimeStepDelay = (arg_extract_FunArg1_value.(arg_lookup "FunArg1")) args
    getAgentsSubscribedToChannel =  (arg_extract_FunArg2_value.(arg_lookup "FunArg2")) args
    agentToAgentMessages = (filter ((not).msg_isbroadcast)) (concat agentMessages)
    broadcastMessages = (filter (msg_isbroadcast)) (concat agentMessages)
    delayQsWithAgentToAgentMsgs = insertAgentToAgentMessagesIntoDelayQs agentToAgentMessages
                                  delayQs getTimeStepDelay
    delayQsWithAToAMsgsAndRBs = createAndInsertRBsIntoDelayQs broadcastMessages delayQsWithAgentToAgentMsgs
                                getAgentsSubscribedToChannel getTimeStepDelay
    messagesToBeSent
     = (cpsafehd "taking head of updatedDelay Qs" delayQsWithAToAMsgsAndRBs)
    finalDelayQs = (tail delayQsWithAToAMsgsAndRBs) ++ [[]]
    aToAMessagesToBeSent = filter ((not).msg_isbroadcast) messagesToBeSent
    rBMessagesToBeSent = filter (msg_isbroadcast) messagesToBeSent
    groupedAgentToAgentMessages = moveMessagesToCorrectToList aToAMessagesToBeSent cleanAToAMsgs
    groupedRoutedBroadcastMsgs  = moveMessagesToCorrectToList rBMessagesToBeSent cleanRBLists
    finalHarnessMessages = cpsafehd "head (finalHarnessMessages)" groupedAgentToAgentMessages
    amendedMyRands              = (drop (72*(length agentMessages)) myrands)
    randomise         | (arg_lookup "maxDelay" args) /= EmptyArg  = True
                      | otherwise                                 = False
    aToAMessagesWithoutHarness    = drop 1 groupedAgentToAgentMessages
    finalAToAMessages | randomise = (finalHarnessMessages :
                                     (map (randomWrap amendedMyRands) aToAMessagesWithoutHarness))
                      | otherwise = groupedAgentToAgentMessages
    finalRBroadcasts  | randomise = (map (randomWrap amendedMyRands) groupedRoutedBroadcastMsgs)
                      | otherwise = groupedRoutedBroadcastMsgs
    finalBroadcasts = cleanBroadcastChannels
```

Figure F.7: Function for updating the simulator state with routed broadcasts and delay queues.

```
-- Takes a list of messages (they should be pre filtered to only include agent to agent messages), delay queues,
-- and a function providing a mapping to the associated delay between two agents. It inserts each message, one by
-- one into the corresponding delay queue.
insertAgentToAgentMessagesIntoDelayQs :: [Msg_t] -> [[Msg_t]] -> (Int -> Int -> Int) -> [[Msg_t]]
insertAgentToAgentMessagesIntoDelayQs        []       finalDelayQs getTimeStepDelay = finalDelayQs
insertAgentToAgentMessagesIntoDelayQs (message : rest)  delayQs    getTimeStepDelay
 = insertAgentToAgentMessagesIntoDelayQs rest updatedDelayQs getTimeStepDelay
   where
     -- Grab the message from/to IDs (if it is a Hiaton it will be (0,0) ).
     agentFromID         = msg_getfromid message
     agentToID           = msg_getid message -- msg_getid gets the 'to' ID of a message.
     -- Lookup the associated delay between the agent sending the message and the one to receive it.
     associatedDelay     = getTimeStepDelay agentFromID agentToID
     -- Insert the message into the delayQs at the correct index. The correct index corresponds
     -- to the amount the message is delayed.
     updatedDelayQs      = insertIntoInnerList associatedDelay message delayQs
```

Figure F.8: Function for inserting direct messages into the relevant delay queue.

