Formalization of Automated Trading Systems in a Concurrent Linear Framework[*]

Iliano Cervesato Sharjeel Khan Giselle Reis Dragiša Žunić
Carnegie Mellon University
iliano@cmu.edu smkhan@andrew.cmu.edu giselle@cmu.edu dzunic@andrew.cmu.edu

We present a declarative and modular specification of an automated trading system (ATS) in the concurrent linear framework CLF. We implemented it in Celf, a CLF type checker which also supports executing CLF specifications. We outline the verification of a representative property of trading systems using generative grammars, an approach to reasoning about CLF specifications.

1 Introduction

Trading systems are platforms where buy and sell orders are automatically matched. Matchings are executed according to the operational specification of the system. In order to guarantee trading fairness, these systems must meet the requirements of regulatory bodies, in addition to any internal requirement of the trading institution. However, both specifications and requirements are presented in natural language which leaves space for ambiguity and interpretation errors. As a result, it is difficult to guarantee regulatory compliance [3]. For example, the main US regulator, the Securities and Exchanges Commission (SEC), has fined several companies, including Deutsche Bank (37M in 2016), Barclay's Capital (70M in 2016), Credit Suisse (84M in 2016), UBS (19.5M in 2015) and many others [6].

Modern trading systems are complex pieces of software with intricate and sensitive rules of operation. Moreover they are in a state of continuous change as they strive to support new client requirements and new order types. Therefore it is difficult to attest that they satisfy all requirements at all times using standard software testing approaches. Even as regulatory bodies recently demand that systems must be "fully tested" [1], experience has shown that (possibly unintentional) violations often originate from unforeseen interactions between order types [10].

Formalization and formal reasoning can play a big role in mitigating these problems. They provide methods to verify properties of complex and infinite state space systems with certainty, and have already been applied in fields ranging from microprocessor design [8], avionics [14], election security [11], and financial derivative contracts [12, 2]. Trading systems are a prime candidate as well.

In this paper we use the logical framework CLF [5] to specify and reason about trading systems. CLF is a linear concurrent extension of the long-established LF framework [7]. Linearity enables natural encoding of state transition, where facts are consumed and produced thereby changing the system's state. The concurrent nature of CLF is convenient to account for the possible orderings of exchanges.

The contributions of this research are twofold: (1) We formally define an archetypal automated trading system in CLF [5] and implement it as an executable specification in Celf. (2) We demonstrate how to prove some properties about the specification using generative grammars [13], a technique for meta-reasoning in CLF.

[*] This paper was made possible by grant NPRP 7-988-1-178 from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.

Submitted to:
Linearity and TLLA 2018

$$\dfrac{\Gamma;\Delta;\Psi \vdash P_0}{\Gamma;\Delta;\Psi,1 \vdash P_0}\ 1_l \qquad \dfrac{\Gamma;\Delta;\Psi,A,B \vdash P_0}{\Gamma;\Delta;\Psi,A\otimes B \vdash P_0}\ \otimes_l \qquad \dfrac{\Gamma,A;\Delta;\Psi \vdash P_0}{\Gamma;\Delta;\Psi,!A \vdash P_0}\ !_l \qquad \dfrac{\Gamma;\Delta \vdash P_0}{\Gamma;\Delta;\cdot \vdash P_0}\ \mathsf{L}$$

$$\dfrac{}{\Gamma;\cdot \vdash 1}\ 1_r \qquad \dfrac{\Gamma;\Delta_1 \vdash A \quad \Gamma;\Delta_2 \vdash B}{\Gamma;\Delta_1,\Delta_2 \vdash A\otimes B}\ \otimes_r \qquad \dfrac{\Gamma;\cdot \vdash A}{\Gamma;\cdot \vdash !A}\ !_r \qquad \dfrac{\Gamma;\Delta \vdash a}{\Gamma;\Delta \vdash a}\ \mathsf{R} \qquad \dfrac{}{\Gamma;a \vdash a}\ \mathsf{init}$$

