Pre-Processing Network Messages of Trading Systems into Event Logs for Process Mining*

Julio C. Carrasquel[1], Sergey A. Chuburov[2], and Irina A. Lomazova[1]

[1] National Research University Higher School of Economics,
Myasnitskaya ul. 20, 101000 Moscow, Russia
jcarrasquel@hse.ru, ilomazova@hse.ru
[2] Exactpro Systems,
Lenina ul. 20, 156013 Kostroma, Russia
sergey.chuburov@exactprosystems.com

Abstract. Process mining is emerging as an important discipline for the analysis, monitoring, and improvement of business and software processes. Methods from process mining are based on the use of formal models and event logs, i.e., describing respectively the expected and observed behavior of system processes. This approach can be leveraged by the software testing industry for the log-based analysis of trading platforms. In this light, this paper presents an approach to extract event logs for process mining from network messages of trading systems. In particular, these messages are Financial Information Exchange (FIX) protocol messages, which are related to trading sessions in order books.

Keywords: process mining, trading systems, financial information exchange (FIX) protocol, event logs, data pre-processing.

1 Introduction

The reliability and robustness of stock trading platforms [10] is widely recognized to be crucial for the stability and integrity of global financial markets [21]. The rapid increase in the volume of transactional data, and the growing complexity of the market rules and infrastructures have turned automated exchanges into very large distributed systems, which present significant testing challenges [9].

Moreover, quality standards to meet, such as the minimization of latency and overhead, make difficult the deployment of intrusive testing instrumentation within trading platforms. This is the reason why logs of these systems are often employed as an alternative to analyze their behavior [12]. A recent endeavor in this direction can be found in [11], where the authors propose a (user-assisted) log analysis framework, powered by different *data science* techniques. The framework is aimed to be a support for test engineers, providing them an understanding of system states and possible behavior deviations. As an example, a practical experience using text analysis and clustering was also introduced in [11] for the diagnosis of *settlement* and *clearing* systems.

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

The usage of logs for analyzing the behavior of trading systems matches with the approach of process mining [4]. Methods from process mining take as input the so-called *event logs*. An event log is related to a system process, and it records a set of *cases*, such that each case represents an execution of the process (a process instance). A case consists of *events*, where each event is related to some process activity. Fueled by event logs, process mining methods allow to construct process models from *observed behavior* (process discovery), to diagnose deviations comparing logs against *expected behavior* described by formal models (conformance checking) [3, 5, 14], and to analyze process performance [13, 15], among other capabilities. Research works have employed process mining to analyze software systems behavior and the interaction of users [18–20].

Thus, process mining can be integrated within the analysis framework of trading systems introduced in [11]. However, as Fig. 1 depicts, it is firstly required to pre-process system data sources (either from logging components or captured from network interfaces) into well-structured event logs that process mining methods may leverage afterwards.

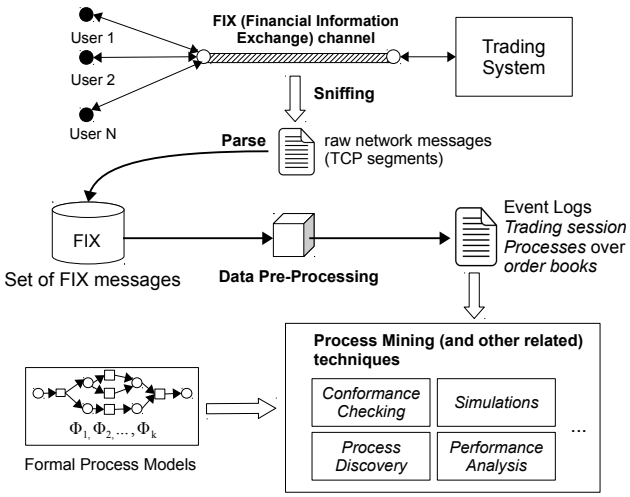

Fig. 1: The research scheme: from FIX messages to event logs for process mining.

