Efficient and Verified Continuous Double Auctions

Mohit Garg and Suneel Sarswat

June 5, 2023

Efficient and Verified Continuous Double Auctions [*]

Mohit Garg[1†]and Suneel Sarswat[2]

[1] Indian Institute of Science, Bengaluru, India
mohitgarg@iisc.ac.in
[2] Tata Institute of Fundamental Research, Mumbai, India
suneel.sarswat@gmail.com

Abstract

Continuous double auctions are commonly used to match orders at currency, stock, and commodities exchanges. A verified implementation of continuous double auctions is a useful tool for market regulators as they give rise to automated checkers that are guaranteed to detect errors in the trade logs of an existing exchange if they contain trades that violate the matching rules. We provide an efficient and formally verified implementation of continuous double auctions that takes $O(n \log n)$ time to match n orders. This improves an earlier $O(n^2)$ verified implementation. We also prove a matching $\Omega(n \log n)$ lower bound on the running time for continuous double auctions. Our new implementation takes only a couple of minutes to run on ten million randomly generated orders as opposed to a few days taken by the earlier implementation. Our new implementation gives rise to an efficient automatic checker.

We use the Coq proof assistant for verifying our implementation and extracting a verified OCaml program. While using Coq's standard library implementation of red-black trees to obtain our improvement, we observed that its specification has serious gaps, which we fill in this work; this might be of independent interest.

1 Introduction

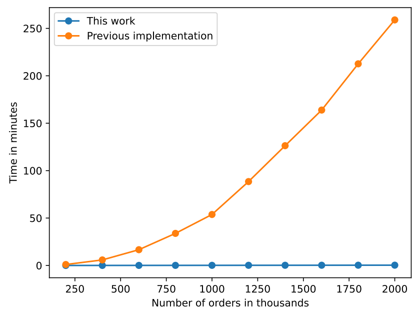
Continuous double auctions are used to match buy and sell orders for a particular product at an exchange. For example, they are used at currency, stock, and commodities exchanges. These exchanges generally use computer software to match orders and are required to adhere to regulatory directives that ensure fairness, safety, and transparency. There are multiple reported incidents where exchanges were found violating the regulatory directives or the stated rules [16, 15, 14, 11]. Bugs in the exchange program can trigger undesirable events leading to significant losses.

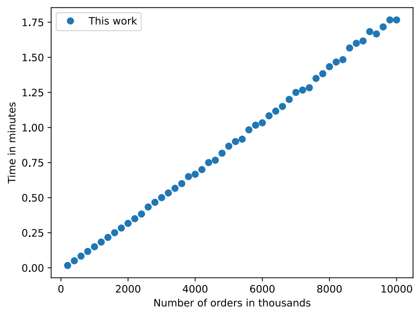
These problems have received attention from the formalization community and form an active research area [9, 10, 7, 8, 6, 4]. In [6], a general model of continuous double auctions is considered, and three simple properties of continuous double auctions are identified and shown to be sufficient to uniquely determine the input-output relation, thus yielding formal specifications for such auctions. A natural algorithm for continuous double auctions is then verified in the Coq theorem prover. As a novel application, the verified program obtained is used to build an automated checker that goes over the trade logs of an exchange, comparing the matchings generated by the exchange against the matchings produced by the verified program, and reporting any mismatches. Given that the specifications ensure a unique input-output

[*]Presented at 24th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-24) [5].

[†]Supported by SERB Core Research Grant (CRG/2022/001176) on "Optimization under Intractability and Uncertainty". A part of this work was done while the author was affiliated with the University of Bremen, Germany.

relation, a mismatch would mean that the exchange program violates the specifications. Such verified programs and accompanying checkers can be extremely useful to the exchanges and the regulators.





The checker obtained in [6] runs reasonably fast on tens of thousands of orders taking only a few seconds and can be useful for products that have a reasonable trade frequency. On the contrary, for products that are traded much more frequently, say receiving ten million orders a day, the checker would take a few days to detect violations, limiting its applicability. The implementation uses the List data structure resulting in an $O(n^2)$-time implementation for n orders.