```haskell
-- Takes a list of messages (they should be pre-filtered to only include broadcast messages), delay queues, a
-- function providing a mapping from the broadcast channel to the agentIDs of those agents subscribed to the
-- channel, and a function providing the associated delay between two agents. It looks up the the agentIDs of those
-- subscribed to the broadcast channel the broadcast messages is sent to. Then it 'explodes' to broadcast, creating
-- a 'RoutedBroadcast' for every agent. The delay between the sender of the original broadcast and the recipient
-- agent is then looked up and the routed broadcast is placed in the relevant delay queue. The final delay queues
-- with each routed broadcast inside is returned.
createAndInsertRBsIntoDelayQs :: [Msg_t] -> [[Msg_t]] -> (Int -> [Int]) -> (Int -> Int -> Int) -> [[Msg_t]]
createAndInsertRBsIntoDelayQs   []  finalDelayQs getAgentsSubscribedToChannel getTimeStepDelay = finalDelayQs
createAndInsertRBsIntoDelayQs (bcastMsg : rest)  intermediateDelayQs getAgentsSubscribedToChannel getTimeStepDelay
  = createAndInsertRBsIntoDelayQs rest updatedDelayQs getAgentsSubscribedToChannel getTimeStepDelay
    where
      -- Grab the from ID of the broadcast sender and the broadcast channel 'to' ID of the broadcast.
      agentFromID                = msg_getfromid bcastMsg
      broadcastChannelToID       = msg_getid bcastMsg -- msg_getid gets the 'to' ID of a message.
      -- Get the list of agents subscribed to the broadcast channel.
      broadcastChannelSubs       = getAgentsSubscribedToChannel broadcastChannelToID
      -- Recurse through the list of subscribed agents:
      updatedDelayQs             = explodeBroadcastAndInsertIntoDelayQs broadcastChannelSubs intermediateDelayQs
        where
          explodeBroadcastAndInsertIntoDelayQs       []         updatedDelayQs = updatedDelayQs
          explodeBroadcastAndInsertIntoDelayQs (agentToID:rest)    delayQs
           = explodeBroadcastAndInsertIntoDelayQs rest delayQsWithNewRBInserted
            where
              newRoutedBroadcast          = (RoutedBroadcast (agentFromID, agentToID) bcastMsg)
              associatedDelay             = getTimeStepDelay agentFromID agentToID
              delayQsWithNewRBInserted    = insertIntoInnerList associatedDelay newRoutedBroadcast delayQs
```

Figure F.9: Function for creating and inserting routed broadcasts into the relevant delay queue.

```
-- moveMessagesToCorrectToList is given a list of messages [Msg_t] and a list of lists
-- of messages [ [ Msg_t] ]. It inserts each message one by one into the correct inner
-- list of the second argument. The correct inner list corresponds to the message's 'to'
-- field in the 'from/to' tuple.
moveMessagesToCorrectToList :: [Msg_t] -> [ [Msg_t] ] -> [ [ Msg_t ] ]
moveMessagesToCorrectToList   []   finalListOfLists = finalListOfLists
moveMessagesToCorrectToList (x:xs)   intermediate
  = moveMessagesToCorrectToList xs (insertIntoInnerList (msg_getToID x) x intermediate)
   where
     msg_getToID = msg_getid -- Using a synonym to make intent clearer.
```

Figure F.10: Function for moving messages to the correct inner list.

F.4 Accessory Functions for Filtering the List of All States

```
si warn list n = if n < (length list)
                 then  list!!n
                 else error warn
```

Figure F.11: Accessory function, providing a 'safe index' into a list.

```
safehd :: [a] -> [Char] -> a
safehd x caller = if not(null x)
                  then head x
                  else error ("safehd - from " ++ caller)
```

Figure F.12: Accessory function, providing 'safe' access to the head of a list.

F.4.1 Old Versus New Versions of Functions

```
sim_gettime :: Simstate_t -> Int
sim_gettime (m, b, c, br) = b
```

Figure F.13: Old accessory function used for retrieving the time of a simulator state

```
sim_gettime :: Simstate_t -> Int
sim_gettime (m, b, c, br, dq, rb) = b
```

Figure F.14: Updated accessory function used for retrieving the time of a simulator state

```
sim_getmymessages :: Simstate_t -> Int -> [Msg_t]
sim_getmymessages (m,b,c,br) idnum = si "sim:1" m idnum
```

Figure F.15: Old accessory function, used for retrieving the messages of an agent with the given ID.