$$\dfrac{\Gamma;\Delta_1 \vdash a \quad \Gamma;\Delta_2,N \vdash F}{\Gamma;\Delta_1,\Delta_2,a \multimap N \vdash F}\ \multimap_l \qquad \dfrac{\Gamma;\cdot \vdash a \quad \Gamma;\Delta,N \vdash F}{\Gamma;\Delta,a \rightarrow N \vdash F}\ \rightarrow_l \qquad \dfrac{\Gamma;\Delta,N[x\mapsto t] \vdash F}{\Gamma;\Delta,\forall x.N \vdash F}\ \forall_l \qquad \dfrac{\Gamma;\Delta;P \vdash P_0}{\Gamma;\Delta,\{P\} \vdash P_0}\ \{\}_l$$

$$\dfrac{\Gamma;\Delta,a \vdash N}{\Gamma;\Delta \vdash a \multimap N}\ \multimap_r \qquad \dfrac{\Gamma,a;\Delta \vdash N}{\Gamma;\Delta \vdash a \rightarrow N}\ \rightarrow_r \qquad \dfrac{\Gamma;\Delta \vdash N[x\mapsto \alpha]}{\Gamma;\Delta \vdash \forall x.N}\ \forall_r \qquad \dfrac{\Gamma;\Delta \vdash P}{\Gamma;\Delta \vdash \{P\}}\ \{\}_r \qquad \dfrac{\Gamma,N;\Delta,N \vdash C}{\Gamma,N;\Delta \vdash C}\ \mathsf{cont}$$

Figure 1: Sequent calculus for a fragment of CLF. N is a negative formula, P is a positive formula, P_0 is either an atom or $\{P\}$, F is any formula, a is an atom, α is an eigenvariable and t is a term.

2 Concurrent Linear Logic and Celf

The logical framework CLF [5] is based on a fragment of intuitionistic linear logic. It extends the logical framework LF [7] with the linear connectives $\multimap$, $\&$, $\top$, $\otimes$, 1 and $!$ to obtain a resource aware framework with a satisfactory representation of concurrency. The rules of the system impose a discipline on when the synchronous connectives $\otimes$, 1 and $!$ are decomposed, thus still retaining enough determinism to allow for its use as a logical framework. Being a type-theoretical framework, CLF unifies implication and universal quantification as the dependent product construct. For simplicity we present only the logical fragment of CLF (i.e., without terms) needed for our encodings. A detailed description of the full framework can be found in [5].

We divide the formulas in this fragment of CLF into two classes: *negative* and *positive*. Negative formulas have right invertible rules and positive formulas have left invertible rules. Their grammar is:

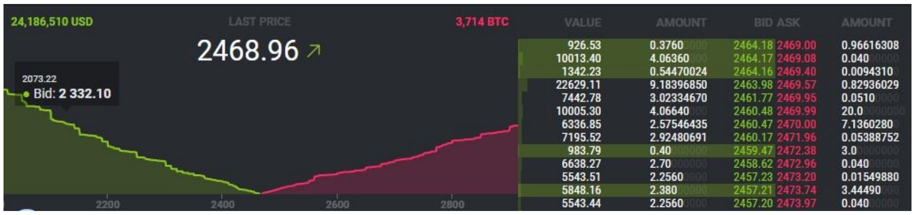
$$\begin{array}{lcll} N,M & ::= & a \multimap N \mid a \rightarrow N \mid \{P\} \mid \forall x.N \mid a & \text{(negative formulas)} \\ P,Q & ::= & P \otimes Q \mid 1 \mid !P \mid a & \text{(positive formulas)} \end{array}$$

where a is an atom (i.e., a predicate). Positive formulas are enclosed in the lax modality $\{\cdot\}$, which ensures that their decomposition happens atomically.

The sequent calculus proof system for this fragment of CLF is presented in Figure 1. The sequents make use of either two or three contexts on the left: Γ contains unrestricted formulas, Δ contains linear formulas and Ψ, when present, contains positive formulas. The decomposition phase of a positive formula on the right is indicated in red and in blue on the left. This phase only ends (via L or R) after the formula is completely decomposed. Note that positive formulas cannot contain negative formulas, so this phase necessarily ends with inits.

Since CLF has both the linear and intuitionistic implications, we can specify computation in two different ways. Linear implication formulas are interpreted as multiset rewriting: the bounded resources on the left are consumed and those on the right are produced. State transitions can be modeled naturally this way. Intuitionistic implication formulas are interpreted as backward-chaining clauses *à la* Prolog, providing a way to compute solutions for a predicate by matching it with the head (rightmost predicate) of a clause and solving the body.

