A complete formalization of an ATS in CLF

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 full specification of an automated trading system (ATS) in a concurrent linear framework CLF, expressed in Celf, an implementation of CLF. Our motivation is on developing a theory for automated meta-reasoning on CLF specification and implement it in Celf. Besides that, we address some challenges in finance industry, in the domain of systematic regulation of trading systems.

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, in recent years global markets have been troubled by several prominent breaches in regulatory compliance. For example, the main US regulator, the Securities and Exchanges Commission (SEC), has fined several companies, including Deutsche Bank (USD 37 million, in 2016), Barclay's Capital (70M in 2016), Credit Suisse (84M in 2016), UBS (19.5M in 2015) and many others [6]. This is not to say that regulatory challenges did not exist in the past as well [3].

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 if regulatory bodies recently demand that systems must be "fully tested" [1], experience has shown that (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 in which exchanges can take place.

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.

Submitted to:
Linearity and TLLA 2018

$$
\frac{\Gamma, a;\Delta \vdash N}{\Gamma;\Delta \vdash a \to N}\to_r \quad
\frac{\Gamma;\cdot \vdash a \quad \Gamma;\Delta,N \vdash F}{\Gamma;\Delta, a\to N \vdash F}\to_l \quad
\frac{\Gamma;\Delta\vdash N[x\mapsto\alpha]}{\Gamma;\Delta\vdash \forall x.N}\forall_r \quad
\frac{\Gamma;\Delta,N[x\mapsto t]\vdash F}{\Gamma;\Delta,\forall x.N\vdash F}\forall_l
$$

$$
\frac{\Gamma;\Delta,a\vdash N}{\Gamma;\Delta\vdash a\multimap N}\multimap_r \quad
\frac{\Gamma;\Delta_1\vdash a \quad \Gamma;\Delta_2,N\vdash F}{\Gamma;\Delta_1,\Delta_2,a\multimap N\vdash F}\multimap_l \quad
\frac{\Gamma;\Delta\vdash\downarrow P}{\Gamma;\Delta\vdash\{P\}}\{\}_r \quad
\frac{\Gamma;\Delta;P\vdash P_0}{\Gamma;\Delta,\{P\}\vdash P_0}\{\}_l
$$

$$
\frac{\Gamma;\Delta\vdash P_0}{\Gamma;\Delta;\cdot\vdash P_0}\mathsf{L} \quad
\frac{\Gamma;\Delta\vdash a}{\Gamma;\Delta\vdash\downarrow a}\mathsf{R} \quad
\frac{\Gamma,N;\Delta,N\vdash C}{\Gamma,N;\Delta\vdash C}\mathsf{cont} \quad
\frac{}{\Gamma;N\vdash N}\mathsf{init} \quad
\frac{}{\Gamma;\cdot\vdash\downarrow 1}1_r \quad
\frac{\Gamma;\cdot\vdash\downarrow A}{\Gamma;\cdot\vdash\downarrow !A}!_r
$$

$$
\frac{\Gamma;\Delta_1\vdash\downarrow A \quad \Gamma;\Delta_2\vdash\downarrow B}{\Gamma;\Delta_1,\Delta_2\vdash\downarrow A\otimes B}\otimes_r \quad
\frac{\Gamma;\Delta;\Psi,A,B\vdash P_0}{\Gamma;\Delta;\Psi,A\otimes B\vdash P_0}\otimes_l \quad
\frac{\Gamma;\Delta;\Psi\vdash P_0}{\Gamma;\Delta;\Psi,1\vdash P_0}1_l \quad
\frac{\Gamma,A;\Delta;\Psi\vdash P_0}{\Gamma;\Delta;\Psi,!A\vdash P_0}!_l
$$

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. The grammar for our formulas is:

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

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. On the right-hand side, the decomposition phase of a positive formula is indicated by a $\downarrow$.