This paper presents an approach to extract event logs for process mining from network messages of trading systems. In particular, we consider Financial Information Exchange (FIX) protocol messages [7]. FIX is a communication standard widely adopted in large-scale trading systems. Besides, in this work we focus on trading session processes in order books. These processes are carried out within trading system cores, and their correct execution is determining. In order books, submitted orders to buy or sell securities from market participants are ranked and crossed for trading, typically supported by a matching engine (see [10] for a detailed description of such processes). In such context, we present the approaches for extracting two types of event logs, each of them with a different notion of a case:

- **Order-based event logs:** Each case refers to the observed trace of an individual order submitted by a market participant, i.e., from an event when an order is submitted until an event in which such order is discarded, i.e., because it is *filled*, *canceled*, etc. Each case is identified by an order identifier.
- **Order-Book-based event logs:** Each case refers to the trading session of a financial security in an order book, i.e., from a first to a last event during a trading day related to orders trading a specific security. Each case is identified by a security identifier.

In this way, process mining methods can leverage *order-based* event logs to diagnose the behavior of executed orders, whereas using *order-book-based* event logs it is possible to analyze states of order books. Based on the approaches provided in this paper, we have developed a toolset (in Java programming language) to extract these two types of event logs from FIX messages; the toolset is available via [1]. The toolset includes a graphical interface to replay order-book-based event logs. The interface also can be used for simulation purposes. The results provided in this paper can be replicated to extract event logs for process mining to analyze other components of trading systems, as well as using other protocols similar to FIX.

The remainder of this paper is structured as follows. Section 2 introduces some basics about the FIX protocol, as well as some basic features of the message set used as input for extracting event logs. Section 3 describes the approaches to extract event logs for process mining from FIX messages. Finally, section 4 presents some conclusions and future work.

2 The Financial Information Exchange (FIX) Protocol

The Financial Information Exchange (FIX) protocol [7] is a point-to-point communication standard for exchanging trading-related messages. It operates at the application layer over the TCP/IP stack. FIX is employed in many large-scale trading systems operating worldwide such as in the National Association of Securities Dealers Automated Quotation (NASDAQ) [16], or in the London Stock Exchange (LSE) [2]. In the following, we describe some basics of the protocol, as well as we present an example of some basic features of a set of FIX messages.

Message format. A FIX message is a sequence of ASCII-encoded tag-value pairs separated by the `0x01` control character; tags are integers indicating the meaning of the value. Fig. 2 depicts a message from a trader, with identifier `User1` (tag 49), to the trading system; The message can be read as follows: Tag 35 refers to the activity (message type) performed by the sender, i.e., the value `D` stands for submit an order; tag 40 indicates which order type (*market*, *limit*, *stop*, *pegged*, etc.) is submitted — in this case, it is a limit order (2); tag 59 refers to the validity, i.e., how long this order will be alive — the value 0 stands for a day order; tag 54 indicates whether the user wants to buy (1) or sell (0). Thus, the user is submitting a `day limit order` configured to buy 40 stocks (tag 38) of the security `VTB24` (tag 55) at a stock price of 100 (tag 44).

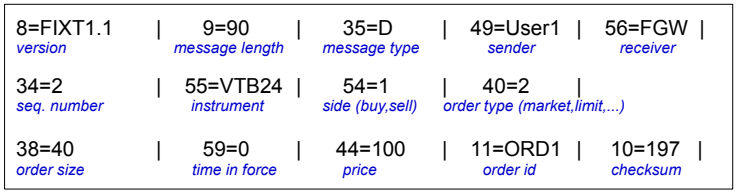

Fig. 2: Example of a Financial Information Exchange (FIX) protocol message.

Table 1: A subset of some FIX messages related to the handling of orders [7].

Client-initiated messages	
Message type ⟨*code*⟩	*Description*
new order - single ⟨*D*⟩	*submit* an order.
order cancel request ⟨*F*⟩	*cancel* an order.
order cancel mass request ⟨*q*⟩	*cancel multiple* orders.
order replace request ⟨*G*⟩	*replace* an order.
System-initiated messages (notifications)	
Message type ⟨*code*⟩	*Description*
execution report ⟨8⟩	Notifies a performed activity over an order.
order cancel reject ⟨9⟩	a cancel/replace request has been rejected.
order mass cancel report ⟨*r*⟩	a mass cancel request has been accepted/rejected.

