Towards a Formal Modelling of Order-driven Trading Systems using Petri Nets: A Multi-Agent Approach*

Julio C. Carrasquel[1], Irina A. Lomazova[1], and Iosif L. Itkin[2]

[1] National Research University Higher School of Economics,
Myasnitskaya ul. 20, 101000 Moscow, Russia
jcarrasquel@hse.ru, ilomazova@hse.ru
[2] Exactpro Systems
iosif.itkin@exactprosystems.com

Abstract. Electronic trading systems provide the computational support for stock exchanges. Liquid markets use order-driven systems, i.e., where client requests, for trading financial instruments, are served through individual orders. This paper presents Petri net models assembling some crucial processes executed within order-driven systems such as orders submission, application of precedence rules, and the order matching mechanism. Such processes were modelled as types of agents running in a multi-agent system (MAS) using nested Petri nets (NP-nets) - a convenient formalism for modelling MAS. With NP-nets, we focus on the control-flow perspective (causal dependence between activities executed by agents) and in the synchronization between agents. Conversely, we have used coloured Petri nets to extend the model including orders as objects with attributes. Thus, this work with Petri nets represents an experimental & initial research phase to validate trading systems using related methods such as process mining, simulations and model checking.

Keywords: Stock trading systems, order-driven systems, Petri nets, multi-agent systems, nested Petri nets, coloured Petri nets.

1 Introduction

Stock exchanges have been historically the forum where market participants trade financial instruments, i.e., shares of a company. Brokers, with access to the stock exchanges, provide the intermediary services to clients for trading. The core of stock exchanges lies in stock trading systems - software solutions supporting all the processes executed to perform trades. In this research, we focus on order-driven systems [4], used by most of the liquid markets, where client's requests for trading are managed as orders; order-driven systems receive

[1] This work is supported by the Basic Research Program at the National Research University Higher School of Economics.

* Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

orders from participants, rank/place orders in order books, and perform trades, among other operations.

Nowadays an important part of the trading volume of stock exchanges have been shifted to order-driven electronic systems. As their trading volume growths, with many participants and processes involved, it is a task of utmost importance the validation of the trading system's correctness. i.e. auditing of the system's processes, performance analysis, verification of its properties, among other tasks. For such tasks, we consider Petri nets [7] [8] - a consolidated formalism for modelling and validating distributed and concurrent systems. With Petri nets, we are able to formally describe the concurrent processes which are executed within order-driven trading systems. Petri net models are an input for validating the system using process mining [9], simulations, and model checking verification [3], among other methods.

This paper presents an exploratory research work where we assemble as Petri nets some important processes executed within order-driven systems. We focus on the modelling of the processes of orders submission, execution of order precedence rules, and the matching mechanism. We model the order-driven system as a multi-agent system (MAS), where agents are conceived as running instances of the mentioned processes, and these execute specific tasks within the processing of orders. This approach represents a theoretical model and do not necessarily reflects real life technical implementations of electronic stock trading systems. The main difference is the sequencing of events into a single thread in the latter. In this work, we use two classes of Petri nets. We present a first model based on nested Petri nets (NP-nets) [6] - an extension of Petri nets where tokens can be Petri nets themselves, thereby making them suitable for modelling MAS. With NP-nets, we focus on the control-flow perspective (causal dependence between activities executed by the system and agents) and in the synchronization between agents. As a second exercise, we provide a model based on coloured Petri nets (CPN) [5] - an extension of Petri nets where tokens have data types (*colors*) associated. Tokens with *colors* assigned allow to model orders as objects with data attributes; the latter allows to impose additional restrictions on the execution of the system's activities according to the order attributes.

The rest of this paper is organized as follows. Section 2 introduces some basic concepts of order-driven trading systems. Section 3 describes the NP-net and CPN models developed. Section 4 presents some conclusions and introduces future research work.

2 Order-driven Trading Systems

Order-driven systems handle requests from clients for trading financial instruments as orders. At the core they contain rule-based processing that stores orders into the order books, applies precedence rules to prioritize orders, and matching mechanisms to perform automated trades between orders. The core is surrounded by order-routing systems, which deliver the orders from the clients and order presentation systems and market data systems send reports to clients about orders and trades. In this paper, we focus on the processes of submitting orders into an order book, and the activities of *rule-based matching systems* including the order precedence rules and the matching procedure.