```
sim_getmymessages :: Simstate_t -> Int -> [Msg_t]
sim_getmymessages (m, b, c, br, dq, rb) idnum = si "sim_getmymessages" m idnum
```

Figure F.16: Updated accessory function, used for retrieving the messages of an agent with the given ID.

```
sim_getmybroadcasts :: Simstate_t -> Int -> [Msg_t]
sim_getmybroadcasts (m,b,c,br) groupnum = si "sim:2" br groupnum
```

Figure F.17: Old accessory function, used for retrieving the broadcast messages sent to a particular channel.

```
sim_getmybroadcasts :: Simstate_t -> Int -> [Msg_t]
sim_getmybroadcasts (m, b, c, br, dq, rb) groupnum = si "sim_getmybroadcasts" br groupnum
```

Figure F.18: Updated accessory function, used for retrieving the broadcast messages sent to a particular channel.

```
sim_getmyroutedbroadcasts :: Simstate_t -> Int -> [Msg_t]
sim_getmyroutedbroadcasts (m, b, c, br, dq, rb) idnum = si "sim_getmyroutedbroadcasts" rb idnum
```

Figure F.19: Newly created accessory function, used for retrieving the routed broadcasts sent to a particular channel.

F.5 Functions Related to FunArg3

```
-- | Transform for sim is intended to transform a user defined agent list to the
--   form [ (wrapper, [0..16]) ]. This transformed list is then passed
--   as the 'agents' argument to sim. This is done (rather than changing the sim
--   function to accept the new form) for backwards compatability.
transformForSim :: [( String, (Agent_t, [Int]) )]  -> [ (Agent_t, [Int]) ]
transformForSim                []                          = []
transformForSim ( (key, (wrapper, bcChannels)) : rest ) = (wrapper, bcChannels) : (transformForSim rest)
```

Figure F.20: Code for the transforming the user defined agent list from the new style to the old.

```haskell
-- | When given a user defined agent list of the form [ (String, (Agent_t, [Int]))], this returns a
--   bidirectional map between each agentLabel :: String and the agent's unique ID :: Int.
generateAgentBimap :: [(String, (Agent_t, [Int]) )] -> Bimap String Int
generateAgentBimap agentList = generateAgentBimap' agentListWithIDs []
  where
    generateAgentBimap' ::    [(String, (Int, Agent_t, [Int]) )] -> [(String, Int)] -> Bimap String Int
    generateAgentBimap' [] finalMappings = (Data.Bimap.fromList finalMappings)
    generateAgentBimap' ((agentLabel, (agentID, wrapper, bcGroups)) : rest) intermediateMappings
      = generateAgentBimap' rest (intermediateMappings ++ [(agentLabel,agentID)])
    agentListWithIDs = addAgentIDs agentList 1 -- Start adding IDs from 1, because 0 is the simulator itself
      where
        -- This adds the same unique ID the sim function would add to each agent. This is just done
        -- in ascending order from the bottom of the list. ID 0 is reserved for the simulator
        --  itself - so IDs should always be added from 1 upwards.
        addAgentIDs :: [(String, (Agent_t, [Int]))] -> Int -> [(String, (Int, Agent_t, [Int]) )]
        addAgentIDs                    []                  agentID = []
        addAgentIDs ( (key, (wrapper, bcGroups)) : rest ) agentID
          = ((key, (agentID, wrapper, bcGroups)) : (addAgentIDs rest (agentID+1)) )
```

Figure F.21: Code for the generating a bidirectional mapping between the agent labels and IDs.

