Abstract

The Flash Crash in 2010 with its long debate about the causes has demonstrated that we still do not fully understand how the interaction of different market participants can have such dramatic results. Moreover, it shows that there is a need for new ways to model financial systems eradicate a flaw. Due to this, we introduce a new approach to model a financial system by using the Stochastic Pi-Machine (SPiM), a programming language which has previously been applied in the area of Biology, to model the information flows in an exchange-based trading environment. We show how to realize the exchange and the behaviour of other market participants in a modular way, such that future users can easily adapt and extend it. Additionally, we also provide tools which improve the usability of SPiM as a general simulation instrument such as a library with functions to simplify the application of lists or a pseudo random number generator. We finally conclude that it is possible to model financial system with SPiM and that SPiM might have its place in the world of financial simulation, provided that the language will be further improved.

Contents List

1 Introduction

The computerization of the global exchanges has been going on for decades and since the rise of the High Frequency Traders a new level of tremendous execution speeds and complex trading algorithms has been reached. However, this development also entails risks, risks which manifest themselves in events like the Flash Crash in 2010, where the aggregated market capitalisation of all listed companies of the Dow Jones Industrial Average dropped by 850 billion Dollars within half an hour before rebounding (Tett, 2011). Especially this incident with its high media coverage and its long going public discussion about the reasons why it had happened, demonstrates that there is an urgent need for new ways to analyse the modern markets.

The financial markets with their network-like nature and their many different market participants are a complex field and difficult to model. As a result, we want to focus our efforts on a tool which has been proven useful in another area with similar complexity. We therefore devote this work to the exploration of the potential of the Stochastic Pi Machine (SPiM) for finance-related issues. Up to now, SPiM has only been used to model processes in a biological context, however, we aim to adopt this programming language to help us in a financial context as well by modelling the information and data flows in the financial world. In a first attempt we intend to use the SPiM to set up a model for a simple financial market model which can be used in the future as a foundation for a comprehensive simulation and analysis of the Hot Potato effect (HPE). This effect, which describes the occurrence of a high trading volume between the high frequency market makers and a low change in the net position of securities, i.e. the security is traded between the market makers back and forth, was a notable feature of the Flash Crash (Kirilenko et al, 2011). It is therefore our goal to give financial researchers a new tool to get a better insight into the interplay of the modern markets, especially with focus on the HPE, and to support further research in the areas of Finance and Economics.

Based on extensive research and with a focus on the Hot Potato effect, we derive a model for an exchange-based trading system with 2 different types of market agents in addition to the exchange. We show how to realize the exchange and the behaviour of the other market participants in a modular way, such that future users can easily adapt and extend it. We also improve the usability of SPiM by providing a

function/process -library, containing tools to manipulate lists and a pseudo random number generator, which simplifies the development process. Furthermore, we will discuss why SPiM is not yet ready for high-scale and complex simulations in the area of finance and what has to be done to improve the SPiM.

The organization of the report is based on the software development cycle. The second chapter gives finance-related background knowledge and also provides a small introduction into SPiM. The third chapter analyses the requirements for the work, the potential challenges, as well as why SPiM has been chosen. In the fourth chapter the main design decisions are presented and the fifth shows the actual implementation of the system and auxiliary instruments. The sixth chapter demonstrates the testing strategies, while the last chapter deals with conclusion.

2 Background knowledge

2.1 Exchange-based Trading

In this section we will explain how an exchange-based trading environment works, what traders are and we will take a look at market makers and how high frequency market makers are different from them.

2.1.1 Exchange and Order book

In an exchange-based trading environment the different traders do not trade directly with each other but via an exchange or to be more precise via the order book of the exchange and two different groups of orders, limit and market orders.

Figure 1 shows a schematic representation of an order book. The order book is basically a storage element which stores the two different types of limit orders, on the one hand there is the bid limit order, which indicates the willingness of a trader to buy a specified volume of the traded good at the specified price, and on the other hand there is the offer limit order, which indicates the willingness to sell the volume at the specified price. There is normally a difference between the lowest price level of all offers, the so called best offer price, and the highest price level of all bids, called the best bid price. This difference is referred to as the spread. Furthermore, the step size between the different price levels cannot be infinitesimally small in practice and therefore the exchange restricts the change of the price between two consecutive price levels to some certain amount, called the ticker size. Regarding the mid price,

this price indicates the price which is exactly in the middle of best bid and best offer price.

When now a new limit order is issued to the exchange, the exchange takes the order, evaluates its type and price and then puts the limit order at the end of the list of orders at the corresponding price (in case the limit order is the first order at this price, it would be the first element of the list), i.e. there is a chronological ordering of the different limit orders.

In order that a trade is executed, the other group of orders is required, market orders. Market orders subdivide themselves into buy and sell market orders and they indicate the intention to immediately buy or sell the traded good. A sell market order, for example, is processed by the exchange by picking the first bid order from the pile of orders at the best bid price level and then matches the market order against the limit order. In case that the market order is completely filled by the limit order, i.e. the issuer or the market order has sold the entire volume to the issuer of the bid limit order, the remaining volume, if any, of the bid order goes back to its position in the order book, i.e. to the start of the best bid order list. However, the market order can also be partially executed, i.e. the first bid order does not cover the entire volume requested by the sell order. When this happens, the next order of the orders stored at the best bid price level is matched against the remaining market order and so on, until the market order is executed completely or the first list of bid orders is exhausted, then the exchange starts to take the orders from the next order list which represents now the best bid orders, i.e. the best bid price moves down and the issuer of the market order now has to sell at an inferior price. The concept is similar for buy market orders and offer limit orders, however, there the price moves up, not down.

In reality, the whole procedure is slightly more difficult, e.g. the different order types have different sub-types or the exchange grants some market participants advantages when it comes to matching, nevertheless, the basic concept remains the same.

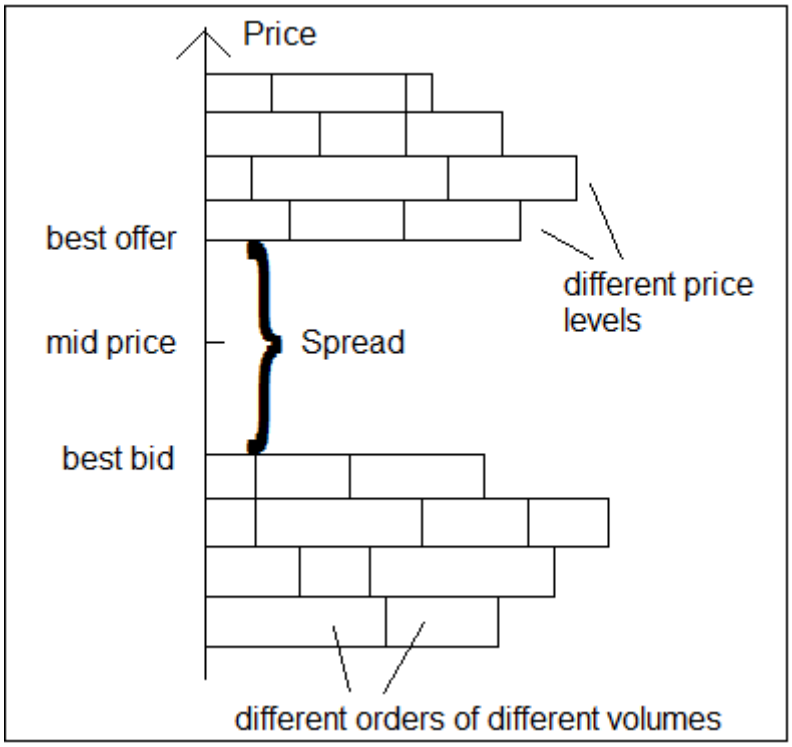

Figure 1: Schematic representation of an order book.

2.1.1.1 Traders

Traders are people or institutions who buy or sell assets at the exchange. In this broad term it does not matter whether they are using limit or market orders to trade the assets, however, Kirilenko et al (2011) provide a subdivision for the term trader and identify the following subtypes of traders, whereas the term inventory refers to the numbers of assets hold by the trader:

- Intermediary Traders: Traders who buy and sell high quantities of the asset in order to stay around some inventory level, i.e. their net position at the end of the day is almost unchanged.

- Fundamental Traders: These are traders which either focus on buying or selling the asset over the day, i.e. there end-of-day change in the inventory is either substantially positive or negative.

- Small Traders: Have only a small trading volume compared to Intermediary and Fundamental Traders.

- Opportunistic Traders: Change their behaviour over the trading day, i.e. sometimes they act like Intermediary Traders and sometimes like Fundamental Traders

2.1.2 Market Makers

2.1.2.1 In General

Market makers are a special form of traders. Their strategy is to provide liquidity to the market by submitting limit orders.

Market makers make their profits from the spread and/or a rebate granted by the exchange operator, for example, the BATS exchange grants a rebate of up to 0.0029 Dollars for every share provided (in form of limit orders) and which has been eventually matched by a market order, i.e. led to a trade.

In order to reduce the risk of being exposed to sudden movements in the price, they try to keep their long, which means they lose money on this positions when the asset price increases, and/or short, the lose money when the price increases[1], positions very short, i.e. they have a high inventory turnover compared to other traders without a significant change in the net position of the inventory, Kirilenko et al (2011) have identified them in their paper as Intermediary Traders.

In the group of market makers there is one important subtype, the high frequency market maker, which is closely related to the Hot Potato Effect (HPE).

2.1.2.2 High Frequency Market Makers

High frequency market makers (HFMM) are a group of market makers which participate in a large number of different trades, even for market makers. Kirilenko et al refers to the HFMM as High Frequency Trader, a subset of the Intermediary Traders, which make up the top 7% of the Intermediary Traders in terms of executed trades. As we can infer from the information provided in the section about the exchange and the order book, to be part in so many transactions requires being at the top of the book as often as possible. To achieve this HFMM rely on highly-automated systems and a high execution speed (compare Avellaneda and Stoikov,

[1] It is possible for a trader to sell an asset which s/he does not possess by borrowing it from another trader and returning the asset at a later date to the trader.

2007). However, the fact that they are frequently at the top of the book can also be dangerous and this will be covered in the next section about the Hot Potato Effect.

2.2 The Hot Potato Effect

As mentioned in the Introduction, the Hot Potato Effect (HPE) describes a situation where a high trading volume between high frequency market makers occurs, while the net position of the securities remains almost unchanged, i.e. the security is traded between the market makers back and forth. But how can this situation happen?

Clack (2012) suggests that the HPE can occur because of inventory-driven trading algorithms which are applied by the HFMM. An inventory-driven trading algorithm, as the name suggests, uses a trading approach which is based on the level of the inventory, e.g. when the inventory is low, the HFMM issues more aggressively bid limit orders and cancels the majority of its offer limit orders, when the inventory is high, the HFMM might do exactly the opposite, i.e. cancelling all the bids and price the offers more aggressively and with a higher volume size. This notion of a high inventory level and a low one already conveys the idea of inventory thresholds, i.e. limits the HFMM does not want to violate. However, what would happen if a HFMM exceeded such a limit? Kirilenko et al (2011) have shown that the HFMM would use market orders to quickly get back within their limits.

To demonstrate how this behaviour can result in a HPE, we are using a scenario. Let us assume there are 2 HFMM, A and B, which are depicted in figure 2, both are currently at their midpoint (current inventory is indicated by the arrow).

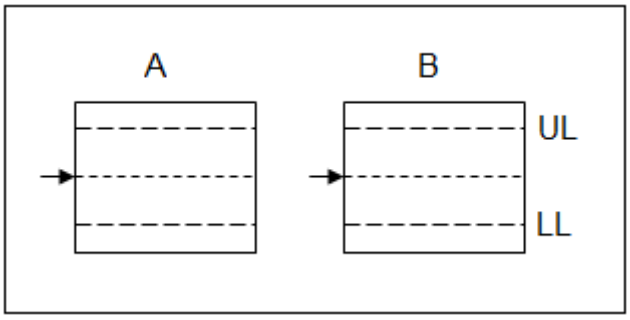

Figure 2: HFMM A and B.

Suddenly a big sell order of a Fundamental Trader hits the bid orders of A and drives the HFMM over its upper limit (denoted by UL).

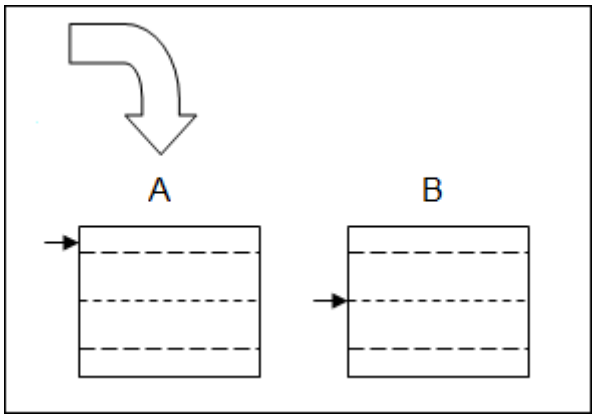

Figure 3: Sell order hits bid orders from A.

If the excess amount of securities cannot be reduced via offer limit orders, the HFMM will start to sell them off via sell market orders, assuming that B has had the best bid orders a lot of the securities, if not all, will move from A to B, who has now exceeded its upper limit:

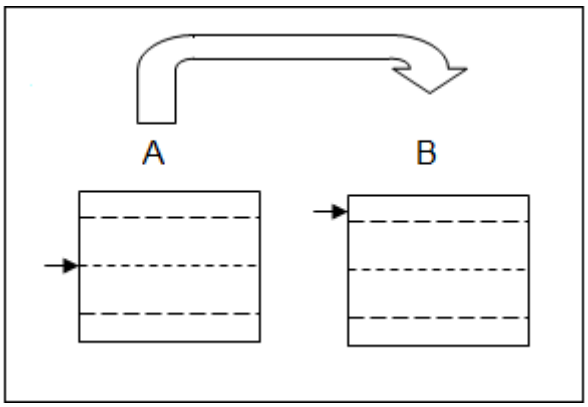

Figure 4: A sells off the securities and B buys them.

Same situation as before, if B cannot sell them via limit orders, he will sell them off via market orders, when A has meanwhile issued bid limit orders, which are positioned at the top of the book, then these orders will get hit by the market orders of B:

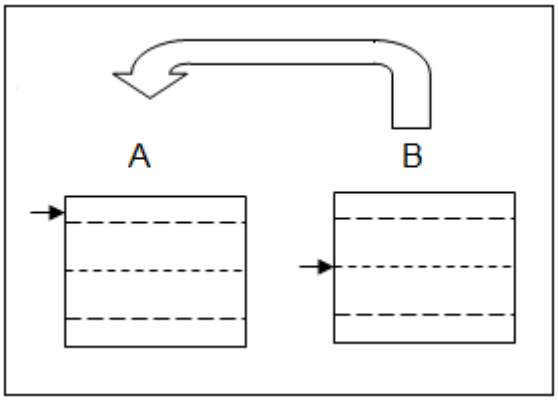

Figure 5: B sells off to A

A is now in the same situation as in the beginning and A will not act differently, i.e. A will sell the securities off to B, then B back to A and so on, a circular flow of the initial sell order of the Fundamental Trader has been created, the Hot Potato loops around.

Moreover, one initial trade has lead to an undetermined number of subsequent trades of the same size, i.e. the traded volume goes up dramatically, although the net change of the inventories (from the point of view of the HFMMs) is marginal. However, this high trading volume can be mistaken by other traders for real liquidity and they start to sell their securities, although the liquidity provided by the HFMMs is actually reduced because the Hot Potato is still circulating and at least one HFMM is always using Market Orders to sell the Hot Potato, i.e. he takes liquidity instead of providing it.

This process does not necessarily need to be started by one big sell order. It could also happen with smaller trades which are distributed over the MMs. The critical point is when one HFMM breaks through its upper limit starts to sell off the securities massively via Market Orders. This creates an avalanche of sell orders from HFMMs but also from other sellers. Furthermore, the appearance of many other sellers could also lead to the creation of new Hot Potatoes, so instead of one Hot Potato circulating around, 2 or more are circulating, forcing the HFMMs to spend even more time on selling via Market Orders than actually providing liquidity to the market.

To put it in a nutshell, as Clack (2012) suggests, the HPE can occur, when two conditions are fulfilled. On the one hand, the HFMM must have a “panic mode”, i.e. they start to sell off (or buy in, in case that the lower limit is violated) the traded asset massively and there must be second HFMM which panics in “synchronous

oscillation", i.e. one HFMM panics and sells off the securities to the other HFMM which then panics and sells them off back to the first HFMM.

2.3 The Stochastic Pi-Machine

This section should help the reader to get a better understanding of the Stochastic Pi-Machine (SPiM). While the first subsection explains what SPiM actually is, the second one gives a small introduction into the syntax of SPiM, such that the reader can comprehend the basic codes.

2.3.1 What is SPiM

The SPiM is a programming language which has been developed by Microsoft Research to model and simulate biological systems. The modelling approach is incremental, i.e. small subsystem can be connected to get larger systems. It is based on the Pi-Calculus and the theoretical groundwork for the machine has been laid by Andrew Phillips and Luca Cardelli in a paper in 2004 (compare Microsoft Research, n.d.). All our models have been created for Visual SPiM v0.1beta.

2.3.2 SPiM-Syntax

The following sub-points give a concise overview of the SPiM syntax and are based on the SPiM Language Definition (Phillips, 2007).

2.3.2.1.1 Processes

Processes are a basic element of SPiM. Figure 6 shows the basic application of a process, i.e. to what a process can evaluate to. Regarding the notation, an "Action" can either be a delay, i.e. the system waits for a stochastically determined amount of time before it proceeds (written: delay@Value), or the procedure of reading or writing to a channel, which is covered in the following section. The semicolon indicates that two things happen subsequently, e.g. "Action; Process()" means that the action has to occur first and there is no way that the process executes without the action executing before. When a process has executed its process body the process usually terminates, in case that the process does not have a recreation instruction in the definition body, i.e. the process calls itself recursively. To indicate that two or more processes are running in parallel, a vertical bar ("|") is used, however, in SPiM this is only a pseudo-parallelism because in SPiM one of the processes gets picked stochastically (therefore Stochastic - PiM) and is evaluated. However, all the other processes, which are running in parallel to the chosen process, are evaluated later

on, provided that they get picked at some point by SPiM. In contrast to this the choice-process, the do-or-structure, does only allow one of the specified processes to run, i.e. when one of the actions in front of the processes gets executed, only the process subsequent to this action gets evaluated, the other competing branches are discarded, this is especially useful when modelling race conditions. While the choice-process is a channel based branching operator, the if-then-else- and the pattern-matching-process are the standard value based branching instruments, the latter is especially valuable when dealing with lists and tuples. The replicate-process is particularly helpful when a process has to be repeatedly executed, i.e. the process is used again and again, whenever the action in front of the process is executed, a copy of the process is created and then evaluated.

```
(*Process definition*)
let Process(Processparameters) = (
    (*Process body*)
)

(*Action followed by an optional process*)
Action; Process()

(*Two processes in parallel*)
(Process1() | Process2())

(*if-then-else branching*)
if booleanValue then
    Process1()
else (*optional*)
    Process2()

(*Pattern matching process*)
match Value
case Value1 -> Process1()
case Value2 -> Process2()
(*...*)
case ValueN -> ProcessN

(*Choice process*)
do Action1; Process1()
or Action2; Process2()
or Action3; Process3()
(*...*)

(*Replicated action plus optional process*)
replicate Action; Process()

(*Creates x instances (x needs to be an integer) of the process*)
x of Process()
```

Figure 6: Basic applications of processes.

2.3.2.1.2 Channels

Channels are used to connect elements of almost every SPiM program. Each channel has a name and two points, an output and an input point, whereas the output point, denoted by an “!”, is used to write to the channel and the input point, denoted by an “?”, is used to read from the channel. Moreover, the information about the existence of a channel, i.e. its name, can be known to all the processes (global channels) or restricted to a certain scope (restricted channels). Names of channels, like values, can be written to other channels and therefore it is possible to share restricted channels.

```
(*Declare a new global channel*)
new a:chan

(*Write x to the channel*)
!a(x)

(*Read y from the channel*)
?a(y)

let test() = (
    (*Declaring a restricted channel *)
    new b:chan
)
```

Figure 7: Basic channel operations.

2.3.2.1.3 Values

SPiM provides the following main types:

- int: Integer numbers like: -5, 0, 2,... etc.
- float: Floating-point numbers like: 13.7603, 2.71828, 3.1415,... etc.
- bool: The Boolean values true and false.
- char: Characters like: ‘a’,’z’,’b’,... etc.
- string: List of chars like: “car”, ”neuron”, ”orange”,... etc.

As figure 8 shows, these values can be globally declared, concatenated to a list, stored in a tuple or simply send over a channel. In combination with values, SPiM also provides arithmetic operators (+,-,*,/) (+ also works as a logical Or and as list append, while * can be used as a logical And) and comparison operators (=,<>,>,<,>=,<=). Moreover, SPiM also provides some built-in functions to simplify

the application of values, there are, for example, functions which are converting an integer value into a float value and vice versa, as well as a function which calculates the square root of a float number.

```
(*Delcaring and assigning a value to a global value-variable*)
val a = 5

(*Integer values concatenated to an empty list*)
1::2::3::[]

(*Values stored in a tuple*)
(Value1,...,ValueN)

(*Writing and reading an integer value to a channel*)
(!c1(4) | ?c1(x))

(*Converts a variable of type int into a variable of type float*)
float_of_int(x)

(*Converts a variable of type float into a variable of type int*)
int_of_float(y)

(*Calculates the square root of a variable of type float*)
sqrt(z)
```

Figure 8: Value declaration, sending via a channel and some built-in functions.

2.3.2.1.4 Types

SPiM supports several types, like Integer (denoted int), Float (float), Boolean(bool), Characters (char), String (string) and polymorphic (‘Name), e.g. a channel of type ‘o can be used to transmit a variable of any type. Figure 9 demonstrates some basic applications of the type system. Especially the third example is of interest because it shows how a global type expression is used.

```
(*Declaring a channel which can carry values of type int*)
new channel1:chan(int)

(*Declaring another channel which can transmit two variables of type float*)
new channel2:chan(float,float)

(*Alternative approach for channel2 with a type expression. Send a 2-tuple*)
type tuple = (float,float)
new channel2Alternative:chan(tuple)

let Process(a:int,b:float,c:list(string)) = (
    (*Processbody*)
)
```

Figure 9: Basic applications of types.

2.3.2.1.5 Plotting

SPiM has also the ability to plot the number of instances of a process at each time stamp. Figure 10 demonstrates the standard plotting instruction, the floating number 100.0 indicates the maximum duration and the “MC()” and “EX()” after the “directive plot” tells the SPiM to plot the instances of the processes MC() and EX().

```
(*Specifies the maximum duration of the simulation*)
directive sample 100.0

(*Specifies the processes to plot*)
directive plot MC(); EX()
```

Figure 10: Basic plotting instructions.

