Modeling Trading Systems using Petri Net Extensions*

Julio C. Carrasquel[1], Irina A. Lomazova[1], and Andrey Rivkin[2]

[1] National Research University Higher School of Economics,
Myasnitskaya ul. 20, 101000 Moscow, Russia
jcarrasquel@hse.ru, ilomazova@hse.ru
[2] Free University of Bozen-Bolzano,
Piazza Domenicani 3, 39100 Bolzano, Italy
rivkin@inf.unibz.it

Abstract. Trading systems have become sophisticated multi-agent infrastructures with complex development cycles. This is why the financial industry constantly seeks for novel approaches to design and validate these systems. We propose the use of models to support such tasks. On the one hand, these models need to describe how objects (e.g., orders to buy/sell securities) are shared by the system and traders. On the other hand, being a dynamic multi-agent system, models of trading systems should have a clear structure, describing how participants interact between each other. In this paper, we address these requirements, integrating notions of various Petri net extensions. In particular, we discuss modeling capabilities/limitations of each extension, and we propose to integrate them into a single approach, allowing for comprehensive modeling of different trading system components.

Keywords: trading systems, financial technology, formal models, Petri nets, multi-agent systems.

1 Introduction

Trading systems are software platforms used in financial markets to support the exchange of financial instruments between market participants.The kind of such instruments depends on the market type that a trading system works with. For instance, in *commodity markets*, participants trade primary goods ranging from cocoa to gold and oil. In *foreign exchange markets*, people trade currencies. We consider trading systems in *stock exchanges*, where participants submit orders to buy/sell securities (e.g., company shares) [13]. Participants trade for different reasons. For example, investors buy securities with promising returns. Companies sell their shares to gain capital for growth. Given these reasons, trading systems have been positioned as a crucial element of the global economy.

* This work is supported by the Basic Research Program at the National Research University Higher School of Economics.
Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2651/paper8.pdf

Today market environment has become more demanding. On the one hand, there has been an increase of the number and the variety of participants, each of which concurrently interacts with a platform according to some trading strategy. On the other hand, in trading platforms, a plethora of market services and rules affecting, for example, order serving and trade execution policies, have to be implemented. Thus, trading systems have turned into large and sophisticated multi-agent infrastructures with complex development cycles, which are more prone to various flaws. Reducing or even eliminating the number of such flaws becomes a crucial task. However, within these systems, common alternatives to analyze implementation flaws such as active testing are sacrificed to minimize latency and overhead, pushing software quality experts to search for clever and less expensive/intrusive solutions [27]. Thus, the financial technology industry constantly seeks for novel approaches to design and validate trading systems. In this regard, one of the recently proposed approaches suggests to analyze the behavior of the platform and its participants using system logs [15, 9].

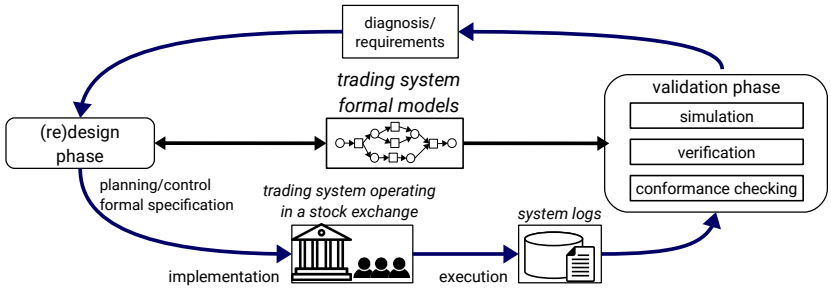


Fig. 1: The central role of formal models for design/validation of trading systems.

In this work, we focus on developing a formalism to support the design and validation phases of trading systems, as illustrated in Fig. 1. To fully support these phases, models constructed with the devised formalism need to provide a holistic view, capturing different aspects of trading systems in a convenient level of abstraction. First, similar to business processes, trading system models need to describe the control-flow of system entities, i.e., activities that participants and a trading platform execute as well as their causal order. Second, as in *data-aware* models, the formalism should allow to describe how objects of the domain (e.g., orders, trades) can be consumed, produced and ultimately shared by the system and processes representing routines executed by the system participants. Finally, being a dynamic multi-agent system by nature, models of trading systems should have a consistent representation, describing how participants (also called agents) asynchronously interact with each other and with the trading platform. Moreover, the number of objects and participants in a model should be able to change dynamically without affecting the model structure. A formalism meeting all such requirements can lay the basis for further development of comprehensive formal specifications of trading systems, and it can be used for running multi-perspective validation methods such as simulation, verification and conformance checking.

In this paper, we delineate key modeling requirements for trading systems, integrating notions of different Petri net extensions. Petri nets are a well-known formalism for modeling and analyzing concurrent distributed systems [25], that provides a graphical notation for visualization and formal semantics allowing to conduct model-based analysis. Moreover, Petri nets represent one of the reference formalisms for conformance checking [8] – an approach allowing to assess correctness of system behavior by comparing its model against concrete executions extracted from system logs. Besides, several extensions of Petri nets have been developed, answering the increasing demand to model different perspectives of distributed systems. To address the modeling requirements of trading systems, we consider the following extensions: (i) colored Petri nets [18], where tokens carry data values of different domains, (ii) nested Petri nets [21], where tokens can be Petri nets themselves, allowing to model multi-agent systems, and (iii) db-nets [23], where non-adjacent net components can share data using a database.

As we show with several examples in this paper, none of these extensions can solely provide a holistic view that could fully cover the modeling requirements of trading systems. Thus, we discuss a possible extension that builds on top of different Petri net classes, and we demonstrate how it can be used to comprehensively model various trading system components. We point out that this solution is not a mere combination of existing Petri net extensions. For example, in contrast with db-nets, where queries are attached to places, we incorporate the concept of *reference tokens* with attached queries. The latter allows to model dynamic lists such as order books independently from the model structure.

The remainder of this paper is structured as follows. In Section 2, we provide an introduction to trading systems. In Section 3, we describe Petri net extensions and we discuss modeling capabilities and limitations using examples of trading system models. In Section 4, we informally present our integrated extension, and we demonstrate how to use it for modeling different components of trading systems. In Section 5, we present the related work. Finally, Section 6 provides some conclusions and future work.

2 Trading Systems

When referring to trading systems, we consider a trading platform and a group of market participants. The former is a software infrastructure supporting the automatic exchange of securities between participants. Fig. 2 depicts a simplified view of the general architecture of trading systems. Communication between participants and a trading platform is typically handled through different network interfaces. For instance, a *trading interface* allows participants to log in the platform and to send/receive trading-related messages. A *market information interface* disseminates market information to users via a real-time protocol (RTP) channel. A *downstream interface* exchanges data with external systems: it sends reports to surveillance authorities, and it forwards executed trades to settlement and clearing systems performing the actual exchange of money and securities.