2.1 Orders

An order, which we denoted as o, is a client instruction to trade a instrument, i.e., shares of a specific company. Among its attributes, an order o has a side $s \in \{buy, sell\}$ indicating whether the client wants to *buy* or *sell*, a price per stock p; and a quantity of stocks q to trade. Orders also include other constraints regarding in which terms the client wants to trade.

Order states. An order may have the following states. An order is *submitted* when it has been received by the system. The system verifies if the order is valid; if so, the order eventually changes its state to *placed* in an order book; otherwise, it is *rejected*. If an order o_1 is traded with another order o_2, then o_1 is *filled* if $q_1 \leq q_2$ where q_1 and q_2 are the stocks quantities of o_1 and o_2 to be traded; otherwise, o_1 is *partially filled* waiting to trade its remainder $q_1{}' = q_1 - q_2$ with other orders. An order also may be *replaced*, *canceled* or *expired*.

Types of orders. In this paper, we focus on two kind of orders whose use is very common, market orders and limit orders. Market orders aim to trade at the best price available, i.e, a client placing a buy market order does not specify a price, so he is willing to buy at the best price available that sellers ask. Instead, limit orders trade at a final price p_{tr} not worse than a (limit) price p, i.e., buy limit orders trade iff $p_{tr} \leq p$, whereas sell limit orders trade iff $p_{tr} \geq p$. We refer to [4] for other types of orders.

2.2 Precedence rules

Order-driven systems use *order precedence rules* to separately rank buy and sell orders. Orders with highest precedence are served first. The system ranks orders using a primary rule which usually is the price. Buy orders with higher prices, and sell orders with lower prices are ranked first in their sides. Market orders always rank highest (their prices are not limited). If two or more orders have the same price, then it is applied a secondary rule; in this work, we consider time as the secondary rule - if two orders have the same price, it will be served the first one submitted in the system. With price and time as our precedence rules, we considered a *price-time policy* for serving orders.

As an example, tables 1(a) and 1(c) show buy and sell orders received by the system. Each row in tables 1(a) and 1(c) represents an order o with a submitted time, its trader, size (quantity of stocks) and price per stock. As each order arrives, these are placed and ranked in the order book based on the *price-time policy.* Table 1(b) shows the order book state after all listed buy and sell orders have been ranked and inserted.

In Table 1(b), buy orders are served from the top to the bottom. Bif's order ranks highest (highest buy price), while Bud's order is ranked last (lowest buy price). Orders of Bea and Ben have the same price, but Bea's order is ranked first since her order arrived before. Sell orders are served from the bottom to the top. Sol's order ranks first (lowest sell price), while Stu is ranked last (highest sell price). Notice that the order book presented in Table 1(b) is a simplified and abstracted version w.r.t to how order books may be presented in real systems.

Table 1: Submitted buy (a) and sell (b) orders, ranked in the order book.

(a) received buy orders

time	trader	size	price
10:01	Bea	3	20
10:08	Ben	2	20
10:15	Bif	4	market
10:18	Bob	2	20.1
10:29	Bud	7	19.8

(b) order book

buyer	size	price	size	seller
Bif	4	market		
		20.1	5	Stu
Bob	2	20.1	2	Sam
Bea	3	20.0		
Ben	2	20.0		
		20.0	6	Sue
Bud	7	19.8	1	Sol

(c) received sell orders

time	trader	size	price
10:05	Sam	2	20.1
10:08	Sol	1	19.8
10:10	Stu	5	20.2
10:20	Sue	6	20.0

2.3 Order matching procedure

The matching mechanism matches highest-ranked buy and sell orders. If the buy order price is greater or equal than the sell order price (the buyer is willing to pay at least what the seller demands) then a match is executed between the two orders. If one order is smaller than the other, the smaller order is filled, and discarded from the order book. The remainder of the largest order is placed in the order book. Such order still ranks highest in its side, so the system will attempt to match it against the next highest-ranking order on the other side of the order book. If two matched orders have the same size, both are filled completely. This process continues as long as the next highest-ranked buy order's price is greater or equal than the next highest-ranked sell order's price. From the example of table 1(b), matchings are executed as follows:

1. Sol's order to sell 1 at 19.8 with Bif's market order to buy 4. Sol's order is filled, and Bifs order is partially filled with a remainder of 3.
2. Bif's remainder order to buy 3 with Sue's order to sell 6 at 20.0. Bifs order is filled, and Sues order is partially filled with a remainder of 3.
3. Sue's remainder order of 3 with Bob's order to buy 2 for 20.1. Bob's order is filled, and Sue's order is partially filled with a remainder of 1.
4. Sue's remainder order of 1 with Bea's order to buy 3 for 20.0. Sue's order is filled, and Bea's order is partially filled with a remainder of 2. Table 2 shows the order book after execution of all possible matchings. Then, the process cannot execute matchings since now the highest-ranked buy order is less than the highest-ranked sell order.