2.4 Additional Software used

Besides of SPiM, other software packages have been used as well.

2.4.1 Amanda

The pure functional programming language Amanda has been used to test some of the algorithms which are used in the simulation, especially the Adder, the Canceller and the Matcher.

2.4.2 Microsoft Office Excel 2007

Microsoft Office Excel has been used to test the pseudo random number generator.

2.4.3 Notepad++

Notepad++ has been used to write the code for the SPiM because the standard development environment does not support bracket-checking and is therefore inconvenient to us for bigger projects.

3 Analysis

In this chapter we will try to investigate the reasons for the Hot Potato effect (HPE), we will justify our model and the application of SpiM, as well as we will cover requirements and challenges of our simulation.

3.1 First Intuition

From the background knowledge Chapter we know that for a HPE to appear it is a crucial incident that a (high frequency) market maker violates one of its threshold limits and then sell off/buy in the excessive long/short position by using market orders. Taking the risk averse nature of market makers into account, it is not very likely that they are violating their limits on purpose, quite the contrary, it is rather justifiable to assume that they try to avoid this situation rigorously by using algorithms which are highly unlikely or even impossible to violate their limits, at least from their perspective. But if that is the case, how is it possible that a hot potato can occur and travel around? How is it possible that one market maker after another violates its inventory threshold? Clack (2012) suggests as one possible cause for the HPE the occurrence of time delays between the execution of a trade and the notification via a trade confirmation message, i.e. the market makers issue their new orders based on outdated inventory numbers. However, to test the validity of this suggestion it is too short-sighted to look at only one market maker completely isolated from all the other market participants, therefore it is necessary to look at the whole picture, the entire system, and model the interplay between the market maker and the other market participants adequately.

3.2 The System

Chapter 2 has presented some of the market agents which interact in an exchange-based trading environment. Although there number seems to be vast, they can be broken down into three different types: the exchange, market makers and fundamental traders.

3.2.1 Exchange

In an exchange-based trading environment, as the name implies, is the exchange the core of the whole system. It is the place where all the trading is executed and all the declarations of intent to trade, i.e. orders, are stored and processed. There is a 2-way communication between the exchange and the other agents of the system. On the

one hand, the exchange takes orders from the other market participants as an input, while on the other hand, status messages about the orders are sent out to the issuers of the orders. For our simulation, such as Clack (2012) proposes, the exchange is also the origin of the time delays which presumably cause the HPE to occur, however, in contrast to the original suggestion we will model the cause for the time delay not after the trade execution but before. Because in our opinion it is more reasonable that a delay occurs due to a huge amount of incoming orders which cannot be processed all at once by the exchange and are therefore stored temporarily in a buffer where a pile of orders builds up. To be more precise, from our point of view the exchange can be overloaded at the input side, e.g. a market maker could issue a cancellation order for a stored limit order, although this limit order will be matched by a market order as soon as the exchange has processed all the excessive orders in the buffer, in case that the market maker issues new limit orders before it gets the notification of the cancellation, this behaviour can also lead to the HPE.

3.2.2 Market Makers

The market makers provide the market with liquidity and make their profit from the spread and/or a rebate granted by the exchange. Therefore they are issuing mainly limit orders. For our simulation the market makers need a deterministic strategy when they are within the boundaries of their inventory and a so called “panic mode” which describes the behaviour when they have violated their limits. The market makers are connected to the exchange, and only to the exchange, by either outputting orders to the exchange or receiving messages from the exchange.

3.2.3 Traders

The term trader refers to all the other non-market-making types of traders, instead of providing, these traders actually take liquidity from the market. Due to the fact that we have subsumed a lot of different traders with different trading behaviours and, at least to some extent, also different trading intention, we suggest to model the behaviour of the traders as a random process[2].

[2] This type of trader is usually called “noise trader” in the literature

3.3 Requirements

Let us take a look at what requirements the system has to fulfil in general and which requirements have to be fulfilled by the individual agents to achieve the stated goals[3]:

3.3.1 In General

3.3.1.1 For general simulations in SPiM

Requirement	Description
Flexibility of the simulation	As this work should also lay the groundwork for further research, the system should be easily adaptable and extendable. More about this can be found in the requirements for the individual agents.
Improve the usability of SPiM	The work should provide tools to simplify the handling of SpiM as an instrument to conduct simulations in the area of finance.

Table 1: Requirements for simulations in SPiM.

3.3.1.2 For the Hot Potato Effect

Requirement	Description
Provide the tools to generate the HPE	This work should provide the tools to show that the interplay of the different market participants in combination with a time delay should lead to a HPE.
Realize a time delay	In order to test the hypothesis that a time delay can be the origin of the HPE, it is needed that the system can be simulated with a time delay.
Provide the tools to display the HPE	It must be possible to display the HPE with the built-in plotting system of SPiM.

Table 2: Requirements regarding a Hot Potato Effect simulation.

[3] A dark blue row indicates a main requirement while the light blue ones constitute sub requirements which belong to the main requirements

3.3.2 Exchange

Requirement	Description
Receive orders	The exchange can receive new orders from the other market participants.
Store unprocessed orders	Incoming orders need to be stored in a buffer until they are processed by the exchange.
Implement a limit order book	The order book is the core of the exchange and stores all the limit orders.
Store orders	The limit order book needs to store orders until they are cancelled or matched by a market order.
Add new orders	New incoming orders can be added to the order book
Execute trades	Market orders get matched against the limit orders on the order book.
Orders can be cancelled	Orders currently on the order book can be cancelled by issuing a cancel order.
Different order durations	The order book should be able to cope with different duration qualifiers, e.g. how long is an order valid, for the different standard order types (bid, offer, buy, sell).
Provide a trade book	The trade book contains all the executed trades and is therefore a valuable instrument to analyse the market events.
Implement exchange rules	The exchange should adhere to some standard exchange rules. Especially the following issues need to be addressed:
Rules regarding price jumps	Trading should be halted in case of strong price movements.
Rules regarding the range of the price	Prices, which are too high or too low compared to the last traded price, should be rejected.
Rule regarding overlapping limit orders	It is possible, especially if a time delay can occur, that an incoming bid order overlaps with the best offer (or vice versa), i.e. the bid offer has the same

	price as the best offer or even higher. The exchange needs to take care of such a situation.
Realize an information system	Needed to keep the other market participants up to date.
Provide status messages	The exchange should be able to send out messages to inform the other market participants about the state of their orders.
Provide market data	The other agents should be able to request market data from the exchange, like the last traded price or the current best bid/offer.
Flexibility of the simulation	The exchange can assist the general requirement by fulfilling the following things:
Adjustable behaviour	It should be possible to change parts of the behaviour of the exchange, e.g. it should be possible to change the algorithm which adds new orders to the book while other parts of the exchange remain unchanged.
Works with different numbers of market makers	The exchange should be able to interact with a different number of market makers, i.e. it does not matter whether two or ten market makers feed orders into the exchange.
Works with different numbers of fundamental traders	Same as above but with fundamental traders.

Table 3: Requirements of the Exchange

3.3.3 Market Maker

Requirement	Description
Issues orders	It should be possible for the market maker to issue orders, whereas the following sub requirements need to be taken into consideration:
Manages inventory	A market maker makes its profit on the spread and on a probable rebate granted by the exchange. However, the market maker is not interested in being exposed to the risk of price movements and therefore the market maker needs to manage its inventory. This requirement is closely related to the order issuing algorithm and the panic mode.
Specify standard algorithm	To simulate an exchange-based trading environment the market maker needs a procedure to specify the volume of its next order and at which price.
Specify the panic mode	For the simulation a panic mode is needed, i.e. what should the market maker do if one of its limits is violated.
Keep track of orders	Issued orders should be stored in an own order book to cancel them later on or to appropriately respond to messages received.
Message management	The market maker should be able to receive messages and process them.
Flexibility of the simulation	The market makers can assist the general requirement by fulfilling the following things:
Changeable order issuing algorithms	It should be possible to change the algorithm, which issues orders, easily, e.g. it should be possible to specify the simulation like this: Two market makers which adhere to algorithm A and two market makers which apply algorithm B.
Changeable message algorithms	Same as above but now the algorithm that deals with incoming messages is changeable.

Table 4: Requirements of the Market Maker

3.3.4 Traders

Requirement	Description
Issues market orders	It should be possible for the trader to issue market orders to take liquidity from the market, whereas the following sub requirements need to be taken into consideration:
Random order volume	The volume demand/supplied by a fundamental trader should be normally distributed, to take into account the unpredictability of the behaviour and the different needs of the different agents, which could act as a trader, e.g. pensions funds, noise traders, venture fund and so on.
Keep track of orders	Issued orders should be stored in an own order book, although no post-issuing-processing has to be conducted it might be a useful tool for analyses.
Message management	The fundamental trader should be able to receive messages and process them.

Table 5: Requirements of the Traders.

3.4 Challenges

The table below provides an overview of the challenges, whereas the column refers to the (part of a) requirement to which the challenge is closely related, while the second one provides detailed information about the challenge itself. Rows which are coloured in dark red indicate a main challenge, while the lighter red specifies sub challenges which belong to main challenges above them.

Requirement/Part of Requirement	Challenge
Create the tools and framework that could be used to display the HPE.	Part of the challenge is that, although the SPiM provides a (graphical) output in form of a chart and a spreadsheet, there is no convenient way to display something in SpiM, because these two possibilities work only with processes, therefore it is necessary to find a way to visualize the HPE.
Display the inventory of the market makers.	For a simulation which deals with the HPE it is definitely no drawback to visualize the inventories of the market makers, however, as described above, it is not possible to output the progress of single number but only processes, therefore a convenient workaround is needed.
Display the amount of orders in the order buffer.	Same things that hold for the market makers, are also holding for the order buffer.
Negative numbers in the built-in output system of SPiM.	Due to the fact that the standard output system of SPiM works with processes, it is not possible to display negative numbers with it, because you cannot have less than zero instances of a process. While this might not be a problem in a biological context, it is one in the context of finance, when we allow the market participants to short sell, i.e. selling assets which they do not own (by borrowing from a broker). To allow the usage of short selling market participants, it is necessary to find a workaround.

Random numbers	The fundamental traders require a random order volume. The problem is that SPiM does not provide a pseudo random number generator, i.e. we need to implement one on our own.
Flexibility of the simulation	There is a trade-off between the flexibility of the simulation and the convenient application of SPiM. The more flexible the simulation should be, the more restricted channels are required. Nevertheless, restricted channels are harder to manage than simple global channels and it is therefore more difficult for potential new users to use to manipulate and extend the system. A good balance between flexibility and convenient application needs to be found.

Table 6: Challenges

Due to the fact that the majority of challenges have to do with the functionality of SPiM, someone might raise the question why SPiM has been used at all. The answer to this question is provided by the next section.

3.5 Why SPiM?

3.5.1 The Pi-Calculus

The Pi-Calculus is a process calculus which was developed by Rober Millner (1999) to describe the communication between parallel processes. The construction of this calculus is simple but it is still very expressive. However, this simplicity has also a drawback. It requires a lot of effort to model complex systems, due to this we have switched to the Stochastic Pi-Calculus, which is based on Pure Pi-Calculus.

3.5.2 SPiM vs. Pure Pi

Pi-Calculus	Stochastic-Pi-Calculus
No (predefined) alphanumeric type system.	Provides Integers, Floats, Chars and Strings.
No (predefined) Boolean values.	True and False are supported.
No (predefined) conditional branching.	If-then-else and pattern-matching is possible.
No (predefined) data structures.	Tuples and lists, as well as the standard operations for lists, like append, are provided.
No (predefined) arithmetic operators.	+,-,/,* are supported.
No (predefined) comparison operators.	=,<>,>,>=, <,<=, are applicable.
Automatic, static analysis is possible.	There is currently no static analyser available, however, it is planned by Microsoft Research to adapt SPiM, such that SPiM programs can be exported to PRISM code, which enables stochastic model-checking.

Table 7: Comparison of Pi-Calculus and Stochastic-Pi-Calculus.

To put it in a nutshell, the SPiM environment hides a lot of the complexity of the pure Pi-Calculus by providing standard operations, data structures and types which are well-known from other programming languages. To be clear, we do not want to imply that these things cannot be realized in Pure Pi-Calculus but they are quite clumsy to implement, let us take the list data structure as an example, in SPiM we would simply use "::" to concatenate an element to the head of a list, in a Pure Pi-Calculus environment we would need to define two restricted channels, one which holds the value (which we would also need to define) and one which holds the actual list, in addition we would also have to implement our own concatenation process and find a definition for the empty list. This all can be done in the Pure Pi-Calculus but why should we bother us with matters of detail when we actually want to model the interplay in a financial system and we are provided with the necessary tool to do so by the SPiM? Therefore we are using SPiM.

4 Design

The figure below shows the overall system with the main agents which have been worked out in the analysis. In the sections 4.1 and 4.2 we will focus on the individual agents and how we have fulfilled the requirement to provide future users with a system that is easily adaptable and extendable. In section 4.3 we will deal with the communication between the different agents.

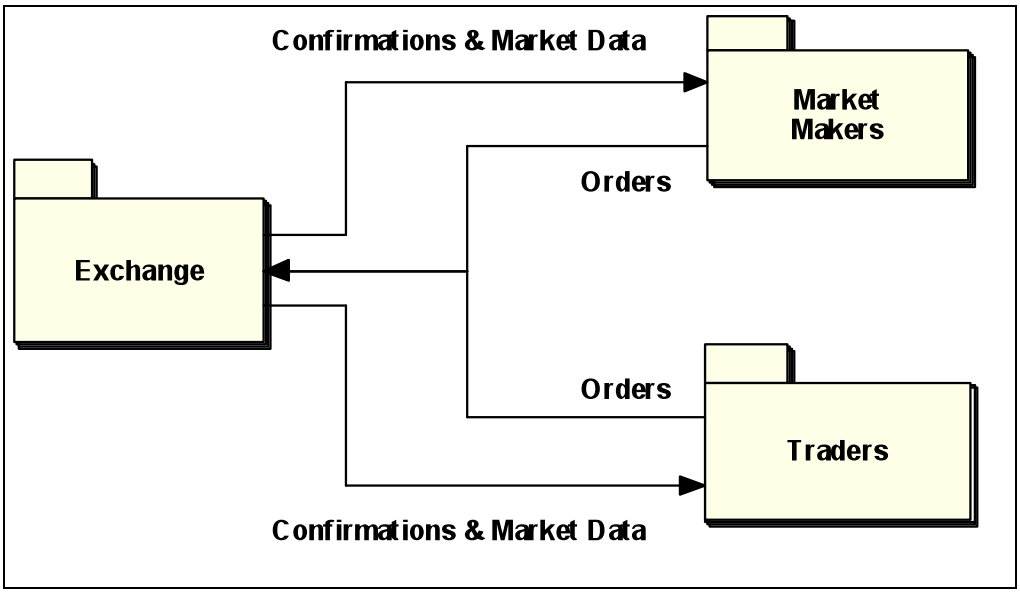

Figure 11: Overview of the system.

4.1 Exchange

In Figure 12 the composition of the Exchange system is depicted. Each subsystem is in charge of one or more specific tasks and closely related to one or more requirements worked out in Chapter 3[4]:

- Order queue: Buffers incoming orders before they get processed.
- Order evaluation: Enforces the exchange rules and decides what to do with an incoming order.
- Order book: Grants access to the underlying data structures, which are storing all the orders, such that they can be manipulated by other subsystems.
- Adder: adds a new order to the order book.
- Matcher: Is in charge of the trade execution and also sets up the trade book.
- Canceller: Deals with cancel orders.
- Market Data Provider: Provides the other market participants with non-order related information, like last traded price.

This modular structure is shaped by the requirement to enable users of this simulation tool to adjust the behaviour of the system to their needs easily. For example, if someone wants to conduct a simulation with a different matching algorithm, s/he only needs to replace the original Matcher with the new one, while the rest of the program remains unchanged. Nevertheless, let us take a more detailed look at each of the subsystems.

[4] external communication is not taken into consideration in this diagram, only internal, for external please see Chapter 4.3

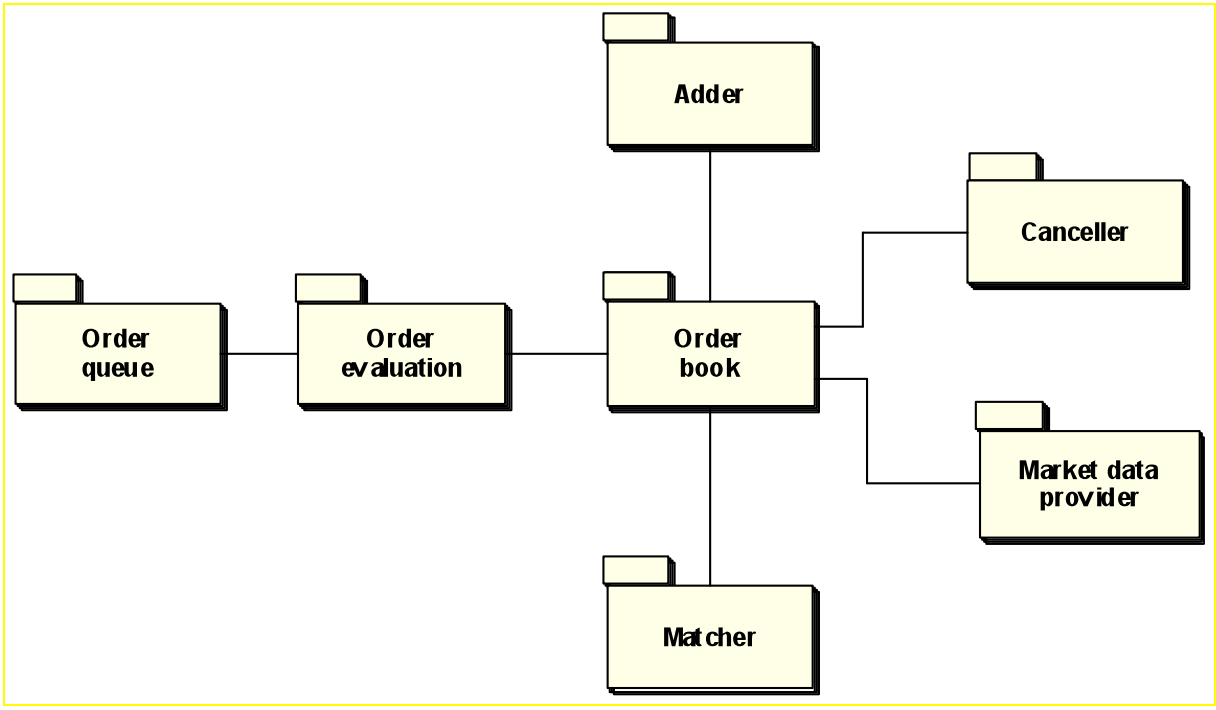

Figure 12: Subsystem of the Exchange system (the lines indicate internal flows of information).

4.1.1 Order Queue

The queue works as a buffer to store orders which have not yet been processed and helps to implement the time delay. This data structure has been chosen because the inherent FIFO-principle of the queue naturally maintains the chronological order of the orders, e.g. it is not possible that one order A, which has been enqueued after order B, gets dequeued before B and therefore gets processed before B.

However, SPiM does not provide the data structure queue and therefore it is needed to be implemented. Figure 13 shows the pseudo code of one way to implement a queue for the SPiM using a list. This approach has some general elements like the fact that it works with a list of polymorphic type, which makes it possible to reuse the code for other projects. However, the queue is especially designed for the application in the simulation conducted in this project.

- On the one hand the queue is realized with global channels, i.e. the channels can be accessed from everywhere and it is not needed to share the information about the channel with all the market participants, this takes into account that for the simulation only one exchange is used and therefore only one queue is used, a more general way would require that each exchange (in case that someone wants to conduct a simulation with more exchanges) creates its own queue and then shares them with the other market

participants, nevertheless, for our purposes this approach is more than sufficient and if someone requires more queues/exchanges, it can be easily changed.

- On the other hand, another difference from a more general approach is the fact that dequeuing from an empty queue does not lead to an error but to the output of an empty-notification, this merges the functionality of a standard dequeue-method with a standard isEmpty-method, the advantage of this approach is that an additional channel is not needed which reduces code complexity because the channel needs to be read from or written to by the exchange or the queue all the time, i.e. when someone reads from the channel the information is deleted and therefore it would be required to update the status of the queue frequently. One disadvantage of this approach is that the empty-notification needs to be of the same type as the elements which are stored in the queue, so it might not be possible to create a notification, e.g. the queue stores all kinds of integers, however, in our simulation it is not a problem, the queue stores items of type order and so the empty-notification is an order with the order type "EMPTY"[5] (compare Lafore, 2003).

```
new enqueue:chan('o)
new dequeue:chan('o)
new dequeueRequest:chan

let Queue(x:list('o)) = (
    do  (?enqueue(newElement); Queue(x+newElement::[])
    or  (?dequeueRequest();
        match x
        case [] -> (!dequeue(empty-notification) | Queue(x))
        case head::tail -> (!dequeue(head) | Queue(tail))
    )
)
```

Figure 13: Pseudo code of an adjusted generic queue.

However, the above implementation has one drawback, it is not possible to enqueue and dequeue elements at the same time, i.e. in parallel. The built-in list of SPiM does not allow this. A truly parallel list could be implemented but this would require to write own functions for e.g. appending elements or other lists and also the built-in pattern-matching would not work anymore. The benefits of working with the built-in data structure outweigh the disadvantages.

[5] To store elements of a different type, the empty-notification needs to be adjusted, but the interface itself is still polymorphic.

Figure 14 shows the interface diagram of the order queue, as we already know from the pseudo code example, the queue has three globally known channels, the dequeueRequest- and the dequeue-channel, which connect the queue to the order evaluation, and the enqueue-channel, which connects the queue to the market makers and the traders.