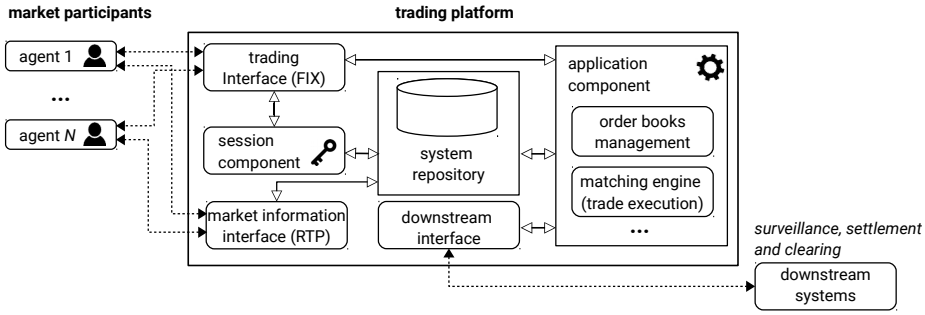


Fig. 2: A simplified view of the general architecture of trading systems.

In this work, we focus on the services provided through the trading interface. This interface is implemented according to a finance-oriented communication standard. We consider as a reference the Financial Information Exchange (FIX) protocol [2], implemented nowadays in most trading platforms. This protocol is organized in two layers: a session and an application layer. At the session layer, the connection between each participant and the platform is managed. On the platform side, the management of user connections is handled by a *session component*. At the application layer, after establishing a connection, agents send trading-related messages to the platform. There can be a large set of trading-related message types according to the services provided by *application components* of a platform. We focus on the application component that manages incoming participant orders to buy/sell securities. As an initial example, Fig. 3 depicts a common message exchange between two participants and a trading platform. The agents initiate (terminate) the communication by exchanging `login(logout)`-type messages with the session component. Once online, each agent sends an order to trade a certain number of stocks of securities, and receive trade notifications in case of a trade.

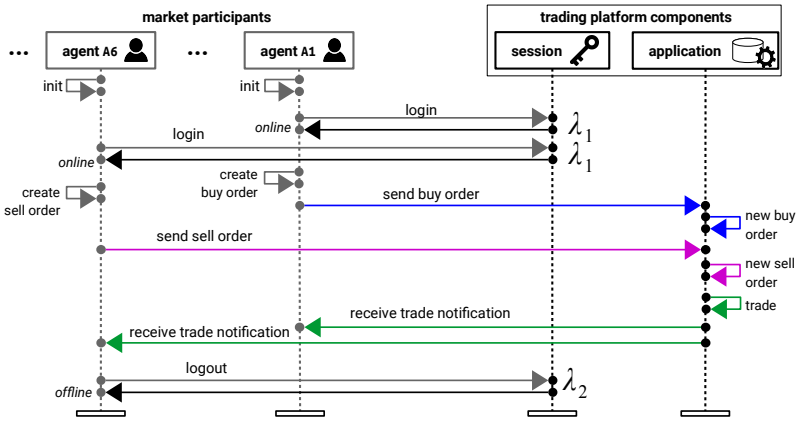


Fig. 3: Message sequence chart describing a typical communication flow between two market participants and components of a trading platform.

Table 1: Example of an order book. For simplicity, we assume that order identifiers are given to orders according to their arrival time.

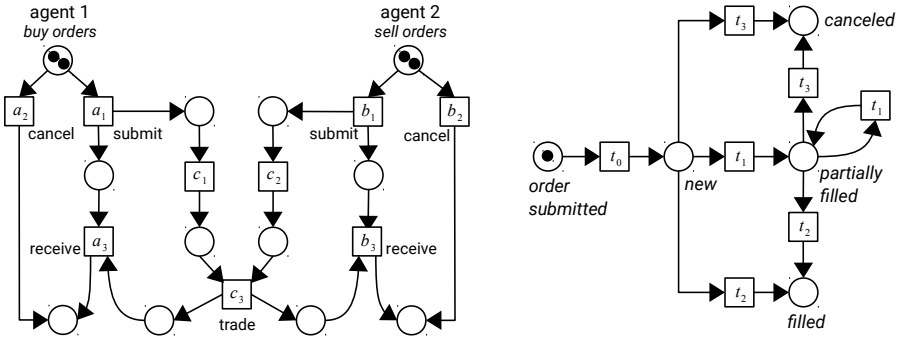
order book `sec1`									
buy side					***sell side***				
position	order id.	agent id.	qty	price	position	order id.	agent id.	qty	price
`# 1`	`O1`	`A1`	4	`22.0 $`	`# 1`	`O6`	`A6`	1	`19.8 $`
`# 2`	`O4`	`A4`	2	`20.1 $`	`# 2`	`O7`	`A7`	6	`20.0 $`
`# 3`	`O2`	`A2`	3	`20.0 $`	`# 3`	`O8`	`A8`	2	`20.1 $`
`# 4`	`O3`	`A3`	2	`20.0 $`	`# 4`	`O9`	`A9`	5	`20.1 $`
`# 5`	`O5`	`A5`	7	`19.8 $`					

In the following, we describe how trading platforms internally manage orders and execute trades. Incoming orders of participants are received and processed sequentially by the corresponding application component, and then they get inserted into *order books*. Order books are lists where orders trading the same security are placed to be matched. Hence, in a trading platform, there can be an order book for each individual security that can be traded in that system. Each order book has a *buy side* and a *sell side*, where buy and sell orders are placed. Orders are placed in order books based on a *precedence rule*. In our work, we consider the *price-time* precedence rule, typically employed in most trading systems. This rule places first buy orders (sell orders) whose prices are the *highest* (*lowest*) in the buy (sell) side; if two orders in the same side have the same price, then the one that arrived earlier is getting placed first. Table 1 displays an order book, whose orders are placed according to the price-time scheme.

Within a trading platform, there is also a *matching engine* that executes trades between orders. For a given order book, this engine takes as input the first orders from the buy side and the sell side. Then, a trade can be executed iff the price of the first buy order is greater or equal than the price of the first sell order (in other words, the best buyer is willing to pay at least as much as the best seller wants). This situation is exemplified by the order book in Table 1, where the price of order `O1` is greater than the price of order `O6`. When a trade is executed between a buy order o_1 and a sell order o_2, which respectively have q_1 and q_2 stocks, $min(q_1, q_2)$ stocks are taken from each order (min returns the minimum of two numbers). The quantities of o_1 and o_2 are decreased according to what they traded, e.g., $q_1{}' = q_1 - min(q_1, q_2)$. For each of the two orders, if its remainder goes to zero, then the order is discarded from the order book (that is, the agent successfully sold/bought what she wanted through that order). Otherwise, the order remains in the order book (still as the first one in its side) waiting to trade its remainder against the next best order of the other side. In Table 1, a trade of 1 stock will be executed with orders `O1` and `O6`. Order `O1` will be partially filled, keeping a remainder of 3 stocks, whereas order `O6` will be filled and consequently discarded from the order book.

