Formal definitions, theorems, and algorithm

1 Introduction

In this document we present the definitions, results, and the algorithm that appear in our paper 'The Design and Regulation of Exchanges: A Formal Approach' alongside their formal versions (included in the shaded boxes), making it convenient for a reader to understand the formalization. These formal statements are taken directly from the accompanying Coq formalization.

2 Definitions

In this section, we introduce and define all the notions that help us establish our results. In the presentation of our work, we make extensive use of sets for ease of readability for a general mathematical audience. In the Coq formalization, we use lists instead. The presentation and the Coq formalization closely mirror each other and the mathematical content is kept intact. The choice of lists instead of sets in the formalization helps us in two major ways: we can use some of the existing publicly available libraries from previous works for the purposes of modelling auctions, and, more importantly, it helps us in extracting a reasonably fast OCaml program for the verified algorithm which we apply on real data generated from an exchange.

2.1 Orders

Definition 1 (order, id, timestamp, qty, price, ids)**.** *An order is a* 4*-tuple* $(id, time, quantity, price)$*. In Coq, we define an order as a Record as follows.*

```
(* Definition of Order *)
Record order:Type:=
Mk_order{
    id: nat;
    otime: nat;
    oquantity: nat;
    oprice: nat;
    oquantity_cond: Nat.ltb oquantity 1 = false
    }.
```

Deviating from earlier works, we use the field oquantity_cond *to specify that quantity is always a positive number. Consequently, the proofs become more concise.*

For a set of orders Ω, we define

$$\mathsf{ids}(\Omega) := \{id \mid \exists \omega \in \Omega \ s.t. \ \mathsf{id}(\omega) = id\}.$$

In our formalization, $\mathsf{ids}(\Omega)$ *allows for multiple copies of the same id (when* Ω *has multiple orders with the same id. Whenever we need the collection of distinct ids of* Ω*, we use* $\mathsf{uniq}\ \mathsf{ids}(\Omega)$*, where* uniq *returns only one copy of each entry in a list.*

```
(*Defintion of ids*)
Fixpoint ids (B:list order):(list nat):=
  match B with
  |nil => nil
  |b::B' => (id b)::ids B'
  end.
```

For a set of orders Ω *with distinct ids and id which appears in* Ω*, with slight abuse of notation, we define* $\mathsf{timestamp}(\Omega, id)$*,* $\mathsf{qty}(\Omega, id)$*, and* $\mathsf{price}(\Omega, id)$ *to be* $\mathsf{timestamp}(\omega)$*,* $\mathsf{qty}(\omega)$*, and* $\mathsf{price}(\omega)$ *(respectively) of the order* $\omega \in \Omega$ *such that* $\mathsf{id}(\omega) = id$*. Below is a function to extract price of an order from a list of order for a given is. Similarly, we define* $\mathsf{timestamp}$ *and* qty *functions.*

```
(*Definition of price(B, i)*)
(*Only used when In i (ids B)*)
Fixpoint price (B: list order)(i:nat):=
  match B with
  |nil => 0
  |b::B' => if (id b) == i then oprice b else price B' i
  end.
```

Similarly, we have defined timestamp and quantity functions in Coq.

We will often have a universe from which the bids and asks arise. To capture this notion, we define an order domain.

Definition 2 (order-domain, admissible)**.** *(B, A) is an order-domain, if B and A are sets of orders.*

In an order-domain (B, A), the first component B represents a set of bids and the second component A represents a set of asks.

An order-domain is called admissible where each id is distinct and each timestamp is distinct.

We do not need to explicitly define order domain in Coq. We use the following instead.

```
(*(B,A) is an order domain*)
(B A : list order)
```

We use a slightly more general definition of admissible in Coq than what we present in the paper. Instead of requiring all ids (timestamps) to be distinct, we just require the ids (timestamps) of the bids to be distinct and the ids (timestamps) of the asks to be distinct. In the following NoDup (stands for ‘No Duplicate’) is a proposition which is True iff there are no duplicates.

```
Definition admissible (B A :list order) :=
(NoDup (ids B))/\(NoDup (ids A))/\(NoDup (timesof B))/\(NoDup (timesof A)).
```

Once we have the notion of an order-domain, we do not need to define bids and asks explicitly; if an order belongs to the first component of an order-domain, it is regarded as a bid and if it belongs to the second component, it is regarded as an ask.

We now define when a bid and an ask are tradable and when an order-domain is matchable.

Definition 3 (tradable, matchable)**.** *Given two orders b (bid) and a (ask), we say b and a are tradable if* $\mathsf{price}(b) \geq \mathsf{price}(a)$*. Given an order-domain (B, A), B and A are matchable if there exists $b \in B$ and $a \in A$ such that b and a are tradable.*

```
Definition tradable (b a: order):= (oprice b >= oprice a).
Definition matchable (B A : list order):=
exists b a, (In a A)/\(In b B)/\(tradable b a).
```

Next, we capture the notion of competitiveness between two asks and between two bids based on price and time.