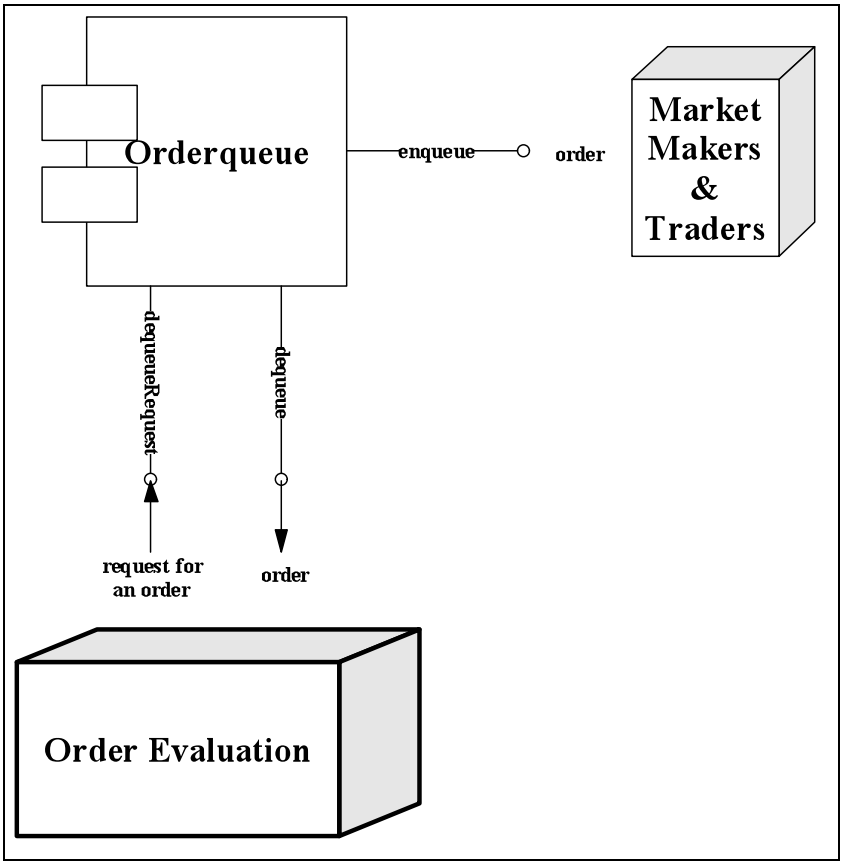

Figure 14: Interface diagram of the order queue.

4.1.2 Order evaluation

The order evaluation enforces the rules of the exchange, like price boundaries or halt of trading due to strong movements in the price. Let us now take a look at the rule enforcing logic.

Figure 15 shows the flowchart of the order evaluation with the rules which will be implemented as a standard rule set for simulations. The shown rules are based on the rules used by the electronic trading platform of the CME to protect the market from strong price movements (CME Group, n.d.):

- Stop Spike Logic (SSL): If active, all market orders will be rejected and only limit orders are processed, i.e. added to the limit order book. This rule is used by the CME to stop trading in cases where the trades would be executed outside of some price limits, e.g. if the matching of market order X led to a decline in the last traded price of more than 300 basis points, the trade would not be executed, the market order would be rejected and further matching would be stopped for some predefined time. However, the order evaluation is only checking whether the logic is active or not and proceeds accordingly, the activation of the SSL itself is conducted by the Matcher.
- Price Banding/Boundary: Used by the CME to reject orders with an allegedly erroneous price. Every order outside some specified price range (with reference to the last traded price) gets rejected. This does not only protect the market itself but also the market participant which has issued the order, when the reason for the order price has been an algorithm going wrong or a typo[6].

As it can be seen from the diagram, the order evaluation takes also care of overlapping limit orders, e.g. when an incoming bid limit order has the same or a higher price than the best offer order stored in the order book, the bid limit order is transformed into a sell market order and matched.

[6] The price banding usually refers to all orders but in our simulation only the limit orders possess a price and therefore they are the only type of orders which need to be checked

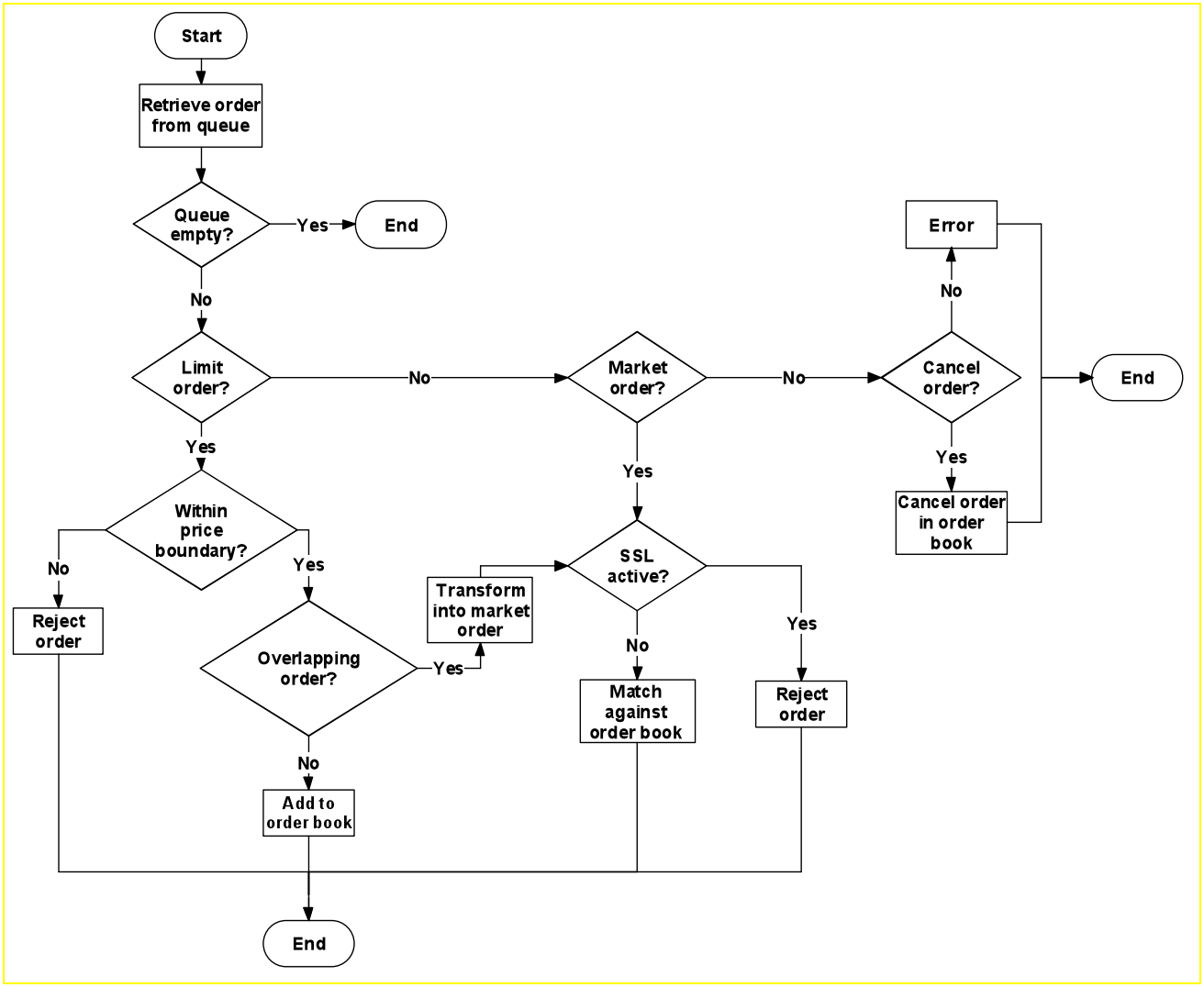

Figure 15: Flowchart of the order evaluation.

From the flowchart above, we can also derive the interface diagram for the communication with other subsystems of the exchange. The tasks of the different channels are as following:

- dequeueRequest: Requests an order from the order queue.
- dequeue: Used to transmit the requested order from the queue to the order evaluation
- addBid: Outputs a bid limit order to the Orderbook-Process to add the order to the order book.
- matchBid: Used to send a sell market order to the Orderbook-Process, such that it gets matched with the stored bid limit orders.

- addOffer: Outputs an offer limit order to the Orderbook-Process to add the order to the order book.

- matchOffer: Used to send a buy market order to the Orderbook-Process, such that it gets matched with the stored offer limit orders.

- obconfirm: This channel is used by the Orderbook-Process to indicate to the order evaluation that the next order can be processed.

- requestMarketData: The OrderEvaluation-Process sends out a restricted channel name on this channel to the MarketDataProvider.

- receiveMarketData: This is the restricted (and therefore represented by a dashed line) channel mentioned above. The channel is used by the MarketDataProvider-Process to provide the order evaluation with necessary market data, such as the last traded price, for the price banding, or the best bid and best offer price.

- ssl: Used by the Matcher to indicate to the order evaluation that the Stop Spike Logic needs to be activated (see Matcher section for more information).

The design decision to use different channels for the different order types has been made in order to reduce the amounts of comparisons, i.e. to enforce the rules of the exchange the order evaluation has to check every order for its type, if the order evaluation had used only one channel to forward all the orders to the order book subsystem, the order book would need to evaluate all the orders again.

Regarding the obconfirm-channel, this channel is on the one hand important, because it prevents orders from slipping through the order evaluation, while an order is still processed, e.g. a market order which is processed by the Matcher could lead to the activation of the SSL, but meanwhile the order evaluation has evaluated another market order and is writing the order to the matchBid/matchOffer-channel, although this order would have been rejected, if the Matcher activated the SSL earlier[7].

[7] This slipping through behaviour can also appear in the real world, however, there is the overall number of orders much higher and therefore a single order which is slipping through does not have a huge impact, while our system is adapted to a smaller scale of orders, due to performance reasons,

Moreover, the majority of channels have globally known names. This design decision has one main drawback, that is, it will be not possible to simulate an environment with more than one exchange, when the standard modules are used, i.e. while the number of market makers and other traders can vary, the maximum number of exchanges is one. However, the advantage of this approach is that it is much easier for other developers to construct new modules for the exchange system, because the main input and output channels are already known and therefore the developer does not need to consider the whole “channel-name-sharing-issue”.

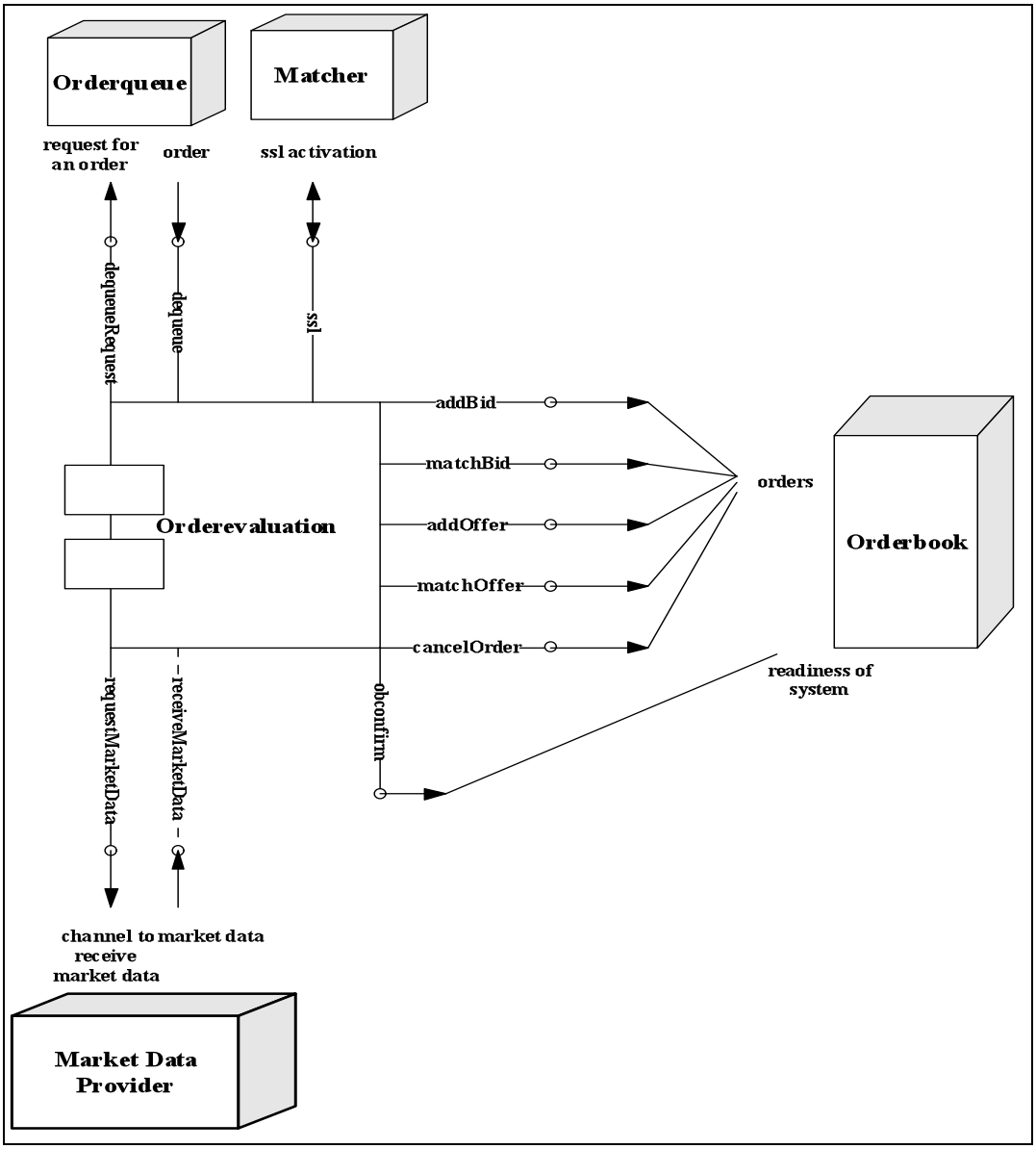

Figure 16: Interface diagram for the communication within the exchange of the order evaluation.

and therefore there would be a proportionally higher impact of a single order slipping through than in the real world.

4.1.3 Order Book

The order book is the central element of the exchange, despite the unambiguous name the range of duty is larger than a mere storage task. The order book also takes in valid orders from the order evaluation and delegates their processing to the other subsystems by granting access to the stored orders and some pre-processing. Moreover, it also takes care of the management of the different order qualifiers. However, let us start with the obvious part of the order book, the underlying data structure.

For the framework conditions of our order book, we are using the assumptions of Clack's (2012) archetypical limit order book, i.e. there are two types of limit orders, bids and offers. In our simulation these two types also have a different ordering paradigm, i.e. for efficient processing bids should be ordered from high prices to low prices while offers should be ordered the other way round, it is a straightforward design decision to give both types their own data structure (subsequently referred to as bid book and offer book). But how should this data structure look like? There are two ways which have been explored during the project, a one list approach and a list of lists approach:

- One list: All the limit orders of one type are stored in one list, where the individual price levels are represented by blocks of orders, when a new order must be added, the Adder would only need to go to the end of the block of the specified price (assuming that there are already orders stored at this price) and insert the new order there, just in front of the next block of orders at an inferior price.
- List of lists: There is one list which represents all the different levels of prices, for which orders have been stored. However, each element of this "price" list is a list too, storing all the different orders for each price level. Figure 17 depicts this approach for a bid book, for the offer book the only difference is that order lists are ordered from low to high.

Both approaches have their advantages and disadvantages. The one list approach is easy to implement and also all the potential operations, like adding a new order or matching a market order against the book, are way easier to realize for the one list approach than for the list of lists approach, however there is one main drawback, the program needs to go through all of the stored orders of a certain price level, e.g.

when someone wants to add an offer at 5.0$ and there are already orders stored at the price levels of 4.8$ and 4.9$, the Adder needs to check all orders at 4.8$, 4.9$ and also 5.0$ (and the first of 5.1$ or for the empty list, in case that the 5.0$ block mark the end of the offer book) to find the position. In contrast, the list of lists approach needs only three comparisons to successfully insert the order, due to this performance advantage the list of lists approach has been chosen to implement the underlying data structures.

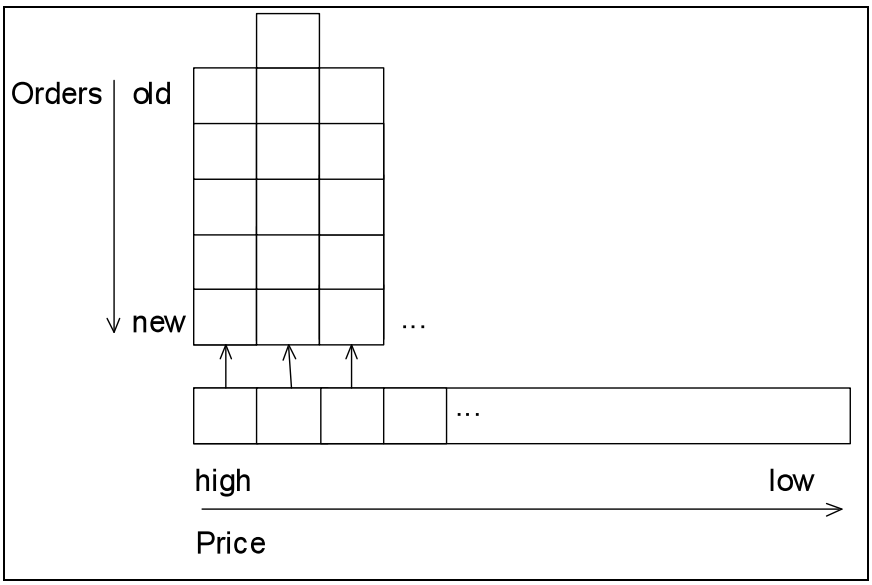

Figure 17: Construction of the bid book.

Now let us evaluate how the Orderbook-Process is connected to the other subsystems of the exchange. We know from the section about the order evaluation that we will need two channels for incoming limit orders, two channels for incoming market orders, one channel for cancellation orders and one channel to signalize the order evaluation that the next order can be processed. Furthermore, we also need channels which connect the order book with the Adder, the Matcher, the Canceller and the Market Data Provider (MDP) to provide them with the orders to process (not necessary for the MDP, because it does not process orders) and grant them access to the bid and offer book. Moreover, we also need channels which can be used to return the two books back to the Orderbook-Process.

Figure 18 shows our approach to design the interface. The top five channels and the obconfirm-channel have already been dealt with in the order evaluation, the only difference is that we are now at the other point of the channel, i.e. instead of writing to the channels, like the order evaluation, the Orderbook-Process reads from them

(except of the obconfirm-channel where it is the other way round). The other seven channels have tasks as follows:

- toAdder: Sends a limit order and the corresponding book to the Adder-Process.
- toMatcher: Sends a market order and the corresponding book to the Matcher-Process.
- toCanceller: Forwards a cancellation order and both books to the Canceller-Process.
- requestBooks: The Orderbook-Process reads a channel name from this channel (see below).
- channelname: This is the channel for which we have received the name from the requestBooks-channel. The channel is used to transmit the two books to the creator of this channel, i.e. this is not a globally known channel but a restricted one (therefore represented by a dashed line in the diagram).
- returnBidBook: Used by the other subsystems to return the (modified) bid book back to the order book.
- returnOfferBook: Used by the other subsystems to return the (modified) offer book back to the order book.

The first two channels and the return-channels have a quite straightforward functionality and why the toCanceller-channel requires both books to be forwarded will be dealt in the section about the Canceller (3.1.6). The requestBooks-channel can be used to easily extend the functionality of the overall exchange, a process can write a restricted channel name (for a channel which is able to transmit the two books) to this channel and then the Orderbook-Process writes the bid and offer book to this channel, such that the other process can use them. This feature will be demonstrated in the context of the Market Data Provider, i.e. this process has not a globally known channel to retrieve the books but has to use the requestBooks-channel.

It can be argued that the number of required channels could easily be reduced, for example, the two return-channels could be replaced by a single one, while this might not be a problem at the level of the Adder or the Matcher, because only one book is returned, it would be problem for the Canceller-Process and any other process, which retrieves both books from the Orderbook-Process, because it is an obstacle for other developers, who wish to implement their own Canceller- or MarketDataProvider-Process. Before they can implement their process, they would need to figure out, how the Orderbook-Process differentiates between the two books, i.e. how does the order book know that the incoming book is, for example, the bid book? Does the order book simply assume that the first book, which is written to the return-channel, is the bid book, a very error-prone approach, or does the Orderbook-Process actually evaluate each of the incoming books, whether they are a bid or an offer book? To avoid this problem, we are using a two-channel approach.

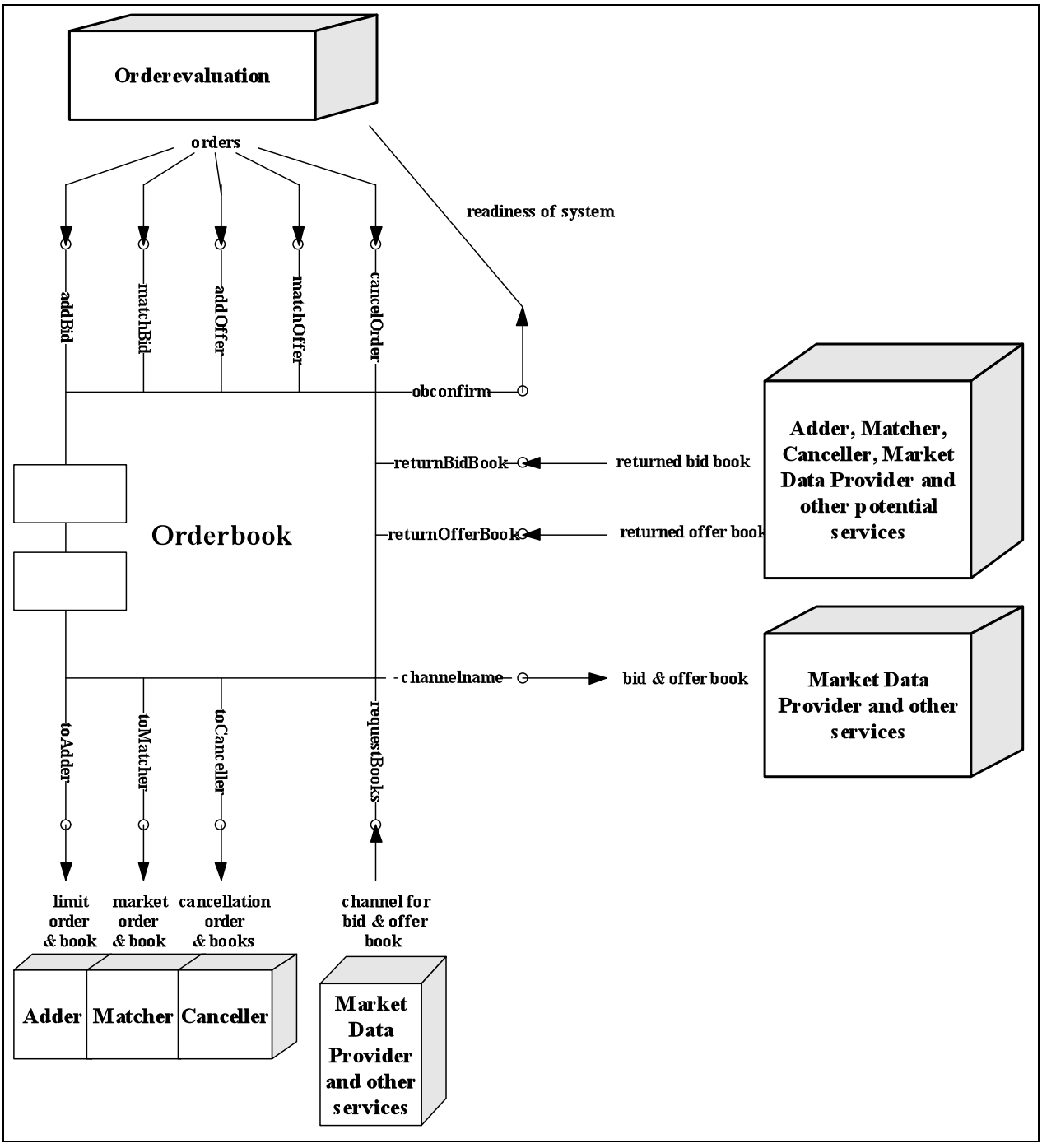

Figure 18: Interface diagram for the communication within the exchange of the order book.