3 Modelling Order-driven trading systems

Distinct sub-processes with a predefined set of tasks may operate concurrently in an order-driven system. Each sub-process may be seen as a kind of agent, which

Table 2: Order book last state (a) and trades (b) after executing matchings.

(a) order book last state

buyer	size	price	size	seller
		20.1	5	Stu
		20.1	2	Sam
Bea	2	20.0		
Ben	2	20.0		
Bud	7	19.8		

(b) trade summary

match	seller	buyer	quantity
1	Sol	Bif	1
2	Sue	Bif	3
3	Sue	Bob	2
4	Sue	Bea	1

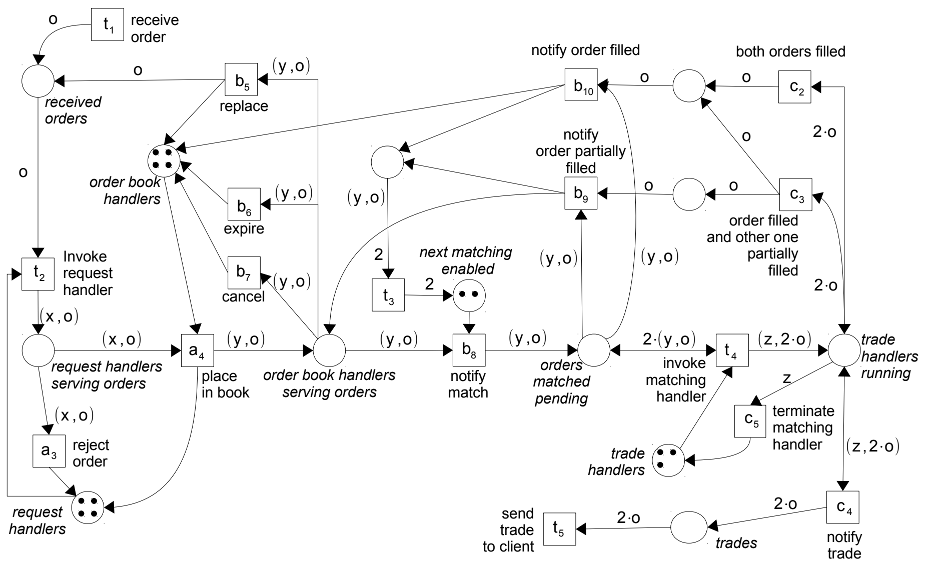
is instantiated upon request, it executes its tasks independently, it sends/receives messages from other agents, and eventually it is terminated upon completion of its tasks. Thus, we model using Petri nets some major components of an order-driven system under a multi-agent system (MAS) approach.

The notion of agents has become an important concept in the software engineering and artificial intelligence (AI) domains, i.e., a MAS approach provides a convenient degree of modularity, and it allows to understand the communication points between sub-processes, which now we treat as agents. This section describes the kind of agents we devised, and the MAS-based nested Petri net (NP-nets) and coloured Petri net (CPN) models. In the following, it is assumed that the reader has a basic understanding of NP-nets and CPNs; we refer to [6] [5] for basic concepts on NP-nets and CPNs.