The majority of our encoding involves clauses in the following shape (for atomic p_i and q_i):
$p_1 \otimes \ldots \otimes p_n \multimap \{q_1 \otimes \ldots \otimes q_m\}$ which is the uncurried version of: $p_1 \multimap \ldots \multimap p_n \multimap \{q_1 \otimes \ldots \otimes q_m\}$.



VALUE	AMOUNT	BID	ASK	AMOUNT
926.53	0.3760	2464.18	2469.00	0.96616308
10013.40	4.06360	2464.17	2469.08	0.040
1342.23	0.54470024	2464.16	2469.40	0.0094310
22629.11	9.18396850	2463.98	2469.57	0.82936029
7442.78	3.02334670	2461.77	2469.95	0.0510
10005.30	4.06640	2460.48	2469.99	20.0
6336.85	2.57546435	2460.47	2470.00	7.1360280
7195.52	2.92480691	2460.17	2471.96	0.05388752
983.79	0.40	2459.47	2472.38	3.0
6638.27	2.70	2458.62	2472.96	0.040
5543.51	2.2560	2457.23	2473.20	0.01549880
5848.16	2.380	2457.21	2473.74	3.44490
5543.44	2.2560	2457.20	2473.97	0.040

Figure 2: Visualization of the market view

This framework is implemented as the tool Celf (`https://clf.github.io/celf/`) which we used for the encodings. Following the tool's convention, variable names start with an upper-case letter.

3 Automated Trading System (ATS)

Real life trading systems differ in the details of how they manage orders (there are hundreds of order types in use [9]). However, there is a certain common core that guides all those trading systems, and which embodies the market logic of trading on an exchange. We have formalized those elements in what we call an automated trading system, or an ATS. Let us introduce some basic notions.

An *order* is an investor's instruction to a broker to buy or sell securities (or any asset type which can be traded). They enter an ATS sequentially and are *exchanged* when successfully matched against opposite orders. In this paper, we will only be concerned with *limit* orders. A *limit order* has a specified limit price, meaning that it will trade at that price or better. In the case of a limit order to sell, a limit price P means that the security will be sold at the best available price in the market, but no less than P. And dually for buy orders. If no exchange is possible, the order stays in the market waiting to be exchanged – these are called *resident orders*.

A *matching algorithm* determines how resident orders are prioritized for exchange, essentially defining the mode of operation of a given ATS. The most common one is *price/time priority*. Resident orders are first ranked according to their price (increasingly for sell and decreasingly for buy orders); orders with the same price are then ranked depending on when they entered.

Figure 2 presents a visualization of a (Bitcoin) market. The left-hand side (green) contains resident buy orders, while the right-hand side contains resident sell orders. The price offered by the most expensive buy order is called *bid* and the cheapest sell order is called *ask*. The point where they (almost) meet is the *bid-ask* spread, which, at that particular moment, was around 2468 USD.

Some of the standard regulatory requirements for real world financial trading systems are: the bid price is always strictly less than the ask price (i.e., no locked – bid is equal to ask – or crossed – bid is greater than ask – states), the trade always takes place at either the bid or the ask price, the price/time priority is always respected when exchanging orders, order priority is transitive, and the system does not prohibit a valid exchange.

4 Formalization of an ATS

We have formalized the most popular components of an ATS in the logical framework CLF and implemented them in Celf. This formalization is divided into three parts. First, we represent the market infrastructure using some auxiliary data-structures. Then we determine how to represent limit orders (although our formalization extends to other types of orders) and how they are organized for processing. Finally we encode the exchange rules which act on incoming orders.

Since we are using a linear framework, the state of the system is naturally represented by a set of facts which hold at that point in time. Each rule consumes some of these facts and generates others, thus reaching a new state. Many operations are dual for buy and sell orders, so, whenever possible, predicates and rules are parameterized by the action (`sell` or `buy`, generically denoted A). The machinery needed in our formalization includes natural numbers, lists and queues. Their encoding relies on the backward-chaining semantics of Celf.

The full encoding can be found at `https://github.com/Sharjeel-Khan/financialCLF`.