Definition 4 (Competitiveness, $\succ$)**.** *A bid b_1 is defined to be more* competitive *compared to another bid b_2, denoted by $b_1 \succ b_2$, iff*

$\mathsf{price}(b_1) > \mathsf{price}(b_2)$ *OR (*$\mathsf{price}(b_1) = \mathsf{price}(b_2)$ *AND* $\mathsf{timestamp}(b_1) < \mathsf{timestamp}(b_2)$*). Similarly, An ask a_1 is considered more* competitive *compared to another ask a_2, denoted by $a_1 \succ a_2$, iff*

$\mathsf{price}(a_1) < \mathsf{price}(a_2)$ *OR (*$\mathsf{price}(a_1) = \mathsf{price}(a_2)$ *AND* $\mathsf{timestamp}(a_1) < \mathsf{timestamp}(a_2)$*).*

In our Coq formalization, we use two definitions for the competitive operator, namely bcompetitive *and* acompetitive*, one for comparing bids and one for comparing asks.*

```
Definition bcompetitive (b b':order):=
(Nat.ltb (oprice b') (oprice b)) ||
((Nat.eqb (oprice b') (oprice b)) &&
(Nat.leb (otime b) (otime b'))).

Definition acompetitive (a a':order):=
(Nat.ltb (oprice a) (oprice a')) ||
((Nat.eqb (oprice a) (oprice a')) &&
(Nat.leb (otime a) (otime a'))).
```

We treat a set of orders also as a multiset where we suppress the quantity field of each order and set its multiplicity equal to its quantity. This view will help us succinctly state the conservation property and ease the formalization substantially. For two sets of orders R and S, we define the relation $R - S$, which corresponds to the usual notion of set difference when viewing R and S as multisets. We continue to use the usual $\setminus$ when the sets are not treated as multisets.

Definition 5 ($-$ between sets of orders)**.** *Let Ω_1 and Ω_2 be sets of orders such that each of them contains orders with distinct ids.*

$\Omega_1 - \Omega_2 :=$

$$\begin{aligned}\{(id,t,q,p) \mid \exists(id,t,q_1,p) \in \Omega_1 \textit{ and } \exists(id,t,q_2,p) \in \Omega_2 \textit{ s.t.}\\ q = q_1 - q_2 \textit{ and } q > 0\\ \textit{OR}\\ \exists((id,t,q,p) \in \Omega_1 \textit{ and } \forall q'(id,t,q',p) \notin \Omega_2\}.\end{aligned}$$

```
(*Defintion of Omega1-Omega2*)
Fixpoint odiff (Omega1 Omega2:list order):(list order).
refine (match Omega1 with
 |nil => nil
 |w1::Omega1' => match (Compare_dec.lt_dec
            ((oquantity w1) - (quant Omega2 (id w1))) 1) with
    |left _ => odiff Omega1' Omega2
    |right _ => (Mk_order (id w1) (otime w1)
                ((oquantity w1) - (quant Omega2 (id w1)))
                (price Omega1 (id w1)) _)::(odiff Omega1' Omega2)
    end
  end).
rewrite PeanoNat.Nat.ltb_nlt. auto.
Defined.
```

2.2 Transactions and matchings

Here, we define transactions, matchings, and associated functions.

In a major modelling decision, we opted to not keep price and timestamp in a transaction which are usually kept in real data for many applications. This greatly simplifies our formalization and exposition without diluting any mathematical content. Based on the application, a transaction between tradable orders b and a can be assigned an appropriate transaction price in the interval $[\mathsf{price}(a), \mathsf{price}(b)]$. In stock markets for example, this price may vary from exchange to exchange. For our work though the role of price and timestamp is used to only decide competitiveness and checking tradability between orders; keeping the price or timestamp in a transaction is completely redundant.

We first define transactions and associated functions.

Definition 6 (transaction, $\mathsf{id}_{\mathsf{bid}}$, $\mathsf{id}_{\mathsf{ask}}$, qty, $\mathsf{ids}_{\mathsf{bid}}$, $\mathsf{ids}_{\mathsf{ask}}$, Qty, Vol)**.** *A transaction is a 3-tuple $(id_b, id_a, quantity)$ of natural numbers where $quantity > 0$. Here id_b and id_a represent the ids of the participating bid and ask.*

We can extract the components of a transaction using the functions $\mathsf{id}_{\mathsf{bid}}$*,* $\mathsf{id}_{\mathsf{ask}}$*, and* qty*. Note that* qty *is overloaded, but that is for ease of presentation; formally they are kept different.*

Given a set of transactions T, id_b and id_a, we define $\mathsf{ids}_{\mathsf{bid}}(T)$ *and* $\mathsf{ids}_{\mathsf{ask}}(T)$ *as follows.*

$$\mathsf{ids}_{\mathsf{bid}}(T) := \{id_b \mid \exists t \in T \textit{ s.t. } \mathsf{id}_{\mathsf{bid}}(t) = id_b\}.$$

$$\mathsf{ids}_{\mathsf{ask}}(T) := \{id_a \mid \exists t \in T \textit{ s.t. } \mathsf{id}_{\mathsf{ask}}(t) = id_a\}.$$

Given a set of transactions T, id_b and id_a we define $\mathsf{Qty}_{bid}(T, id_b)$ and $\mathsf{Qty}_{ask}(T, id_a)$ as follows.

$$\mathsf{Qty}_{bid}(T, id_b) := \sum_{t \in T:\ \mathsf{id}_{\mathsf{bid}}(t) = id_b} \mathsf{qty}(t),$$

which represents the total traded quantity of id_b in T. Similarly, we define the total traded quantity of id_a in T:

$$\mathsf{Qty}_{ask}(T, id_a) := \sum_{t \in T:\ \mathsf{id}_{\mathsf{ask}}(t) = id_a} \mathsf{qty}(t).$$

For ease of readability, whenever it is clear from context, we will use just Qty instead of Qty_{ask} and Qty_{bid}.