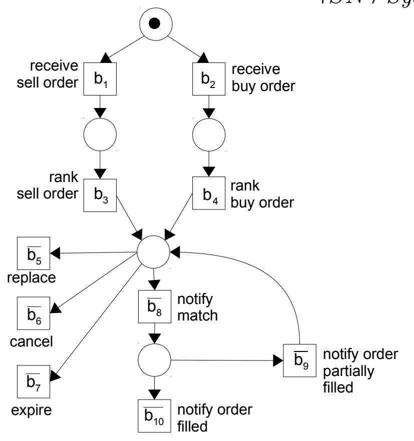
3.1 Agents

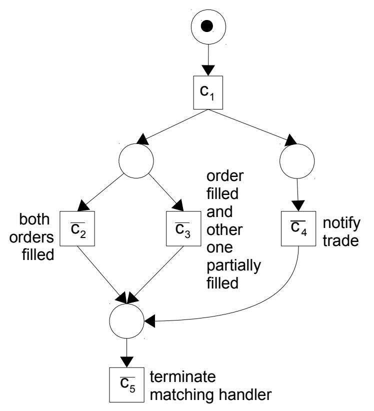
We have focused in the orders submission into the system, insertion and ranking on the order book, and the matching mechanism. Thus, we conceive a MAS model with the following kind of agents:

- (x) *Request handler agent.* It handles an order submission into the system. It verifies whether or not the order is valid. If is valid, such order is forwarded to what we denote as an *order book handler* agent. Otherwise, the order is rejected and discarded.
- (y) *Order book handler agent.* It is initiated upon a call from a *request handler agent.* It places and rank the received order in the proper order book side. When a matching is executed, against other order handled by other agent, it takes the order from its respective order book side, and it sends the order to a *trade handler agent.* Eventually, this agent receives a response indicating whether or not its order was filled. If so, the agent terminates. If not, it keeps handling the partially filled order that keeps stored in the order book. The agent also handles the replace, cancel or expire events of an order.
- (z) *Trade handler agent.* It is initiate upon the reception of two orders, a buy and sell order, that have matched. This agent sends the trade to the clients. In addition, it checks if the orders are filled. If they have been partially filled, the agent places their remainders in the order book. Finally, it sends back to the *order book handler agents* which handle the buy and sell orders, informing whether or not the orders they handle are filled or not.

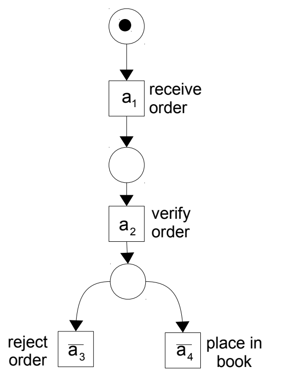


(SN) System net




c_1
both orders filled $\overline{c_2}$
$\overline{c_3}$ order filled and other one partially filled
$\overline{c_4}$ notify trade
$\overline{c_5}$ terminate matching handler

(EN_y) *Order book handler agent y* (EN_z) *Trade handler agent z*



(EN_x) *Request handler agent x*

Fig. 1: NP-net model for the order-driven system with the *system net* and *agents*

3.2 Nested Petri Net Model

The NP-net model is depicted in Fig. 1 showing the *system net SN*, and the *element nets EN_x*, EN_y, and EN_z, which describe the activities of agents of type x, y and z. The places *request handlers*, *order book handlers*, and *trade handlers* contain black dots indicating respectively the number of agents (system resources) of type x, y, and z available to be instantiated.

A submitted order o is produced in the place *received order* when the transition t_1 (receive order) fires. If there is an available resource at the *request handlers* place, the transition t_2 (invoke request handler) fires: it consumes a black dot and a variable o, and produces a pair (x, o) at the place *request handlers serving orders* meaning that an agent x will handle an order o. Agent x may execute its internal transitions, described in EN_x, for verifying the validity of order o. If the order o is not valid, it is performed a *vertical synchronization step* [6] involving the firing of transitions a_3 and $\overline{a_3}$ (reject order). Thus, the agent x and the current order o disappears, and a black dot is produced at the *request handlers* place indicating that an agent resource has been released. If the order o was valid, and there is an available resource at the *order book handlers* place, then it is performed other vertical synchronization step with transitions a_4 and $\overline{a_4}$ (place in book); thus, agent x disappears, but it is produced a pair (y, o) in the place *order book handlers serving orders* - it indicates that agent y (*order book handler*) will handle now the order o.

As depicted by the net EN_y, an agent y may fire transitions b_1 (receive sell order) and b_3 (rank sell order) if o is a sell order; otherwise, if fires b_1 (receive buy order) and b_4 (rank buy order). Transitions b_3 and b_4 represent the activites of placing and ranking o in its corresponding order book side. Later, either one among of the activities $\overline{b_5}$ (replace order), $\overline{b_6}$ (expire order), $\overline{b_7}$ (cancel order) may be executed (thus, terminating agent y) or transition $\overline{b_8}$ (notify match) is fired; if so, in the system net SN, the current agent y and the order o, represented as (y, o) will be transferred to the *orders matched pending* place.