The rules $\{\}_r$ and $\{\}_l$ are responsible for starting a phase of decomposition of positive formulas. The 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 modelled 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)

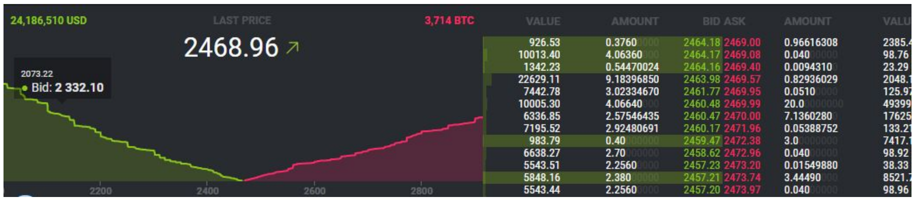

Figure 2: Visualization of the market view

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: $p_1 \multimap \ldots \multimap p_n \multimap \{q_1 \otimes \ldots \otimes q_m\}$.

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 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* essentially defines the mode of operation of a given ATS, since it determines how resident orders are prioritized for exchange. 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 parametrized 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:

- $\mathtt{queue}(Q)$
- $\mathtt{priceQ}(A,\ P,\ Q)$
- $\mathtt{actPrices}(A,\ L)$
- $\mathtt{time}(T)$

Predicate $\mathtt{queue}(Q)$ 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 $\mathtt{priceQ}(A,\ P,\ Q)$ 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 $\mathtt{actPrices}(A,\ L)$ contains the exchange prices available in the market, i.e., all the prices on the x-axis of Figure 2 with non-empty columns. This list is kept sorted in ascending order. Predicate $\mathtt{time}(T)$ keeps track of time, staring from `z` it increases with every operation.

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

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

4.2 Orders' structure

An order is represented by a linear fact $\mathtt{order}(O,\ A,\ P,\ ID,\ N)$, 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 via and added to the order queue for processing:

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

The predicate is transformed into a term $\mathtt{ordIn}(O,\ A,\ P,\ ID,\ N,\ T)$ containing the same arguments plus the timestamp T. This term is added to the order queue $\mathtt{queue}(Q)$ 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 market prices – namely the bid (best offer to buy) and ask (best offer to sell).

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 to the market, distinguishing whether there are other resident orders at the same price or not. See Figure3. The (backward chaining) predicate store is provable when the order cannot be exchanged.

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

Figure 3: Adding limit orders to the market

The first line is the same for both. Given the order at the front of the order queue, the predicate dual will bind A' to the dual action of A (i.e., if A is buy, A' will be sell and vice-versa). Then actPrices(A', L') binds L' to the list of active prices of A'. The incoming order can be added to the market only if there is no dual resident order at an acceptable price. For example, if A is buy at price P, any resident sell order with price *P or less* would be an acceptable match. The predicate store holds iff there is no acceptable match.

The second line of each rule distinguishes whether the new order to be added is the first one at that price (upper rule — notInList(L, P) or not. If it is, the active price list is updated by back-chaining on insert(L, P, LP), and rewriting actPrices(A, L) to actPrices(A, LP). Additionally, a new price queue is created with that order alone (priceQ(A, P, consP(ID, N, T, nilP))). If there are resident orders at the same price (and action), the existing price queue is extended with the new order by back-chaining with extendP(PQ, ID, N, T, PQ') and by rewriting priceQ(A, P, PQ) to priceQ(A, P, PQ'). Both rules increment the time by one unit.

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. See Figure 4.

$$
\begin{array}{ll}
\texttt{limit/1:} & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \\
& \texttt{exchange}(A, L', P, X) \otimes \texttt{priceQ}(A', X, \texttt{consP}(ID', N, T', \texttt{nilP})) \otimes \texttt{remove}(L', X, L'') \otimes \texttt{time}(T) \\
\multimap & \{\texttt{queue}(Q) \otimes \texttt{actPrices}(A', L'') \otimes \texttt{time}(\texttt{s}(T))\}
\end{array}
$$