3 Petri Net Extensions

Petri nets. A Petri net consists of two kinds of nodes: places and transitions. Places (drawn as circles) represent conditions or resource buffers, whereas transitions (drawn as boxes) denote system activities. Places store tokens (drawn as black dots), and they model resources, local threads, etc. Distribution of tokens across places represents a state of a Petri net, which is called a marking. A transition is enabled to fire iff each of its input places contains at least one token. Firing of an enabled transition consumes a token from each input place and produces a new token in each output place. Fig. 4(a) depicts a small example, where two net components denote agents, whereas a central component consumes orders and produces trades. Whilst we focus on modeling trading platforms and participants, in other cases it might be of interest to analyze atomic elements, i.e., evolution of orders across their lifetime within a platform. The latter can be modeled as a Petri net (see Fig. 4(b)) in which places account for order states and transitions denote activities over orders.



(a) Interaction of two agents and a middleware receiving orders and producing trades.

(b) Handling process of orders.

Fig. 4: Examples of ordinary Petri nets for modeling elements of trading systems.

Colored Petri nets. When constructing trading system models, it is needed to describe how domain objects such as orders are handled. For example, one may need to design how the application component of a platform inserts orders in an order book or how order attributes are modified. For this task we consider colored Petri nets (CPNs). In CPNs, tokens carry values of different data types. Fig. 5 presents a CPN modeling the application component of a trading platform. It models the reception of orders, how these orders are placed in order books according to an employed priority scheme, and how trades are produced. Orders are defined as tuples `o1 = (id,sec,q,p,s,st)` where `id` is an order identifier, `sec` is the security that the order trades, `q` and `p` denote respectively the stock quantity and price per unit, `s` is the order side (buy or sell), and `st` is the order state. Place `O` is the entry point where orders are received.

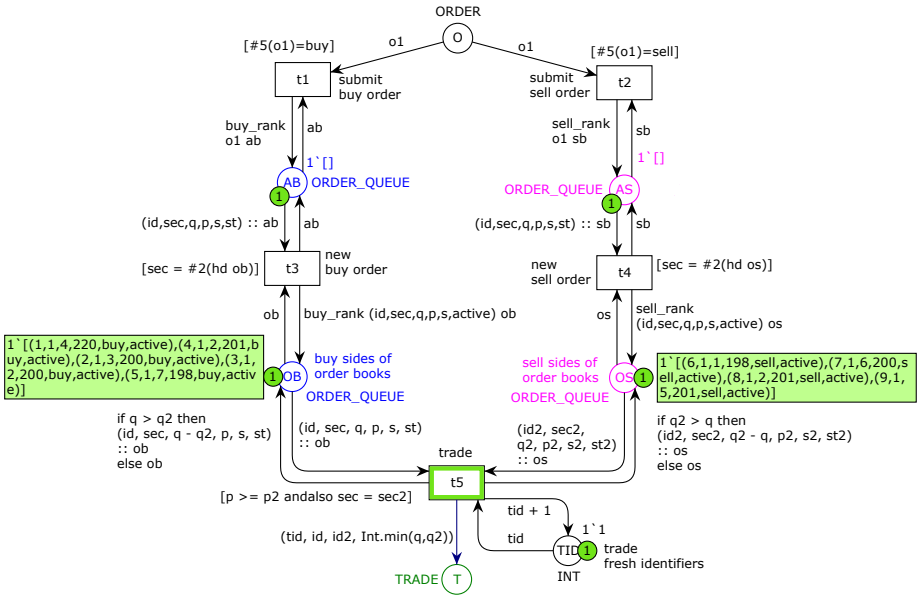


Fig. 5: A CPN model (constructed with CPN Tools [1]) describing the application component of a trading platform.

When there is an order in place `O`, either transition `t1` (`submit buy order`) or `t2` (`submit sell order`) can fire, depending on the order side `s`. When `t1` or `t2` fires, an order is inserted in place `AB` (resp., `AS`) for arriving buy (resp., sell) orders. Two list tokens represent an order book, i.e., the two tokens in places `OB` and `OS` model the order book in Table 1. More pairs of list tokens may be added to include more order books. Insertion of orders in an order book is modeled as follows. When transition `t3` fires, an order in place `AB` is transferred to a list token in place `OB`. The same principle holds on the sell side. The matching between first buy and sell orders is done as follows. For two list tokens representing an order book, transition `t5` takes the first elements of each list – the first buy order and the first sell order. The firing of `t5` is possible if the price `p` of the first buy order is greater or equal than the price `p2` of the first sell order. Then, when `t5` fires, it is produced a token in place `T` denoting a trade. The remainder (if any) of each first order is placed back as the first element of its list.

CPNs provide a language for declaring expressions, formed by variables, constants and functions. In CPN Tools, this language is CPN ML – an implementation of the Standard Meta Language (SML). For example, to insert a buy order in an order book side, a function `buy_rank` is used. The function performs a priority insertion of the order in the list. Thus, precedence rules such as the price-time scheme can be easily implemented. CPNs also provide boolean guards, extending the definition of transition enabling. For example, transition `t5` fires only after the order price values have been checked against the guard assigned to `t5`.

Albeit CPNs meet various requirements to describe data aspects of trading systems, this extension falls short for modeling large multi-agent systems. As we have discussed before, one of the important requirements concerns the ability of having a clear representation of participants, describing how they interact with a trading platform. It is true that subnets denoting participants may be added in a CPN, and such subnets may be connected with the trading platform model using channels (e.g., place `O` in Fig. 5). However, the latter leads to the increasing complexity of net models that, in most cases, may be unreadable and impractical for analysis, especially in the presence of a large number of participants. Hence, we proceed to present another Petri net extension, where participants can be modeled as dynamic objects with explicitly specified behavior.

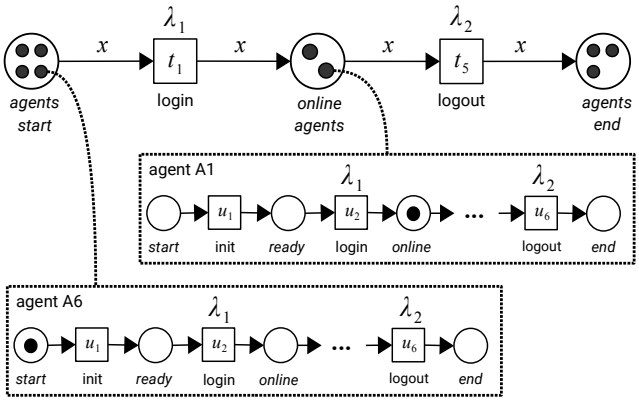


Fig. 6: A NP-net modeling a fragment of a platform session component.