Protocol layers and message types. The FIX protocol is divided in two layers: a session and an upper application layer. The session layer handles the maintenance of a session between a user and the system. Once a session is established, the upper application layer in each entity is enabled to transmit. FIX messages can be either session-level messages (*logon*, *logout*, *heartbeat*, etc.) or application specific. The message type is indicated in tag 35 (as depicted in Fig. 2). Trading systems provide a large set of application-related message types, i.e., for order handling, quote negotiation, market data subscriptions, etc. For instance, Table 1 presents a subset of application message types related to the order handling between clients and a trading system.

Table 2: Some features of a network message set captured during a trading day.

TCP Segments (including just control segments)	988803
Valid FIX Messages	552935
First message (*event*) sent at	18-02-2019 02:14:31
Last message (*event*) sent at	18-02-2019 17:29:06
Number of individual orders to buy or sell	64539
Number of financial securities being traded	1144
Number of distinct market participants	593
FIX *Execution report* messages	138392

Message set for extracting *event logs*. As shown by Fig. 1, FIX messages used to extract event logs are captured from a system network interface during a trading day. The messages are encapsulated in network packet payloads, i.e., TCP segments. For this reason, we implemented a parser used in the toolset we developed (available via [1]) to extract FIX messages from TCP payloads. As Table 2 exemplifies, it is possible to extract basic information from a set of FIX messages, captured during a trading day, such as the total amount of orders executed, number of securities traded, number of market participants, etc.

3 Extracting Event Logs for Process Mining from FIX Messages

In this section we present approaches to extract event logs for process mining from a set of FIX messages. A set of FIX messages captured during a trading day contains the observed behavior of several trading system components, so event logs of different processes may be extracted. In this work, we consider specifically event logs related to the trading session process in *order books*, where orders from market participants to buy or sell securities are submitted, ranked and crossed [10]. We present techniques for extracting two types of event logs — *order-based* and *order-book-based* event logs. On the one hand, in *order-based* event logs each case is related to the trace of an individual order, so they allow to diagnose the observed behavior of a given set of orders. On the other hand, in *order-book-based* event logs each case refers to the trading session of a financial security in an order book, so this type of event logs can be useful to analyze states of order books. We point out that the performance in practice of these approaches depends on the number of input FIX messages. Notice that we consider a set of already captured messages (*post-mortem* data) and not online pre-processing, which can be instead a subject for further research.

In the following, we introduce some basic definitions related to event logs for process mining. Afterwards, we describe in detail the techniques for extracting the two mentioned types of event logs.

3.1 Basic definitions

An *event log* L is a finite set of *cases* $\{c_1, c_2, ..., c_{|L|}\}$ related to a specific process. Each case $c \in L$ represents an execution of the process, and it consists in an *ordered sequence of events* $\{e_1, e_2, ..., e_{|c|}\}$. Each *event* e is related to the occurrence of an *activity* $a \in \mathcal{A}$, where $\mathcal{A}$ is the set of all activities. An event may have a timestamp t and other domain-specific attributes. Below we define the set of activities $\mathcal{A}$ for both types of event logs presented in this work. It represents some of the activities executed over orders. The set is defined using the values of the FIX tag 150 (of *execution report* messages) plus a *submit* activity (executed when a participant submits an order using a *new order* $\langle D \rangle$ message).

$$\mathcal{A} = \{new, replace, reject, cancel, expire, trade, trade\_cancel\} \cup \{submit\}$$