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

$$
\begin{array}{ll}
\texttt{limit/3:} & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \\
& \texttt{exchange}(A, L', P, X) \otimes \texttt{priceQ}(A', X, \texttt{consP}(ID', N', T', \texttt{nilP})) \otimes \texttt{remove}(L', X, L'') \otimes \\
& \texttt{nat-great}(N, N') \otimes \texttt{nat-minus}(N, N', N'') \\
\multimap & \{\texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N'', T), Q)) \otimes \texttt{actPrices}(A', L'')\}
\end{array}
$$

$$
\begin{array}{ll}
\texttt{limit/4:} & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \\
& \texttt{exchange}(A, L', P, X) \otimes \texttt{priceQ}(A', X, \texttt{consP}(ID', N', T', \texttt{consP}(ID1, N1, T1, L))) \otimes \\
& \texttt{nat-great}(N, N') \otimes \texttt{nat-minus}(N, N', N'') \\
\multimap & \{\texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N'', T), Q)) \otimes \texttt{actPrices}(A', L') \otimes \\
& \texttt{priceQ}(A', X, \texttt{consP}(ID1, N1, T1, L))\}
\end{array}
$$

$$
\begin{array}{ll}
\texttt{limit/5:} & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \\
& \texttt{exchange}(A, L', P, X) \otimes \texttt{priceQ}(A', X, \texttt{consP}(ID', N', T', L)) \otimes \texttt{nat-less}(N, N') \otimes \\
& \texttt{nat-minus}(N', N, N'') \otimes \texttt{time}(T) \\
\multimap & \{\texttt{queue}(Q) \otimes \texttt{actPrices}(A', L') \otimes \texttt{priceQ}(A', X, \texttt{consP}(ID', N'', T', L)) \otimes \texttt{time}(\texttt{s}(T))\}
\end{array}
$$

Figure 4: Exchanging limit orders

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, or only a part of the available orders. The arithmetic comparison and operations are implemented in the usual backward-chaining way using a unary representation of natural numbers.

All five rules start the same way: the first line binds L' to the list of active prices of the dual orders. The backward-chaining predicate $\texttt{exchange}(A, L', P, X)$ holds iff there is a matching resident order. In this case, it binds X to the best available market price The first order in the price queue for X has priority and will be exchanged.

Let N be the quantity in the incoming order and N' be the quantity of the resident order with highest priority. There are three cases:

- $N = N'$ (1^{st} and 2^{nd} clauses in Figure 4): Both orders will be completely exchanged. The incoming order is removed from the order queue by continuing with $\texttt{queue}(Q)$. The resident order is removed from its price queue and we distinguish two cases:
  - It was the last element in the queue (1^{st} clause): then the fact $\texttt{priceQ}(\_, \_, \_)$ is not rewritten and X is removed from the list of active prices by the back-chaining predicate $\texttt{remove}(L', X, L'')$. This list is rewritten from $\texttt{actPrices}(A', L')$ to $\texttt{actPrices}(A', L'')$.
  - Otherwise (2^{nd} clause), the resident order is removed from the price queue by rewriting $\texttt{priceQ}(A', X, \texttt{consP}(ID', N, T', \texttt{consP}(ID1, N1, T1, L)))$ to $\texttt{priceQ}(A', X, \texttt{consP}(ID1, N1, T1, L))$.

- $N > N'$ (3^{rd} and 4^{th} clauses): The incoming order will be partially exchanged, and not leave the order queue as long as there are matching resident orders. At each exchange its quantity is updated to N'', the difference between N and N', computed by the backward-chaining predicate $\mathtt{nat\text{-}minus}(N, N', N'')$. The order queue is rewritten to $\mathtt{queue}(\mathtt{front}(\mathtt{ordIn}(\mathtt{limit}, A, P, ID, N'', T), Q))$. The resident order will be completely consumed and removed from the market. Therefore, we need to distinguish two cases as before (last element in its price queue — 3^{rd} clause — or not — 4^{th} clause).
- $N < N'$ (5^{th} clause): The incoming order is completely exchanged and removed from the order queue (rewritten to $\mathtt{queue}(Q)$). The resident order is partially exchanged and its quantity is updated to N'', computed by the back-chaining clause $\mathtt{nat\text{-}minus}(N', N, N'')$. Notice that the price queue is rewritten with the order in the same position, so its priority does not change.