Two tokens of type (y, o) at the place *orders matched pending* represent two orders (a buy and a sell order) that are matched. When transition t_4 (*invoke matching handler*) fires, it consumes such couple of pairs (y, o), and a black dot from the *trade handlers* place; then, t_4 produces a new pair $(z, 2 \cdot o)$ in the place *trade handlers running* - it means that an agent z will handle the matching of the buy and sell order. Notice that t_4 also produces back the two tokens of type (y, o); the latter means that the two agents of type y will wait for a response to be notified if the orders that they manage are filled.

The behavior of agent z is described by EN_z. It executes transition $\overline{c_4}$ to send a message to the clients about the trade performed. It also executes $\overline{c_2}$ if the orders have been filled or $\overline{c_3}$ if one of the two orders is just partially filled; the latter is done to inform the previous two agents of type y, that wait for such response at the *orders matching matching* place. When receiving such response, each of the two agents of type y will be able to fire either b_9 (notify order partially filled) or b_{10} (notify order filled) according to the case. If b_9 is fired, then the pair (y, o) will be transferred from the *order matched pending* to the *order book handlers serving orders* place, indicating that y will attempt to eventually match the remainder of order o; otherwise, if b_{10} is fired, the pair (y, o) simply disappears indicating the completion of y, and producing a black dot at the *orders book handlers* place. Notice that the place *next matching enabled* constrains the execution of transition b_8 (notify match) to at most two times in a round (for a buy and a sell order), and no other matching can be executed until an agent z gives back the previously explained response to the two agents y handling the buy and sell order. The motivation of this is to give highest priority to be matched again to the orders processed by agent z (and that rank highest in their sides of the order book) in case one of them were just partially filled and transferred back to the order book.

3.3 Coloured Petri Net Model

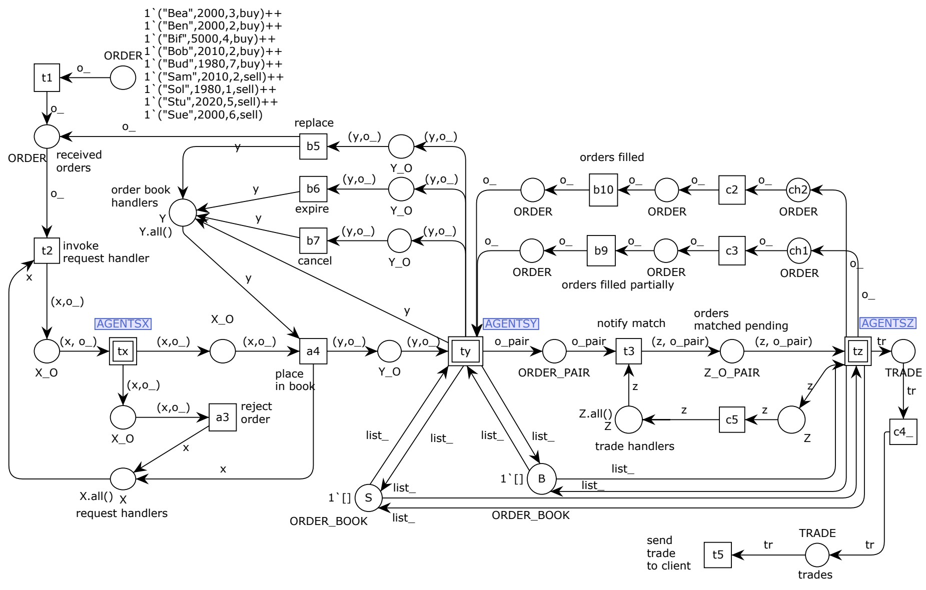


Fig. 2: The *system net* or top page of the CPN model for the order-driven system.