The order book, which we want to model, has to take care of some processing of the incoming orders as well. This processing is closely related to the order duration qualifiers, for the model we will implement four duration qualifiers which are based on the qualifiers used by the CME electronic trading platform (CME Group, n.d.), and which are as follows:

- Good ‘Till Cancellation (GTC): The order is valid until execution or the issuer sends out a cancellation order.
- Good ‘Till Date (GTD): The order is valid until execution, cancellation or the specified time has been expired.
- Fill And Kill (FAK): These orders are immediately executed partly or as a whole.
- Fill Or Kill (FOK): If the order cannot be executed as a whole at once, the order will be rejected.

From these four qualifiers only GTD and FOK need some pre-processing. While GTD needs a counter within each stored order which works as a time stamp, a function which reduces this time stamp and one function which cancels all the orders which have dropped down to zero, FOK orders need a function which returns the volume of the corresponding book, e.g. for a FOK-Sell-Order we would need the volume of the bid book.

From all of the above, the following flowchart can be derived. The diagram shows the procedures which are conducted within the Orderbook-Process.[8]

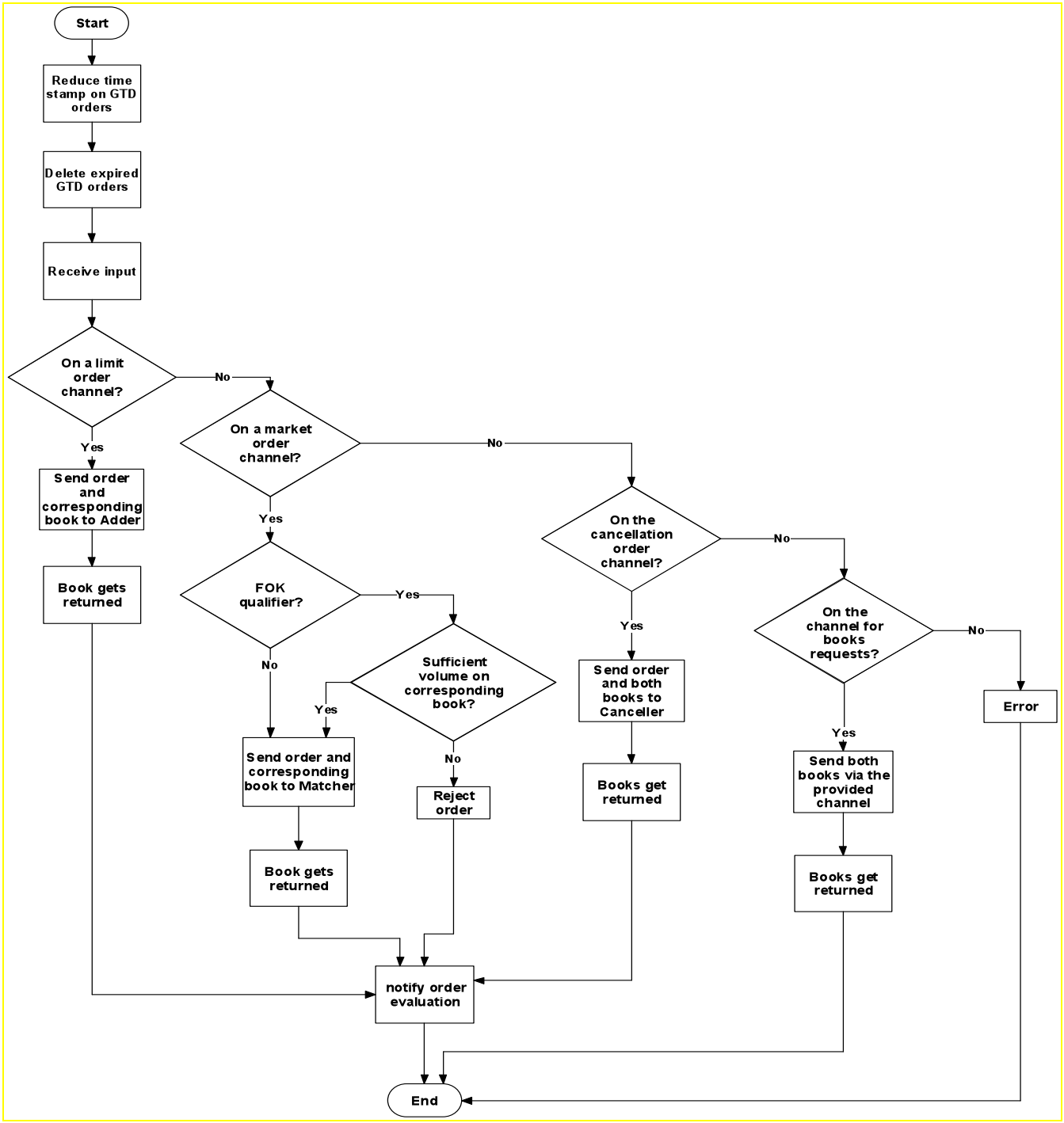

Figure 19: Flowchart of the order book.

Furthermore, some might criticise that the order duration qualifier are assigned to the order book, i.e. in case that someone wants to change the qualifiers s/he needs to change the Orderbook-Process. While the reason for implementing the necessary concepts to realize the GTD qualifiers on the order book level is simply that all the required elements, i.e. the bid and the offer book, are there, any other procedure would still need to request the books from the Orderbook-Process. Moreover, it can be argued that this design does not consider orders which are rejected at the order

evaluation level, however, in our model we are assuming that the rejection at the order evaluation level happens so quickly that the time this requires is negligible and therefore this design concept is appropriate. In contrast, the assignment of the FOK-qualifier to the order book is not that straightforward, it can be argued that the check could have been done on the level of the Matcher as well, nevertheless, this would require an additional layer in the Matcher because the Matcher is recursively called and otherwise the volume of the remaining book would be calculated over and over again, which would have a huge impact on the performance (the recurring comparison of market order volume and calculated volume has also a negative influence but, in comparison to the calculation, this one can be neglect). This additional layer has been deemed to be not worth the effort and therefore the FOK-qualifier has been designed on the level of the order book.

4.1.4 Adder

The Adder has only one purpose, to add new orders to the order book quickly and in an efficient way, such that future orders can be added quickly as well or that the Matcher can match market orders against the book rapidly.

The flowchart in figure 20 shows a way of how to achieve these goals. The Adder takes the first order list of the book, in case that the book is not empty, takes from this list the first order and checks the price of the order to add against the retrieved order, if they have the same price, the order gets added to the end of the list, i.e. next to the empty list, this allows the Matcher to match quickly in chronological order, if the price of the order to add is greater than the price of the stored order and it is a bid order, than a new list, containing the order to add, is put in front of the current order list, if the price is smaller than the next order list is chosen and it starts from the beginning. The only difference between adding a bid and adding an offer is that the bid book is ordered such that the high price levels are close to the head of the list, i.e. from high to low, while the offer book is ordered from low to high. This price ordering of the order lists and the chronological ordering of the orders within the order lists make it quite easy for the Matcher to match (see later).

What we would like to point out is that even a trivial Adder requires a lot of reconstruction work. Whereas we mean with reconstruction work, the work which has to be done to put the book back together. For example, the Adder has run through several order lists and has now found the correct position for the order. As a result,

the Adder must now put the order, which has been used to conduct the price comparison, back onto the list. Then the new limit order must be appended to the end of the list. Eventually this new order list must be appended to the remaining order lists, which represent more inferior prices, and also to all the order lists, which have been evaluated so far and have superior prices. Only then the book is ready to be returned to the Orderbook-Process. This can be tedious for people with little experience in Stochastic Pi- or Pure Pi-Calculus. Therefore it will be demonstrated in the Implementation Chapter how to set up an Adder, which has the same functionality as the Adder depicted below, but reduces the reconstruction work to a minimum by using the library that has been developed in the context of this project.

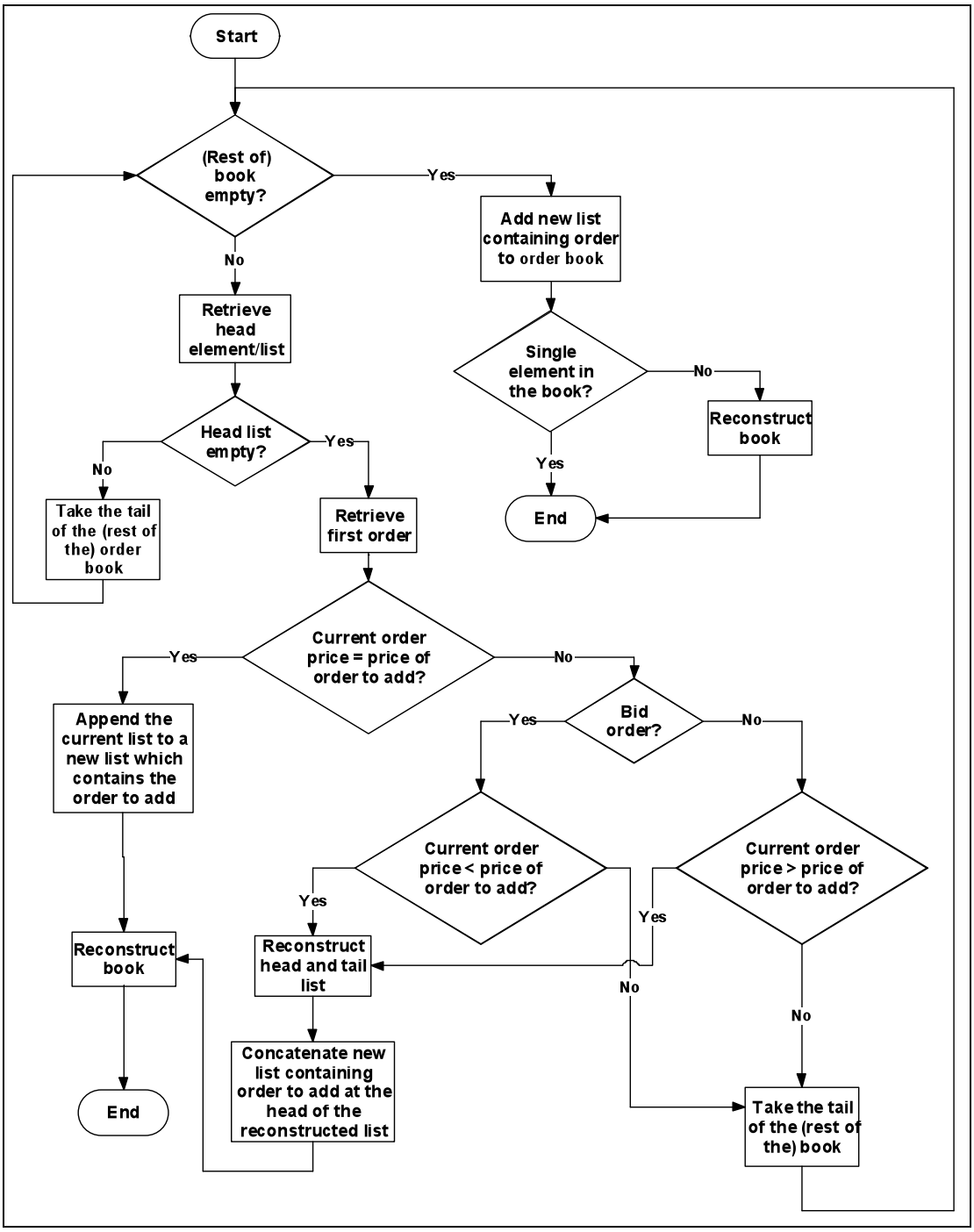

Figure 20: Flowchart of how a new order is added to the order book.

From the point of view of the interfaces, the Adder does not need any new globally known channels to achieve its goal, i.e. all the channels depicted in figure 21 have already been described in previous sections.

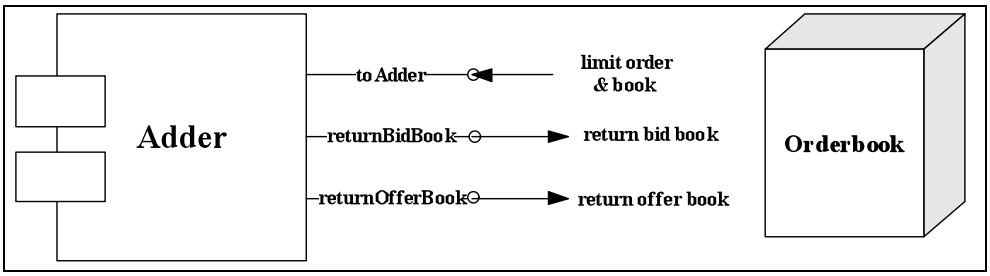

Figure 21: Interface diagram for the communication within the exchange of the Adder.

4.1.5 Matcher

The Matcher is in charge of the actual trade execution, i.e. matching incoming market orders with the limit orders stored on the book, and manages also the trading book.

For our model we want the Matcher to conduct the standard FIFO-matching-algorithm, which discriminates the orders primarily by price and secondarily by time, this is also one of the matching algorithms used by the CME electronic trading platform (CME Group, n.d.). Due to the way of how the Adder has stored the orders in the book, this matching approach is realized by a quite straightforward Matcher.

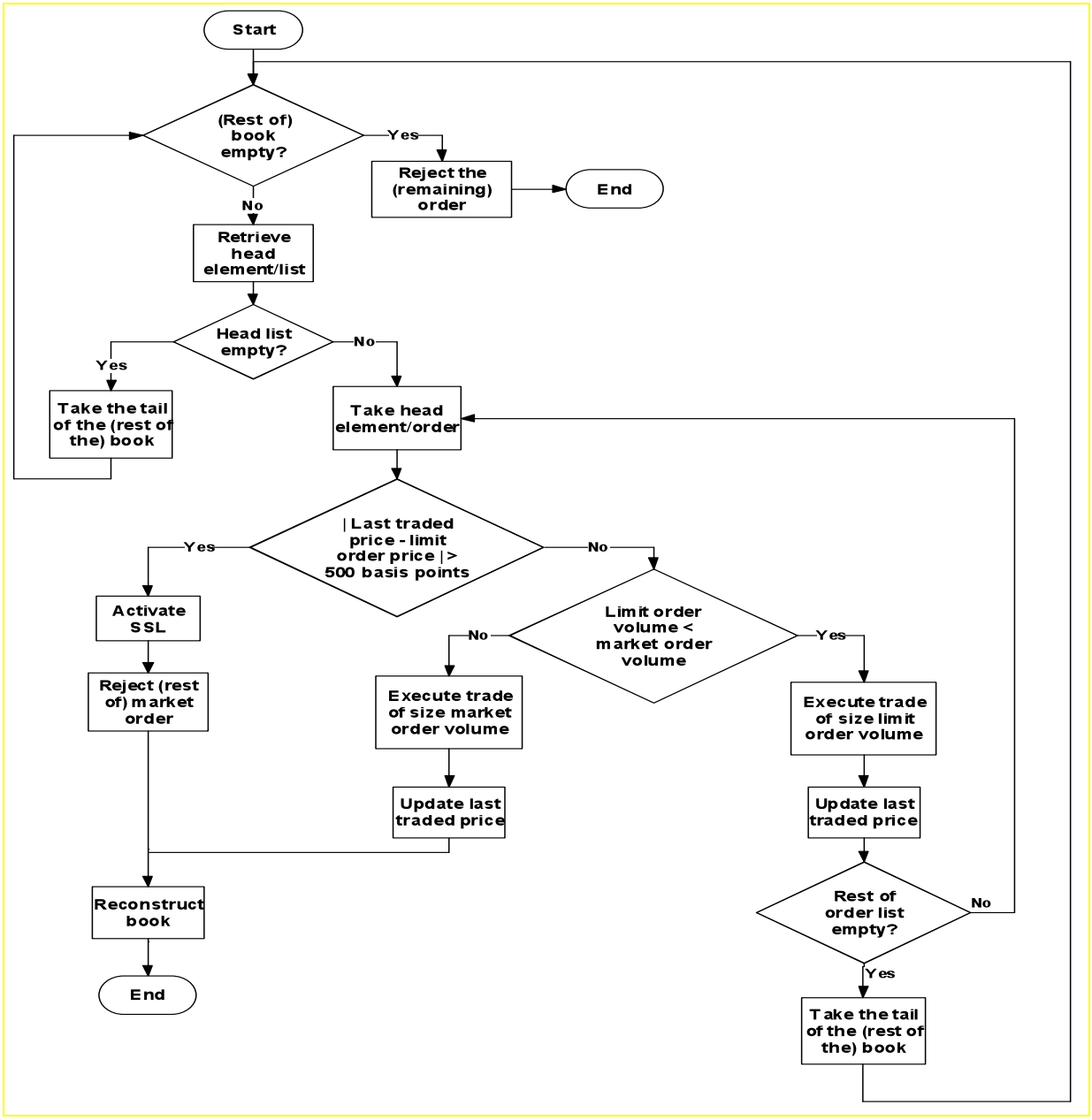

Figure 22: Flowchart of how an order is matched.

The flowchart in figure 22 provides an overview of the procedure, the Matcher takes the first list of orders from the book and matches the market order against the limit orders in the list until either the market order is completely executed (i.e. outstanding volume drops to zero) or the order list is exhausted which leads the Matcher to take the next order list.

On the first glimpse it might seem that the Matcher used for our simulation does not differentiate between a sell and a buy market order and indeed the process is almost the same for both market orders except of one little detail which is not depicted in the

chart, this detail is hidden in the process boxes with the description “Execute trade of size ...” (ETSV). These ETSV processes are quite abstract and entail different processes, they need to calculate the remaining volume of the not completely matched order, contact the issuers of the executed orders and also update the trade book and for this update the Matcher needs to know, whether it is matching a sell or a buy market order, therefore it is not sufficient to forward only the book, the volume of the market order and the message channel from the issuer of the market order. The Matcher derives this information from the order, which the Matcher has to match, i.e. the order is data structure[9].

Regarding the data structure of the trade book, because this book only needs to store the executed trades in a chronological order, it is reasonable to use a single list as underlying data structure and concatenate the data of every executed trade in the form of a 3-tuple (market order type, price, volume) to it.

Moreover, the branching/decision – element where it is asked whether the absolute value of the difference between the last traded price and the price of the limit order is greater than 500 basis points refers to the Stop Spike Logic (SSL) and should prevent massive price movements (CME Group, n.d.). Whenever the next traded price would exceed the last traded price by at least 500 basis points, the Matcher does not execute the trade, activate SSL and rejects the (remaining) order. The threshold of 500 basis points has been chosen arbitrarily. To signalize the order evaluation that SSL has been activated a globally known channel is required from which the order evaluation can read, therefore the ssl-channel has been set up.

Concerning the ssl-channel, as indicated by the diagram, the Matcher does not only read from the channel but also needs to write to the channel. This has to do with the fact that reading from a channel destroys the message, i.e. another process cannot access the message anymore. This can lead to the following situation. The Matcher detects a price jump and therefore wants to activate SSL. Firstly, the Matcher reads from the ssl-channel to destroy the current message, this is very important because otherwise there would be 2 different messages on the ssl-channel. Secondly, the Matcher writes the Boolean value true to the channel. The first step must be conducted before the books are returned and the second one can be realized either before the return or in parallel to the return. Otherwise the system would either block,

[9] A tuple, to be precisely.

if the second step happened before the returning of the books, because no one would read from the channel, or, in case that the first step happens after the return of the book, it cannot be guaranteed that the order evaluation does not read from the channel before the Matcher, i.e. there would be a race condition. However, also the order evaluation needs to write a value back to the channel after reading, otherwise the Matcher will block the next time when it tries to read from the ssl-channel. Therefore a writing-instruction should always be provided when working with this kind of channels. A more critical version of this case is the next channel.

The Matcher also requires channel to share the information about the last traded price with the other subsystems. Figure 23 shows the interface diagram of the Matcher, as it can be seen, the new channel ltp, which stands for “last traded price”, is comparable to the ssl-channel, i.e. the Matcher is reading from and writing to this channel. However, the Matcher does not only use the channel to communicate with other subsystems but also uses the channel to store the value of the last traded price, this design decision has a lot to do with the actual implementation, as we know from the Chapter Background Knowledge, every process terminates after executing its process body, however, we repeatedly need the Matcher, so we have to recreate it, there are two common ways, either by using replicate or by calling, i.e. creating, the process at every possible termination point, the first approach is the more convenient way because it is less error-prone, e.g. by forgetting a termination point (could happen easily by many if-then-else or pattern-matching operations) or accidentally spawning two copies of the process, however, it is not possible to use replicate with changing parameter values, i.e. replicate creates an exact copy of the process, and therefore it would not be possible to use replicate if the last traded price was stored as a parameter of the Matcher. Nevertheless this approach has one drawback, the MarketDataProvider-Process (MDP) wants to access the ltp-channel too. While the Matcher does not need to compete for the ssl-channel with the order evaluation, because usually one of the two processes is blocked while the other on is working[10], this is not the case with the MDP. Therefore it is important to write back to the ltp-channel as soon as possible, a good approach is to write to the channel directly after reading from it and in parallel to the remaining process body, such that the own processing is not blocked..

[10] The toMatcher-channel block the Matcher as long as the order evaluation works and obconfirm blocks the order evaluation as long as the Matcher works.