Given a set of transactions T, we define the total volume of T, denoted by $\mathsf{Vol}(T)$, as the sum of the quantities of the transactions in T. Formally,

$$\mathsf{Vol}(T) := \sum_{t \in T} \mathsf{qty}(t).$$

Often in Coq we keep two definitions for the same object, one is constructive whereas the other is propositional. We formally establish that these definitions are equivalent, and depending on the context one may be more useful than the other.

```
(* Definitions of ids_bid *)
Fixpoint ids_bid_aux (T: list transaction):(list nat):=
  match T with
  |nil => nil
  |t1::T' => (idb t1)::(ids_bid_aux T')
  end.

Definition fun_ids_bid (T:list transaction) :=
(uniq (ids_bid_aux T)).

Definition ids_bid (I:list nat)(T:list transaction):=
(forall i, In i I ->(exists t, (In t T)/\(idb t = i))) /\
(forall t, In t T ->(exists i, (In i I)/\(idb t = i))) /\
(NoDup I).
```

```
(*Definition of Qty_bid*)
Fixpoint Qty_bid (T: list transaction) (i:nat): (nat):=
  match T with
  |nil => 0
  |t::T' => if (idb t)==i then tquantity t + (Qty_bid T' i)
            else (Qty_bid T' i)
  end.
```

```
(*Definition of Vol*)
Fixpoint Vol (T: list transaction):(nat):=
  match T with
  |nil => 0
  |t::T' => tquantity t + (Vol T')
  end.
```

Next, we capture the notion of when a transaction can arise from an order-domain.

Definition 7 (over, valid)**.** *We say a transaction t is over the order-domain (B, A) iff* $\mathsf{id}_{\mathsf{bid}}(t) = \mathsf{id}(b)$ *for some* $b \in B$ *and* $\mathsf{id}_{\mathsf{ask}}(t) = \mathsf{id}(a)$ *for some* $a \in A$.

We say that a transaction t is valid w.r.t order-domain (B, A) iff there exists $b \in B$ and $a \in A$ such that

- $\mathsf{id}_{\mathsf{bid}}(t) = \mathsf{id}(b)$ *and* $\mathsf{id}_{\mathsf{ask}}(t) = \mathsf{id}(a)$.
- $\mathsf{price}(a) \leq \mathsf{price}(b)$ *(i.e., b and a are tradable)* .
- $\mathsf{qty}(t) \leq \min(\mathsf{qty}(b), \mathsf{qty}(a))$.

We say that a set of transactions T is valid over an order-domain (B, A) if each transaction in T is valid over (B, A).

```
Definition over (t : transaction)(B A : list order):=
exists b a, (In a A)/\(In b B)/\(idb t = id b)/\(ida t = id a).

Definition valid (t : transaction)(B A : list order):=
exists b a, (In a A)/\(In b B)/\
(idb t = id b)/\(ida t = id a)/\(tradable b a)/\
(tquantity t <= oquantity b)/\(tquantity t <= oquantity a).

Definition Tvalid (T : list transaction)(B A : list order):=
forall t, (In t T) -> (valid t B A).
```

Given a set of transactions, we would like to extract out the partial bids or the asks that got traded. Observe that to extract out these orders, we would need to determine the price and the timestamp of those orders, which is not available in the transactions; these sets can be determined if we know the underlying order-domain which gives rise to the transactions. Thus, we define the functions Bids and Asks as follows.

Definition 8 (Bids, Asks)**.** *Given a set of transactions T over an admissible order-domain (B, A) we define* $\mathsf{Bids}(T, B)$ *and* $\mathsf{Asks}(T, A)$ *as follows.* $\mathsf{Bids}(T, B)$ *is the set of all the bids participating in T where the quantity of a bid b is set to the total traded quantity of b in T. Formally,*

$$\mathsf{Bids}(T, B) := \{(id, \mathsf{timestamp}(B, id), \mathsf{Qty}_{bid}(T, id), \mathsf{price}(B, id)) \mid id \in \mathsf{ids}_{\mathsf{bid}}(T)\}.$$

Similarly,

$$\mathsf{Asks}(T, A) := \{(id, \mathsf{timestamp}(A, id), \mathsf{Qty}_{ask}(T, id), \mathsf{price}(A, id)) \mid id \in \mathsf{ids}_{\mathsf{ask}}(T)\}.$$