Nested Petri nets. Nested Petri nets (NP-nets) represent an extension of classical Petri nets where tokens can be Petri nets themselves. These tokens, called *net tokens*, account for agents whose behavior is described using Petri nets. Net tokens reside in a Petri net called the *system net*, denoting the environment in which agents interact. Thus, NP-nets allow to model multi-agent systems. Fig. 6 depicts a NP-net modeling a fragment of a platform session component. The system net models the platform side, whereas net tokens represent market participants. Places in the system net model connection states between agents and a trading platform. It provides a clear visualization, displaying which agents are at a given connection state, and how they move to other states in synchronization with the system. The way net tokens progress through this component depends on their inner states. For instance, transition t_1 (activity `login`) fires, if a transition u_2 is enabled in an agent workflow (as these transitions are linked by a *synchronization label* λ_1). When u_2 and t_1 are enabled, both transitions fire simultaneously, updating the session component and the agent involved: the net token is transferred to the place *online agents*, whereas her inner state is updated to *online*. This synchronization mechanism provides a way to model the communication flow at the session layer, depicted in Fig. 3. In general, NP-nets have different firing steps: (i) an *autonomous step*, which is the firing of a transition in a net token, (ii) a *transfer step*, in which net tokens are con-

sumed/produced in the system net without affecting their inner states, (iii) a *vertical synchronization step*, which is the simultaneous firing of a transition in the system net and transitions in net tokens, as it was shown for t_1 and u_2, and (iv) a *horizontal synchronization step*, which is the simultaneous firing of two transitions in two net tokens residing in the same place of a system net. The main limitation of NP-nets lies in their inability to describe how net tokens and a system net share data. As explained before, when modeling trading systems, it is required to model how participants can send orders and receive trades from the trading platform. In this sense, we proceed with a Petri net extension where non-adjacent net components send/receive data through a shared database.

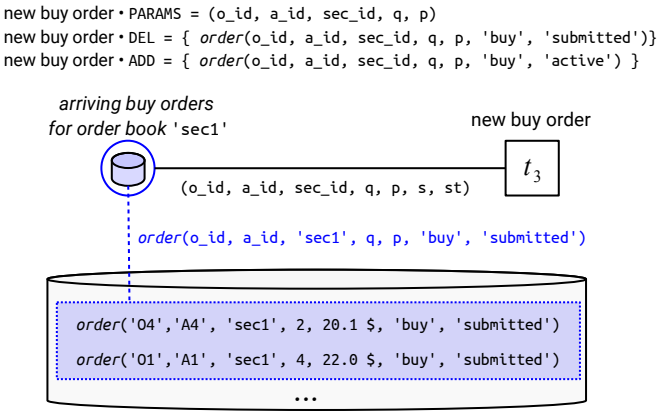


Fig. 7: Example of a db-net modeling insertion of buy orders in an order book.

DB-nets. This extension combines CPNs and relational databases into a single formalism. A db-net is structured in three layers: (i) a *persistence layer* consisting of a relational database. A database stores a finite set of facts (hereafter referred to as records), such that the structures of records are defined by *relation schemas*; (ii) a *data logic layer* which consists of a set of queries and actions used to read/update records in the database, and finally (iii) a *control layer*, consisting of a Petri net incorporating several features of CPNs. Fig. 7 describes buy order insertions in an order book, where a security `sec1` is traded, using db-nets. The database stores, among other records, two orders. For instance, the record *order*(`'O1','A1','sec1'`, 4, `22.0$`, `'buy','submitted'`) models an order sent by the participant `A1`, to buy 4 stocks of `sec1` at `22.0$` per stock. To access records in the database, db-nets employ special places called *view places* that work akin to views in relational databases. More specifically, every view place is having a query assigned to it that populates the place's content with tokens extracted from the query answer. Let us come back to the db-net in Fig. 7. The query *order*`(o_id,a_id,'sec1',q,p,'buy','submitted')` is equivalent to the SQL query `SELECT o_id, a_id, sec_id, q, p, s, st FROM order WHERE sec_id='sec1' AND s='buy' AND st ='submitted'`, returning all orders planning to buy `'sec1'` and whose state is `'submitted'`. In Fig. 7, this query returns two records highlighted in the database, resulting in the view

place being marked with two tokens that represent such records in the net. Every view place can be connected to transitions only via read arcs. This implicitly results in creating tokens that are "copies" of extracted records.

In db-nets, transitions are equipped with actions that allow to update the database. For example, upon firing, transition t_3 invokes an action `new buy order` assigned to it. Assume that t_3 has fired and read the token `('O1','A1', 'sec1',4,22.0$,'buy','submitted')` from the view place. The action assigned to t_3 updates the database by first *deleting* a record corresponding to order `O1`, and then *adding* to the database the same order, but now with its state value changed to `submitted`. In this way, manipulation of orders within the order books can be simulated.

In trading systems, the number of securities that are traded may change dynamically. Thus, there can be a variable amount of order books in the system. To deal with this variability, order books need to be independent from the model structure, and to be conceived as dynamic objects, like participants. Fig. 7 shows why db-nets cannot satisfy such requirement. The query to a collection of orders, representing an order book section, is attached to view places. Under this static modeling strategy, the size of the model structure increases according to the number of order books. This issue does not allow to conceptualize a clever management of order books, being created or closed upon demand. We aim to model order books as a combination of dynamic list tokens (e.g., as in the CPN of Fig. 5). In such model, participants should be able to manipulate remotely elements of such lists. This approach can be implemented if queries are attached to tokens (and not to places). Thus, the number of order books can be variable in the model. Such approach is explained in the following section, where we integrate notions of the introduced Petri net extensions into a single solution, addressing all the aforementioned requirements for modeling trading systems.

4 An Integrated Extension for Modeling Trading Systems

In this section, we propose the integration of the considered Petri net extensions into a single formalism for modeling trading systems. We conceive a Petri net combining the characteristics of CPNs and NP-nets. In addition, following the approach of db-nets, we include in our extension a relational database and queries to consume database records. However, in contrast with db-nets, these queries are going to be directly assigned to *reference tokens*. Such tokens are carrying direct links to records extracted from the underlying database using such queries. Moreover, any manipulation done with these tokens are mirrored on the records they are pointing at. For example, a token that is consumed by a transition will have its data deleted from the database.

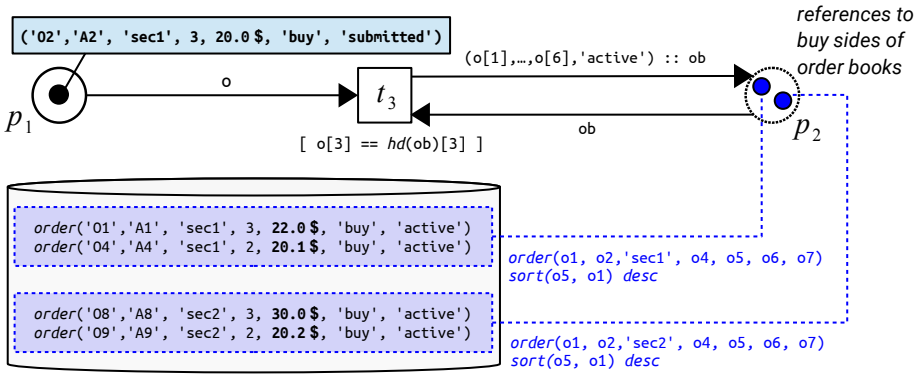


Fig. 8: Example of reference tokens in the proposed extension.