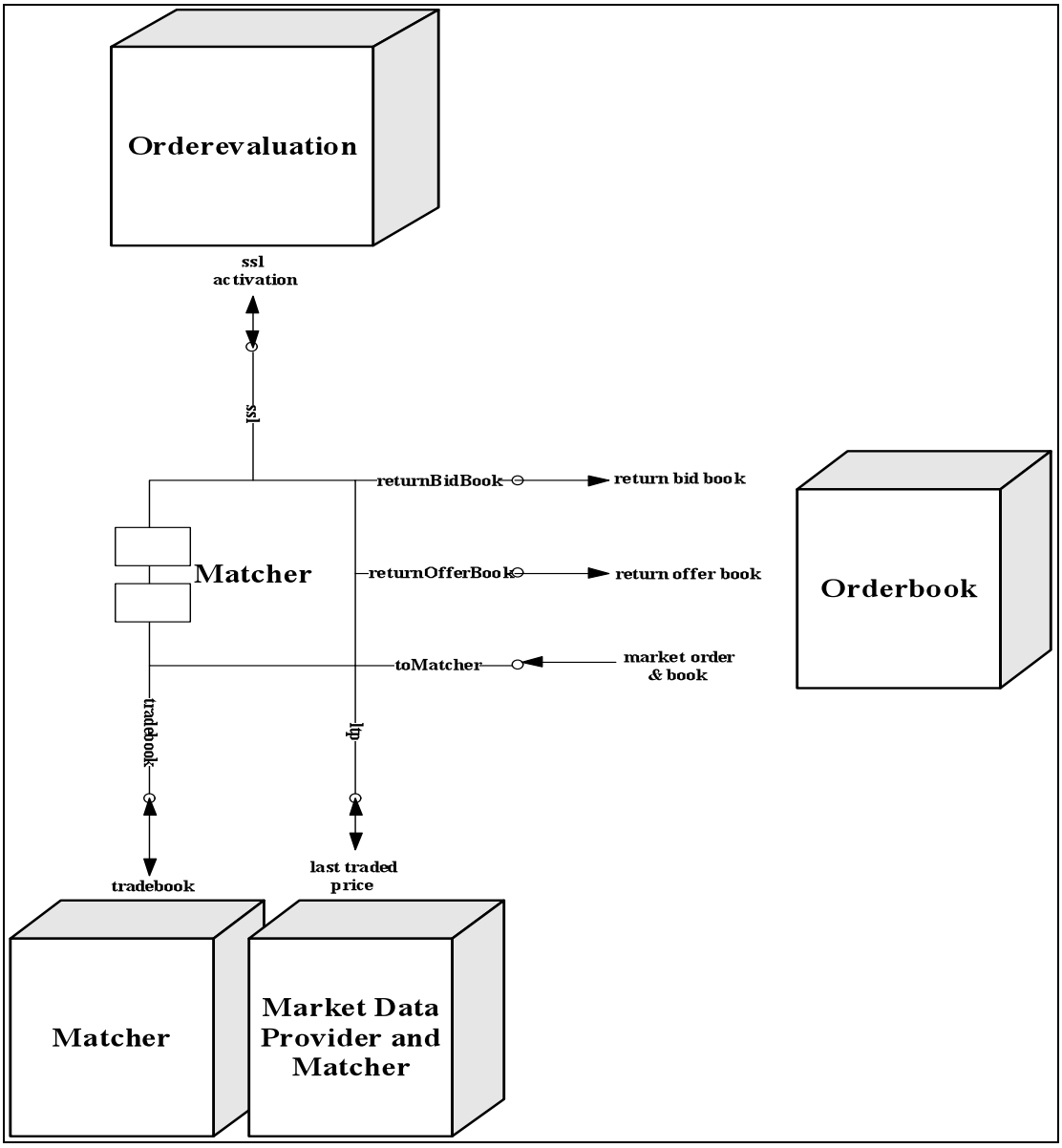

Figure 23: Interface diagram for the communication within the exchange for the Matcher.

4.1.6 Canceller

The Canceller has the task to delete orders from the books, although this sounds not very hard to realize, it is actually quite sophisticated.

The reason for this is that in our simulation the order type of each stored orderID is not stored, i.e. there is no data structure the Canceller-Process could access to determine whether the limit order with the specified orderID is a bid or an offer (market orders cannot be cancelled because they are executed/rejected immediately in our simulation), therefore the Canceller needs to search both books for the order and then cancels the order from the book where it has been found, although these

two steps can be combined into one, there is still the need for checking whether the order was on one of the books or not.

Figure 24 shows what has to be done on each of the two books to cancel a single order. The Process takes the first list of orders from the book and compares the id of the order to delete with the limit orders in the list until either the order has been found (the process returns the reconstructed book and a “found”-notification, which could be a Boolean value) or the order list is exhausted which leads the process to take the next order list. When the process reaches the end of the book without finding the order a “not found”-notification and the book are returned.

On top of this depicted process there needs to be another process which actually compares the results from the two cancellation operations, i.e. has one of the processes returned a “found”-notification then confirm the cancellation to the issuer of the cancellation order, has nothing been found then reject the order. To realize this in SpiM would require at least two recursions and 3 processes, one searching through the individual order lists at each price level, one which changes the price level and the last one needs to compare the results, a quite sophisticated and error-prone approach (the problems of SPiM with recursion will be discussed in the Chapter Conclusion), however, with the help of the library, which has been developed for this project, the needed effort and the susceptibility to error can be reduced. This will be done by mapping the newly implemented takedropwhile function over the entire book, which uses an evaluation function which searches for the order.

Nevertheless, someone could still claim that this approach is inefficient, compared to a solution which is based on a data structure that stores the mapping from orderID to order type, however, on the one hand this approach would require an additional data structure which needs to be managed and on the other hand the strategy used in our simulation can also be seen as a demonstration of the power of the implemented library.

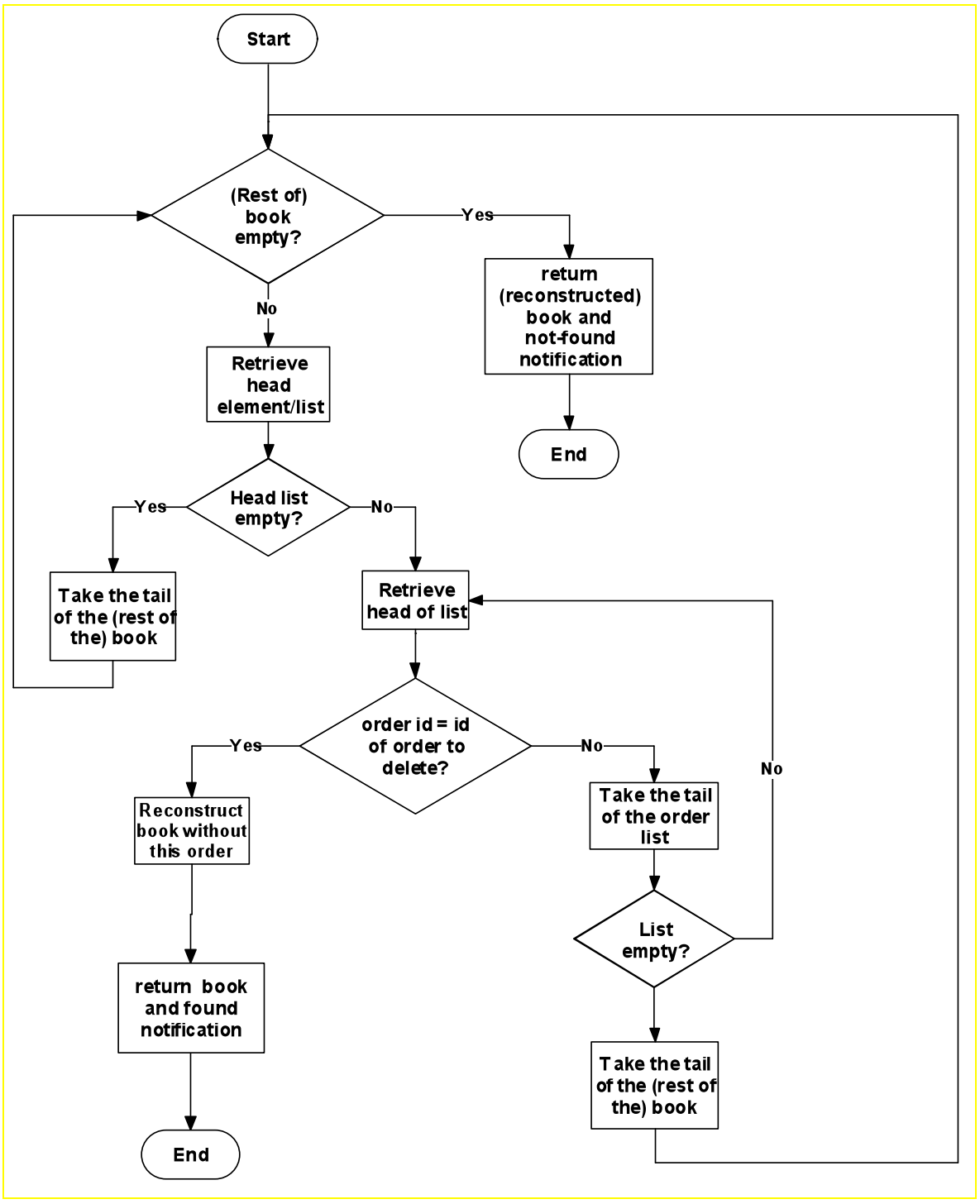

Figure 24: Cancel-Operation which has to be conducted on bid and offer book.

Regarding the interface diagram in figure 25, we have covered all the globally known channels, which are required by the Canceller, in previous sections.

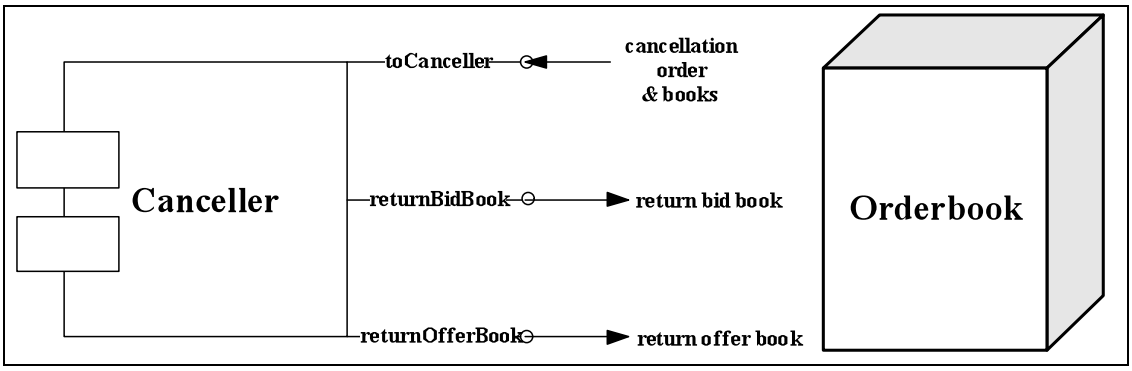

Figure 25: Interface diagram for the communication within the exchange.

4.1.7 Market Data Provider

The Market Data Provider (MDP) has the task to provide internal (like the order evaluation) and external (like the market makers) agents with valuable data about the market.

Because we have already covered all the global channels which are used by the MDP, there is not that much to say about the interface which is depicted in figure 26. What we want to point out is that the MDP does not only read from the ltp-channel but also writes to the channel, this is necessary because otherwise the Matcher cannot access the last traded price anymore, i.e. the MDP retrieves the last traded price and subsequently writes the value back onto the ltp-channel.

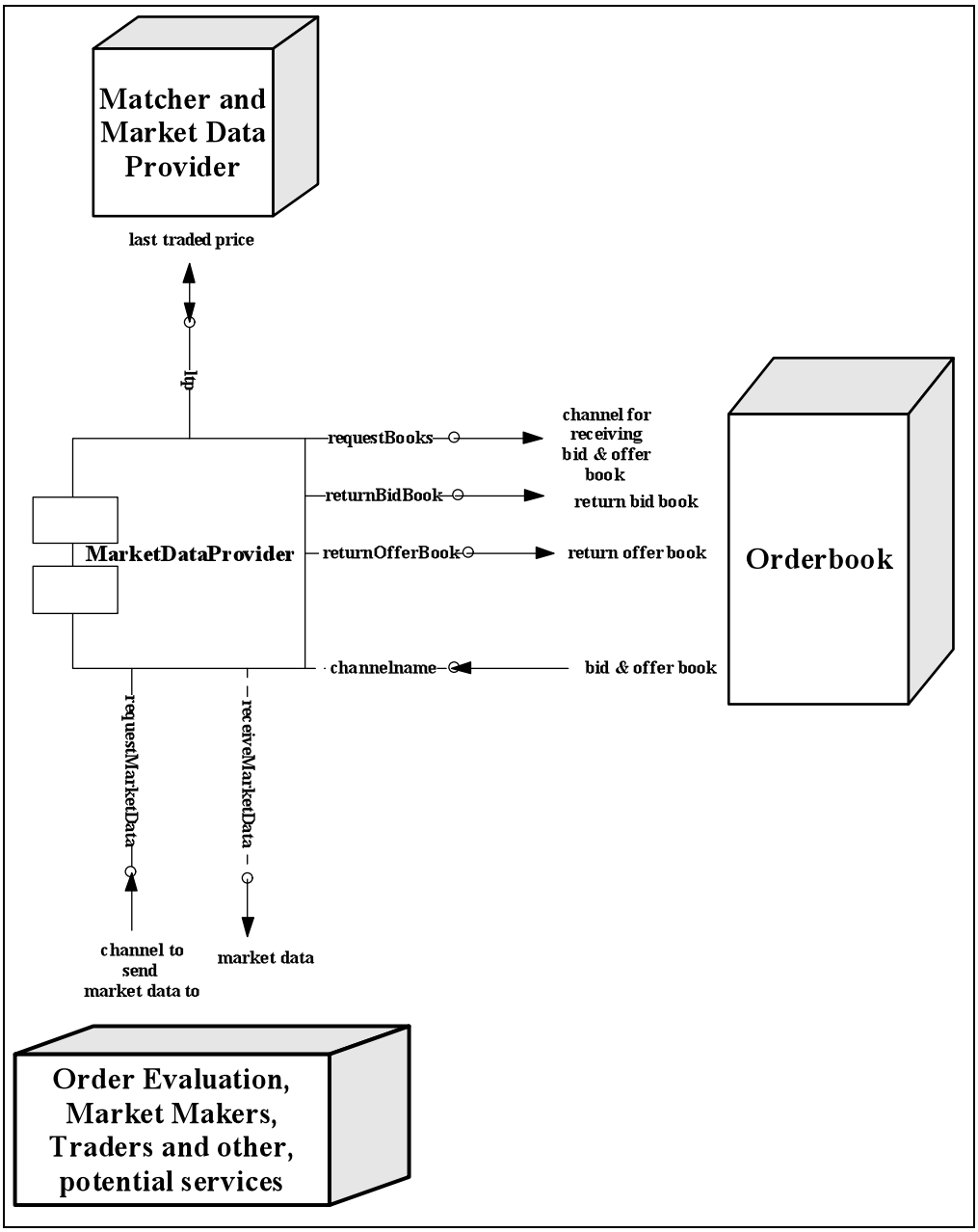

Figure 26: Interface diagram for the communication within the exchange of the Market Data Provider.

For our simulation we want the MDP to provide the last traded price, the best bid price and the best offer price, the flowchart in figure 27 depicts this behaviour, the -1.0 are used as a placeholder to indicate that the corresponding book is empty. Furthermore, we do not take into account that there could not be a last traded price, because this situation would mean that the asset has never been traded on the exchange before. This could happen in the case of an initial public offering (IPO), however, an IPO is a quite specific situation, which we will not cover with the

standard model and therefore we are assuming that there is always a last traded price, i.e. the ltp-channel gets pre-initialized[11].

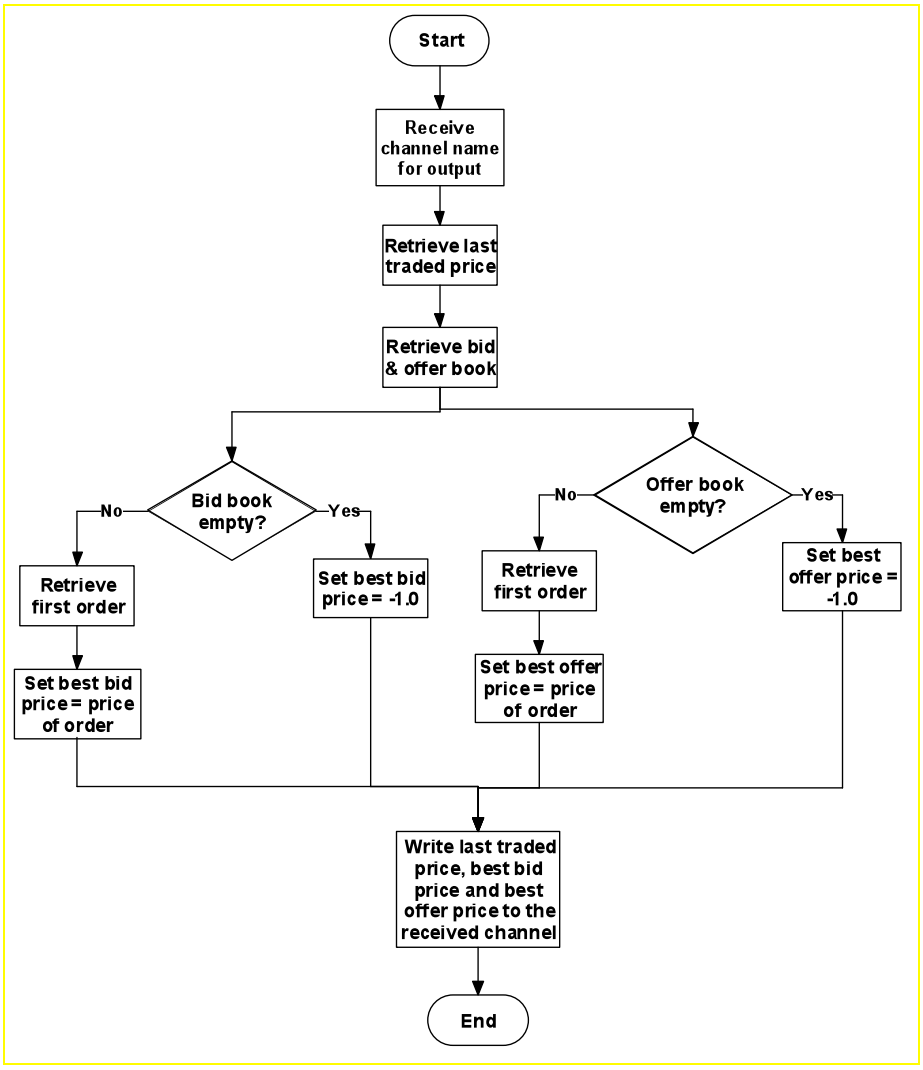

Figure 27: Flowchart of the Market Data Provider.

Some people might argue that it would be sufficient that the Orderbook-Process actually takes care of the best bid and best offer price and simply forwards these two prices to the MDP, instead of both books. However, this design decision, to forward the complete books to the MDP, has been made in order to enable an easy replacement of the standard MDP, with an MDP which provides the inquirer with other market data, e.g. the current volume of the books.

[11] For example with the end-of-day market price from the previous trading day.

4.1.8 Component Overview

Before we continue to the Market Makers and Traders, let us take a look at the whole picture. Figure 28 shows all the components in one diagram. Some of the channels are not directly connected but via boxes, otherwise the clarity would suffer. This figure gives a good impression about the complexity of the exchange system.

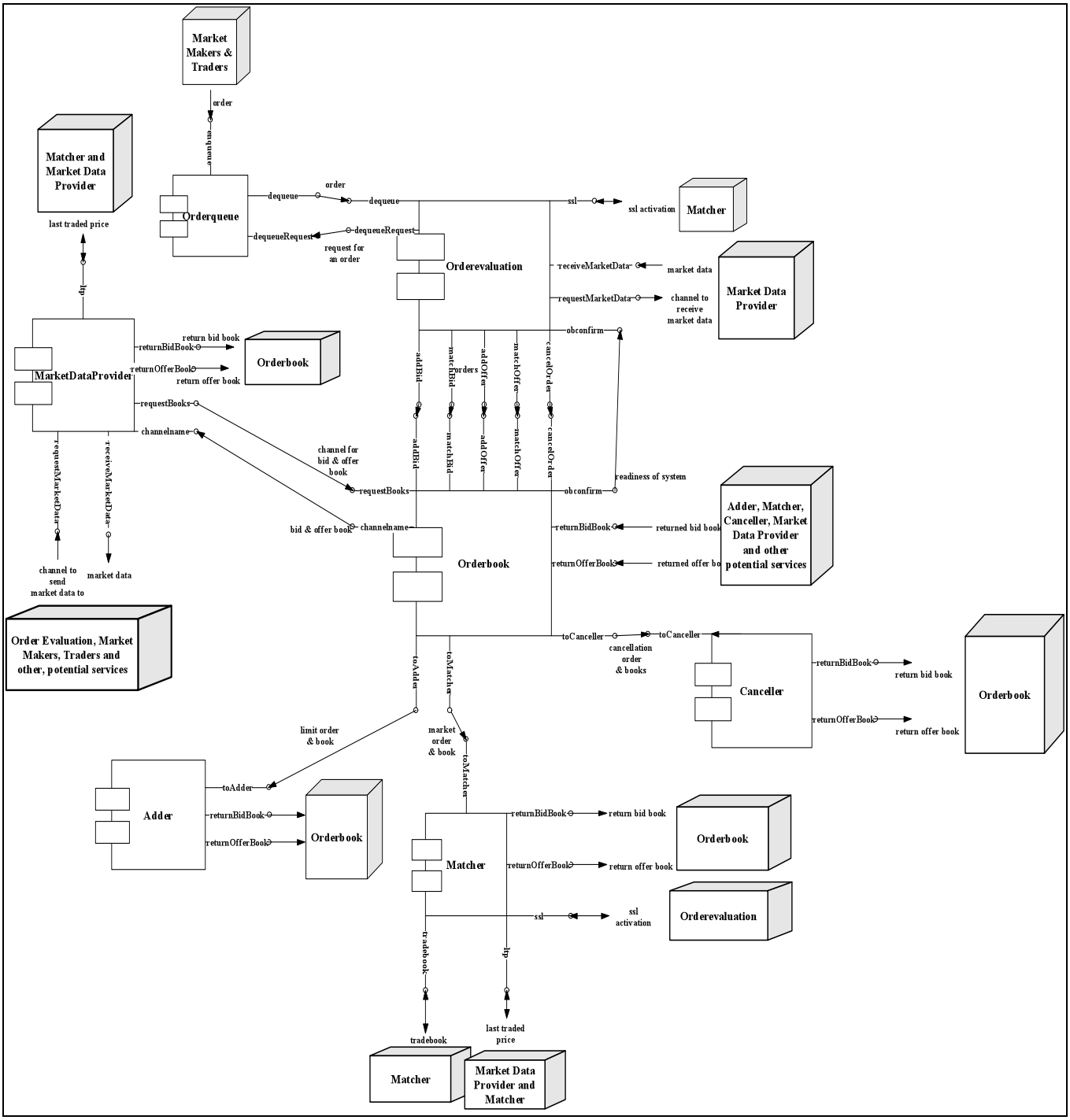

Figure 28: Overview of the exchange system.