We often simply write $\mathsf{Bids}(T)$ *and* $\mathsf{Asks}(T)$ *instead of* $\mathsf{Bids}(T, B)$ *and* $\mathsf{Asks}(T, A)$ *whenever B and A are clear from the context.*

```
(*Definitions of Bids*)
Fixpoint bids_aux (T: list transaction)(B:list order)
(Bi :list nat):(list order).
refine ( match Bi with
  |nil => nil
  |i::Bi' => match (Compare_dec.lt_dec (Qty_bid T i) 1) with
    |left _ => bids_aux T B Bi'
    |right _ => (Mk_order
                i (timestamp B i) (Qty_bid T i ) (price B i) _ )
                :: (bids_aux T B Bi')
 end
 end). rewrite PeanoNat.Nat.ltb_nlt. auto.
 Defined.

Definition bids (T: list transaction)(B:list order):=
uniq (bids_aux T B (ids B)).

Definition Bids
(B:list order)(T: list transaction)(B': list order):=
subset (ids_bid_aux T) (ids B') ->
(forall b, In b B -> (exists t, (In t T)/\(idb t = id b)/\
(oquantity b = Qty_bid T (id b))/\
(exists b', (In b' B')/\(id b = id b')/\
(otime b = otime b')/\(oprice b = oprice b'))))/\
(subset (ids_bid_aux T) (ids B))/\(NoDup B).
```

Finally, we define a matching, which is a set of transactions that can simultaneously arise from an order-domain. We also define the canonical form of a set of transactions, which will be often applied to matchings.

Definition 9 (Matching, Canonical form)**.** *We say a set of valid transactions M over an order-domain (B, A) is a matching over (B, A) iff $\forall b \in B$, $\mathsf{qty}(b) \geq \mathsf{Qty}(M, \mathsf{id}(b))$ and $\forall a \in A$, $\mathsf{qty}(a) \geq \mathsf{Qty}(M, \mathsf{id}(a))$.*

We define the canonical form of a set of transactions M, denoted by $\mathcal{C}(M)$, as follows.

$$\mathcal{C}(M) := \Bigg\{ (id_b, id_a, q) \mid \exists m \in M \ \ s.t. \ \ \mathsf{id}_{\mathsf{bid}}(m) = id_b, \mathsf{id}_{\mathsf{ask}}(m) = \\ id_a \ \text{ and } q = \sum_{\substack{m: \\ \mathsf{id}_{\mathsf{bid}}(m) = id_b, \\ \mathsf{id}_{\mathsf{ask}}(m) = id_a}} \mathsf{qty}(m) \Bigg\}.$$

Note that in a canonical form, for each participating bid and ask pair, there is a unique transaction.

```
Definition Matching (M: list transaction)(B A: list order):=
  (Tvalid M B A)/\
  (forall b: order, In b B ->
                  (Qty_bid M (id b)) <= (oquantity b))/\
  (forall a: order, In a A->
                  (Qty_ask M (id a)) <= (oquantity a)).
```

```
(*Definitions of canonical form*)
Fixpoint cform_aux (T: list transaction)(Bi Ai :list nat):
(list transaction).
```

```
refine ( match (Bi,Ai) with
    |(nil, _) => nil
    |(_, nil) => nil
    |(i::Bi', j::Ai') =>
      match (Compare_dec.lt_dec (Qty T i j) 1) with
        |left _ => cform_aux T Bi' Ai'
        |right _ => (Mk_transaction i j (Qty T i j) _ )::
                    (cform_aux T Bi' Ai')
 end
 end). rewrite PeanoNat.Nat.ltb_nlt. auto.
 Defined.

Definition cform (M: list transaction):(list transaction) :=
uniq (cform_aux M (ids_bid_aux M) (ids_ask_aux M)).

Definition CanonicalForm (M M': list transaction):=
(***M is canForm of M'***)
(forall m, In m M -> (Qty M' (idb m) (ida m) = tquantity m)/\
(exists m', (In m' M')/\(idb m = idb m')/\(ida m = ida m')))/\
(forall m', In m' M' ->
exists m, (In m M)/\(idb m' = idb m)/\(ida m' = ida m)/\
(Qty M' (idb m) (ida m) = tquantity m))/\(NoDup M).
```

2.3 Order-book and process

We now formally define instructions and order-book.

Definition 10 (command, instruction, order-book)**.** *Buy, Sell and Del are called commands.*

An instruction is a pair (Δ, ω) where Δ is a command and ω is an order. For convenience we represent (Δ, ω) instruction by Δ ω. Also sometimes we represent (Del, ω) instruction simply by Del $\mathsf{id}(\omega)$; this is done because for a Del command, only the id of the order matters.

An order-book is a list where each entry is an instruction.

```
(*Definition of command*)
Inductive command:Set:=
  |buy
  |sell
  |del.
```

```
(*Definition of instruction*)
Record instruction :Type:=
Mk_instruction {cmd : command; ord : order}.
```

We need to impose two technical conditions that an order-book should meet. Firstly, the timestamps of the orders must be increasing. Secondly, the ids of the orders in the non-Del instructions in the order-book must be all distinct. We will relax the second condition slightly that will help us in certain applications. We allow an order to have an id identical only to the id appearing in an immediately preceding Del order instruction. This will later help us in our application to implement an 'update' instruction, by replacing it with a delete instruction followed with a Buy or Sell instruction carrying the same id. We now formally define a 'structured' order-book that formally captures these conditions.