Fig. 8 illustrates the concept of reference tokens. Place p_1 contains an order to buy `sec1`, waiting to be inserted in its order book side. In place p_2, each token has a query attached to it. A result set of each of such queries is a list of orders, modeling an order book buy side. In our approach, reference tokens are treated as lists of records, corresponding to the result sets of queries. To order result sets returned by the queries, we propose to enrich the latter with special *sort* clauses. In Fig. 8, result sets are sorted in descending order, based on two record attributes, namely the price and the order number. When evaluating the guard expression in transition t_3, it is checked whether the third component of the head of a list (hd`(ob)[3]`) from p_2, representing an orders security value is equal to the security value of the incoming order in place p_1. The guard of t_3 evaluates to true (thus making t_3 enabled) if a selected binding contains an element from the reference token in p_2, that relates to orders trading `sec1`. Let us consider the firing of t_3, essentially consists of three phases. First, on token consumption, the query result set of the token is actually consumed, i.e., records are deleted from the database. Second, on token production, the reference token is placed back in place p_2, but with a new element added to its list: the consumed order from p_1. The latter is done using the expression on the arc (t_3, p_2). Finally, the transition firing concludes updating the database based on records in the list of the produced token. In what follows, we explain, using several fragments, how this integrated extension can model different components of trading systems.

4.1 General Scheme of a Trading System Model

Using our proposal, a trading system is modeled as a multi-agent system, consisting of a system net and net tokens. The system net models the session and application components of a trading platform as two disjoint components. Net tokens, denoting participants, reside within the session component (as in Fig. 6). Fig. 9 shows a top view of the trading system model. The application component models order insertion in order books and trade executions. Agents interact with the application component via a database. Database records are orders, trades, or agent data, based on a schema derived from the entity-relationship diagram

of Fig. 10. For example, *trade*('T1', 1, 20.0$, 'A1','A6','O1','O6') models a trade between a buyer A1 and a seller A6. Participants and the application component consume/produce records from/to the database using reference tokens. Records in the database are non ordered, so the application component manages orders using reference tokens, whose queries are equipped with *sort* clauses. Table 2 shows how four reference tokens can model sections of an order book.

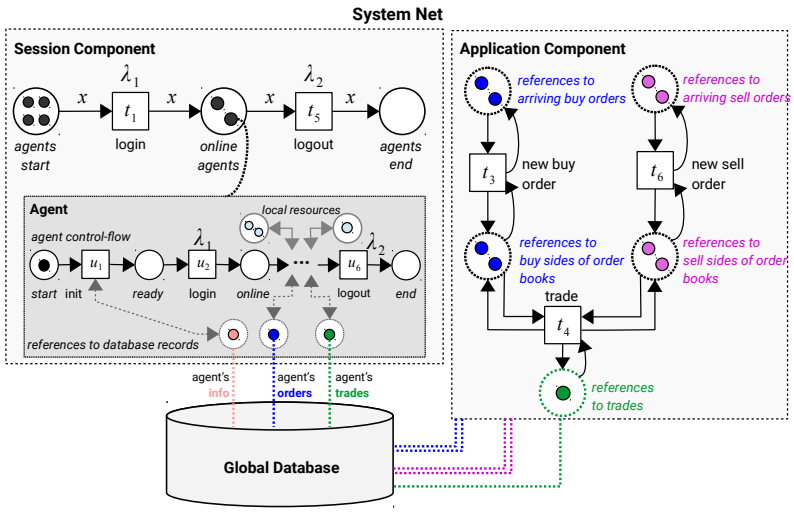


Fig. 9: Top view of a trading system model. Some inscriptions are omitted.

Table 2: Four reference tokens with their queries can model an order book.

order book `sec1`	
place (location)	**query attached to the reference token**
arriving buy orders	*order*(o1,o2,'sec1',o4, o5,'buy','submitted') *sort*(o5,o1) *desc*
arriving sell orders	*order*(o1,o2,'sec1',o4, o5,'sell','submitted') *sort*(o5,o1) *asc*
buy side	*order*(o1,o2,'sec1',o4, o5,'buy','active') *sort*(o5,o1) *desc*
sell side	*order*(o1,o2,'sec1',o4, o5,'sell','active') *sort*(o5,o1) *asc*

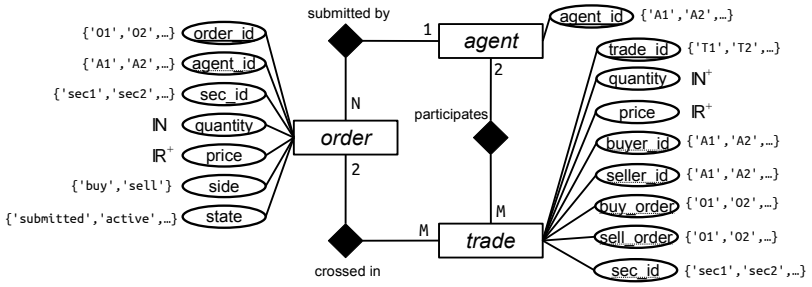


Fig. 10: Entity-relationship diagram of the database model.

4.2 Modeling Market Participants

Participants can be modeled as net tokens, nested within the session component of the system net. Each net token has an inner Petri net workflow (see Fig. 11) composed by different aspects of an agent trading process (control-flow, references to data, and local resources). We describe these aspects, as well as the activities of order submission and trade reception.

Agent control-flow. The control-flow is the agent process backbone. It consists of a set of activities that an agent executes, and it establishes a causal ordering, e.g., activity `submit buy order` executes iff the `login` activity was executed before. As we resort to the same synchronization steps of nested Petri nets, some transitions in the workflow are linked with transitions in the session component using synchronization labels. In this way, we model the connection management between a participant and the session component of a trading platform.

References to shared data. An agent may have references to shared data, for example, to orders that the agent submitted or to upcoming trades. In Fig. 11, reference tokens are created upon the firing of transitions u_1 and u_2. When u_1 fires, it creates a reference to a record $agent(\texttt{'A1'})$, as specified by the inscription $agent(\nu)$ on the arc (u_1, r_1) (ν is a unique $fresh$ agent identifier). Other agents subsequently firing u_1 would insert $agent$ $(\texttt{'A2'})$, $agent(\texttt{'A3'})$, etc. After u_2 fires, references to agent's orders and future trades are created. The queries of these references use the identifier `A1`.