By convention, the time is only updated once an order is completely processed and removed from the order queue.

5 Towards 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 property. Here we show that the `bid` price B is always less than the `ask` price S, or, equivalently, the system is never in a "crossed or locked market" 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. These states are generated from a single seed context containing just `gen`.

Definition 1 *The following generative grammar, denoted Σ_{NCL}[1], produces only contexts where $B < S$.*

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

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

Intuitively, to show that the market is never in a crossed or locked 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.

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

Theorem 1 (No crossed or locked market) *If* actPrices(buy, *BP*) *and* actPrices(sell, *SP*) *and* maxP(*BP*, *B*) *and* maxP(*SP*, *S*)*, then* $B < S$*.*

Proof The proof proceeds by case analysis on the state transition rules of the ATS for limit orders (case of buy actions, i.e., $A =$ buy). We focus on the rules that change linear facts actPrices(buy, *BP*) and actPrices(sell, *SP*), which are limit/empty, limit/1, and limit/4, above.

The limit/empty rule. By inspecting the rule $\sigma =$ limit/empty we see that it rewrites *BP*, the list of buy prices, to BP' by expanding it, namely it inserts a fresh buy price *P*, which is by definition less than *S* (minP(*SP*, *S*) $\otimes P < S$). Thus BP' is *BP* extended by a fresh price *P* which is less than the minimal sell price at that moment.

Notice that the limit/empty does not assume that there is a pre-existing order on either buy or sell side (*BP*,*SP* could be nil).

The proof will follow the scheme which can be illustrated as follows:

$$\begin{array}{ccc} \texttt{gen} & & \texttt{gen} \\ \downarrow \varepsilon & & \downarrow \varepsilon' \\ \Gamma,\Delta & \xrightarrow{\sigma} & \Gamma',\Delta' \end{array}$$

We have the following cases:

- Case $\varepsilon =$ gen/00. This case applies as limit/empty can be performed even if there are no resident orders in the market (both *BP*,*SP* are empty). If the context Δ is obtained by $\varepsilon =$ gen/00 then $\Delta = \{$actPrices(buy, nil) $\otimes$ actPrices(sell, nil)$\}$. The rule $\sigma =$ limit/empty rewrites the context Δ to Δ' where $\Delta' = \{$actPrices(buy, BP') $\otimes$ actPrices(sell, nil)$\}$, where $BP' = P ::$ nil (in general adding a new fresh price to the list of prices is handled by the insert(*BP*, *P*, BP') rule). Then, ε' which generates Δ' in Σ_{NCL} is gen/10.
- Case $\varepsilon =$ gen/10. If Δ is obtained by $\varepsilon =$ gen/10 we have $\Delta = \{$actPrices(buy, *BP*) $\otimes$ actPrices(sell, nil)$\}$, and $\sigma =$ limit/empty rewrites Δ to Δ' where $\Delta' = \{$actPrices(buy, BP') $\otimes$ actPrices(sell, nil)$\}$. Then, ε' which generates Δ' in Σ_{NCL} is gen/10.
- Case $\varepsilon =$ gen/01. If Δ is obtained by $\varepsilon =$ gen/01 we have $\Delta = \{$actPrices(buy, nil) $\otimes$ actPrices(sell, *SP*)$\}$, and σ rewrites Δ to Δ' where $\Delta' = \{$actPrices(buy, BP') $\otimes$ actPrices(sell, *SP*)$\}$, where $BP' = P ::$ nil. Then, ε' which generates Δ' in Σ_{NCL} is gen/11.
- Case $\varepsilon =$ gen/11. Similarly to the previous case, we conclude that the derivation ε' which generates the context Δ' is gen/11.