4.1 Infrastructure

The trading system's infrastructure is represented by the following four linear predicates: $\texttt{queue}(Q)$, $\texttt{priceQ}(A,\ P,\ Q)$, $\texttt{actPrices}(A,\ L)$, and $\texttt{time}(T)$. $\texttt{queue}(Q)$ represents the queue in which orders are inserted for processing. As orders arrive in the market, they are assigned a timestamp and added to Q. For an action A and price P, the queue Q in $\texttt{priceQ}(A,\ P,\ Q)$ contains all resident orders with those attributes. Due to how orders are processed, the queue is sorted in ascending order of timestamp. We maintain the invariant that price queues are never empty. Price queues correspond to columns in the graph of Figure 2. For an action A, the list L in $\texttt{actPrices}(A,\ L)$ contains the exchange prices available in the market, i.e., all the prices on the x-axis of Figure 2 with non-empty columns. Note that the bid price is the maximum of L when A is `buy` and the minimum when A is `sell`. The time is represented by the fact $\texttt{time}(T)$ and increases as the state changes.

The `begin` fact is the entry point in our formalization. This fact starts the ATS. It is rewritten to an empty order queue, empty active price lists for `buy` and `sell`, and the zero time:

$$\texttt{begin} \multimap \{\texttt{queue}(\texttt{empty}) \otimes \texttt{actPrices}(\texttt{buy},\ \texttt{nil}) \otimes \texttt{actPrices}(\texttt{sell},\ \texttt{nil}) \otimes \texttt{time}(\texttt{z})\}$$

4.2 Orders' Structure

An order is represented by a linear fact $\texttt{order}(O,\ A,\ P,\ ID,\ N)$, where O is the type of order, A is an action, P is the order price, ID is the identifier of the order and N is the quantity. P, ID and N are natural numbers. In this paper, O is always `limit`. An order predicate in the context is consumed and added to the order queue for processing via the following rule:

$$\texttt{order}(O,\ A,\ P,\ ID,\ N) \otimes \texttt{queue}(Q) \otimes \texttt{time}(T) \otimes \texttt{enq}(Q,\ \texttt{ordIn}(O,\ A,\ P,\ ID,\ N,\ T),\ Q') \multimap \{\texttt{queue}(Q') \otimes \texttt{time}(\texttt{s}(T))\}$$

The predicate is transformed into a term $\texttt{ordIn}(O,\ A,\ P,\ ID,\ N,\ T)$ containing the same arguments plus the timestamp T. This term is added to the order queue Q by the (backward-chaining) predicate `enq`. This queue allows the sequential processing of orders given their time of arrival in the market, thus simulating what happens in reality. The timestamp is also used to define resident order priority. Sequentiality is guaranteed as all state transition rules act only on the first order in the queue. Nevertheless, due to Celf's non-determinism, orders are added to the queue in an arbitrary order.

4.3 Rules for Handling Order Matching

According to the matching logic, there are two basic actions for every order in the queue: exchange (partially or completely) or add to the market (becomes resident). The action taken depends on the order's limit price (at which it is willing to trade), as well as the bid and ask prices.

We show only the rules used in Section 5. The complete set can be found at `https://github.com/Sharjeel-Khan/financialCLF/blob/master/doc/ATScomplete.pdf`.

Adding orders to the market An order is added to the market when its limit price P is such that it cannot be exchanged against opposite resident orders. Namely when $P < ask$ in the case of a buy order, and when $P > bid$ in the case of a sell order. There are two rules for adding an order, depending on whether there are other rules at the same price in the market or not. The rule below corresponds to the latter case. The (backward chaining) predicate `store` is provable when the order cannot be exchanged.

$$\begin{array}{rl} \mathtt{limit/empty:} & \mathtt{queue}(\mathtt{front}(\mathtt{ordIn}(\mathtt{limit}, A, P, ID, N, T), Q)) \otimes \mathtt{dual}(A, A') \otimes \mathtt{actPrices}(A', L') \otimes \\ & \mathtt{store}(A, L', P) \otimes \mathtt{actPrices}(A, L) \otimes \mathtt{notInList}(L, P) \otimes \mathtt{insert}(L, P, LP) \otimes \mathtt{time}(T) \\ \multimap & \{\mathtt{queue}(Q) \otimes \mathtt{actPrices}(A', L') \otimes \mathtt{priceQ}(A, P, \mathtt{consP}(ID, N, T, \mathtt{nilP})) \otimes \mathtt{actPrices}(A, LP) \\ & \quad \otimes \mathtt{time}(\mathtt{s}(T))\} \end{array}$$