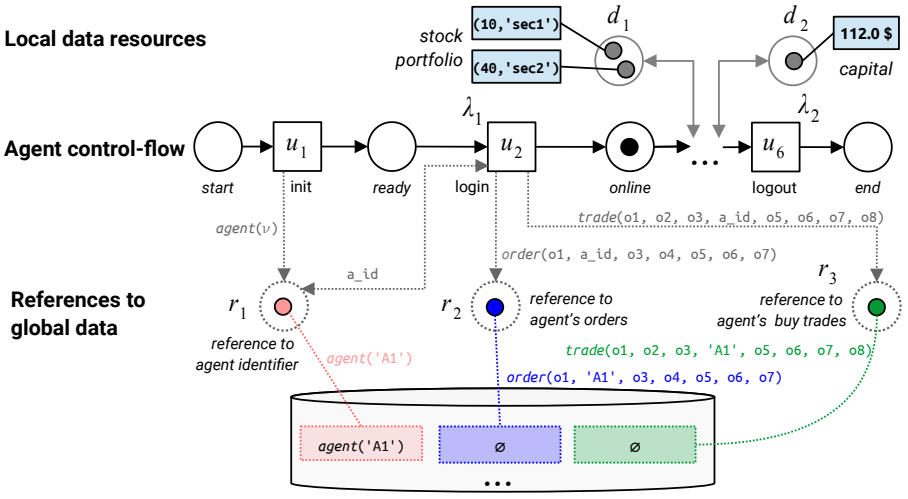


Fig. 11: Fragment of a market participant model.

Agent local resources. Agent local resources (non visible by other agents or the system net) are modeled as places storing tokens with data values. In Fig. 11, places d_1 and d_2 represent the agent's *stock portfolio* and *capital*.

The *stock portfolio* is a set of pairs denoting stock quantities of securities that an agent trades, whereas the *capital* is the available balance to buy more stocks.

Creation and submission of orders. An agent can use her local resources, for instance, to create and submit buy orders. In Fig. 12, the firing of transition u_3 produces an order (yet a local data token) in place *order created*. When transition u_4 fires, the order is consumed, and it is produced as a record in the database. The latter is accomplished using the reference token to agent's orders in the shared database.

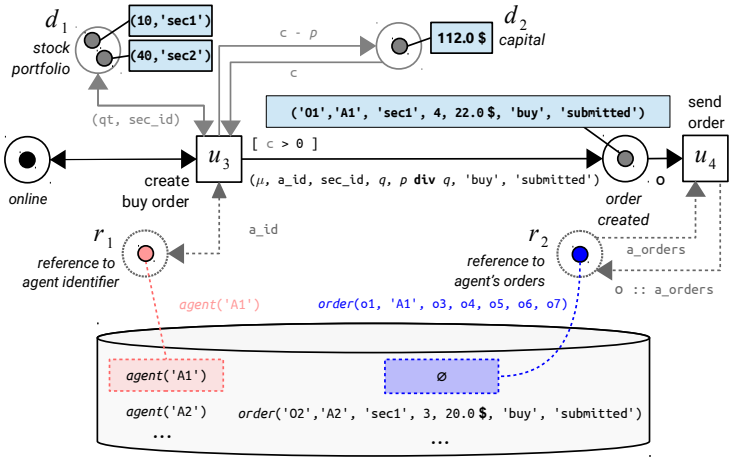


Fig. 12: Agent model fragment regarding the creation/submission of buy orders.

Reception of trade notifications. Fig. 13 depicts the reception of a trade notification in which an agent was involved as a buyer. The buyer updates her trade portfolio based on the purchased quantity of stocks informed in the record.

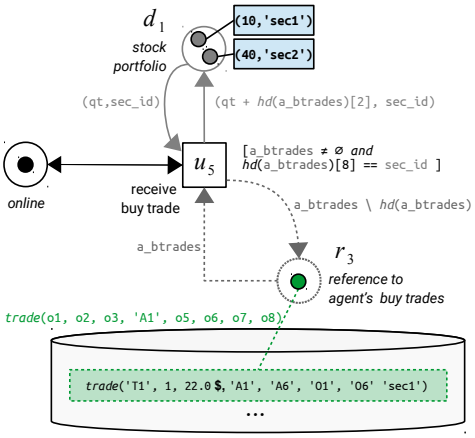


Fig. 13: Agent model fragment describing the reception of trades for a buyer.

4.3 Modeling the Application Component of a Trading System

The application component processes orders submitted by agents. As explained before, all orders in the database trading the same security are organized using four different reference tokens, representing thereby the different sections of an order book. As depicted in Fig. 9, these reference tokens are located in different places of the application component: two tokens for arriving buy/sell orders, and other two tokens for the buy/sell side of the order book. Using this design, we explain the insertion of an order into an order book side and the execution of trades.

Order insertion in an order book side. The insertion of an arriving order into an order book is managed by changing the state attribute of an order from `submitted` to `active`. In this case, the order will be referenced by the reference token which acts as the buy or sell side of an order book. This procedure is exemplified in Fig. 14. When transition t_3 fires, it accesses the list of orders referenced by the token at place *references to arriving buy orders*, removes the first record of this list (that is `O1`, which is the next arriving buy order to be served), resulting in the simultaneous deletion of the same record in the database, and then inserts this record in the list carried by the reference token at place *references to buy sides of order books*. Note that the newly generated token is going to reference a new record as its state is set to `active`.

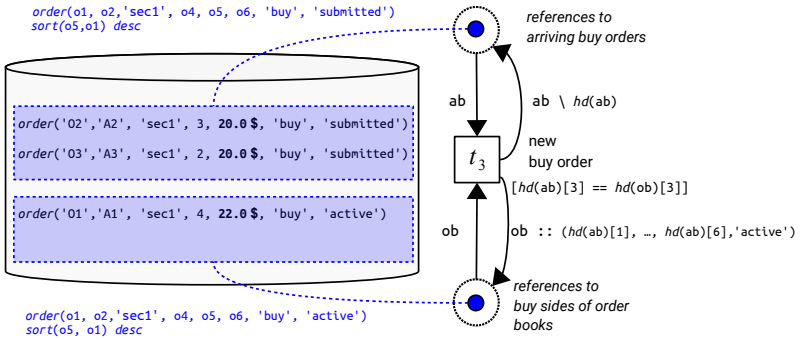


Fig. 14: Model fragment regarding insertion of orders into an order book.

Trade execution. As explained before, if the first two orders of each side of an order book can be matched, then they can produce a trade. We recall that a trade execution is possible iff the price of the first buy order is greater or equal than the price of the first sell order. Fig. 15 presents a fragment of the application component modeling this situation. The guard in transition t_4 (activity `trade`) models the inequality condition between the fifth attributes (the price) of the first records in each side. Then, when transition t_4 fires, it consumes these first orders, it produces a trade record with the stock quantity traded, and finally it places back these first orders to their sides in case they have any stock remainder.

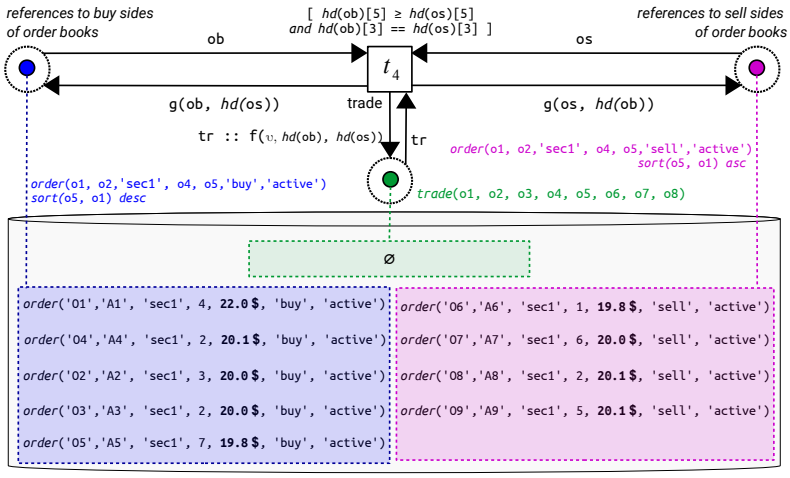


Fig. 15: Model fragment describing trade execution between first orders.