3.2 Extraction of Order-based Event Logs

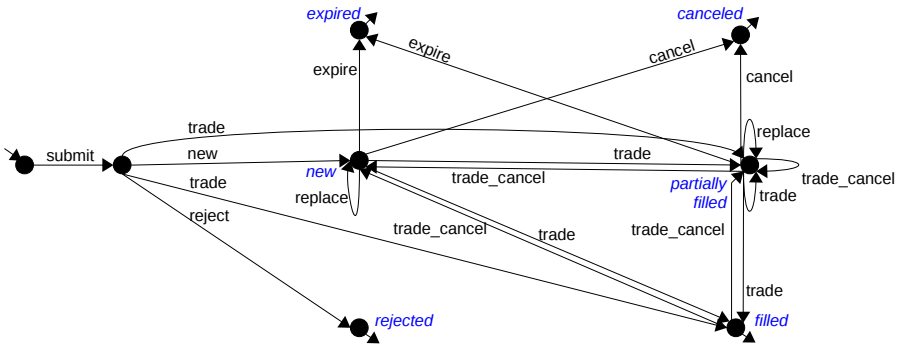

Fig. 3: *Order handling process* as a *transition system*; the node with a small inbound arrow represents the initial state, whereas nodes with small outbound arrows represent the final states.

We consider the extraction of order-based event logs. An order-based event log is related to handling process of individual orders. Each case in this log is an observed trace for a specific order — from an initial event when an order was submitted by a participant until an event when the order was discarded (because it was $filled$, *canceled*, etc.). Fig. 3 depicts the order handling process as a *transition system* — nodes represent states of an order, whereas transitions denote *activities* fired over such order. Each activity a belongs to the set $\mathcal{A}$ presented previously indicating an activity fired over a specific order.

Algorithm 1 presents the procedure we implemented to extract an order-based event log L given a time ordered set $\mathcal{M}$ of FIX messages. We extract orders trading the same financial security, whose identifier $secId$ is given as input. We consider a relationship 1:1 between a FIX message and an *event*, so in each iteration through the set $\mathcal{M}$ (ll. 2-35), a message m is used to create an event e. Thus, we use tag fields contained in the FIX messages to indicate states and attributes of orders when events occur.

Two kind of messages are used in Algorithm 1: *new orders*$\langle D \rangle$ and *execution reports*$\langle 8 \rangle$ (see Table 1). For each processed message m (ll. 2-35), if m is a *new order* message, then a *submit* event is created with some initial order attributes. Otherwise, if m is an *execution report*, then an event e is created, and added to its respective case c in the log L identified by the order identifier (tag 37).

Each event e is structured as a tuple $(id, a, s, t, p, q, \mathfrak{s}, [tr])$ where: id is a case (order) identifier (tag 37); $a \in A$ is an *activity* executed (tag 150); t is a timestamp (tag 60); s, q and p are the current *state*, *size*, and *price* of the order after activity a fired (tags 38, 151, and 44); and $\mathfrak{s} \in \{buy, sell\}$ indicates an order side (tag 54); an event e also may have a *trade identifier* tr (tag 880) if the activity a is a *trade* or a trade cancel — the motivation is to relate two events referring to the same trade (or trade cancellation) in two distinct cases.