Exchanging orders An order is exchanged when its limit price P satisfies $P \leq bid$, in the case of sell orders, or $P \geq ask$ for buy orders. We present only the exchange rules used in the proof, but in total there are five of them. The (backward chaining) predicate `exchange` binds X to the exchange price (either *bid* or *ask*). The two cases below distinguish between an incoming order that "consumes" all the quantity available at price X (`limit/4`), or only a part of the available orders (`limit/2`). The arithmetic comparison and operations are implemented in the usual backward-chaining way using a unary representation of natural numbers.

$$\begin{array}{rl} \mathtt{limit/2:} & \mathtt{queue}(\mathtt{front}(\mathtt{ordIn}(\mathtt{limit}, A, P, ID, N, T), Q)) \otimes \mathtt{dual}(A, A') \otimes \mathtt{actPrices}(A', L') \otimes \\ & \mathtt{exchange}(A, L', P, X) \otimes \mathtt{priceQ}(A', X, \mathtt{consP}(ID', N', T', \mathtt{consP}(ID1, N1, T1, L))) \otimes \\ & N > N' \otimes (N - N' = N'') \\ \multimap & \{\mathtt{queue}(\mathtt{front}(\mathtt{ordIn}(\mathtt{limit}, A, P, ID, N'', T), Q)) \otimes \mathtt{actPrices}(A', L') \otimes \\ & \mathtt{priceQ}(A', X, \mathtt{consP}(ID1, N1, T1, L))\} \end{array}$$

$$\begin{array}{rl} \mathtt{limit/4:} & \mathtt{queue}(\mathtt{front}(\mathtt{ordIn}(\mathtt{limit}, A, P, ID, N, T), Q)) \otimes \mathtt{dual}(A, A') \otimes \mathtt{actPrices}(A', L') \otimes \\ & \mathtt{exchange}(A, L', P, X) \otimes \mathtt{priceQ}(A', X, \mathtt{consP}(ID', N, T', \mathtt{nilP})) \otimes \mathtt{remove}(L', X, L'') \otimes \mathtt{time}(T) \\ \multimap & \{\mathtt{queue}(Q) \otimes \mathtt{actPrices}(A', L'') \otimes \mathtt{time}(\mathtt{s}(T))\} \end{array}$$

5 Towards a Mechanized Verification of ATS Properties

Using our formalization we are able to check that this combination of order-matching rules does not violate the expected ATS properties. Here we show that the *bid* price is always less than the *ask* price, or, equivalently, the system is never in a locked-or-crossed state.

Although CLF is a powerful logical framework fit for specifying the syntax and semantics of concurrent systems, stating and proving properties about these systems goes beyond its current expressive power. For this task, one needs to consider states of computation, and the execution traces that lead from one state to another. Recent developments show that CLF contexts can be described in CLF itself through the notion of generative grammars [13]. Using such grammars plus reasoning on steps and traces of computation, it is possible to state and prove meta-theorems about CLF specifications. This method is structured enough to become the meta-reasoning engine behind CLF [4].

Since our goal is to eventually formalize the proofs, we develop them using the method just mentioned. We start by defining a generative grammar that precisely captures the set of states satisfying the property considered. This is achieved by observing that *bid* is the maximum of L in $\mathtt{actPrices}(\mathtt{buy}, L)$ and *ask* is the minimum of $\mathtt{actPrices}(\mathtt{sell}, L)$. So the grammar only generates `actPrices` facts if the lists L_B used for buy and L_S for `sell` are such that $\mathtt{max}(L_B) < \mathtt{min}(L_S)$. The start symbol is gen.