5 Related Work

The increasing amount and heterogeneity of participants and market rules has driven the financial industry to look for novel approaches to design and validate trading systems. In this regard, passive testing techniques have drawn the attention of practitioners [12, 16]. These techniques meet industrial demands of minimizing latency and overhead as they explore system behavior based on output logs, avoiding to inject intrusive control in online systems. On this subject, different authors advocate the use of formal models (see, for example, [4, 5]), sometimes even in conjunction with traditional data science techniques or process mining [15, 19, 20, 24, 30]. This shows that formal models for validating concurrent end-to-end system processes are highly relevant in the industrial context. For instance, formal models can be compared against recorded behavior of trading systems to detect system deviations.

However, to detect deviations that occur due to agent or data corruption, integrated formalisms modeling agent and data aspects become essential. As it has been proposed in [10, 22], extended conformance checking techniques, may be applied between integrated models and event logs that focus on the information about participants within some multi-agent system. While both studies focus on nested Petri nets, [10] offers a preliminary investigation on the feasibility of conformance checking in the case of trading systems, whereas [22] presents a formal approach for checking perfect fitness of a nested Petri net w.r.t a log of a multi-agent system. This paper proposes a possible formalism for constructing such integrated models. Notably, there is already an approach suggesting how to obtain event logs of trading systems from sets of FIX messages, which are captured from network interfaces of trading platforms [9]. Also, this work can be seen as a development of an earlier study based on nested and colored Petri nets for formal modeling of order-driven trading systems [11].

To model trading systems, we are interested in using Petri-net based techniques that can describe persistent data storage with all the complexity of an underlying data model to capture data manipulation, as well as to represent autonomous agents together with a logic that allows them to interact with each other and the environment. As we have already demonstrated before, existing Petri net formalisms do not fully capture all the requirements needed to faithfully model trading systems. We briefly discuss formalisms used in this work and other Petri net classes related to them.

In order to model autonomous agents that interact with each other and the environment, we considered nested Petri nets [21]. Tokens in NP-nets can be Petri nets themselves (referred to as net tokens). As described in Section 3, there are different synchronization steps in NP-nets, allowing to model several interaction patterns between agents and the environment. NP-nets belong to a family of Petri nets called nets-within-nets. From this class of nets, it is noteworthy to mention object Petri nets (OPN), introduced in [31], and adopted in various applied frameworks for modeling and analysis of multi-agent systems (see, e.g., [6, 7]). In contrast with NP-nets, OPNs also support reference semantics. In this type of semantics, a remote object can be referenced by multiple tokens (using special identifiers) stored in different places, resembling the concept of pointers in programming languages. This is a clear difference with respect to NP-nets, which support value semantics, where every single net token is represented as a different object (e.g., a market participant) locally assigned to a specific place.

In Section 3, we showed that colored Petri nets could be used for describing processes that manipulate complex data objects. In CPNs, colors abstractly account for data types, and the control threads (i.e., tokens), traversing the net, carry data conforming to colors. However, when it comes to the analysis of CPN models, one needs to severely restrict the contribution of data by requiring colors to have a finite domain [17]. This, in turn, led to the de facto adoption of the CPN formalism with bounded color domains only. Different approaches have been studied in order to overcome this limitation. For example, ν-Petri nets [29] allow to model processes running on top of an abstract object domain. Single values from this domain are assigned to tokens, whereas transitions can compare these tokens only for equality, but can also generate tokens with values, that are distinct from all those in the current marking. The last feature makes the formalism of ν-Petri nets very appealing to model cases when one needs to secure possibly infinite provision of fresh data objects (for example, generation of a new identifier) into the process. It is worth mentioning an approach studied in [26] that essentially extends the ν-PN formalisms along two dimensions. First, the authors use Petri nets with identifiers that do not deal only with single values, but allow tokens to carry vectors of identifiers from the same abstract object domain. Second, the formalism allows to capture information aspects of the modeled data-aware process by introducing a sophisticated information model allowing to express relations between objects as well as constraints on top of them and CRUD (create, read, update, and delete) operations to manipulate such objects.

There are still other data-aware Petri net classes in which tokens are associated to more complex data structures such as nested relations [14] or XML documents [3]. Even though all these approaches offer different ways for modeling and manipulating data, there is a common limitation that all of them (apart from [26]) share: data elements are "locally" attached to tokens, while no native support for global, persistent relational data is provided.

Given that the data representation and manipulation are crucial for modeling trading systems, in this paper we opted for db-nets [23]. Similarly to the formalism studied in [26], db-nets provide a way for modeling complex dynamic systems by realizing the clear separation of concerns between control flow and data-related aspects. As we demonstrated in Section 3, the persistent data in db-nets are represented using a single relational database. Nevertheless, the approach can be seamlessly extended towards the support of multiple data storages. In addition, a variant of db-nets has been applied for model-based simulation of software systems. More specifically, in [28] the authors demonstrated how to model and test (via simulation) enterprise integration patterns using bounded db-nets and their temporal extensions.

6 Conclusions

In this work, we primarily focused on formal modeling of trading systems in the context of Petri net theory so as to provide a basis for testing their correctness using trading system logs. To this end, we identified and presented a number of crucial design requirements that would allow to capture a holistic view over the systems under consideration. Then, it has been shown how such requirements could be addressed using currently existing formalisms of Petri nets. We argued that each of such formalisms can be suitable for representing only a certain type of the trading system abstraction (and supported our observations with concrete examples), and that currently there is no approach that could fully address all the requirements. Therefore, we informally proposed a new framework that combines relevant features of the discussed Petri net formalisms, and we demonstrated using detailed examples how it can be employed for modeling different components of trading systems.

A next step is to thoroughly formalize the proposed approach by combining already existing theories of CPNs, db-nets and NP-nets, and to develop a prototype supporting modeling and simulation. We also plan to study how to analyze the models of the devised formalism. Indeed, given such a rich setting, performing formal analysis can be a complex task. In this case, one can proceed independently following two directions. First, it is possible to focus only on the simulation task supported by the prototypical implementation of the proposed approach. This can be further applied to perform model-based passive testing. More specifically, one would need to develop a conformance checking technique that accounts both for data and agents interacting with the system net, and that uses execution traces of the trading system software. Notably, the latter can be extracted from trading system logs following the approach presented in [9]. Second, we are planning to investigate how various available theoretical results on

the verification of NP-nets and db-nets can be adopted in the context of the new formalism. We are particularly interested in studying a notion of soundness (a property that guarantees the absence of livelocks, deadlocks, and other domain-related anomalies) that is specific for the scenario of trading systems, and a corresponding compositional verification approach that would allow to check the soundness property separately on every trading agent net.