Algorithm 1: Extraction of Order-based Event Logs

Input: $\mathcal{M}, secId$;
Output: L;
$\mathcal{D} \leftarrow \emptyset,\ L \leftarrow \emptyset,\ I \leftarrow \emptyset$;
foreach $m \in \mathcal{M}$ **do**
  if $secId \neq m.securityId$ **then**
    continue;
  endif
  $t \leftarrow m.transactTime$; // timestamp (tag 60)
  $\mathfrak{s} \leftarrow m.side$; // tag 54
  $cid \leftarrow m.clOrdId$; // client-side order identifier (tag 11)
  $p \leftarrow m.ordType =$ "$market$" ? "$market$" : $m.price$; // tag 44
  $msgType \leftarrow m.msgType$; // tag 35
  if $msgType =$ "D" **then**
    $a \leftarrow$ "$submit$";
    $s \leftarrow$ " $-$ ";
    $q \leftarrow m.qty$; // stock quantity to trade (tag 38)
    $e \leftarrow (cid, a, s, t, q, p, \mathfrak{s})$;
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{e\}$; // storing *submit* events to be added to cases
  endif
  if $msgType =$ "8" **then**
    $id \leftarrow m.orderId$; // (system-side) order identifier (tag 37)
    $a \leftarrow m.execType$; // an activity from the set $\mathcal{A}$ (tag 150)
    $s \leftarrow m.ordStatus$; // current order state (tag 39)
    $q \leftarrow m.leavesQty$; // current stock quantity (tag 151)
    $tr \leftarrow$ **null**;
    if $a =$ "$trade$" $\vee$ $a =$ "$trade\_cancel$" **then**
      $tr \leftarrow m.trdMatchId$; // trade identifier (tag 880)
    endif
    if $L.contains(id) =$ **false then**
      $c \leftarrow$ **new** $case()$;
      $L.put(id, c)$;
      $I \leftarrow I \cup \{(id, cid)\}$; // relation of system and client order ids.
    endif
    $e \leftarrow (id, a, s, t, q, p, \mathfrak{s}, tr)$; // create an event related to order id
    $L.get(id).add(e)$; // add the event into its correponding case
  endif
endfor
foreach $e \in \mathcal{D}$ **do**
  foreach $(id, cid) \in I$ **do**
    if $e.id = cid$ **then**
      $e.id \leftarrow id$;
      $L.get(id).add(0, e)$; // place *submit* event at the case start
      break;
    endif
  endfor
endfor
return L;

Notice that *new order* $\langle D \rangle$ messages do not contain the system-side order identifiers that we use as case identifiers (tag 37), but just a client-side identifier (tag 11). In Algorithm 1, both identifiers are extracted from *execution report* $\langle 8 \rangle$ messages to relate to which case each new order message (*submit* event) belongs. Thus, *submit* events are added at the beginning of each case (ll. 36-44). Table 3 presents an event log extracted by Algorithm 1. It describes the execution history of 4 *limit* buy orders and 1 *market* sell order in an order book. Fig. 4 depicts a *directly-follows graph*, obtained using Disco [8], which summarizes the observed behavior of orders in the event log of Table 3.

Table 3: An order-based event log consisting of 5 individual orders.

case	*order* (*id*)	*activity* (*a*)	*state* (*s*)	*timestamp* (*t*)	*size*	*price*	*side*	*trade_id*
1	Pl	submit	-	07.536000	100	9.0	*buy*	
1	Pl	new	*new*	07.537557	100	9.0	*buy*	
1	Pl	trade	*filled*	07.581175	0	9.0	*buy*	VE
2	Pm	submit	-	07.544000	100	8.9	*buy*	
2	Pm	new	*new*	07.545718	100	8.9	*buy*	
2	Pm	trade	*filled*	07.581175	0	8.9	*buy*	VF
3	Pn	submit	-	07.565000	100	8.57	*buy*	
3	Pn	new	*new*	07.566645	100	8.57	*buy*	
3	Pn	trade	*filled*	07.581175	0	8.57	*buy*	WG
4	Po	submit	-	07.572000	100	8.45	*buy*	
4	Po	new	*new*	07.573880	100	8.45	*buy*	
4	Po	cancel	*canceled*	11.236553	0	8.45	*buy*	
5	Pp	submit	-	07.579000	400	market	*sell*	
5	Pp	trade	*partially_filled*	07.581175	300	market	*sell*	VE
5	Pp	trade	*partially_filled*	07.581175	200	market	*sell*	VF
5	Pp	trade	*partially_filled*	07.581175	100	market	*sell*	WG
5	Pp	cancel	*canceled*	07.536000	0	market	*sell*	

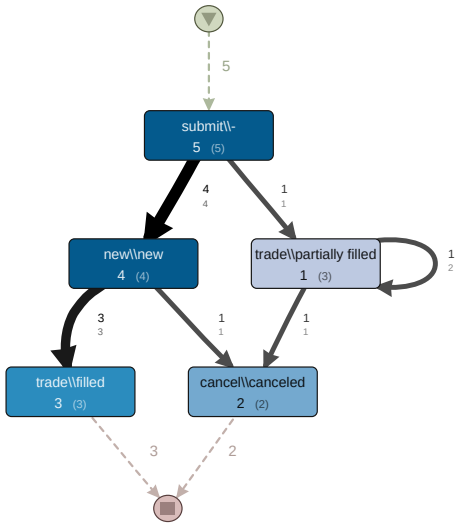

Fig. 4: Directly-follows graph describing execution of orders recorded in Table 3.