```
-- Generate an agent bidirectional mapping from the agentLabel in the list of user defined agents to their Int ID.
biMap = (generateAgentBimap) listOfAgents
getCorrespondingAgentIdentifier :: AgentIdentifier_t -> AgentIdentifier_t
-- When given an (AgentLabel label) we return an error if it does not exist within the BiMap
-- or (if it does exist) the corresponding (AgentID id).
getCorrespondingAgentIdentifier (AgentLabel label) = correspondingAgentID
  where
    lookupValue    = Data.Bimap.lookup label biMap -- Returns a Maybe Int
    extractedValue | lookupValue == Nothing
                     = error ("getCorrespondingAgentIdentifier: No mapping from AgentLabel "++(show label))
                   | otherwise              = fromJust lookupValue
    correspondingAgentID = AgentID extractedValue
-- When given an (AgentID id) we return an error if it does not exist within the BiMap
-- or (if it does exist) the corresponding (AgentLabel label).
getCorrespondingAgentIdentifier (AgentID id)       = correspondingAgentLabel
  where
    lookupValue    = Data.Bimap.lookupR id biMap -- Returns a Maybe String
    -- If the lookup returns a nothing - return an error. Otherwise extract the String from the Just constructor.
    extractedValue | lookupValue == Nothing
     = error ("getCorrespondingAgentIdentifier: No mapping from AgentID "++(show id))
                   | otherwise              = fromJust lookupValue
    -- Wrap the extractedValue into an AgentLabel constructor so it can be returned:
    correspondingAgentLabel = AgentLabel extractedValue
-- Wrap the function in a FunArg3 constructor so it can be included in the list of args:
funArg3 = FunArg3 (Str "FunArg3", getCorrespondingAgentIdentifier)
```

Figure F.22: Code for the getCorrespondingAgentIdentifier function and how to create a FunArg3.

F.6 BATS Messages and FIX Tags

F.6.1 BATS Messages

The declaration `data BATSMessage =` ... is quite large and so the code for it is split over a number of pages.

```
data BATSMessage
  = BATSNewOrderMessage
    { _BNO_SendingTime_FT          :: Tag (TagQualifier SendingTime_FT)
    , _BNO_ClOrdID_FT              :: Tag (TagQualifier ClOrdID_FT)
    , _BNO_OrderQty_FT             :: Tag (TagQualifier OrderQty_FT)
    , _BNO_OrdType_FT              :: Tag (TagQualifier OrdType_FT)
    , _BNO_Side_FT                 :: Tag (TagQualifier Side_FT)
    , _BNO_Symbol_FT               :: Tag (TagQualifier Symbol_FT)
    , _BNO_RoutingInst_BT          :: Tag (TagQualifier RoutingInst_BT)
    , _BNO_ExecInst_FT             :: Tag (TagQualifier ExecInst_FT)
    , _BNO_Price_FT                :: Tag (TagQualifier Price_FT)
    , _BNO_TimeInForce_FT          :: Tag (TagQualifier TimeInForce_FT)
    , _BNO_ExpireTime_FT           :: Tag (TagQualifier ExpireTime_FT)
    }
    | ...
```

Figure F.23: Part 1 of the data declaration for a BATSMessage.

```
data BATSMessage
  = ...
    |  BATSCancelOrderRequestMessage
    { _BCOR_SendingTime_FT :: Tag (TagQualifier SendingTime_FT)
    , _BCOR_OrderID_FT     :: Tag (TagQualifier OrderID_FT)
    }
    | BATSModifyOrderRequestMessage
    { _BMOR_SendingTime_FT :: Tag (TagQualifier SendingTime_FT)
    , _BMOR_OrderID_FT     :: Tag (TagQualifier OrderID_FT)
    , _BMOR_Price_FT       :: Tag (TagQualifier Price_FT)
    , _BMOR_OrderQty_FT    :: Tag (TagQualifier OrderQty_FT)
    , _BMOR_OrdType_FT     :: Tag (TagQualifier OrdType_FT)
    }
    | BATSOrderAcknowledgmentMessage
    { _BOA_SendingTime_FT  :: Tag (TagQualifier SendingTime_FT)
    , _BOA_Symbol_FT       :: Tag (TagQualifier Symbol_FT)
    , _BOA_OrderID_FT      :: Tag (TagQualifier OrderID_FT)
    , _BOA_TransactTime_FT :: Tag (TagQualifier TransactTime_FT)
    , _BOA_ClOrdID_FT      :: Tag (TagQualifier ClOrdID_FT)
    , _BOA_LeavesQty_FT    :: Tag (TagQualifier LeavesQty_FT)
    }
    | ...
```

Figure F.24: Part 2 of the data declaration for a BATSMessage.