References

1. CPN Tools - A tool for editing, simulating, and analyzing Colored Petri nets. `https://www.cpntools.org`
2. FIX Standards - FIX Trading Community. `https://www.fixtrading.org/standards/`
3. Badouel, E., Hélouët, L., Morvan, C.: Petri nets with structured data. Fundam. Inform. **146**(1), 35–82 (2016). https://doi.org/10.3233/FI-2016-1375
4. Böhmer, K., Rinderle-Ma, S.: A systematic literature review on process model testing: Approaches, challenges, and research directions. CoRR **abs/1509.04076** (2015), `http://arxiv.org/abs/1509.04076`
5. Broy, M.: Seamless model driven systems engineering based on formal models. In: Formal Methods and Software Engineering. pp. 1–19. Springer Berlin Heidelberg (2009)
6. Cabac, L.: Modeling Petri net-based multi-agent applications. Ph.D. thesis, University of Hamburg (2010), `http://www.sub.uni-hamburg.de/opus/volltexte/2010/4666/index.html`
7. Cabac, L., Haustermann, M., Mosteller, D.: Software development with petri nets and agents: Approach, frameworks and tool set. Sci. Comput. Program. **157**, 56–70 (2018). https://doi.org/10.1016/j.scico.2017.12.003
8. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: Conformance Checking - Relating Processes and Models. Springer (2018). https://doi.org/10.1007/978-3-319-99414-7
9. Carrasquel, J.C., Chuburov, S.A., Lomazova, I.A.: Pre-Processing Network Messages of Trading Systems into Event Logs for Process Mining (2019), to appear
10. Carrasquel, J.C., Lomazova, I.A.: Modelling and Validation of Trading and Multi-Agent Systems: An Approach Based on Process Mining and Petri Nets. In: Proc. of the ICPM Doctoral Consortium. CEUR Workshop Proceedings, vol. 2432 (2019)
11. Carrasquel, J.C., Lomazova, I.A., Itkin, I.L.: Towards a Formal Modelling of Order-driven Trading Systems using Petri Nets: A Multi-Agent Approach. In: Lomazova, I., Kalenkova, A., Yavorsky, R. (eds.) Modeling and Analysis of Complex Systems and Processes (MACSPro). CEUR Workshop Proceedings, vol. 2478 (2019)
12. Cavalli, A.R., Higashino, T., Núñez, M.: A survey on formal active and passive testing with applications to the cloud. Annales des Télécommunications **70**(3-4), 85–93 (2015). https://doi.org/10.1007/s12243-015-0457-8
13. Harris, L.: Trading and Exchanges: Market Microstructure for Practitioners. Oxford University Press (2003)
14. Hidders, J., Kwasnikowska, N., Sroka, J., Tyszkiewicz, J., Van den Bussche, J.: Dfl: A dataflow language based on petri nets and nested relational calculus. Inf. Syst. **33**(3), 261–284 (2008)
15. Itkin, I., Gromova, A., Sitnikov, A., Legchikov, D., Tsymbalov, E., Yavorskiy, R., Novikov, A., Rudakov, K.: User-assisted log analysis for quality control of distributed fintech applications. In: Proc. of AITest 2019,. pp. 45–51. IEEE (2019). https://doi.org/10.1109/AITest.2019.000-9

16. Itkin, I., Yavorskiy, R.: Overview of Applications of Passive Testing Techniques. In: Lomazova, I., Kalenkova, A., Yavorsky, R. (eds.) Modeling and Analysis of Complex Systems and Processes (MACSPro). CEUR Workshop Proceedings, vol. 2478 (2019)
17. Jensen, K.: Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use - Volume 1, Second Edition. Monographs in Theoretical Computer Science. An EATCS Series, Springer (1996). https://doi.org/10.1007/978-3-662-03241-1
18. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer, 1st edn. (2009)
19. Keith, B., Vega, V.: Process mining applications in software engineering. In: Trends and Applications in Software Engineering. pp. 47–56. Springer International Publishing, Cham (2017)
20. Liu, C., van Dongen, B.F., Assy, N., van der Aalst, W.M.P.: A general framework to identify software components from execution data. In: Proc. of ENASE. pp. 234–241. SciTePress (2019). https://doi.org/10.5220/0007655902340241
21. Lomazova, I.A.: Nested Petri Nets - a Formalism for Specification and Verification of Multi-Agent Distributed Systems. Fundamenta Informaticae **43**, 195–214 (2000)
22. Mecheraoui, K., Carrasquel, J.C., Lomazova, I.A.: Compositional conformance checking of nested petri nets and event logs of multi-agent systems. CoRR **abs/2003.07291** (2020), `https://arxiv.org/abs/2003.07291`
23. Montali, M., Rivkin, A.: DB-Nets: On the Marriage of Colored Petri Nets and Relational Databases". In: Koutny, M., Kleijn, J., Penczek, W. (eds.) Transactions on Petri Nets and Other Models of Concurrency XII. LNCS, vol. 10470, pp. 91–118. Springer (2017)
24. Mull, J.: Mind the gap between testing and production: applying process mining to test the resilience of exchange platforms (2019) (October 2019), `https://tinyurl.com/y55sndcv`
25. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
26. Polyvyanyy, A., van der Werf, J.M.E.M., Overbeek, S., Brouwers, R.: Information systems modeling: Language, verification, and tool support. In: Proc. of CAiSE 2019. Lecture Notes in Computer Science, vol. 11483, pp. 194–212. Springer (2019). https://doi.org/10.1007/978-3-030-21290-2_13
27. Protsenko, P., Khristenok, A., Lukina, A., Alexeenko, A., Pavlyuk, T., Itkin, I.: Trading Day Logs Replay Limitations and Test Tools Applicability. In: Proceedings. International Conference on Tools and Methods of Program Analysis (TMPA 2014) pp. 46–53 (2014)
28. Ritter, D., Rinderle-Ma, S., Montali, M., Rivkin, A.: Formal foundations for responsible application integration. Information Systems (2019). https://doi.org/10.1016/j.is.2019.101439, to appear
29. Rosa-Velardo, F., de Frutos-Escrig, D.: Name creation vs. replication in petri net systems. Fundam. Inform. **88**(3), 329–356 (2008)
30. Saint-Pierre, C., Cifuentes, F., Bustos-Jiménez, J.: Detecting anomalies in DNS protocol traces via passive testing and process mining. In: Proc. of CNS. pp. 520–521. IEEE (2014). https://doi.org/10.1109/CNS.2014.6997534
31. Valk, R.: Petri nets as token objects: An introduction to elementary object nets. In: Proc. of ICATPN 1998. pp. 1–25 (1998). https://doi.org/10.1007/3-540-69108-1_1