Thus, order-based event logs are suitable to analyze the behavior of a set of individual orders. Using this type of event logs it is possible to detect deviations of specific orders, i.e., to detect non-allowed transition movements checking against some reference model like the transition system in Fig. 3. Desired properties can be specified based on order attributes, i.e., using temporal logics [3], to verify whether or not orders in an event log meet these properties.

3.3 Extraction of Order-Book-based Event Logs

Order-based event logs are limited to diagnose the behavior of individual orders, rather than capturing together how orders interact in an order book. The latter is useful to analyze states of order books. Hence, in this part we consider *order-book-based* event logs. In this type of logs, each case refers to a trading session of a financial security in an order book, i.e., from a first to a last event during a trading day involving the trading of a specific security. Each case is identified by a security identifier (FIX tag field 48). Thus, the trade of several financial securities during a trading day, each of them in a different order book, can be recorded in an event log.

Each event e in an order-book-based event log L_{OB} is defined as a tuple $(secId, a, t, o_1, [o_2], [tr])$ where: $secId$ is a case (security) identifier; $a \in \mathcal{A}$ is an activity fired; t is a timestamp; o_1 is an order involved in e, whereas o_2 an optional second order for events where two orders interact, i.e., in *trade* or *trade_cancel* activities; and finally, tr — an optional trade identifier. Orders o_1 and o_2 are tuples of the form $(id, s, q, p, \mathfrak{s})$ indicating an order identifier id, a current state s, size q, price p, and a side $\mathfrak{s}$. For these attributes, we use the same correspondence of FIX tags previously described for order-based event logs.

Let L be an order-based event log, i.e., obtained from a set $\mathcal{M}$ of FIX messages as explained in Section 3.2. Algorithm 2 extracts an order-book-based event log L_{OB} given L as input. We assume that orders in L may trade different securities, so events in L also have an attribute $secId$ indicating the securities that orders trade. As an example, Table 4 shows an event log L_{OB} extracted by Algorithm 2. This event log consists of just one case, the trading of a single financial security in an order book, since it used the same data of orders for the log of Table 3.

3.4 Replay of Order-book-based Event Logs

We have developed a graphical interface (see Fig. 5) to replay order-book-based event logs, providing a convenient visualization of order book states (available via [1]). For example, event number 10 in the event log of Table 4 presents the situation of four buy limit orders, and a market sell order trading with the highest ranked buy order. While order-book-based event logs do not present directly such order book state, the replay capabilities of the interface show directly such situation. The interface operates in two modes: either reading event logs from a file, or receiving events in stream via sockets. The latter allows to connect the inteface with modelling and simulation tools.

Algorithm 2: Extraction of Order-Book-based Event Logs

Input: L;
Output: L_{OB};
$\mathcal{E} \leftarrow \emptyset,\ L_{OB} \leftarrow \emptyset$;
foreach $c \in L$ **do**
  foreach $e \in c$ **do**
    $o_1 \leftarrow (e.id, e.s, e.q, e.p, e.\mathfrak{s})$;
    $o_2 \leftarrow$ **null**;
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\ (e.secId, e.a, e.t, o_1, o_2, e.tr)\ \}$; `// assume secId in e`
  endfor
endfor
$\mathcal{E} \leftarrow sort(\mathcal{E})$; `// sort all events by timestamp and activity priority`
foreach $e_1 \in \mathcal{E}$ **do**
  if $e_1.tr \neq null$ **then**
    foreach $e_2 \in \mathcal{E}$ **do**
      if $e_1.a = a_2.a\ \wedge\ e_1.tr = e_2.tr$ **then**
        $o_2 \leftarrow (e_2.id, e_2.s, e_2.q, e_2.p, e_2.\mathfrak{s})$;
        $e_1.o_2 \leftarrow o_2$; `// merge together orders o1 and o2 in trade event`
        $\mathcal{E} \leftarrow \mathcal{E} - \{e_2\}$;
        break;
      endif
    endfor
  endif
  if $L.contains(e_1.secId) =$ **false then**
    $c \leftarrow$ **new** $case()$;
    $L.put(e_1.secId, c)$;
  endif
  $L.get(e_1.secId).add(e_1)$; `// add event to corresponding trading session`
endfor
return L_{OB};