Definition 11 (structured order-book)**.** *An order-book* $\mathcal{I} = [(\Delta_0, \omega_0), \cdots, (\Delta_n, \omega_n)]$ *is called structured if the following conditions hold.*

- *For all* $i \in \{0, \cdots, n-1\}$, $\mathsf{timestamp}(\omega_i) < \mathsf{timestamp}(\omega_{i+1})$.
- *For all* $i \in \{0, 1, \cdots, n\}$*, at least one of the following three conditions hold.*
  - $\Delta_i = \mathsf{Del}$.
  - $\mathsf{id}(\omega_i) \notin \mathsf{ids}\{\omega_0, \cdots, \omega_{i-1}\}$.
  - $\mathsf{id}(\omega_i) = \mathsf{id}(\omega_{i-1})$ *and* $\Delta_{i-1} = \mathsf{Del}$.

```
Definition structured (I :list instruction):=
(forall t:nat,  (t+1) <= (length I) ->
(cmd (nth t I tau0) = del)\/
(~In (id (ord (nth t I tau0))) (ids (tilln (orders I) (t-1))))\/
((id (ord (nth t I tau0))) = (id (ord (nth (t-1) I tau0)))/\
(cmd (nth (t-1) I tau0) = del)))/\
Sorted (Nat.ltb) (timesof (orders I))/\
NoDup ((timesof (orders I))).
```

We now define a process which represents an abstract online algorithm that will be fed resident bids and asks and an instruction, and it will output a set of transactions and resulting resident bids and asks.

Definition 12 (process)**.** *A process is a function* $(B, A, \tau) \mapsto (B', A', M)$ *that takes as input sets of orders* B*,* A *and an instruction* τ *and outputs sets of orders* B'*,* A' *and a set of transactions* M*.*

We do not need to explicitly define process in Coq. It can be simply described by its type.

```
(*P is a process*)
(P :(list order)->(list order) -> instruction ->
(list order)*(list order)*(list transaction))
```

The input to a process is an order-domain, which represents the resident orders in the system, and an instruction. If this instruction, is a delete id instruction, then process is supposed to delete all resident orders with that id, and the 'effective' order-domain potentially gets reduced. Otherwise, if the instruction is a buy/sell order, then the 'effective' order-domain needs to include that order. We define Absorb that takes an order-domain and an instruction as input and outputs the 'effective' order-domain.

Definition 13 (Absorb)**.** *For an order-domain* (B, A) *and an instruction* τ *we define* $\mathsf{Absorb}(B, A, \tau) :=$

$$\begin{cases} (\{\beta \in B | \mathsf{id}(\beta) \neq id\}, \{\alpha \in A | \mathsf{id}(\alpha) \neq id\}) \text{ if } \tau = \textit{Del id} \\ (B \cup \{\beta\}, A) \text{ if } \tau = \textit{Buy } \beta \\ (B, A \cup \{\alpha\}) \text{ if } \tau = \textit{Sell } \alpha. \end{cases}$$

```
Definition Absorb (B A: list order)(tau: instruction):=
match (cmd tau) with
|buy => ((ord tau)::B, A)
|sell => (B, (ord tau)::A)
|del => (delete_order B (id (ord tau)), delete_order A (id (ord tau)))
end.
```

To a process, we will usually feed inputs that satisfy certain properties, and such inputs we refer to as legal-inputs and are defined as follows.

Definition 14 (legal-input)**.** *We say an order-domain (B, A) and an instruction τ forms a legal-input if B and A are not matchable and τ is such that $(B', A') = \mathsf{Absorb}(B, A, \tau)$ is an admissible order-domain.*

```
Definition Legal_input (B A :list order)(tau: instruction):=
admissible (fst (Absorb B A tau)) (snd (Absorb B A tau))/\not (matchable B A).
```

Finally, we define Iterated.

Definition 15 (Iterated)**.** *Given a process P, an order-book $\mathcal{I}$ and a natural number k, we define $\mathsf{Iterated}(P, \mathcal{I}, k)$ to be the output of P at time k when it is iteratively run on the order-book $\mathcal{I}$. When $k > \mathsf{length}(\mathcal{I})$, $\mathsf{Iterated}(P, \mathcal{I}, k)$ returns $(\emptyset, \emptyset, \emptyset)$. Otherwise, $\mathsf{Iterated}(P, \mathcal{I}, k)$ can be computed recursively as per the following algorithm.*

Algorithm 1 Iteratively running a process on an order-book

```
function Iterated(Process $P$, Order-book $\mathcal{I}$, natural number $k$)
    if $k = 0$ or $k > \mathsf{length}(\mathcal{I})$ then return $(\emptyset, \emptyset, \emptyset)$
    $(B, A, M) \leftarrow \mathsf{Iterated}(P, \mathcal{I}, k-1)$
  ▷ $B$ & $A$ are resident orders & $M$ is the matching outputted at time $k-1$.
    $\tau \leftarrow k^{\text{th}}$ instruction in $\mathcal{I}$
    return $P(B, A, \tau)$
```

```
(*Definition of iterated*)
Fixpoint iterate
(P: (list order)->(list order) -> instruction ->
(list order)*(list order)*(list transaction))
(I : list instruction)(k:nat) :=
match k with
  |0 => (nil, nil, nil)
  | S k' => let it:=(iterate P I k') in
               P (Blist it) (Alist it) (nth k' I tau0)
end.

Definition iterated
(P: (list order)->(list order) -> instruction ->
(list order)*(list order)*(list transaction))
(I : list instruction)(k:nat) :=
if (Nat.ltb (length I) k) then (nil,nil,nil) else iterate P I k.
```

2.4 Three natural properties of a process

Having setup the definitions, we will now state the three natural properties formally.