```
data BATSMessage
  = ...
    | BATSOrderRejectedMessage
    { _BOR_SendingTime_FT  :: Tag (TagQualifier SendingTime_FT)
    , _BOR_Symbol_FT       :: Tag (TagQualifier Symbol_FT)
    , _BOR_ClOrdID_FT      :: Tag (TagQualifier ClOrdID_FT)
    , _BOR_TransactTime_FT :: Tag (TagQualifier TransactTime_FT)
    , _BOR_OrdRejReason_FT :: Tag (TagQualifier OrdRejReason_FT)
    , _BOR_Text_FT         :: Tag (TagQualifier Text_FT)
    }
    | BATSOrderModifiedMessage
    { _BOM_SendingTime_FT   :: Tag (TagQualifier SendingTime_FT)
    , _BOM_Symbol_FT        :: Tag (TagQualifier Symbol_FT)
    , _BOM_TransactTime_FT  :: Tag (TagQualifier TransactTime_FT)
    , _BOM_OrderID_FT       :: Tag (TagQualifier OrderID_FT)
    , _BOM_Price_FT         :: Tag (TagQualifier Price_FT)
    , _BOM_OrderQty_FT      :: Tag (TagQualifier OrderQty_FT)
    , _BOM_OrdType_FT       :: Tag (TagQualifier OrdType_FT)
    }
 -- | BATSOrderRestated        - Currently not implemented.
 -- | BATSUserModifiedRejected - Currently not implemented.
```

Figure F.25: Part 3 of the data declaration for a BATSMessage.

```
data BATSMessage
  = ...
    | BATSOrderCancelledMessage
    { _BOC_SendingTime_FT   :: Tag (TagQualifier SendingTime_FT)
    , _BOC_Symbol_FT        :: Tag (TagQualifier Symbol_FT)
    , _BOC_ClOrdID_FT       :: Tag (TagQualifier ClOrdID_FT)
    , _BOC_TransactTime_FT  :: Tag (TagQualifier TransactTime_FT)
    , _BOC_OrderID_FT       :: Tag (TagQualifier OrderID_FT)
    }
 -- | BATSCancelRejectedMessage - Currently not implemented.
    | BATSOrderExecutionMessage
    { _BOEX_SendingTime_FT  :: Tag (TagQualifier SendingTime_FT)
    , _BOEX_Symbol_FT       :: Tag (TagQualifier Symbol_FT)
    , _BOEX_ClOrdID_FT      :: Tag (TagQualifier ClOrdID_FT)
    , _BOEX_TransactTime_FT :: Tag (TagQualifier TransactTime_FT)
    , _BOEX_OrderID_FT      :: Tag (TagQualifier OrderID_FT)
    , _BOEX_ExecID_FT       :: Tag (TagQualifier ExecID_FT)
    , _BOEX_LastShares_FT   :: Tag (TagQualifier LastShares_FT)
    , _BOEX_LastPx_FT       :: Tag (TagQualifier LastPx_FT)
    , _BOEX_LeavesQty_FT    :: Tag (TagQualifier LeavesQty_FT)
    , _BOEX_ExecType_FT     :: Tag (TagQualifier ExecType_FT)
    , _BOEX_TradeLiquidityIndicator_BT :: Tag (TagQualifier TradeLiquidityIndicator_BT)
    , _BOEX_ContraBroker_FT :: Tag (TagQualifier ContraBroker_FT)
    } deriving (Show, Eq)
```

Figure F.26: Part 4 of the data declaration for a BATSMessage.

F.6.2 FIX Tags

Each FIX tag includes a declaration for the tag, a neutral definition of that tag with an `EmptyTagValue` and an instance of that tag into the `TagInterface` class. There are too many to include in the appendix so only a few are shown. The entire

implementation is available to view in the modules Messages.FIXTags and Messages.BATSTags.