Table 4: Order-book-based event log, using the data of orders in Table 3. In this example, there is just one case consisting of 14 events where 5 orders interact.

			$order\ 1\ (o_1)$					$[order\ 2\ (o_2)]$				
$secId$	a	t	id_1	s_1	q_1	p_1	$\mathfrak{s}_1$	$[id_2]$	$[s_2]$	$[q_2]$	$[p_2]$	$[\mathfrak{s}_2]$
1	submit	07.536000	Pl	-	100	9.0	*buy*					
1	new	07.537557	Pl	*new*	100	9.0	*buy*					
1	submit	07.544000	Pm	-	100	8.9	*buy*					
1	new	07.545718	Pm	*new*	100	8.9	*buy*					
1	submit	07.565000	Pn	-	100	8.57	*buy*					
1	new	07.566645	Pn	*new*	100	8.57	*buy*					
1	submit	07.572000	Po	-	100	8.45	*buy*					
1	new	07.573880	Po	*new*	100	8.45	*buy*					
1	submit	07.579000	Pp	-	400	market	*sell*					
1	trade	07.581175	Pp	*partially_filled*	300	market	*sell*	Pl	*filled*	0	9.0	*buy*
1	trade	07.581175	Pp	*partially_filled*	200	market	*sell*	Pm	*filled*	0	8.9	*buy*
1	trade	07.581175	Pp	*partially_filled*	100	market	*sell*	Pn	*filled*	0	8.57	*buy*
1	cancel	11.236553	Po	*canceled*	0	8.45	*buy*					
1	cancel	11.236553	Pp	*canceled*	0	market	*sell*					

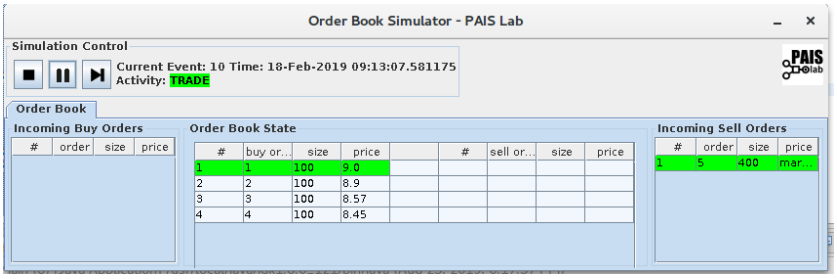

Fig. 5: Interface prototype for replay and visualization of order book states.

4 Conclusions and Future Work

In this paper, we presented an approach to extract event logs for process mining from Financial Information Exchange (FIX) protocol messages of trading systems. We present the extraction of two types of event logs: *order-based* and *order-book-based* event logs. In order-based event logs each case refers to the trace of an order. This allows to synthesize order behavior in process models, and to verify if these orders hold some desired properties. In order-book-based event logs, each case represents a trading session in an order book (related to a single security). Order-book-based event logs can be replayed to analyze order book states. We assumed independence between securities (isolated cases) for reducing the complexity when analyzing and replaying order books states [17]. We selected as event attributes some major order features (state, size, price, side, etc.). We studied event logs with two order types — limit orders and market orders (the latter behave as aggressive limit orders). We also developed a program (available via [1]) for extracting order book trading sessions such that all orders involved are exclusively limit or market orders. Our research is dealing with the construction of event logs for more complex scenarios integrating other order types, i.e., pegged orders, orders with stop conditions, or with non-visible quantities. Finally, it is of interest to integrate explicitly the behavior of market participants in the logs. In such context, a line of our research [6] addresses the development of a formal modelling language that can be suitable to describe the dynamics of trading sessions in order books, integrating the interaction of market participants. On the one hand, models based on such formalism may be useful for simulation. On the other hand, the models are aimed to be compared against event logs, like the ones presented in this paper, for conformance checking.