We say a process P satisfies `positive bid-ask spread`, `price-time priority`, and `conservation` if for all order-domains (B, A) and an instruction τ such that (B, A) and τ forms a legal-input, $P(B, A, \tau) = (\hat{B}, \hat{A}, M)$ and $(B', A') = \mathsf{Absorb}(B, A, \tau)$ implies the following three conditions.

1. `Positive Bid-Ask Spread`: $\hat{B}$ and $\hat{A}$ are not matchable.

```
Definition Condition1 (M: list transaction)
(B A hat_B hat_A: list order) (tau: instruction):
Prop:= not (matchable hat_B hat_A).
```

2. `Price-Time Priority`: If a less competitive order ω is being traded in M, then all orders that are more competitive than ω must be fully traded in M. Formally,

   a. $\forall a, a' \in A',\ a \succ a'$ and $\mathsf{id}(a') \in \mathsf{ids}_{\mathsf{ask}}(M) \implies \mathsf{Qty}(M, id(a)) = \mathsf{qty}(a)$

   b. $\forall b, b' \in B',\ b \succ b'$ and $\mathsf{id}(b') \in \mathsf{ids}_{\mathsf{bid}}(M) \implies \mathsf{Qty}(M, id(b)) = \mathsf{qty}(b)$.

```
Definition Condition2a (M: list transaction)(B:list order):Prop:=
forall b b', (In b B)/\(In b' B)/\
(bcompetitive b b'/\~eqcompetitive b b')/\
(In (id b') (ids_bid_aux M)) ->
(Qty_bid M (id b)) = (oquantity b).

Definition Condition2b (M: list transaction)(A:list order):Prop:=
forall a a', (In a A)/\(In a' A)/\
(acompetitive a a'/\~eqcompetitive a a')/\
(In (id a') (ids_ask_aux M)) ->
(Qty_ask M (id a)) = (oquantity a).
```

3. `Conservation`: P does not lose or add orders arbitrarily. For this we have the following technical conditions.

   a. M is matching over the order-domain (B', A')

   b. $\hat{B} = B' - \mathsf{Bids}(M, B')$

   c. $\hat{A} = A' - \mathsf{Asks}(M, A')$.

```
Definition Condition3a (M: list transaction)
(B A: list order) (tau: instruction):Prop:=
let B' := (fst (Absorb B A tau)) in
let A' := (snd (Absorb B A tau)) in
Matching M B' A'.

Definition Condition3b (M: list transaction)
(B A hat_B: list order) (tau: instruction):Prop:=
let B' := (fst (Absorb B A tau)) in
hat_B === (odiff B' (bids M B')).

Definition Condition3c (M: list transaction)
(B A hat_A: list order) (tau: instruction):Prop:=
let A' := (snd (Absorb B A tau)) in
hat_A === (odiff A' (asks M A')).
```

In Coq, we define the proposition Properties as follows. If a process P satisfies the above three conditions, then and only then 'Properties P' is True.

```
Definition Properties (Process: (list order) ->(list order) ->
instruction -> (list order)*(list order)*(list transaction)):=
forall A B tau, Legal_input B A tau ->
let B' := (fst (Absorb B A tau)) in
let A' := (snd (Absorb B A tau)) in
let hat_B := (Blist (Process B A tau)) in
let hat_A := (Alist (Process B A tau)) in
let M := (Mlist (Process B A tau)) in
Condition1 M B A hat_B hat_A tau /\
Condition2a M B'/\ Condition2b M A'/\
Condition3a M B A tau /\
Condition3b M B A hat_B tau /\
Condition3c M B A hat_A tau.
```

In the above we use Blist, Alist, and Mlist, which are defined as follows.

```
Definition Blist
(p: (list order)*(list order)*(list transaction)) :=
match p with (x, y,z) => x end.

Definition Alist
(p: (list order)*(list order)*(list transaction)) :=
match p with (x, y,z) => y end.

Definition Mlist
(p: (list order)*(list order)*(list transaction)) :=
match p with (x, y,z) => z end.
```

3 Verified algorithm

Here we introduce a natural algorithm for continuous double auctions.

Algorithm 2 Process for continuous market

```
function PROCESS_INSTRUCTION(Bids $B$, Asks $A$, Instruction $\tau$)
    if $\tau =$ Del $id$ then Del_order$(B, A, id)$
    if $\tau =$ Buy $\beta$ then Match_bid$(B, A, \beta)$
    if $\tau =$ Sell $\alpha$ then Match_ask$(B, A, \alpha)$
```

In the Coq formalization of Process_instruction, we sort the list of asks and bids by their competitiveness before calling a subroutine; as a result, the most competitive bid and ask are on top of their respective lists.

```
Definition Process_instruction (B A:list order)(tau: instruction):
((list order)*(list order)*(list transaction)):=
match (cmd tau) with
    |del => Del_order B A (id (ord tau))
    |buy => Match_bid B (sort acompetitive A) (ord tau)
    |sell => Match_ask (sort bcompetitive B) A (ord tau)
end.
```