As a second exercise, we developed a CPN model, developed with CPN Tools [1]; it is a hierarchical net organized in a set of *pages* where the *top page* models the *system net* while *sub-pages* emulate the behavior of agents x, y and z. Due to space limitations we just show the *system net* (Fig. 2) and the net for agent y (Fig. 3). The complete CPN model is available via [2]. The *system net* maintains a similar structure w.r.t the NP-net. CPN extends the model assigning data types (colors) to tokens; thus, tokens become objects with attributes. In addition, declarations with CPN ML [5] provide definitions for *color sets* and variables of the model, i.e, variables `i`, `p`, `q`, `s` stand for the name (identifier) i, price p, size q, and side $s \in \{buy, sell\}$ of an order. An order is token assigned with the color set `ORDER`.

```
colorset SIDE = with buy | sell;
var i, i2: STRING;
var q, q2: INT;
var s, s2: SIDE;
colset ORDER = product STRING * INT * INT * SIDE;
var o_: ORDER;
```

Order book. An order book (from the color set `ORDER_BOOK`) is a sorted list $l = \{o_1, o_2, ..., o_n\}$ such that the first element o_1 of l is the highest ranked order, and the last element o_n is the lowest ranked order. Thus, for a specific financial instrument, we devised to model an order book as two sorted lists l_S and l_B, such that l_B stores buy orders, and l_S stores sell orders. The *system net* of the CPN depicted in Fig. 2 shows places B and S - tokens in B and S are sorted lists of the color set `ORDER_BOOK`. The place B stores lists of buy orders, and S stores lists of sell orders. Since it is assumed to work with a single financial instrument, the initial marking of S and B are unique empty lists (denoted as $[\,]$). If we were working with m financial instruments, then there would be m lists in each of these places.

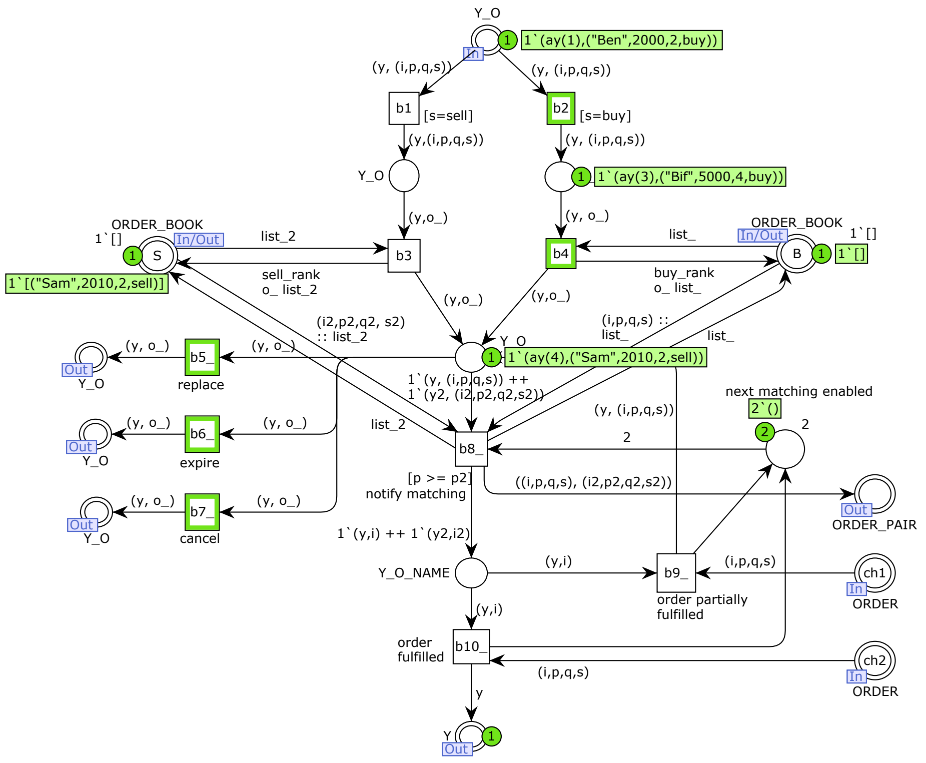


Fig. 3: CPN *sub-page* for execution of agents of type y (*order book handlers*).