4.2 Market Makers & Traders

We will first take a look at the market makers (MM) in the system, afterwards we will explain why the design of the traders is not that much different and how the message system gets its messages. Again we have the requirement that the simulation should be easily adjustable for a user, therefore we are using the modular approach again. Figure 29 displays the individual parts of a MM which have the following tasks:

- Market maker core: Refers to the basic structure which manages the data structures and to which the other subsystems are attached to.
- Order algorithm: This subsystem deals with the issuing of orders.
- Message system: Processes all the messages addressed to the MM.

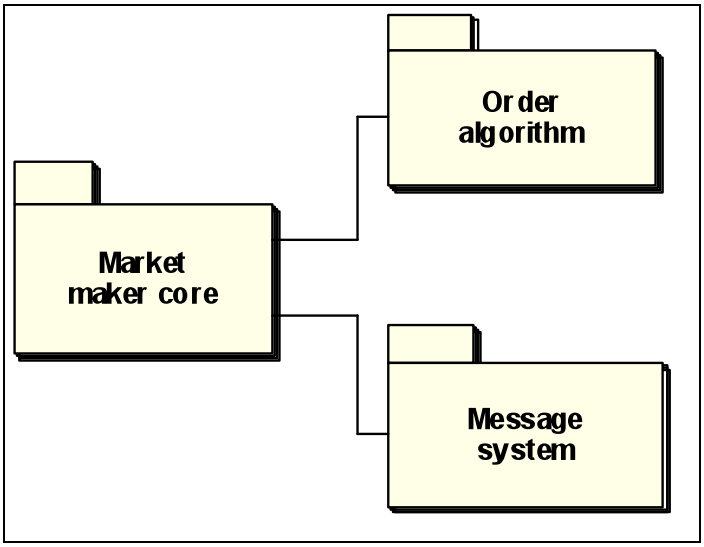

Figure 29: Schematic illustration of a market maker.

This architecture has the advantage that the behaviour of a MM, when orders have to be issued or a message is received, can easily be changed and also independently from each other, i.e. if someone wants to test the MM with a new algorithm to issue orders, s/he can do so without touching the message system. For the further investigation of the market maker, we will focus on the two main subsystems, the order algorithm and the message system, we do not cover the core in this section because the design of the standard core is comparable to the order book but far less complex, i.e. in only delegates incoming data to the other two subsystems and provides them with other data, like the current inventory or the market maker's order book, however, we have devoted a section in the Implementation chapter to the core.

4.2.1 Order algorithm

The order algorithm, in our standard model, is directly in charge of issuing all kind of orders and indirectly of managing the inventory, i.e. the algorithm enforces all specified limits or any other requirement related to the inventory. For our simulation we will use an algorithm which is based on Clack's (2012) archetypical market maker algorithm.

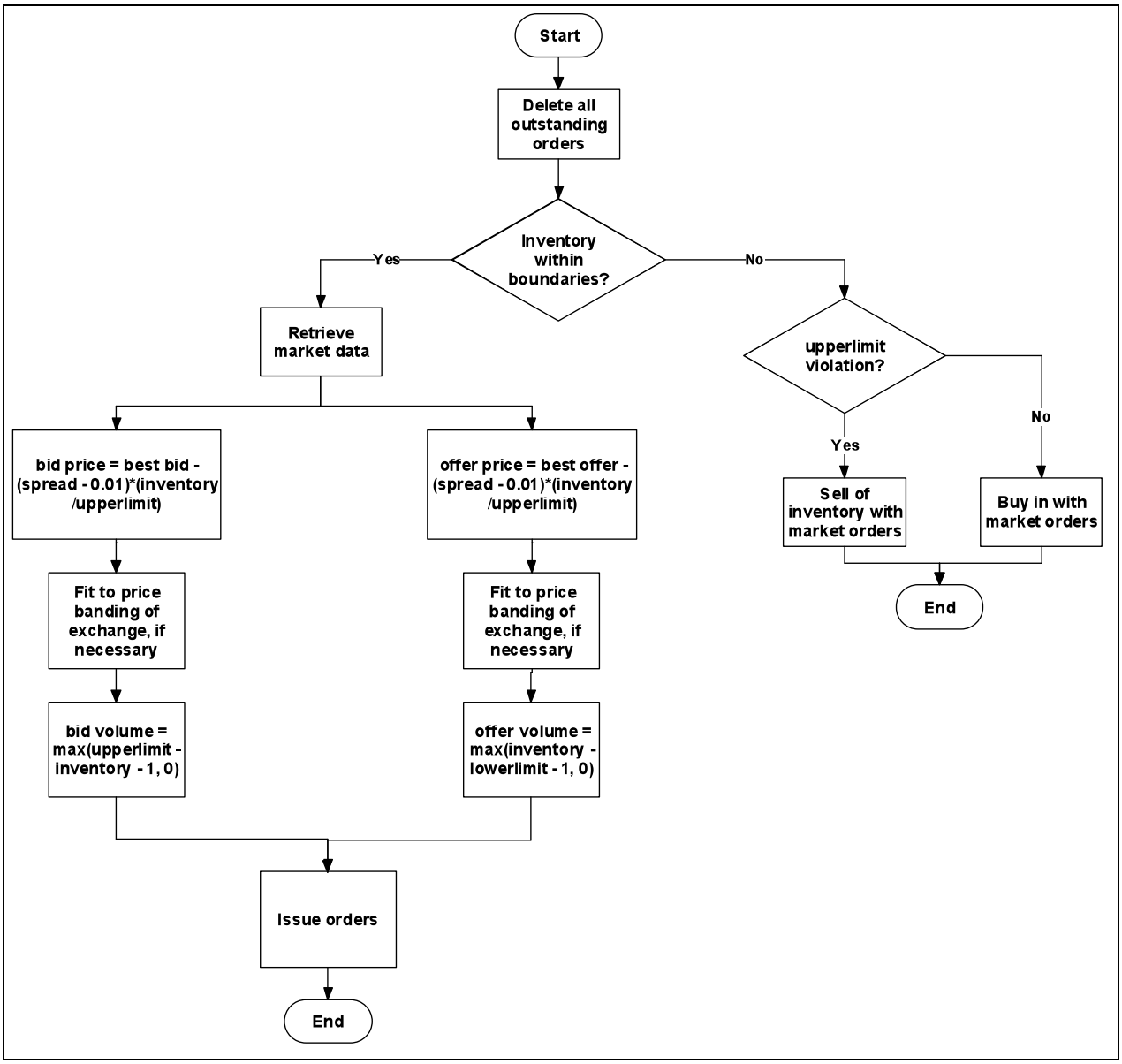

Figure 30: Flowchart of the standard order algorithm.

The algorithm proposed by Clack has the advantage that it is inherently stable and only a time delay can lead to instability, i.e. a violation of the inventory limits. It should be pointed out that the depicted algorithm would not take care of a situation where one or both sides of the order book are empty, i.e. there is no best bid or offer price, this issue has been resolved by using the last traded price plus/minus half of the

price limit (price banding) on incoming orders enforced by the exchange as an substitute, this has been omitted in the above diagram for reasons of clarity and comprehensibility. The 0.01 in the price calculation refers to the smallest change in the price, the ticker size, and has been chosen arbitrarily for the simulation.

4.2.2 Message system

The message system manages incoming messages from external entities, to do so it usually requires the inventory and the order book of the MM (alternative realizations might need different resources). In figure 31 the standard message system, which has been designed as standard module for our simulation tool kit, can be seen.

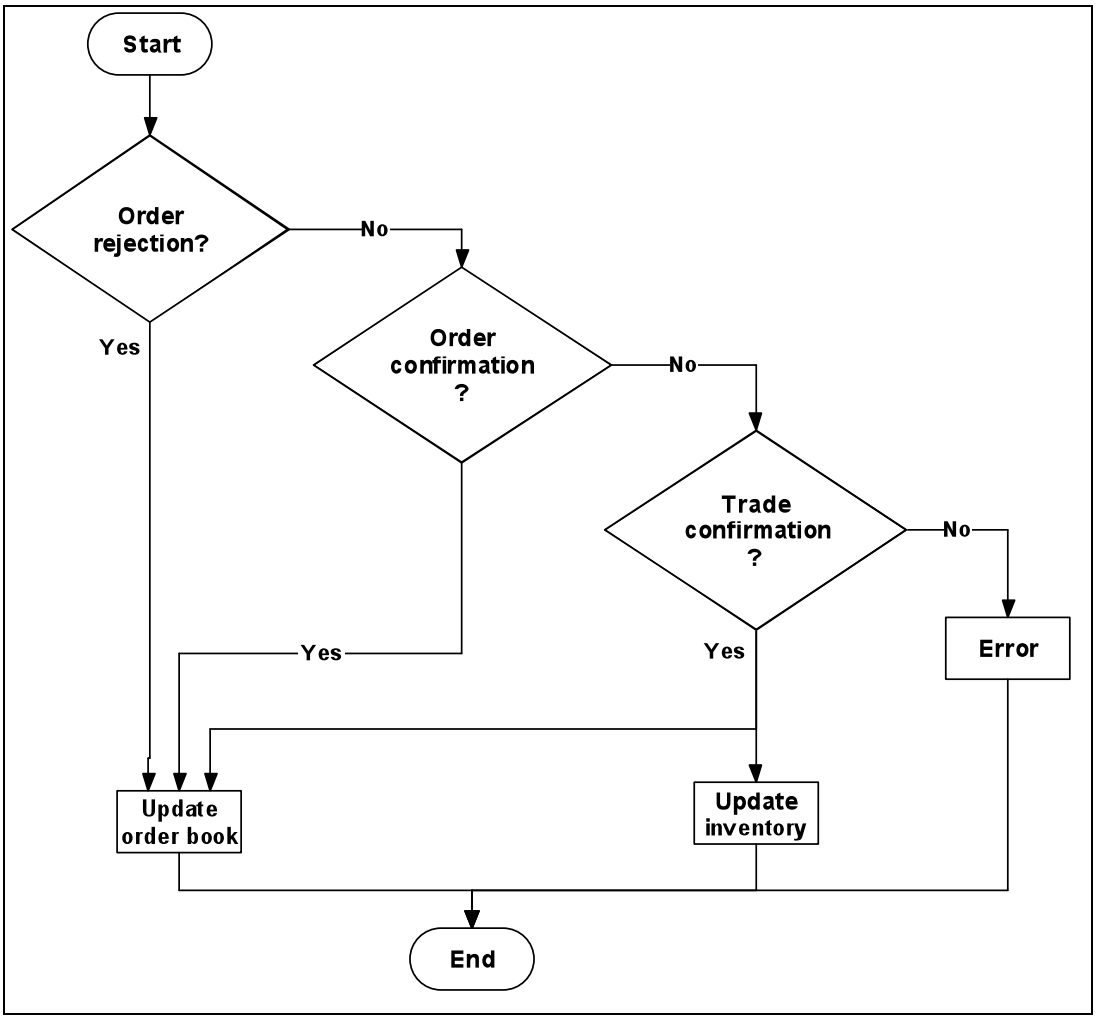

Figure 31: Flowchart of the standard message system.

The message system, when used in a simulation, takes care of three different kinds of messages: rejection, order confirmation and trade confirmation messages. Rejection messages are issued by the exchange when an order from the market maker got rejected, order confirmation messages confirms non-trade-related issues,

e.g. when a cancellation order has successfully been executed, the market maker receives an order confirmation, however, we know that the term order confirmation is usually used to indicate that the exchange has received the order, nevertheless, our standard model assumes that the delivery of an order is guaranteed and therefore we can use this message type more generally, which has the advantage that we do not need to cope with many different message types, i.e. one for notifying a cancellation one for indicating that a GTD order has been expired and so on. The trade confirmation messages on the other hand to notify the market maker of executed trades.

4.2.3 Traders

The design of the trader is very similar to the design of the market maker, i.e. it uses the same modular approach of a core, an order algorithm and a message system, however, in the standard model the requirements of the traders are not as complex as the one of the market makers, the traders are only required to take liquidity from the exchange, i.e. to issue buy or sell orders Moreover, the traders do not need to take care of their inventory, i.e. a pseudo random number generator fulfils the requirements for the order issuing behaviour of the trader sufficiently.

However, some people might argue that the division of market makers and other traders are not really necessary, provided that they more or less only differ by different order algorithm and message systems, and indeed the structure is the same and it is more of a naming issue to keep the worked out agents from the Analysis Chapter separated, however, the actual implementation of market makers and other traders strengthens this division, e.g. both require different parameters to be initiated.

4.2.4 Orders and Status Messages

So far we have not covered how the status-message-communication between the exchange and the other market participants work, i.e. how does a market maker receive its status messages for its orders? Due to the flexible nature of our simulation, a globally known channel is not feasible, someone would always have to change the number of channels and even more, how would the exchange now, which channel belongs to which market participant?

This problem can be easily resolved with storing a trader-id in each order, which would identify the correct channel, however, the inflexibility would still remain.

Nevertheless, our solution is not that different from the trader-id approach. Instead of storing an id, we are storing a restricted channel within each order. This restricted channel is one of the parameters of the market makers and traders and has to be created right at the beginning of the simulation and is afterwards never changed, i.e. the same channel is forwarded whenever the market maker or trader recreates. The subsystems of the exchange can use this channel to send messages to the other market participants, i.e. as long as a subsystem is able to acquire an order, it is also able to send a message to the issuer of that order. The actual message flows can be viewed in the Data Flow Diagram of the next section.

4.3 The Flow of Data

This section gives an overview of the interaction between the different market participants and an insight into information flows within the exchange.

The figure below shows the most important flows of data in our standard model between the different agents and within the exchange, the internal streams of the exchange have been explicitly depicted because they have been deemed to be more complex than their counterparts in the market makers and the traders and so this diagram might help the reader to gain a better understanding of the interplay between the different subsystems within the exchange.

As it can be seen below, the order queue is the only way for the external entities to make an input to the exchange, i.e. they have to feed all their orders into the queue. In contrast, the external entities receive messages directly from different subsystems of the exchange. There are two ways how this can be implemented, on the one hand, with a multiple-channel approach, i.e. every market participant has its own channel for each message type or, on the other hand, a one-channel approach, where every message type is received over the same channel. The first approach has the advantage that every channel can be exactly tailored to the requirements of each individual message type this would minimize the amount of useless data which has to be sent over the channel, e.g. aside of other fields which indicate the order-id or the volume, a trade confirmation message needs at least one number field of type float to indicate at which price the trade has been executed, a rejection message, on the other hand, does not need such a field at all and therefore would need to use a placeholder like 0.0 when using a one-channel approach. This might also reduce the danger of misinterpreting which part of the message stands for which information in a

bigger simulation with a more complex message system. However, the main drawback of a multiple-channel approach is that it would require three restricted channels for each of the market makers and traders in our simulation and the information about these three channels needs to be forwarded to the exchange via the orders, making the whole order issuing system and processing within the exchange more prone to errors. Therefore it has been decided to use the one-channel approach and interpret every message on the level of the message system, i.e. every message has a type field, which indicates whether the message is a rejection, order confirmation or trade confirmation message, from this field the message system infers what to do.

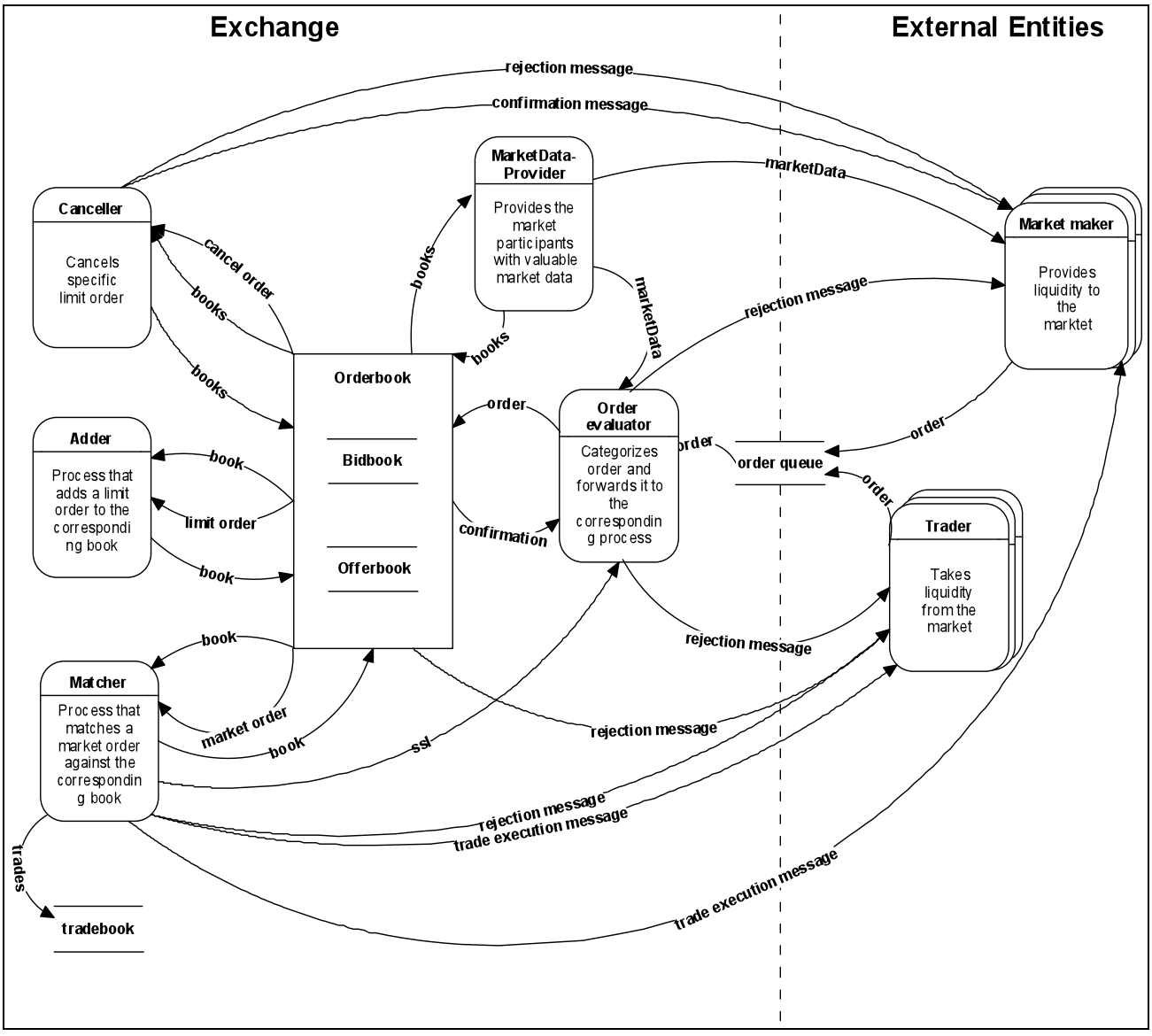

Figure 32: Dataflow diagram of the system.

5 Implementation

In this chapter we will look at the actual implementation, we will present the function library, which improves the usability of the SPiM significantly, and also parts of the implementation of the simulation, which demonstrates how we have circumvented some of the shortcomings in SPiM, especially the challenge of how to display the inventory of the market makers in the built in plotting tool of SPiM. Finally, we will also give an explanation, how we have realized the time delay.

5.1 Function Library

5.1.1 List Manipulation

While SPiM incorporates some of the standard list operations known from functional programming languages, like an operator which concatenates 2 lists, it does not provide the convenient ways to manipulate the data stored in the list like the functional languages, this makes the application of lists in SPiM cumbersome, for example the implementation of the order book for the simulation conducted in this work will require several lists and these lists will be searched for specific orders, the time code of some orders will be adjusted and orders will be deleted. To implement the whole process of traversing, manipulating and rebuilding the list for each of the individual tasks would lead to a substantial amount of redundant code. This section will provide the implementation of some of the more sophisticated operations to manipulate lists and so reduce the effect of redundant code. Additional functions can be found in the appendix D.

5.1.1.1 Filter

```
let xfilter(functionIn:chan('o),functionOut:chan(bool),x:list('o),
intermed:list('o),result:chan(list('o))) = (
    match x
    case [] -> !result(intermed)
    case head::tail -> (!functionIn(head) | ?functionOut(a);
                            if a then

    xfilter(functionIn,functionOut,tail,intermed+(head::[]),result)
                            else

    xfilter(functionIn,functionOut,tail,intermed,result)
                          )
)

let filter(functionIn:chan('o),functionOut:chan(bool),x:list('o),
result:chan(list('o))) = (
    xfilter(functionIn,functionOut,x,[],result)
)
```

Figure 33: Implementation of the filter function for SPiM.

The filter function has the task to filter a list for specified elements. To work with the function the developer needs to set up an evaluation function which takes in a single element of the list on the functionIn-channel, evaluates the element whether it meets the conditions, e.g. the third element of the 3-tuple is smaller than 5, and eventually outputs the result of the evaluation on the functionOut-channel. The filter function itself uses an accumulative recursion to solve the problem, when the evaluation function results true, xfilter recreates itself with the intermed-parameter appended to a list containing the evaluated element and the tail of the list as list-parameter, in case that the evaluation function returns false, the xfilter is recreated with the previous intermed-parameter but still with the tail. The process filter is a shortcut, such that the user does not need to consider the initial value of the intermed-parameter. When the end of the list is reached, the value of intermed is written to the result-channel.

Some people might argue that a stack recursion would solve the problem too and that in a much concise way and, as figure 34 demonstrates, it can be done. This time, when the evaluation function results true, a new result-channel is created and the Filter-Process is recreated with the new channel and the rest of the list, in case that the evaluation function returns false, the Filter-Process is recreated with the previous result-channel. When the end of the list is reached, the stack collapses and every element, which has evaluated to false, is bypassed when the resulting list is rebuilt. This is how it should work and it actually works that way as long as less than four restricted channels are needed for the stack because as soon as the program would need to go back more than three channels, SPiM cannot evaluate the statement anymore, i.e. after the third recursion SPiM “forgets” previously defined, restricted channels, which makes it highly advisable to use accumulative recursion.

```
let filter(functionIn:chan('o),functionOut:chan(bool),x:list('o),
result:chan(list('o))) = (
    match x
    case [] -> !result([])
    case first::rest -> (!functionIn(first) | ?functionOut(a);
                            if a then (
                                new c:chan(list('o)) (
                                    filter(functionIn,functionOut,rest,c) |
                                    ?c(res); !result(first::res)
                                )
                            )
                            else
                                filter(functionIn,functionOut,rest,result)
                        )
)
```

Figure 34: Implementation of the filter function with a stack recursion for SPiM.

5.1.1.2 Map

```
let xmap(functionIn:chan('o),functionOut:chan('p),x:list('o),
intermed:list('p),result:chan(list('p))) = (
    match x
    case [] -> !result(intermed)
    case head::tail -> (!functionIn(first) | ?functionOut(a);
                        xmap(functionIn,functionOut,tail,
                        intermed+(a::[]),result:chan(list('p)))
                        )
)

let map(functionIn:chan('o),functionOut:chan('p),x:list('o),
result:chan(list('p))) = (
    xmap(functionIn,functionOut,x,[],result)
)
```

Figure 35: Implementation of the map function for SPiM.