Algorithm 3 Matching an ask

```
function MATCH_ASK(Bids $B$, Asks $A$, order $\alpha$)                ▷ $\alpha$ is an ask.
    if $B = \emptyset$ then return $(B, A \cup \{\alpha\}, \emptyset)$
    $\beta \leftarrow \mathsf{Extract\_most\_competitive}(B)$                ▷ Note: $B \leftarrow B \setminus \{\beta\}$.
    if $\mathsf{price}(\beta) < \mathsf{price}(\alpha)$ then return $(B \cup \{\beta\}, A \cup \{\alpha\}, \emptyset)$
                                                  ▷ From now on $\beta$ and $\alpha$ are tradable.
    if $\mathsf{qty}(\beta) = \mathsf{qty}(\alpha)$ then $m \leftarrow (\mathsf{id}(\beta), \mathsf{id}(\alpha), \mathsf{qty}(\alpha))$
                        return $(B, A, \{m\})$
    if $\mathsf{qty}(\beta) > \mathsf{qty}(\alpha)$ then $m \leftarrow (\mathsf{id}(\beta), \mathsf{id}(\alpha), \mathsf{qty}(\alpha))$
                        $B' \leftarrow B \cup \{(\mathsf{id}(\beta), \mathsf{timestamp}(\beta), \mathsf{qty}(\beta) - \mathsf{qty}(\alpha), \mathsf{price}(\beta))\}$
                        return $(B', A, \{m\})$
    if $\mathsf{qty}(\beta) < \mathsf{qty}(\alpha)$ then $m \leftarrow (\mathsf{id}(\beta), \mathsf{id}(\alpha), \mathsf{qty}(\beta))$
                        $\alpha' \leftarrow (\mathsf{id}(\alpha), \mathsf{timestamp}(\alpha), \mathsf{qty}(\alpha) - \mathsf{qty}(\beta), \mathsf{price}(\alpha))$
                        $(B', A', M') \leftarrow \mathsf{Match\_ask}(B, A, \alpha')$
                        $M \leftarrow M' \cup \{m\}$
                        return $(B', A', M)$
```

```
(*Definition of Match_ask(B,A,a)*)
Fixpoint Match_ask (B A: list order)(a :order):
((list order)*(list order)*(list transaction)).

destruct B as [|b B']. (*b is most competitive bid*)
exact (nil, a::A, nil). (*When B is empty*)

(*From now on B is not empty*)
destruct (oprice a - oprice b).

(*First case: price a <= price b, i.e., b & a tradable*)

(*destruct ((oquantity a) - (oquantity b)).*)
refine ( match
(Compare_dec.lt_eq_lt_dec (oquantity b) (oquantity a))  with

(*Subcase: qty b=qty a*)
 |inleft (right _) =>
 (B', A, ((Mk_transaction (id b) (id a) (oquantity a)
                            (oquantity_cond a))::nil))

(*Subcase qty b>qty a*)
 |inright _ =>
 ((Mk_order (id b) (otime b)
   ((oquantity b) - (oquantity a)) (oprice b) _)::B', A,
   ((Mk_transaction (id b) (id a) (oquantity a)
   (oquantity_cond a))::nil))

 (*Subcase: qty b < qty a*)
  |inleft (left _) =>
  let BAM := (Match_ask B' A (Mk_order (id a) (otime a)
  ((oquantity a) - (oquantity b)) (oprice a) _ )) in
  (Blist BAM, Alist BAM, (Mk_transaction (id b) (id a)
  (oquantity b) (oquantity_cond b))::(Mlist BAM))
 end ).


rewrite PeanoNat.Nat.ltb_nlt; apply liaforrun;auto.
rewrite PeanoNat.Nat.ltb_nlt; apply liaforrun;auto.

(*Second case: price a > price b, i.e., b & a not tradable*)
exact (b::B', a::A, nil). Defined.
```

The Match_Bid subroutine is symmetric to the Match_Ask subroutine and we do not present it explicitly here.

Algorithm 4 Deleting an order

```
function DEL_ORDER(B, A, id)
    if $id \in \mathsf{ids}(B)$ then $B \leftarrow \mathsf{remove}(B, id)$
    if $id \in \mathsf{ids}(A)$ then $A \leftarrow \mathsf{remove}(A, id)$
    return $(B, A, \emptyset)$
```

```
Definition Del_order (B A:list order)(i:nat):
((list order)*(list order)*(list transaction)):=
(delete_order B i, delete_order A i, nil).
```

4 Lemmas and Theorems

In the following Coq statements, '$A === B$' for lists A and B represents that list A is a permutation of list B. If we think of A and B as sets, then '$A === B$', translates to set equality $A = B$.