Agents. CPNs do not follow the *nets-within-nets* paradigm as NP-nets do, i.e., tokens cannot be Petri nets themselves in CPN. Hence, CPNs become less suitable to model multi-agent systems (MAS). As a way to model MAS in CPN, we have devised the use of a *hierarchical* CPN model in the following way: the *system net* is the *top page* of the CPN model, whereas the internal net structure of agents x, y, and z, described in Fig. 1, have been modelled as *sub-pages* in the CPN model. The *sub-pages* with the tasks performed by agents x, y, and z are linked with the *top page* by *substitution transitions* `tx`, `ty`, and `tz`. For instance, Fig. 3 shows the *sub-page* for agents of type y where a few agents are handling orders, i.e, agent `ay(1)` is about to process Ben's order; agent `ay(3)` is waiting to execute b_4 for ranking Bif's order, and agent `ay(4)` already placed Sam's order in the order book, so its order eventually may be matched, or its order may be replaced, expired, or canceled. Hence, agents are emulated by object tokens which navigate through a shared structure which is the model *sub-page*. Note also that each *sub-page* is connected by input and output channels with places of the *system net* using the *ports-sockets mechanism* provided by hierarchical CPNs. Through these channels, an agent into the *sub-page* for agents y (Fig. 3) is able to place an order into a list of the places B and S (an order book side), it is notified if an order has been filled, among other communication points. This principle is the same for agents of type x and z.

Order precedence rules. Arc inscriptions of the used CPN model are allowed to have functions as expressions. Thus, we were able to model order precedence rules (using a *price-time policy*) on the lists stored in places S and B, which represent the buy and sell sides of an order book. As depicted by Fig. 3, when an agent y fires b_4 (rank buy order) it is consumed the order `o_` that the agent y is handling, and the list of sell orders `list_`; as a result, the transition produces in place B a *reorganized* list executing the function `buy_rank (o_, list_)` which has been implemented as follows:

```
fun buy_rank (i,p,q,s) [] = [(i,p,q,s)] |
buy_rank (i,p,q,s) ((i2,p2,q2,s2)::list_) =
if p > p2 then
  (i,p,q,s)::(i,2,q2,s2)::list_
else
  (i2,p2,q2,s2)::(buy_rank (i,p,q,s) list_)
```

The function first checks the case if `list_` is empty; if so, it returns the list with the single order `(i,p,q,s)`; otherwise, it compares the order `(i,p,q,s)` with the first element of the list, `(i2,p2,q2,s2)` (the operator `::` stands for concatenation). If the price `p` of the new order is strictly greater (>) than the price `p2` of the first order in the list, it is returned a new list where the new order `(i,p,q,s)` is placed first since now it is the order which ranks highest; otherwise, `(i2,p2,q2,s2)` is maintained as the highest ranked order, and a recursive call of `buy_rank (i,p,q,s), list_` is done with `list_` as the rest of the list without the first element we already compared. This principle is applied in the function `sell_rank` using the operator *strictly less than* (<) since sell orders whose price is lowest are ranked first. Note that albeit *time semantics* yet are not used in our work, the use of strict operators, $<$ and $>$, allow to give precedence to orders who were submitted before.

Guards. Boolean expressions known as *guards* [5] are used as additional constraints for a transition to be enabled, i.e., in Fig. 3, transition `b8_` (notify matching) will be enabled if $\texttt{p} \geq \texttt{p2}$ is evaluated to true, i.e., if the price `p` of the highest ranked buy order is greater or equal than the price `p2` of the highest ranked sell order. This shows on how the CPN allows to impose additional constraints on the execution of an activity based on the data perspective, rather than just based on the causal dependence between activities.

4 Conclusions and Future Work

In this paper we have developed experimental Petri net models for some components of order-driven trading systems: orders submission, ranking of orders in an order book, and the matching mechanism. We conceived a multi-agent system (MAS), where each process of the system has been devised as a kind of agent. We aim to focus on nested Petri nets (NP-nets), where tokens can be Petri net themselves, so we have shown how its semantics allow to suitably model MAS. We focused on the control-flow perspective, i.e., causal dependence between activities executed by the system net and the agents, and on their synchronization points, for instance, taking advantage of *vertical synchronization steps.* Notice that a MAS model is capable to respect the sequential priority execution of order matchings in an order book, i.e., in Fig. 1, when two agents y_1 and y_2 with orders o_1 and o_2 respectively are matched, then no other orders can *match* until the trade (o_1, o_2) is handled completely by an agent z.