Definition 1 *The generative grammar Σ_{NCL}*[1] *produces only contexts where bid < ask.*

$$\begin{array}{lcllcl}
\texttt{buy} & : & \texttt{action}. & \texttt{actPrices} & : & \texttt{action} \to \texttt{listP} \to \texttt{prop}. \\
\texttt{sell} & : & \texttt{action}. & \texttt{minP} & : & \texttt{listP} \to \texttt{exp nat} \to \texttt{prop}. \\
\texttt{gen} & : & \texttt{prop}. & \texttt{maxP} & : & \texttt{listP} \to \texttt{exp nat} \to \texttt{prop}.
\end{array}$$

$$\begin{array}{lcl}
\texttt{gen/00} & : & \texttt{gen} \multimap \{\texttt{actPrices}(\texttt{buy},\ \texttt{nil}) \otimes \texttt{actPrices}(\texttt{sell},\ \texttt{nil})\}. \\
\texttt{gen/10} & : & \texttt{gen} \otimes (L_B \neq \texttt{nil}) \multimap \{\texttt{actPrices}(\texttt{buy},\ L_B) \otimes \texttt{actPrices}(\texttt{sell},\ \texttt{nil})\}. \\
\texttt{gen/01} & : & \texttt{gen} \otimes (L_S \neq \texttt{nil}) \multimap \{\texttt{actPrices}(\texttt{buy},\ \texttt{nil}) \otimes \texttt{actPrices}(\texttt{sell},\ L_S)\}. \\
\texttt{gen/11} & : & \texttt{gen} \otimes \texttt{maxP}(L_B,\ B) \otimes \texttt{minP}(L_S,\ S) \otimes B < S \otimes (L_B \neq \texttt{nil}) \otimes (L_S \neq \texttt{nil}) \\
 & & \multimap \{\texttt{actPrices}(\texttt{buy},\ L_B) \otimes \texttt{actPrices}(\texttt{sell},\ L_S)\}.
\end{array}$$

Intuitively, to show that the market is never in a locked-or-crossed state, we show that, given a context generated by the grammar above, any possible step that can be taken by the specified ATS will result in another context that can also be generated by the proposed grammar. Coupled with the fact that computation starts at a valid context, this shows that the property is always preserved.

Theorem 1 (No locked-or-crossed market) *For every reachable state, if* $\texttt{actPrices}(\texttt{buy},\ L_B)$ *and* $\texttt{actPrices}(\texttt{sell},\ L_S)$ *and* $\texttt{maxP}(L_B,\ B)$ *and* $\texttt{minP}(L_S,\ S)$*, then $B < S$.*

Proof The proof will follow a general scheme which can be illustrated as follows:

$$\begin{tikzcd}
\texttt{gen} \arrow[d, "\varepsilon"] & \texttt{gen} \arrow[d, "\varepsilon'"] \\
\Gamma, \Delta \arrow[r, "\sigma"] & \Gamma', \Delta'
\end{tikzcd}$$

meaning that, if $\Delta \in L(\Sigma_{NCL})$, then any transition rule σ that operates on Δ extended by some Γ will generate a new context Γ', Δ' such that $\Delta' \in L(\Sigma_{NCL})$. Therefore, the proof proceeds by case analysis on σ. We consider only the rules that change linear facts $\texttt{actPrices}(\texttt{buy},\ L_B)$ and $\texttt{actPrices}(\texttt{sell},\ L_S)$, namely `limit/empty`, `limit/2`, and `limit/4` above. Moreover, we restrict ourselves to the case of `buy` orders. The case for `sell` is analogous.

Case $\sigma = \texttt{limit/empty}$: This rule rewrites L_B, the list of buy prices, to a list L'_B which extends L_B by a new price P. Since `store` was provable, we know that P is less than the minimum sell price in the market. Notice that `limit/empty` does not assume that there is a preexisting order on either buy or sell side (L_B, L_S could be `nil`).

We have the following cases for ε:

- $\varepsilon = \texttt{gen/00}$: In this case, $\Delta = \{\texttt{actPrices}(\texttt{buy},\ \texttt{nil}), \texttt{actPrices}(\texttt{sell},\ \texttt{nil})\}$. The rule `limit/empty` rewrites Δ to $\Delta' = \{\texttt{actPrices}(\texttt{buy},\ L'_B), \texttt{actPrices}(\texttt{sell},\ \texttt{nil})\}$, where $L'_B = P :: \texttt{nil}$ (computed by the $\texttt{insert}(L_B,\ P,\ L'_B)$ rule). In this case, ε' which generates Δ' in Σ_{NCL} is `gen/10`.
- $\varepsilon = \texttt{gen/10}$: In this case, $\Delta = \{\texttt{actPrices}(\texttt{buy},\ L_B), \texttt{actPrices}(\texttt{sell},\ \texttt{nil})\}$. Thus, the rule `limit/empty` rewrites Δ to $\Delta' = \{\texttt{actPrices}(\texttt{buy},\ L'_B), \texttt{actPrices}(\texttt{sell},\ \texttt{nil})\}$. In this case, ε' which generates Δ' in Σ_{NCL} is `gen/10`.
- $\varepsilon = \texttt{gen/01}$: In this case, $\Delta = \{\texttt{actPrices}(\texttt{buy},\ \texttt{nil}), \texttt{actPrices}(\texttt{sell},\ L_S)\}$. Thus, the rule `limit/empty` rewrites Δ to $\Delta' = \{\texttt{actPrices}(\texttt{buy},\ L'_B), \texttt{actPrices}(\texttt{sell},\ L_S)\}$, where $L'_B = P :: \texttt{nil}$. In this case, ε' which generates Δ' in Σ_{NCL} is `gen/11`.
- $\varepsilon = \texttt{gen/11}$: Similarly to the previous case, we can conclude that the derivation ε' which generates the context Δ' is `gen/11`.

[1] *NCL* stands for non-crossed and non-locked market.

Case $\sigma = \texttt{limit/4}$: This rule rewrites L_S, the list of sell prices, to a list L'_S which consists of L_S without a price P. Notice that `limit/4` assumes that there is a pre-existing order on the sell side, $L_S \neq \texttt{nil}$. Therefore, the rules `gen/00` and `gen/10` are not applicable in this case (they generate contexts that cannot be operated by `limit/4`). We are left with two cases for ε.

- $\varepsilon = \texttt{gen/01}$: In this case, $\Delta = \{\texttt{actPrices(buy, nil)}, \texttt{actPrices(sell,}\ L_S)\}$. According to `limit/4` we transition from Δ to $\Delta' = \{\texttt{actPrices(buy, nil)}, \texttt{actPrices(sell,}\ L'_S)\}$, where L'_S is L_S without S, which is obtained from $\texttt{minP}(L_S,\ S)$.

  We distinguish two subcases, either $L'_S = \texttt{nil}$ or $L'_S \neq \texttt{nil}$. If $L'_S = \texttt{nil}$ then $\varepsilon' = \texttt{gen/00}$ rewrites Δ to Δ'. Otherwise if $L'_S \neq \texttt{nil}$ then $\varepsilon' = \texttt{gen/01}$.

- $\varepsilon = \texttt{gen/11}$: In this case, $\Delta = \{\texttt{actPrices(buy,}\ L_B), \texttt{actPrices(sell,}\ L_S)\}$. According to `limit/4` we transition to $\Delta' = \{\texttt{actPrices(buy,}\ L_B), \texttt{actPrices(sell,}\ L'_S)\}$.

  Analogous to the previous case, we have two solutions for ε' depending on whether $L'_S = \texttt{nil}$ or $L'_S \neq \texttt{nil}$. They are: $\varepsilon' = \texttt{gen/10}$ and $\varepsilon' = \texttt{gen/11}$, respectively.

Case $\sigma = \texttt{limit/2}$: This rule also reduces the list of sell prices L_S to L'_S by removing its minimum S. It rewrites the context $\Delta = \{\texttt{actPrices(buy,}\ L_B), \texttt{actPrices(sell,}\ L_S)\}$ to the context $\Delta' = \{\texttt{actPrices(buy,}\ L_B), \texttt{actPrices(sell,}\ L'_S)\}$. Notice that `limit/2` assumes that there is a pre-existing order on the sell side, $L_S \neq \texttt{nil}$. Similarly to the previous case we are left with two cases for ε.

- $\varepsilon = \texttt{gen/01}$: Then ε' is either `gen/00` (in the case of $L'_S = \texttt{nil}$), or `gen/01`.
- $\varepsilon = \texttt{gen/11}$: Then ε' is either `gen/10` (in the case of $L'_S = \texttt{nil}$), or `gen/11`.

The proof proceeds similarly for the other cases.

6 Conclusion and Future Work

We have formalized the core rules guiding the trade on most of the exchanges worldwide. We have done this by formalizing an archetypal automated trading system in the concurrent logical framework CLF, with an implementation in Celf.