References

1. Laboratory of Process-Aware Information Systems (PAIS Lab) - Projects - Modelling and Validation of Trading Systems. `https://pais.hse.ru/en/research/projects/tradingsystems`
2. London Stock Exchange - MIT 202 - FIX Trading Gateway Issue 11.8, 2018
3. van der Aalst, W., de Beer, H., van Dongen, B.: Process Mining and Verification of Properties: An Approach Based on Temporal Logic. In: Meersman, R., Tari,

Z. (eds.) On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE. LNCS, vol. 3760, pp. 130–147. Springer (2005)
4. van der Aalst, W.: Process Mining: Data Science in Action. Springer, 2nd edn. (2016)
5. Carmona, J., van Dongen, B., Solti, A., Weidlich, M.: Conformance Checking: Relating Processes and Models. Springer (2018)
6. 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: van Dongen, B., Claes, J. (eds.) Proc. of the ICPM Doctoral Consortium. CEUR Workshop Proceedings, vol. 2432 (2019)
7. FIX Community - Standards: `https://www.fixtrading.org/standards/`
8. Fluxicon: Disco. `https://fluxicon.com/disco/`
9. Government Office for Science (United Kingdom): The Future of Computer Trading in Financial Markets: An International Perspective. Final Project Report (2012)
10. Harris, L.: Trading and Exchanges: Market Microstructure for Practitioners. Oxford University Press (2003)
11. 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: IEEE International Conference On Artificial Intelligence Testing (AITest). pp. 45–51. IEEE (2019)
12. 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)
13. Jaisook, P., Premchaiswadi, W.: Time Performance Analysis of Medical Treatment Processes by using Disco. In: 13th International Conference on ICT and Knowledge Engineering (ICT Knowledge Engineering 2015). pp. 110–115 (2015)
14. Kalenkova, A.A., Ageev, A.A., Lomazova, I.A., vander Aalst, W.: E-government services: Comparing real and expected user behavior. In: Teniente, E., Weidlich, M. (eds.) Business Process Management Workshops. pp. 484–496. Springer (2018)
15. Mannhardt, F., Arnesen, P., Landmark, A.: Estimating the Impact of Incidents on Process Delay. In: 1st International Conference on Process Mining (ICPM). pp. 49–56. IEEE (2019)
16. NASDAQtrader (FIX): `https://www.nasdaqtrader.com/Trader.aspx?id=FIX`
17. Protsenko, P., Khristenok, A., Lukina, A., Alexeenko, A., Pavlyuk, T., Itkin, I.: Trading Day Logs Replay Limitations and Test Tools Applicability. International Conference on Tools and Methods of Program Analysis (TMPA 2014) pp. 46–53 (2014)
18. Rubin, V., Mitsyuk, A., Lomazova, I., van der Aalst, W.: Process Mining Can Be Applied to Software Too! In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ACM (2014)
19. Sahlabadi, M., Muniyandi, R., Shukur, Z.: Detecting Abnormal Behavior in Social Network Websites by using a Process Mining Technique. Journal of Computer Science **10**, 393–402 (2014)
20. Shershakov, S., Rubin, V.: System Runs Analysis with Process Mining. Modeling and Analysis of Information Systems **22**, 818833 (2015)
21. U.S Securities and Exchange Comission: Commission Roundtable on Technology and Trading: Promoting Stability in Todays Markets (2012). `https://www.sec.gov/news/otherwebcasts/2012/ttr100212.shtml`