```
data ClOrdID_FT = ClOrdID_FT deriving (Show, Eq)
clOrdID_FT :: Tag (TagQualifier ClOrdID_FT)
clOrdID_FT = Tag
  { tagType = FIXTag, tagName = "ClOrdID", tagID = 11, tagValue = EmptyTagValue }
instance TagInterface ClOrdID_FT where
  data TagQualifier ClOrdID_FT
    = ClOrdID_FTVal AgentInt Int deriving (Show, Eq)
  validQualifier any qualifier | (agentID >= 0) && (uniqueID >= 0) = True
                               | otherwise  = error "validQualifier ClOrdID: "
     ++ "You either have an invalid agentID or uniqueID (one is negative)."
    where
      (ClOrdID_FTVal agentID uniqueID) = qualifier
```

Figure F.27: Part 1 of the implementation of FIX tags.

```
data ExecInst_FT = ExecInst_FT deriving (Show, Eq)
execInst_FT :: Tag (TagQualifier ExecInst_FT)
execInst_FT  = Tag { tagType = FIXTag, tagName = "ExecInst", tagID = 18, tagValue = EmptyTagValue  }
instance TagInterface ExecInst_FT where
  data TagQualifier ExecInst_FT
    = ExecInst_FTVal_ISO
    deriving (Show, Eq)
  validQualifier BYX qualifier | qualifier == ExecInst_FTVal_ISO = True
                               | otherwise                       = False

data LastPx_FT = LastPx_FT deriving (Show, Eq)
lastPx_FT :: Tag (TagQualifier LastPx_FT)
lastPx_FT  = Tag { tagType = FIXTag, tagName = "LastPx", tagID = 31, tagValue = EmptyTagValue  }
instance TagInterface LastPx_FT where
  data TagQualifier LastPx_FT
    = LastPx_FTVal Double
    deriving (Show, Eq)
  validQualifier any qualifier | (num >= 0) = True
                               | otherwise  = error "validQualifier LastPx_Ft: You "
                                 ++ " cannot have a negative LastPx_FT"
    where
      (LastPx_FTVal num) = qualifier
```

Figure F.28: Part 2 of the implementation of FIX tags.

F.6.3 Tag Accessory Functions

```
-- | Returns true if given an EmptyTagValue
isEmptyTagValue :: TagValue a -> Bool
isEmptyTagValue EmptyTagValue = True
isEmptyTagValue anyOther      = False

-- | String expression which is the default output when showing an EmptyTagValue.
defaultShowEmptyTagValue = "Empty"

-- | Polymorphic function which sets any tag's value.
setTagValue :: Tag a -> a -> Tag a
setTagValue tag value = tag { tagValue = TagValue value }

-- | Polymorphic function which extracts any tag's value from the TagValue constructor.
extractTagValue :: TagValue a -> Maybe a
extractTagValue (TagValue any) = Just any
extractTagValue EmptyTagValue  = Nothing
```

Figure F.29: Accessory functions for tags.

```haskell
-- | Takes a tag, a tag value, and an exchange. Checks whether the value is
--   valid for that exchange and if it is it returns the tag with that
--   value set. If it is not valid for that exchange an error is
--   returned and the program is halted.
checkAndSetTagValue :: Exchanges -> Tag (TagQualifier tag)
                    -> TagQualifier tag -> Tag (TagQualifier tag)
checkAndSetTagValue exchange tag tagValue
   | validQualifier exchange tagValue = setTagValue tag tagValue
   | otherwise = error ("Your tagValue of "++(show tagValue)++" is "
     ++ " invalid for the exchange: "++(show exchange))
```

Figure F.30: Accessory functions for tags part 2.