Encoding orders in a market as linear resources results in straightforward rules that either consume such orders when they are bought/sold, or store them in the market as resident orders. Moreover the specification is modular and easy to extend with new order types, which is often required in practice. This was our experience when adding market and immediate-or-cancel types of orders to the system. The concurrent aspect of CLF simulates the non-determinism when orders are accumulated in the order queue, but, as explained, orders from the queue are processed sequentially[2].

Using our formalization we were able to prove a standard property about a market working under these rules. Namely we prove that at any given state the *bid* price is smaller than the *ask*, i.e., the market is never in a locked-or-crossed state. This was done using generative grammars, an approach motivated by our goal to automate meta reasoning on CLF specifications (not implemented in the current version of Celf). Recent investigations indicate that this approach can handle many meta-theorems [13, 4], and ours is yet another example.

[2] As far as we know, no real life trading system performs parallel order matching and execution.

This specification is an important case study for developing the necessary machinery for automated reasoning in CLF. It is one more evidence of the importance of quantification over steps and traces of a (forward-chaining) computation. It is interesting to note that our example combines forward and backward-chaining predicates, but the generative grammar approach still behaves well. In part because we are only concerned with a linear part of the context. In the meantime, we are investigating other properties of financial systems that present interesting challenges. For example, when trying to show that the trade always take place at prices *bid* or *ask*, we are effectively deliberating about a *transition step* as opposed to a *context*. This may result on new and interesting methods to add to CLF's meta-reasoning engine.

Concurrently, we plan to formalize other models of financial trading systems, as this is a relevant application addressing some critical challenges.

References

[1] Financial Conduct Authority (2018): *Algorithmic Trading Compliance in Wholesale Markets*. Available at `https://www.fca.org.uk/publication/multi-firm-reviews/algorithmic-trading-compliance-wholesale-markets.pdf`.

[2] Patrick Bahr, Jost Berthold & Martin Elsman (2015): *Certified symbolic management of financial multi-party contracts*. In: ICFP 2015, Vancouver, Canada, pp. 315–327.

[3] Jan De Bel (1993): *Automated Trading Systems and the Concept of an Exchange in an International Context Proprietary Systems: A Regulatory Headache*. U. Pa. J. Int'l Bus. L 14(2), pp. 169–211.

[4] Iliano Cervesato & Jorge Luis Sacchini (2013): *Towards Meta-Reasoning in the Concurrent Logical Framework CLF*. In: EXPRESS/SOS 2013, Buenos Aires, Argentina, pp. 2–16.

[5] Iliano Cervesato, Kevin Watkins, Frank Pfenning & David Walker (2003): *A Concurrent Logical Framework I: Judgments and Properties*. Technical Report CMU-CS-02-101, Carnegie Mellon University.

[6] Matthew Freedman (2015): *Rise in SEC Dark Pool Fines*. Review of Banking and Financial Law 35(1), pp. 150–162.

[7] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics*. J. ACM 40(1), pp. 143–184.

[8] Robert Jones, John O'Leary, Carl-Johan Seger, Mark Aagaard & Thomas Melham (2001): *Practical formal verification in microprocessor design*. IEEE Design Test of Computers 18(4), pp. 16–25.

[9] Phil Mackintosh (2014): *Demystifying Order Types*. Available at `http://www.smallake.kr/wp-content/uploads/2016/02/KCG_Demystifying-Order-Types_092414.pdf`.

[10] Grant Olney Passmore & Denis Ignatovich (2017): *Formal Verification of Financial Algorithms*. In: CADE 26, Gothenburg, Sweden, pp. 26–41.

[11] Dirk Pattison & Carsten Schürmann (2015): *Vote Counting as Mathematical Proof*. In: 28th Australasian Joint Conference on Artificial Intelligence, Canberra, Australia.

[12] Simon Peyton Jones, Jean-Marc Eber & Julian Seward (2000): *Composing Contracts: An Adventure in Financial Engineering (Functional Pearl)*. ICFP '00, pp. 280–292.

[13] Robert J. Simmons (2012): *Substructural logical specifications*. Ph.D. thesis, Carnegie Mellon University.

[14] Jean Souyris, Virginie Wiels, David Delmas & Hervé Delseny (2009): *Formal Verification of Avionics Software Products*. In: Proceedings of the 2nd World Congress on Formal Methods, FM '09, pp. 532–546.