4.1 Maximum matching

Lemma 1 (Maximum matching)**.** *Let P be a process that satisfies `positive bid-ask spread` and `conservation`. For all order-domain and instruction pairs $((B, A), \tau)$ that form legal-inputs, if $P(B, A, \tau) = (\hat{B}, \hat{A}, M)$, then for all matchings M' over $\mathsf{Absorb}(B, A, \tau)$, $\mathsf{Vol}(M) \geq \mathsf{Vol}(M')$.*

```
Definition MaxMatch (M: list transaction) (B A: list order):=
forall M', Matching M' B A -> Matching M B A ->
(Vol M') <= (Vol M).
```

```
Theorem Maximum_Maching
(Process: list order->list order -> instruction ->
(list order)*(list order)*(list transaction))
(B A hat_B hat_A : list order)(tau:instruction):

let B' := (fst (Absorb B A tau)) in
let A' := (snd (Absorb B A tau)) in

let hat_B := (Blist (Process B A tau)) in
let hat_A := (Alist (Process B A tau)) in

let M := (Mlist (Process B A tau)) in

Condition1 M B A hat_B hat_A tau->
Condition3a M B A tau ->
Condition3b M B A hat_B tau ->
Condition3c M B A hat_A tau ->

not (matchable B A) ->
NoDup (ids B') -> NoDup (ids A') ->

MaxMatch (Mlist (Process B A tau)) B' A'.
```

4.2 Local Uniqueness

Theorem 1. *Let P_1 and P_2 be processes that satisfy* ***`price-time priority`****,* ***`positive bid-ask spread`****, and* ***`conservation`****. For all order-domain instruction pairs $((B, A), (\Delta, \omega))$ that form legal-inputs, if for each $i \in \{1, 2\}$, $P_i(B, A, (\Delta, \omega)) = (\hat{B}_i, \hat{A}_i, M_i)$, then $(\hat{B}_1, \hat{A}_1, \mathcal{C}(M_1)) = (\hat{B}_2, \hat{A}_2, \mathcal{C}(M_2))$. Furthermore, the following statements hold for each i.*
(1) $\hat{B}_i$ and $\hat{A}_i$ are not matchable.
(2) The timestamps of orders in $\hat{B}_i$ and $\hat{A}_i$ are distinct and form a subset of the set of timestamps of the orders in $B \cup A \cup \{\mathsf{timestamp}(\omega)\}$.
(3) The ids of orders in $\hat{B}_i$ and $\hat{A}_i$ are distinct and form a subset of $\mathsf{ids}(B \cup A) \cup \{\mathsf{id}(\omega)\}$.
(4) $\Delta = \mathsf{Del} \implies \mathsf{id}(\omega) \notin \mathsf{ids}(\hat{B}_i \cup \hat{A}_i)$.

```
Theorem Local_uniqueness
(Process1 Process2: list order->list order -> instruction->
(list order)*(list order)*(list transaction))

(B1 B2 A1 A2 : list order)(tau:instruction):

B1 === B2 -> A1 === A2 -> Legal_input B1 A1 tau ->
Properties Process1 -> Properties Process2 ->

(cform (Mlist (Process1 B1 A1 tau))) ===
(cform (Mlist (Process2 B2 A2 tau)))

/\

(Blist (Process1 B1 A1 tau)) === (Blist (Process2 B2 A2 tau))

/\

(Alist (Process1 B1 A1 tau)) === (Alist (Process2 B2 A2 tau))

/\

(*Properties (1) - (4)*)
(included (timesof (Blist (Process1 B1 A1 tau)))
(timesof ((ord tau)::B1)))/\
(included (timesof (Alist (Process1 B1 A1 tau)))
(timesof ((ord tau)::A1)))/\

(not (matchable (Blist (Process1 B1 A1 tau))
(Alist (Process1 B1 A1 tau))))/\
admissible (Blist (Process1 B1 A1 tau))
(Alist (Process1 B1 A1 tau))/\

(included (ids (Blist (Process1 B1 A1 tau)))
(ids ((ord tau)::B1)))/\
(included (ids (Alist (Process1 B1 A1 tau)))
(ids ((ord tau)::A1)))/\

((cmd tau) = del ->
~ In (id (ord tau)) (ids (Blist (Process1 B1 A1 tau))))/\
((cmd tau) = del ->
~ In (id (ord tau)) (ids (Alist (Process1 B1 A1 tau)))).
```

4.3 Global uniqueness

Theorem 2. *Let P_1 and P_2 be processes that satisfy `positive bid-ask spread`, `price- time parity` and `conservation`. Then, for all structured order-books $\mathcal{I}$ and natural numbers k if $\mathsf{Iterated}(P_1, \mathcal{I}, k) = (B_1, A_1, M_1)$ and $\mathsf{Iterated}(P_2, \mathcal{I}, k) = (B_2, A_2, M_2)$, then $(B_1, A_1, C(M_1)) = (B_2, A_2, C(M_2))$.*

```
Theorem global_unique (P1 P2 :(list order)->(list order) ->
instruction -> (list order)*(list order)*(list transaction))

(I : list instruction)(k:nat):

(Properties P1) /\(Properties P2) /\ structured I  ->

(cform (Mlist (iterated P1 I k))) ===
(cform (Mlist (iterated P2 I k)))

/\

(Blist (iterated P1 I k)) === (Blist (iterated P2 I k))

/\

(Alist (iterated P1 I k)) === (Alist (iterated P2 I k)).
```

4.4 Process_instruction has the desired properties

Theorem 3. *Process_instruction satisfies `positive bid-ask spread`, `price- time priority`, and `conservation`.*

```
Theorem Process_correct :

Properties Process_instruction.
```