The basic structure of the map-Process is quite similar to the filter-Process, however, instead of searching a list, map manipulates the elements of the list. Therefore, in order to use the map-Process, a function to manipulate the single elements is required. This function will take in elements on the functionIn-channel, manipulate them and then output them on the functionOut-channel. Once again an accumulative recursion is used where the result is written to the result-channel when the end of the list is reached.

5.1.1.3 Foldl

```
let foldl(functionIn:chan('o,'o),functionOut:chan('o),intermed:'o,x:list('o),
result:chan('o)) = (
    match x
    case [] -> !result(intermed)
    case first::rest -> (!functionIn(intermed,first) | ?functionOut(res);
                            foldl(functionIn,functionOut,res,rest,result)
                        )
)
```

Figure 36: Implementation of the foldl function for SPiM.

The foldl-Process takes the value of the intermed argument and the head of the list and forwards both to a predefined function. The output of the function is the new intermed argument while the foldl-Process is recreated with the tail of the list. When the end of the list is reached, the intermed is outputted to the result channel. One way of how someone could see the procedure, is that foldl replaces all the concatenation operators, which hold together the list, with the predefined function.

5.1.1.4 Example

The example demonstrates how to use the three functions in combination. The task is to search a list of 2-tuples, for all 2-tuples where the second element exceeds some threshold. Afterwards all the second elements of the filtered list should be summed up.

```
let Result(a:int) = !c1()

let TupleFilter(input:chan((int,int)),result:chan(bool)) = (
    ?input((a,b));
    (if b > 3 then !result(true) else !result(false) | TupleFilter(input,result))
)

let SecondElement(input:chan((int,int)),result:chan(int)) = (
    ?input((a,b)); (!result(b) | SecondElement(input,result))
)

let Addition(input:chan(int,int),result:chan(int)) = (
    ?input(a,b); (!result(a+b) | Addition(input,result))
)

let LibraryExample(x:list((int,int))) = (
    new a:chan((int,int)) new b:chan(bool) new filter_result:chan(list((int,int)))(
        filter(a,b,x,filter_result) | TupleFilter(a,b) |
        ?filter_result(filteredList); (
            new c:chan((int,int)) new d:chan(int)
            new map_result:chan(list(int)) (
                map(c,d,filteredList,map_result) | SecondElement(c,d) |
                ?map_result(manipList); (
                    new e:chan(int,int) new f:chan(int)
                    new foldl_result:chan(int) (
                        foldl(e,f,0,manipList,foldl_result) | Addition(e,f) |
                        ?foldl_result(summation); Result(summation)
                    )
                )
            )
        )
    )
)

run (LibraryExample((2,4)::(1,6)::(4,1)::[]))
```

Figure 37: Example Program with the library functions filter, map and foldl.

For the first part of the example the process filter and the evaluation process TupleFilter is required. TupleFilter takes in a 2-Tuple via the input-channel, checks whether the second element is bigger than 3, writes the result to the result-channel and then recreates itself, such that the process filter can use TupleFilter for the next element of the list too. The filtered list is then manipulated by the process map and the SecondElement-Process, which takes in a 2-Tuple and outputs the second element. Afterwards the process foldl is used on the manipulated list to sum it up, therefore the Addition-Process is applied, which takes in two integers and returns the sum of them. Finally, a Result-Process is created which holds the result, before the system stops to run.

This example contains nine restricted channels and it might be tempting to reuse some of the channels, e.g. channel “a” and “c” are both transmitting integer 2-tuples, however, this approach would lead to errors because there is still one TupleFilter-

Process running (because of the recreation, when the last element is checked) when the process map is initiated, i.e. it could happen that the process TupleFilter reads from the channel and not the process SecondElement, however, this leads to a deadlock, because map is waiting for the SecondElement-Process to write to the channel d, while SecondElement is still waiting for an input and even TupleFilter cannot do anything, because there is no filter-Process anymore which could read from the result-channel.

5.1.2 Pseudo Random Number Generator

Another factor which aggravates the application of SPiM for finance-related issues is the generation of (pseudo) random numbers. Random numbers play a very important role in Finance, especially in the area of pricing, for example a Monte-Carlo-Simulation relies heavily on the generation of random numbers, however, SPiM does not provide a method to generate random numbers, all the stochastic behaviour is realized by the selection of the next firing channel. As a result, there is no short and concise way to implement even a standard pricing method like the Monte-Carlo-Method (compare Wilmott, 2007). To mitigate this drawback, this section will show how to implement a pseudo-random number Generator (PNG) in SPiM.

5.1.2.1 Channel-Approach

The first attempt to solve the random number problem was based on the channel-picking-algorithm of SPiM. As it can be seen in Figure 38 the concept was straightforward, a process was provided which had the only task to set up numerous output channels with different numbers of type float but the same channel name, if somewhere in the program a random number had been required, a single, matching input channel would have been sufficient to obtain an uniformly distributed random number, i.e. SPiM would randomly choose one of the output channels to forward its specified value to the input channel.

```
new udRnd:chan(float)

let randomUniform() = (
    replicate !udRnd(0.00) | replicate !udRnd(0.01) | replicate !udRnd(0.02) |
    replicate !udRnd(0.03) | replicate !udRnd(0.04) | replicate !udRnd(0.05) |
    (*...*)
    replicate !udRnd(0.97) | replicate !udRnd(0.98) | replicate !udRnd(0.99) |
    replicate !udRnd(1.00)
)
```

Figure 38: Code of the attempt to set up a PNG by using channels.

However, there is one major disadvantage with this approach, the performance. The way of how SPiM picks a channel is computationally very expensive[12]. The reason for this is that all the possible paths are evaluated which slows down the simulation tremendously (Cardelli and Phillips, 2004). Due to the fact that a simulation could need hundreds or even thousands of random numbers this approach has turned out to be not feasible, which is why we focus on another PNG.

5.1.2.2 Algorithm AS 183

The algorithm AS 183 is a multiplicative congruential PNG and was developed by Wichmann and Hill (1982). The main reason for using this algorithm is its simple structure, except of four modulus calculations all the required calculations are standard (built- in) arithmetic operations, while more modern generators like the Mersenne Twister (MT) tend to require more sophisticated operations, in case of the MT, bitwise comparison operators and shift operators are needed. However, when it comes to cycle length the AS 183 cannot compete with a modern PNG and also when a great number of random numbers are required there are better solutions than the AS 183, for example the MT creates quickly with every call 624 random numbers in the standard version (compare with Matsumoto and Nishimura, 1998). Nevertheless, this work has not the aspiration to implement everything in the most efficient way but to explore the potential of SPiM for finance-related issues, for this case, the performance of AS 183 is more than sufficient.

5.1.2.2.1 Code

```
new unifRnd:chan(float)

let PRGenerator(ix:int,iy:int,iz:int) = (
    (
        ix := 171*(ix-(ix/177)*177)-2*(ix/177) |
        iy := 172*(iy-(iy/176)*176)-35*(iy/176) |
        iz := 170*(iz-(iz/178)*178)-63*(iz/178)
    );
    (
        if ix < 0 then ix := ix+30269 |
        if iy < 0 then iy := iy+30307 |
        if iz < 0 then iz := iz+30323
    );
    (
        urnd := Modulo((float(ix)/30269.0 +
                        float(iy)/30307.0 +
                        float(iz)/30323.0),1.0);
        (!unifRnd(urnd) | PRGenerator(ix,iy,iz))
    )
)
```

[12] More about how SPiM chooses channels can be found in the section about the time delay

Figure 39: Pseudo code of the AS 183 implementation for the SPiM.

Figure 39 shows the pseudo code of the AS 183 algorithm, the main difference between pseudo code, how it is used in this example, and the actual implementation of the algorithm is that values can be assigned to a variable directly, instead of using input- and output-channels which have no additional purpose than to conduct calculations, i.e. the calculation is entered into the output-channel while the result is read from the input-channel in parallel. The part (i_-(i_/1__)*1__) in the first three equations calculates the remainder of the integer division i_ divided by 1__, someone might argue that this calculation could have been done in an own process, as a substitute for a built in modulo operator, however, in case of integers there is no value added in doing so, such a process would, whenever used, require to set up a restricted result channel, to initialize the process and then, in parallel, it must be read from the result channel to get the remainder. In the case where the remainder is further processed this would only increase the complexity of the code unnecessarily. In contrast, this argumentation does not hold for a number of type float, where the calculation of the remainder of a division is more sophisticated, because the type of the values needs to be changed two times, therefore a Modulo-Process has been implemented (see Figure 40). The final result is then written to the globally known unifRnd-channel.

```
let Modulo(dividend:float,divisor:float,result:chan(float)) = (
    new calculator:chan(float) (
        (if dividend > 0.0 then
            !calculator(float_of_int(int_of_float(dividend/divisor)))
        else
            !calculator(float_of_int(int_of_float((-1.0*dividend)/divisor)))
        ) | ?calculator(x); !result(dividend-(divisor*x))
    )
)
```

Figure 40: Implementation of the modulo function for float numbers.

5.1.2.2.2 Output

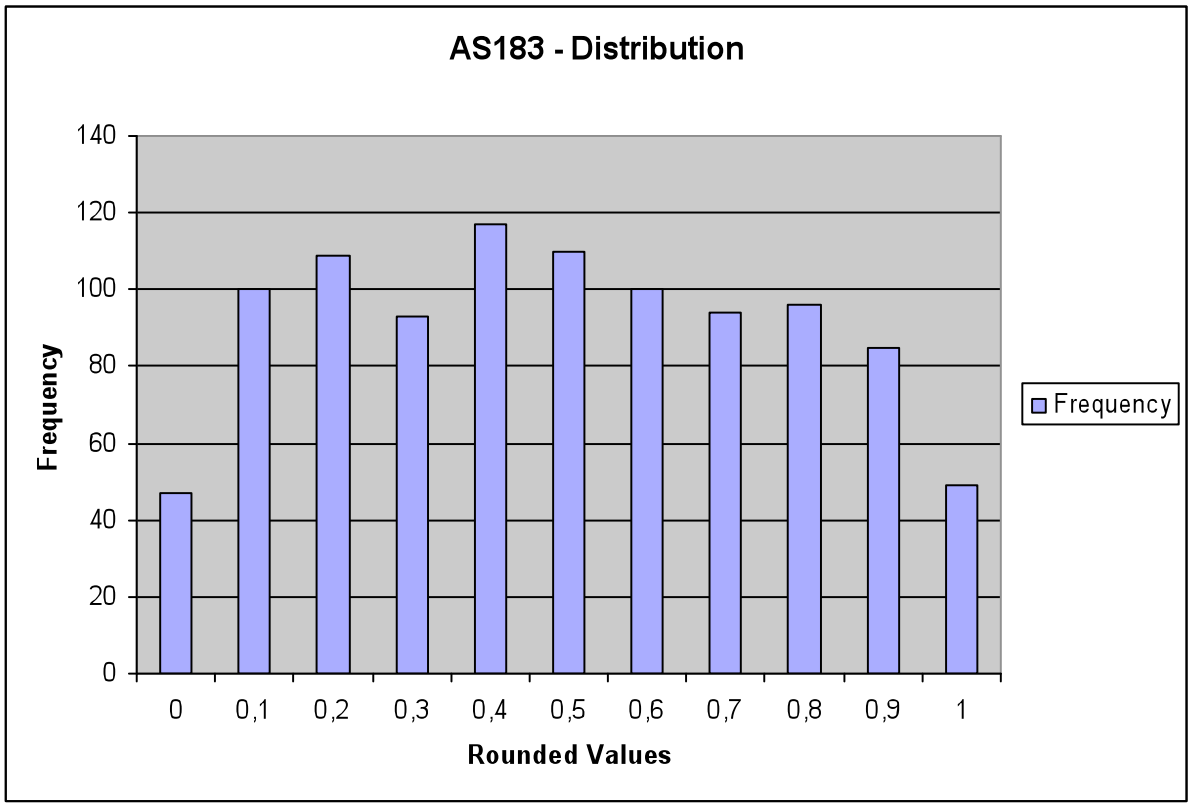

Figure 41: Distribution of the AS183 algorithm (1000 numbers).

Applying a Chi^2-Test for uniform distributions on the result, we get a Chi^2 value of 8.16. This satisfies the threshold condition to keep the Zero-Hypothesis at a significance level of 5% and 10 degrees of freedom (threshold is at 18.31), , i.e. the statistical test indicates that the numbers generated are uniformly distributed (Field, 2009).

5.1.2.3 Normal-Distribution

To get a normal distribution of random variables we do not need to entirely start over, the following algorithm creates normally distributed variables based on uniformly distributed ones, which are created by the AS 183 algorithm.

```
let xStdRandomGauss(sumsi:float,counter:int,result:chan(float)) = (
    if counter = 0 then
        !result(sumsi-6.0)
    else
        ?unifRnd(number); xStdRandomGauss(sumsi+number,counter-1,result)
)

let StdRandomGauss(result:chan(float)) = xStdRandomGauss(0.0,12,result)
```

Figure 42: Pseudo code of the generator for normally distributed random variables.

The algorithm uses the fact that the sum of 12 uniformly distributed random numbers (between 0 and 1) minus 6 equals a normally distributed random number (Wilmott, 2007). The summation is conducted by the auxiliary process xStdRandomGauss and is based on an accumulative recursion. In case that a developer wants to use the generator, s/he only needs to set up a channel for floats, create the process StdRandomGauss with the channel as the parameter and then read from the channel.

To be more precise, the algorithm generates a standard normal distribution, to get a normal distribution with different parameters, the following relation can be used, for which a library function has also been created:

$$□=□ \ *□+\mu \tag{1}$$

Where Y stands for a standard-normally distributed variable, σ is the desired standard deviation and μ is the mean of the new variable.

5.1.2.4 Log-Normal-Distribution

The log-normal distribution is widely used in the field of Finance, especially for the purpose of modelling the compounded rate of return (Wilmott, 2007). A log-normally distributed variable is calculated by:

$$□=□□ \tag{2}$$

Where Y denotes a normally distributed variable and e stands for Euler's number. Although the formula is not very sophisticated it raises some new problems, because SPiM does not have a built-in operator to calculate exponential functions and also Euler's number is not part of the standard environment. However, these two problems were solved by implementing a Taylor series expansion to approximate the value of □□, as shown by figure 43.

```
let xFactorial(fact:float,intermed:float,result:chan(float)) = (
    if fact > 0.0 then
        xFactorial(fact-1.0,intermed*fact,result)
    else
        !result(intermed)
)

let Factorial(x:float,result:chan(float)) = (
    xFactorial(x,1.0,result)
)

let xPow(basis:float,exponent:float,intermed:float,result:chan(float)) = (
    if exponent > 0.0 then
        xPow(basis,exponent-1.0,intermed*basis,result)
    else
        !result(intermed)
)

let Pow(basis:float,exponent:float,result:chan(float)) = (
    xPow(basis,exponent,1.0,result)
)

let xe(x:float,steps:float,intermed:float,result:chan(float)) = (
    if steps <> 0.0 then
        (new a:chan(float) new b:chan(float) (
            Pow(x,steps,a) | Factorial(steps,b) |
            ?a(power);?b(factor);xe(x,steps-1.0,intermed+power/factor,result)
        ))
    else
        !result(intermed)
)

let e(x:float,result:chan(float)) = xe(x,10.0,1.0,result)
```

Figure 43: Taylor approximation of Euler's number to the power of x.

The figure shows how □□ can be approximated in SPiM. The process xe does the actual approximation by summing up the individual parts of the Taylor Series, whereas the series starts with the smallest value (except of the initial value of the intermed), i.e. at each recursion, until the parameter steps equals zero, (3) is calculated by using the Pow-Process and the Factorial-Process and added to the intermed-value, when the parameter steps has reached zero, the intermed-value, i.e. the final approximation, is written to the result-channel. By increasing the initial steps parameter, the precision of the approximation can be improved. The processes e, Pow and Factorial are actually only shortcuts to improve the usability.

□□□□□□/□□□□□ (3)

5.2 The System

This section shows the implementation of some of the more sophisticated processes from the exchange and the market makers. In the subsection about the market makers we will demonstrate how to display the inventory of the market makers, which solves also the other challenge of how to display the amount of orders in the order queue.

5.2.1 Exchange

In this section we will show how the Adder has been implemented, because the Adder is one of the processes which benefits the most from the library.

5.2.1.1 Takedropwhile

To understand the behaviour of the Adder it is necessary to understand another library process first, the takedropwhile process, this process combines the functionality of the dropwhile-Process and the takewhile-Process.

```
let dropwhile(functionIn:chan('o),functionOut:chan(bool),
x:list('o),result:chan(list('o))) = (
    match x
    case [] -> !result([])
    case first::rest -> (!functionIn(first); ?functionOut(a); (
                            if a then
                                dropwhile(functionIn,functionOut,
                                rest,result)
                            else
                                !result(first::rest)
                        ))
)

let takewhile(functionIn:chan('o),functionOut:chan(bool),
x:list('o),result:chan(list('o))) =
xtakewhile(functionIn,functionOut,x,result,[])

and xtakewhile(functionIn:chan('o),functionOut:chan(bool),
x:list('o),result:chan(list('o)),intermed:list('o)) = (
    match x
    case [] -> !result(intermed)
    case first::rest -> (!functionIn(first); ?functionOut(a); (
                            if a then
                                xtakewhile(functionIn,functionOut,rest,
                                result,intermed+(first::[]))
                            else
                                !result(intermed)
                        ))
)

let takedropwhile(functionIn:chan('o),functionOut:chan(bool),x:list('o),
result:chan(list('o),list('o))) = (
    new resTake:chan(list('o)) new resDrop:chan(list('o)) (
        takewhile(functionIn,functionOut,x,resTake);
        dropwhile(functionIn,functionOut,x,resDrop) |
        ?resTake(a); ?resDrop(b); !result(a,b)
    )
)
```

Figure 44: Implementation of dropwhile, takewhile and takedropwhile.

Figure 44 shows the actual implementation of the takedropwhile-Process, the process first applies takewhile onto the list, i.e. as long as the evaluation process, which is connected to the takedropwhile-Process via the functionIn- and functionOut-channel, returns true as an output, the evaluated element is stored in the intermed-value, should the evaluation process return false, the takewhile-Process outputs the intermed-value onto the result-channel, i.e. in this case to the takedropwhile-Process. Secondly, dropwhile is applied onto the list, this process does the opposite than takewhile, i.e. as long as the evaluation returns true, the evaluated elements are dropped, in case that false is returned, the dropwhile-Process outputs the remaining list. Takedropwhile takes these two results and writes them to the result-channel, i.e. in case that someone wants to search a list for an element, manipulate the element

and then reconstruct the list, takedropwhile reduces the amount of channels and the overall complexity tremendously.

5.2.1.2 The Adder

Figure 45 shows the actual implementation of how the Adder adds orders to a bid or an offer book. Regarding the replicate at the beginning, this recreates the Adder-Process automatically whenever the toAdder-channel is executed and therefore we do not need to worry that the Adder-Process terminates in one of its branches without recreating. It can also be seen that our Adder does not need any parameters, this has to do with the fact that all the necessary channels, which connect the Adder with the rest of the system, are globally known and the only resources the Adder requires are an order and a bid or an offer book, which the Adder will both receive from the Orderbook-Process. Nevertheless, the basic concept of the implementation is as followed:

1) The necessary channels for the takedropwhile-Process and the evaluation process must be set up and the two processes must be initiated. The evaluation, which is displayed in Figure 46, takes in an order list from the book and checks the price of the order at the top of the list against the price of the order which should be added, in case of a bid limit order it returns true when the current price level of the order list is higher than the price of the order to add and vice versa if an offer needs to be added. Due to this behaviour, the takedropwhile-Process returns on the twd_result-channel a take-value which contains, in case of a bid, all the order lists which have a higher price level and a drop-value which contains all the order lists which have an equally high or lover price level.

2) In case that the drop-value is actually empty, the Adder appends the take-value to a new order book, which contains only the order to add in a list and writes it to the corresponding return-channel (for bid or offer book). Should drop be non-empty, pattern-matching is used to retrieve the head of the drop-value, i.e. the order list at the top. This list should not be empty, by the internal logic and therefore raises an error, if it is empty. The second pattern-matching, i.e. the pattern-matching of the head, to retrieve the information which is stored in the list can be usually done at the level of the first pattern-matching as well, however, SPiM does not support this “list of lists”-pattern-matching.

3) The reason for actually retrieving the head list and the top order of this list is that the orders at the top of drop can either have a lower price, in case of a bid limit order, or the same price and therefore we need to compare the price of the order to add with the price of an order at the top list.

4) In case that the prices are equal, the Adder appends the head list to a new list which contains the order to add, such that the order to add is now the last element of the head list and then the head list gets concatenated to the tail list again and the take-part is appended to this list of order lists. In case that the price of the order to add is lower, in case of a bid order, than the price represented by the head order list, the order to add forms a new order list which is concatenated to the reunited head and tail list, the take-part is then appended to the new list of lists and the result is outputted to the returnBidBook-channel.

```
let Adder() = (
    replicate ?toAdder((id,otype,oqualifier,price,volume,
    timestamp,issuer),book); (
        new twd_result:chan(orderbook,orderbook)
        new fIn:chan(list(order))
        new fOut:chan(bool) (
            priceCompare(fIn,price,fOut);
            takedropwhile(fIn,fOut,book,twd_result) |
            ?twd_result(take,drop);
            match drop
            case [] ->  if otype = bid then
                            !returnBidBook(take+(((id,otype,oqualifier,
                            price,volume,timestamp,issuer)::[])::[]))
                        else
                            !returnOfferBook(take+(((id,otype,oqualifier,
                            price,volume,timestamp,issuer)::[])::[]))
            case head::tail ->  (
                match head
                case [] -> Error("Adder - matching of the head of the drop")
                case (lid,ot,oq,pr,vol,tst,i)::rest -> (
                    if otype = bid then (
                        if price = pr then
                            !returnBidBook(take+((head+(id,otype,oqualifier,
                            price,volume,timestamp,issuer)::[])::tail))
                        else
                            !returnBidBook(
                                take+(((id,otype,oqualifier,price,
                                volume,timestamp,issuer)::[])::(head::tail))
                            )
                    )
                    else (
                        if price = pr then
                            !returnOfferBook(
                                take+((head+((id,otype,oqualifier,
                                price,volume,timestamp,issuer)::[]))::tail)
                            )
                        else
                            !returnOfferBook(take+(((id,otype,oqualifier,
                            price,volume,timestamp,issuer)::[])::drop))
                    )
                )
            )
        )
    )
)
```

Figure 45: Implementation of the Adder.

```
let priceCompare(input:chan(list(order)),
orderToAddPrice:float,result:chan(bool)) = (
        ?input(orderlist);
        match orderlist
        case [] -> (!result(true);
                    priceCompare(input,orderToAddPrice,result))
        case (id,otype,oqualifier,
              lprice,volume,timestamp,issuer)::tail -> (
            if otype = bid then (
                !result(lprice>orderToAddPrice);
                priceCompare(input,orderToAddPrice,result)
            )
            else (
                !result(lprice<orderToAddPrice);
                priceCompare(input,orderToAddPrice,result)
            )
        )
)
```

Figure 46: Implementation of the evaluation process for the Adder.