The limit/1 rule. By inspecting the rule $\sigma =$ limit/1 we see that it rewrites *SP*, a list of sell prices, to SP' by shrinking it, namely it removes a sell price *P* from the list of sell prices *SP*. So it rewrites the context $\Delta = \{$actPrices(buy, *BP*) $\otimes$ actPrices(sell, *SP*)$\}$ to the context $\Delta' = \{$actPrices(buy, *BP*) $\otimes$ actPrices(sell, SP')$\}$. Removing a price from the list of prices is handled by the remove predicate; remove(*SP*, *P*, SP').

Notice that the limit/1 assumes that there is a pre-existing order on the sell side, $SP \neq$ nil.

Considering that gen/00 and gen/10 do not qualify to generate Δ, we are left with two cases for ε.

- Case $\varepsilon = \texttt{gen/01}$. Context Δ is obtained by $\varepsilon = \texttt{gen/01}$, thus we have $\Delta = \{\texttt{actPrices(buy, nil)} \otimes \texttt{actPrices(sell,}\ SP)\}$, and by $\sigma = \texttt{limit/1}$ we transition from Δ to Δ' where $\Delta' = \{\texttt{actPrices(buy, nil)} \otimes \texttt{actPrices(sell,}\ SP')\}$, and where SP' is SP without S (recall that minP(SP, S)).

  We distinguish two subcases, either $SP' = \texttt{nil}$ or $SP' \neq \texttt{nil}$. If $SP' = \texttt{nil}$ then there exists ε' that leads from Δ to Δ' in Σ_{NCL} and $\varepsilon' = \texttt{gen/00}$. Otherwise if $SP' \neq \texttt{nil}$ then $\varepsilon' = \texttt{gen/01}$.

- Case $\varepsilon = \texttt{gen/11}$. We have $\Delta = \{\texttt{actPrices(buy,}\ BP) \otimes \texttt{actPrices(sell,}\ SP)\}$, and by $\sigma = \texttt{limit/1}$ we transition to Δ' where $\Delta' = \{\texttt{actPrices(buy,}\ BP) \otimes \texttt{actPrices(sell,}\ SP')\}$.

  Depending on whether $SP' = \texttt{nil}$ or $SP' \neq \texttt{nil}$, we have to solutions for ε': $\varepsilon' = \texttt{gen/10}$ and $\varepsilon' = \texttt{gen/11}$, respectively.

The limit/4 rule. By inspecting the rule $\sigma = \texttt{limit/4}$ we see that it also reduces the list of sell prices SP to SP'. Namely it removes S as minimum sell price from SP to obtain SP'. So it rewrites the context $\Delta = \{\texttt{actPrices(buy,}\ BP) \otimes \texttt{actPrices(sell,}\ SP)\}$ to the context $\Delta' = \{\texttt{actPrices(buy,}\ BP) \otimes \texttt{actPrices(sell,}\ SP')\}$.

Notice that limit/4 rule assumes that there is a pre-existing order on the sell side, $SP \neq \texttt{nil}$.

Similarly to the previous case we are left with two cases for ε.

- Case $\varepsilon = \texttt{gen/01}$. Then ε' that lead from Δ to Δ' in Σ_{NCL} is either $\varepsilon' = \texttt{gen/00}$ (in the case of $SP' = \texttt{nil}$), or $\varepsilon' = \texttt{gen/01}$ otherwise.
- Case $\varepsilon = \texttt{gen/11}$. Then ε' that lead from Δ to Δ' in Σ_{NCL} is either $\varepsilon' = \texttt{gen/10}$ (in the case of $SP' = \texttt{nil}$), or $\varepsilon' = \texttt{gen/11}$ otherwise.

The proof goes similarly for sell orders.

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 a 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 type of orders to the system. The concurrent aspect of Celf 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 market under these rules. Namely we prove that at any given state 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). It has been noted in [13] that this is a reasonable way to proceed.

This specification is an important case study for developing the necessary machinery for automated reasoning in Celf. At the same time we plan to formalize other models of financial trading systems, as this is a relevant industry application addressing some long lasting challenges.

[2] No real life trading systems perform parallel order matching and execution.

However, as stated before, our primary focus and motivation is on the development of meta-theory for mechanized reasoning in CLF and Celf.

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.