We also designed a CPN model to express attributes of orders; this allowed to model explicitly the use of precedence rules over orders. Guards, logic expressions, on transitions, are a way to impose additional constraints on the execution of an activity based on the order attributes. Future research will address the necessary formal definitions for our MAS models. The use of Petri nets describing these processes of a trading system is a milestone in our research. Based on such models, Fig. 4 shows some validation tasks which are matter of future research:

Conformance checking [10]. This method, from the process mining field [9], aims to compare how aligned is a model (describing the system's expected behavior) with respect to real *event logs* (describing the real behavior). Hence, for this task we will take real-life event logs from a specific electronic trading system. Thus, we may find deviations which are either desirable (handling unforeseen, but valid circumstances) or undesirable (fraud, inefficiencies). Several metrics and other methods have been proposed to measure the deviation between a specification model and the traces seen in the log; however, the use of MAS on our study case opens an interesting problem on how to apply conformance checking on MAS-oriented models, i.e. in *nets-within-nets.*

Simulation and performance analysis. Simulation helps to identify errors on the system's execution, and to carry out a performance analysis. For instance, if we include *time semantics* on the Petri net models, we are able through simulations to identify bottlenecks, i.e., waiting time for orders to be served by an

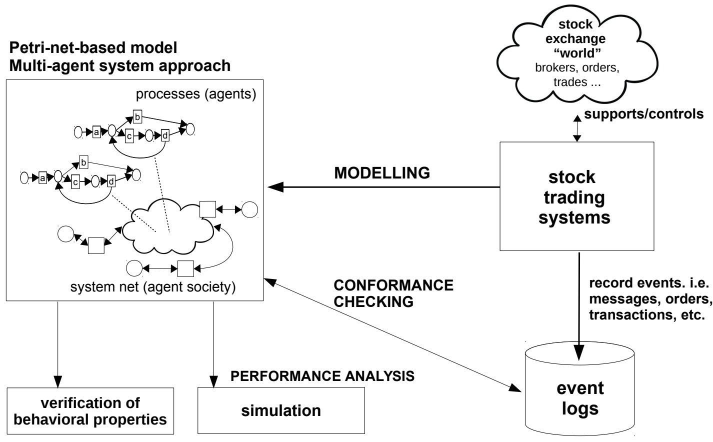

Petri-net-based model
Multi-agent system approach
processes (agents)
system net (agent society)
stock exchange "world" brokers, orders, trades ...
supports/controls
MODELLING
stock trading systems
CONFORMANCE CHECKING
record events. i.e. messages, orders, transactions, etc.
PERFORMANCE ANALYSIS
verification of behavioral properties
simulation
event logs

Fig. 4: Resarch approach for the overall validation of trading systems.

agent, time needed for an agent to complete its activities, etc. *Time semantics* would allow to model other crucial concepts of trading, such as *trading sessions* and *orders validity and expiration* that we did not tackle in this work. We may use well-known tools for Petri nets, such as CPN Tools, which execute Petri net models, and allow software extensions; thus, we may develop an *order book interface*, whose orders and state is affected by the execution of a Petri net.

Verification of behavioral properties. Exhaustive and automated formal verification of the functional and non-functional properties of the system by means of the *model checking* approach [3]. For instance, we may state properties in *temporal logic*, and to check based on *state space exploration* whether or not the model (specification of the trading system) satisfies these properties.

References

1. CPN Tools. `https://www.cpntools.org`.
2. J. C. Carrasquel and I. A. Lomazova - Complete CPN model of the order-driven trading system. via Google Drive. `https://bit.ly/2UEBMws`.
3. C. Baier and J. Katoen. *Principles of Model Checking.* The MIT Press, 2008.
4. L. Harris. *Trading and Exchanges: Market Microstructure for Practitioners.* Oxford University Press, 2003.
5. K. Jensen and L. M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems.* Springer, 2009.
6. I. A. Lomazova. Nested Petri Nets - a Formalism for Specification and Verification of Multi-Agent Distributed Systems. *Fundamenta Informaticae*, 43:195–214, 2000.
7. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
8. W. Reisig. *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies.* Springer, 2013.
9. W. Van der Aalst. *Process Mining: Data Science in Action.* Springer, 2nd edition, 2016.
10. W. van der Aalst, A. Adriansyah, and B. van Dongen. Replaying History on Process Models for Conformance Checking and Performance Analysis. *WIREs Data Mining and Knowledge Discovery*, 2:182–192, 2012.