5.2.2 Market Maker

In this section we will take a closer look at the implementation of the core of the MarketMaker-Process, we will see how different connection latencies can be realized in the system and we will also demonstrate how we have resolved the challenge of plotting values with the built-in plotting tool of SPiM.

5.2.2.1 Market Maker Core

The market maker core is the foundation of the market maker, in this section we will explain its implementation.

Figure 47 shows the pseudo code version of the core. The do-or approach has been chosen over an implementation based on the two sub-processes running in parallel. This has been done because both processes require similar resources and therefore they cannot run independently from each other without using at least one additional channel to share these resources. To avoid the necessity of this channel, we have implemented a do-or approach.

```
let MarketMaker(MarketMakerData) = (
    do  ?issueOrder();
        (!toOrderAlgo(inventory,limits,resultChan) |
        resultChan(newOrders); MarketMaker(newMarketMakerData))
    or ?receive(m);
       (!toMessageSys(m,inventory,orderbook,resultChan) |
       ?resultChan(xzy); MarketMaker(newMarketMakerData))
)
```

Figure 47: Pseudo code of the market maker core.

Regarding the connecting channels in figure 47, the toOrderAlgo- and the toMessageSys-channel are globally known in our simulation because this enables different MM to have access to the same processing-processes, i.e. more than one MM can easily use one message or order strategy, which improves code reusability substantially, to avoid the necessity of hard-coding the different channel names into the core, the channel names are implemented as parameters of the MM, i.e. whenever a MarketMaker-Process is created, the channels need to be defined, this allows for a high level of flexibility when simulating and also makes it possible to model that a MM could change its behaviour over time, e.g. in case that the MM makes substantial losses it could recreate itself with a different order algorithm channel. However, the high modularity and reusability of code have also a drawback, it is vital to use restricted channels, instead of globally-known ones, for the implementation of the resultChan-channels because otherwise data integrity could be violated, i.e. a MM receives the wrong return values. This can easily be the case when many MM use the same message system or order algorithm process.

Some readers might wonder where the corresponding output-version of the issueOrder-channel sits. First of all, we should explain why there is actually a need for an issueOrder-channel. This channel blocks the do-branch because every branch of a choice-Process needs to be blocked by an action, otherwise SPiM returns a syntax error[13]. We came up with several strategies of how to solve this issue:

1) Implement the two sub-processes, i.e. order issuing and message system in parallel. As mentioned before, this would require an additional channel and the problem is that if someone wanted to write their own sub-process they would now also need to take care of the interaction with the other sub-process. For example, someone implements a new message system which requires a new element which is usually only used by the order issuing algorithm. With a do-or approach, the developer would simply take the market core and add the new element to the toMessageSys-channel and can now test the message system with different order issuing algorithms. However, if the two sub-processes run in parallel, the developer would need to change every order issuing algorithm. Therefore we have abandoned this approach.

[13] This is understandable because the whole do-or structure is built on the concept that the first action, which is executed, determines the branch that will be executed.

2) Set up an enqueueRequest-channel at the level of the order queue and read from the enqueue-channel in parallel, comparable to the situation on the dequeue side of the queue. The drawback of this solution is that the whole concept of connection latency (see later) would be obsolete, because only one MM would send its order at a time and the queue is waiting for him.

3) An alternative form of solution number 2 would be, to set up a second or-branch in the order queue which is blocked by enqueueRequest[14]. This would lead to the situation that more than one MM would try to enqueue an order and therefore the connection latency would still make a difference. However, this approach would shift away the queue from a mere data structure to a control element, which would reduce the reusability of the code for other tasks, therefore we have chosen approach number 4 to resolve the problem.

4) Use a replicate function on an issueOrder-output-channel. This approach more or less takes the enqueueRequest-channel out of the order queue and puts the channel in an own process with the replicate function in front of it (and renames the channel to issueOrder). This gives us all the advantages of solution number 3 but not the drawback and therefore we have used it.

5.2.2.2 Connection Latency

As indicated in the Introduction, the different market participants have different execution speeds and an accurate model should take this fact into consideration.

There are various ways how the different execution speeds can be realized in SPiM, for example with delays. Figure 48 depicts our approach. The solution is based on the interaction rate x which can be assigned to any channel, the lower this rate is, the longer it takes, on average, for the transmission to be executed, i.e. the execution time is actually probabilistic. To be more precise, an order is created by an order algorithm process, this algorithm then sets up a channel with the interaction rate x, whereas this rate is determined by the market participant who wishes to issue an order, i.e. it is one of the values which have to be written to the toOrderAlgo-channel. Afterwards a Connection-Process is created with the channel as a parameter and then the order is written to the channel, when the Connection-Process eventually receives the order, it forwards it immediately to the order queue via the enqueue-

[14] The counterpart would sit where in the current version the issueOrder-channel sits.

channel. Advantages of this implementation are that it models quite naturally the idea of a cable and its latency and that it is easily realized. One drawback of this solution is that, due to the probabilistic nature, an order could be overtaken by another order of the same issuer. However, we have considered this by preventing the issuer of the order from issuing new orders until the order has been written to the enqueue-channel.

```
let Connection(c:chan(order)) = ?c(order); !enqueue(order)

let OrderAlgorithm() = (
    (*...*)
    new c:chan@x Connection(c); !c(order)
)
```

Figure 48: Implementation of the different execution speeds.

To simulate a system without a time delay at the level of the exchange, i.e. the exchange processes any incoming order immediately, it is required to rewrite the Connection-Process, such that it outputs to the order evaluation directly and not to the order queue.

5.2.2.3 Following the Inventory

As we know from the Analysis chapter, one of the main problems when working with SPiM is that the built-in plotting window can only display the number of instances of a process[15], not a value. This forces the user to read through the debug output or the confusing reactions list or table and makes it difficult to follow changing values, like an inventory. To solve the problem, we had to find a way of how we could turn a value into a process. In this section, we will demonstrate how we have implemented a way to plot a positive integer value in the plotting window, such that the changing values of the inventories from market makers or the amount of orders in the queue can be displayed.

Figure 49 shows the procedure based on an inventory counter for a market maker. Every asset, which is added to the inventory of the market maker, leads to the creation of exactly one MM1-Process (MM1 indicates that it is the inventory of market maker 1), and every asset that is sold leads to the termination of one MM1-Process by writing to the assetSold1-channel. However, to simplify the application we have also provided the MassCreate- and the x1MassCreate-Process, the latter is used to

[15] The process gets displayed by adding him to the “directive plot” instruction. More about this can be found in the Background Reading Chapter.

recursively create the MM1-Processes, i.e. the market maker only needs to provide the amount which has been acquired and x1MassCreate creates the corresponding instances of MM1-Processes, however, this approach would require that every market maker has a hard-coded reference to its x*Number*MassCreate-Process, which is not very efficient, and therefore the MassCreate-Process has been developed, this process only needs an integer value to identify which x*Number*MassCreate-Process should be executed to increase the correct counter, because the different integer values can be assigned to the different market makers as a parameter, this is more code efficient. MassTerminate and x*Number*Terminate are the counterparts of the Create-Processes, i.e. instead of creating a counter-Process, they write to the corresponding channel to terminate the process and therefore reducing the number of processes displayed in the plotting window.

```
new assetSold1:chan

let MM1() = ?assetSold1()

let x1MassCreate(x:int) = (
    if x>0 then
        (MM1() | x1MassCreate(x-1))
    else
        ()
)

let MassCreate(x:int,MM:int) = (
    if MM=1 then
        x1MassCreate(x)
    else
        ()
)

let x1MassTerminate(x:int) = (
    if x>0 then
        (!assetSold1() | x1MassTerminate(x-1))
    else
        ()
)

let MassTerminate(x:int,MM:int) = (
    if MM=1 then
        x1MassTerminate(x)
    else
        ()
)
```

Figure 49: Implementation of the inventory counter.

This approach solves also, at least to some extent, the problem of displaying negative inventory numbers, e.g. when the market makers have conducted short-

selling. In order to display a negative number, an initial amount of instances of the counter process needs to be created, this can be easily done with the command “x of Process()”, where x is an integer, when the market maker now short-sells an asset, one of the counter processes gets terminated. Some readers might now wonder why we have implemented our recursive approach, although there is a command which creates a number of instances of a process. The problem with this command is that it does not work with a variable, i.e. you cannot write an “x of Process()”-command into a process body, whereas x is a parameter of the process, this would raise a syntax error, you can only write, for example: “5 of Process()”. Therefore we have implemented the recursive counter process concept.

5.2.3 Time Delay

In this section we will explain how the time delay occurs in our system and how a system without time delay can be implemented. This explanation is based on the work of Andrew Phillips and Luca Cardelli (2004).

To implement a time delay in SPiM is not really difficult or to be more precise, the way of how we have designed our system will automatically lead to a time delay at the level of the exchange. The scenario which we have illustrated in the analysis, i.e. a lot of orders pile up in the order queue, overloading the exchange, will eventually occur. This has to do with how SPiM picks the channels. Basically it is a two steps procedure:

1) Pick a channel name out of all the unguarded channels, whereas unguarded means that there is not another channel in front of the channel, i.e. the output channel !x();Process() is unguarded, however, if we put the output channel !b() in front of this statement, i.e. !b();!x;Process(), !x() would now be guarded by !b().

2) Match an input- with an output-version of the chosen channel name.

While the second one is conducted randomly by the built-in procedure of SPiM i.e. all input-output combinations are equally likely to get executed, the first step is different. For picking the actual channel name SPiM uses a so called activity function, which basically looks like this (the real implementation is based on lists):

$$\square\square\square\square = \square\square\square * \square\square\square\square - \square\square\square\square \qquad (4)$$

Where Act_x stands for the activity on channel x, In_x and Out_x are the unguarded input and output instances of channel x and Mix_x are all the input-output combinations which cannot be executed, i.e. they are excluding each other, for example, when a choice process has at the beginning of the do-branch an input version of the channel x and on one of the or-branches an output version of the same channel, they would be counted by In and Out, although they cannot interact with each other, Mix takes this situation into account. The channel activity is directly related to the likelihood that a channel gets picked, i.e. the more possible input-output combinations, the more likely it is that the channel name gets picked by SPiM.

Regarding our model, there is only one order evaluation process, i.e. one dequeueRequest-channel (Activity = 1), while many market makers and other traders try to enqueue an order to the order queue (Activity of enqueue-channel > 1). Therefore the likelihood that an order is dequeued from the order queue is lower than the probability that one market maker or other trader enqueues an order, i.e. the queue grows.

However, there is even more to take into consideration, because when the order evaluation forwards the order to the order book it uses another channel which has only one input-output combination, i.e. the likelihood that the order gets forwarded is again lower than the likelihood to enqueue an order and so on. With many market makers or other traders the exchange could be even trapped in a deadlock, i.e. it never gets the permission to retrieve an order from the order queue.

Therefore it is required to slow down the other market participants or increase the likelihood for the exchange. With the Connector-Process we provide one solution to the problem, i.e. this approach reduces the number of input-output combinations of the enqueue-channel, because this puts a channel in front of the output-version of the enqueue-channel, therefore the channel is no longer unguarded, and therefore improves the likelihood of the order evaluation to dequeue an order. Further ways to mitigate the problem would include using delay-actions on the side of the market makers and the traders or by just forcing SPiM after a certain amount of enqueued orders to execute one of the channels with less activity (could be realized by putting a counter into the order queue, when the counter hits a certain threshold, the order

queue does not read from the enqueue-channel and therefore the activity of this channel would drop to zero). However, this shows that further research has to be done on how to calibrate the model parameters accurately.

Nevertheless, as it is natural for the system that a time delay occurs, how did we manage to get a system without any time delay? The solution is to make the system almost completely deterministic, this can be achieved by reducing the activity of all the unguarded channels to zero i.e. we have to put another channel in front of them, such that they are guarded. Except for the one channel we wish to execute, to achieve this in our model, we have to change the dequeueRequest-channel to an issueOrder-channel for the other market participants, i.e. this is the only random part which remains, at least from the point of view of the main connections between the different agents, the behaviour within the agents can still be random dependent on how the algorithms are implemented. When one of the agents has been chosen, he does all the relevant steps to issue, because the order evaluation is waiting for an order to come in, no one writes to the issueOrder-channel and therefore no other market participant can create and issue an order in parallel to the firstly chosen agent. When the order evaluation has received the order, the chosen agent recreates itself and the order evaluation forwards the order to the order book (or rejects it). The obconfirm-channel prevents the order evaluation from immediate recreation, i.e. as long as the order has not been processed and the order books are back in the Orderbook-Process, no one writes to the issueOrder-channel and therefore none of the agents creates a new order. Eventually the order evaluation gets recreated and the next agent is chosen to create an order[16].

[16] Some of the notification channels, for sending messages, from the channels need also to be adjusted because some of them are implemented in parallel to the return of the order books, i.e. the agent could be chosen by the order evaluation again before the agent has been notified that, for example, one of its orders has been matched.

6 Testing

Because the SPiM has a stochastic nature and the simple output of data is not possible (the built-in print-function leads to an error), it is not very easy to test software for the SPiM. Nevertheless, to check the validity of the code I have used a dynamic testing approach, which was mainly based on a Blackbox-method.

6.1 Test Processes

To circumvent the drawbacks of SPiM when it comes to testing, I have created two additional processes.

6.1.1 The Error-Process

The task of the Error-Process is to substitute the print-function of SPiM in case of an error. The concept of the process is quite simple: Whenever something happens within the simulation, which should not happen, an Error-Process is created. To identify, which of the various positions where an Error-Process can be created, has eventually caused the creation, each Error-Process contains a String as parameter.

6.1.2 The Result-Process

As the name already implies, the Result-Process is used to present results. Whenever the developer wants to get the value of a variable, s/he can implement the creation of the Result-Process in parallel (when the simulation continues) or as a final execution. As the process takes in a parameter of polymorphic type, it can also take in a String, i.e. replace the Error-Process, however, the reason for this lexical and functional difference between these two testing processes is that this approach simplifies the error management in more sophisticated simulations.

When the developer has to deal with numerous Result-Processes in his/her code, s/he might consider using a 2-tuple as parameter, where the first element indicates the position and the second element the actual value.

6.2 Testing approach

To test our system we have mainly relied on a functional testing approach, i.e. we have fed our subsystems with different inputs and then compared the result with the expected outcome (compare Patton, 2005). The reason for this is quite simple. For our work it is important that the different modules behave correctly, these modules consist of different processes which interact with each other. It does not really make

sense to test each of these processes individually. Even if we know that each process works perfectly, we cannot infer that the whole subsystems behave how it should behave and therefore we have focused on a functional testing approach on the top processes.

6.3 Test Case

Our general approach was comparable to a standard functional testing approach (compare Patton, 2005).

1) Check the specification: What should the module do?
2) Create input: How can we test the specification?
3) Specify the output: What can we expect from this input?
4) Execute the system with the input: What do we really get?
5) Compare expected with actual output: Everything alright?

For example, the Adder was tested by setting up a process which continuously forced the Adder to add orders to a book and the final book was then checked against another book, which we have created with an Adder implementation for Amanda. A more detailed example test case, with a slightly different approach, can be seen in section 6.4.

6.4 Example Test Case – Pseudo Random Number Generator

This test case shows how we have proven that the pseudo random number generator (PNG) functions properly.

Our approach was to isolate the PNG from other components and then let it produce random numbers. Figure 50 shows how we have conducted this approach. We have set up a Test-Process which takes in an integer value x and a list y[17]. In the case that the integer value is non-zero, the Test-Process reads a random number created by the PNG from the unifRnd-channel and then recreates itself, while decrementing the integer value and concatenating the random number to the list. When the integer value drops to zero, a Result-Process with the list as a parameter is created and we can see the final list. However, a list of random numbers alone is usually not really

[17] SPiM tends to have a problem, when a type for a list is defined and then an empty list is forwarded (syntax error), therefore we are using the polymorphic type here.

helpful, aside from cases where the list would contain the same number again and again or the number would not be in the range between zero and one. Therefore we have also realized the algorithm with an Excel-spreadsheet and created 1000 random numbers, after testing that the results of the algorithm are uniformly distributed (see Implementation Chapter for more), we have matched the results of the SPiM-Implementation with the results of the Excel-spreadsheet. Due to the fact that the results were equal on the first 50 numbers and due to the highly deterministic nature of how the AS183 algorithm creates random numbers, we have inferred that the PNG functions properly.

```
new c1:chan

let Result(x:'o) = !c1()

let Test(x:int,y:list('o)) = (
    if x = 0 then
        Result(y)
    else
        ?unifRnd(u); Test(x-1,u::y)
)
```

Figure 50: Testing environment for the pseudo random number generator.

7 Conclusion

In this project, we have used the Stochastic Pi-Machine (SPiM) to model a financial system for the first time.

We have successfully demonstrated how SPiM can be used to model the behaviour of the different market participants and have also provided a standard model kit. This model kit can be used as a foundation by financial researches to create their own simulations in the SPiM environment.

Furthermore, we have also improved the usability of SPiM by providing tools which tackle some of the major shortcomings of SPiM. We have developed a library which allows users to manipulate lists more efficiently, we are also providing a pseudo random number generator, which allows the user to incorporate random numbers with different distributions in his models, and we have also solved the problem of how to display integer values in the built-in plotting window of SPiM, which is another important achievement. Future researchers can use these instruments to efficiently

modify or extend our standard framework to get new insights in the nature of modern markets.

Regarding SPiM, although this work shows that SPiM can be used to model a financial system it is still questionable, whether SPiM will become a popular tool in this area, the whole system could have been realized with another programming language as well and compared to common functional programming languages, which use syntax similar to SPiM, it is still very inconvenient to work with SPiM. For example, Pattern-matching does not support the matching of lists within tuples or other lists, i.e. more than one match-case-construct is required, the wildcard operator and the built-in print-function do not work and also the performance must be improved, because at the moment it can take seconds to create one normally distributed random number. Another important point is the problem with the restricted channels, as mentioned in the Implementation Chapter, SPiM can lose restricted channels which makes it hard to test a simulation. Does the system simulation not work because has process has been incorrectly implemented, or does it not work because SPiM has lost one or another restricted channel, leaving the system in a deadlock. However, should Microsoft Research fix these problems and also release a way to statically analyse the SPiM-Code (which is planned) the research potential of SPiM could increase substantially and so would the significance of this work.

Concerning further development of this work, the next steps to improve this work, would need to focus on the calibration of the individual modules, such that their behaviour is accurately when interacting with other modules, for example, the potential starvation problem of the exchange, which we have mentioned in the Implementation Chapter. Of particular interest, and closely related to the calibration of the system, is also the implementation of further delays. While we have already implemented the latency of the different market participants when they are issuing orders, our model does not incorporate any other deliberately implemented delay[18], like the potential delay on the message site, i.e. what happens if the trade confirmation messages are delayed[19].

[18] The overall delay of the system is a built-in feature of SPiM.
[19] Which was the original suggestion of Clack (2012).

Bibliography

Avellaneda, M. & Stoikov S. (2007), *High-frequency trading in a limit order book.* [online] Available at: <http://www.finance-concepts.com/images/fc/HighFrequencyTrading.pdf> [Accessed 16 June 2012]

BATS BZX (2012), *Exchange Fee Schedule 2012*. [online] Available at: <http://cdn.batstrading.com/resources/regulation/rule_book/BATS-Exchanges_Fee_Schedules.pdf> [Accessed 26 August 2012]

Cardelli, L. & Phillips, A. (2004), *A Correct Abstract Machine for the Stochastic Pi-calculus.* [online] Cambridge, UK: Microsoft Research. Available at: <http://research.microsoft.com/apps/pubs/default.aspx?id=65228> [Accessed 2 July 2012]

Clack, C. (2012), *The Instability of Inventory-Driven Computer Trading Algorithms*. Working Paper, UCL

CME Group (n.d.), Client Systems Wiki – Matching Algorithm. [online] Chicago, USA. Available at: <http://www.cmegroup.com/confluence/display/EPICSANDBOX/Match+Algorithms> [Accessed 11 August 2012]

CME Group (n.d.), Globex Reference Guide. [online] Chicago, USA. Available at: <http://www.cmegroup.com/globex/files/GlobexRefGd.pdf> [Accessed 11 August 2012]

Field, A. (2009), *Discovering Statistics using SPSS*. Second Edition, Second Reprint, Los Angeles, Sage Publications

Hill, I. D. & Wichmann, B. A. (1982), Algorithm AS 183: An Efficient and Portable Pseudo-Random Number Generator. *Applied Statistics*, 31 (2), 188-190

Kirilenko, A., Kyle, S., Samadi, M. & Tuzun, T. (2011), The Flash Crash: The Impact of High Frequency Trading on an Electronic Market. [online] Available at SSRN: <http://ssrn.com/abstract=1686004> or <http://dx.doi.org/10.2139/ssrn.1686004>

Lafore, R. (2003), Data Structures and Algorithms in JAVA. Second Edition, Indianapolis, Sams Publishing

Matsumoto, M. and Nishimura, T. (1998), Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generation*.

Microsoft Research (n.d.), *Information text about SPiM*. [online] Cambridge, UK: Microsoft Research. Available at: < http://research.microsoft.com/en-us/projects/spim/>

Milner, R. (1999), *Communicating and Mobile Systems: The Pi Calculus*. First Edition, *Cambridge University Press*

Patton, R. (2005), *Software Testing*. Second Edition, Indianapolis, Sams Publishing

Phillips, A. (2007), *The SPiM Language*. [online] Cambridge, UK: Microsoft Research. Available at: <http://research.microsoft.com/en-us/projects/spim/language.pdf> [Accessed 4 July 2012]

Pressman, R (2010), *Software Engineering*. Seventh Edition, Singapore, McGraw Hill Higher Education.

Serguieva, A (2011), *Software Engineering: Modelling*. [Lecture] University College London.

Tett, G. (2011), Flash crash threatens to return with a vengeance. [online] Financial Times. Available at: <http://www.ft.com/cms/s/0/b008c4c4-3226-11e1-b4ba-00144feabdc0.html#axzz24yfP3WYD> [Accessed 4 July 2012]

Wilmott, P. (2007), *Paul Wilmott Introduces Quantitative Finance*. Second Edition, West Sussex, John Wiley & Sons Ltd