In this work, we drastically improve the running time by producing an $O(n \log n)$-time implementation of continuous double auctions using Red-black trees instead of Lists. We formally prove in the Coq proof assistant that the previous verified implementation and our implementation will produce exactly the same output, thus showing adherence to the specifications. Furthermore, we prove that any algorithm must take $\Omega(n \log n)$ time showing that the running time of our implementation is asymptotically tight. We run the two implementations on randomly generated data and report the running times. In particular, for ten million orders our implementation takes less than two minutes to run, at least a thousandfold improvement.

Our new implementation, like the previous one, gives rise to an automated checker, which is guaranteed to detect any violation of the specifications from the trade logs if there exists one.

Organization of the rest of the paper. In Section 2, we provide a background of continuous double auctions, that we later use to build on. In Section 3, we state the results that we obtain in this work. In Section 4 we describe the natural algorithm for continuous double auctions with an emphasis on the running time of the previous implementation and the new implementation. In Section 5 we describe our improved implementation. In Section 6 we show our implementation is efficient and compare the running time of the new implementation with the previous one. Our Coq formalization and experimental demonstrations are available on [2].

2 Background

We now describe the model of an exchange where a fixed product (for example, a particular stock of a company) is traded. The exchange receives instructions from the traders (buyers and sellers) of the product. Each instruction comprises of a command and an order. Buy, Sell, and Delete are the possible commands. An order $w = (i, t, p, q)$ accompanying a Buy or Sell command comprises of a unique identification number i, a unique timestamp t, a limit price p, and a maximum quantity q. A Buy (i, t, p, q) instruction from a trader, say John, means that at time t John requests the exchange to buy q units of the product, and he has a budget of p (cents) per unit of the product. This instruction from John is assigned a unique id i by the exchange. Similarly, a Sell (i, t, p, q) instruction from Mary means that at time t she requests the exchange to sell q units of the product for at least p (cents) per unit. Her Sell order is assigned the id i by the exchange. Orders accompanying Delete commands consist of an id and timestamp, but no price or quantity: $(i, t, *, *)$. Buy and Sell orders are traditionally referred to as bids and asks, respectively.

On receiving an instruction, say a bid Buy $w = (i, t, p, q)$ from John, the exchange instantly matches w with existing unmatched or partially matched asks that arrived before and are still in the system (resident asks) and have a limit price of at most p. If there is more than one ask that is matchable with w, they are prioritized based on price-time priority (first by the competitiveness of their price and then by their timestamps). Transactions between w and these matchable asks are generated of the largest possible quantity, which is at most q. These transactions form a matching. If the matching is of total quantity q, John's order is completely exhausted and it leaves the system completely. Otherwise, if the size of the matching is $q' < q$, the system keeps the bid $(i, t, p, q - q')$ as a resident bid for future matches. A resident order leaves the system if its quantity gets exhausted by future matches or if it gets deleted by a Delete command. On receiving an ask, the behavior of the exchange is symmetric, where it matches the ask with the resident bids based on their price-time priority.

The exchange algorithm can be thought of as an online algorithm P that maintains a set of resident bids B and a set of resident asks A and gets one instruction at a time. On receiving a new instruction Command w, it generates a matching M and updates the sets of resident bids and asks to $\hat{B}$ and $\hat{A}$.

$$(B, A,\ \text{Command } w) \overset{P}{\mapsto} (\hat{B}, \hat{A}, M)$$

Given the above description of the exchange, it was observed in [6] that the following three properties must be satisfied by P.

- Positive bid-ask spread: The most competitive bid in $\hat{B}$ has a lower limit price than the most competitive ask in $\hat{A}$, i.e., no transaction is possible among resident orders.
- Price-time priority: If Command w is a bid: if w gets traded with an ask a', then each resident ask $a \in A$ which is more competitive than a' must get fully matched in M. A symmetric statement holds when Command w is an ask.
- Conservation: The system does honest arithmetic and does not modify orders arbitrarily. For example, if a bid $b = (i, t, p, q)$ gets $q' < q$ quantity traded with w, then $(i, t, p, q - q') \in \hat{B}$.

These properties are formally represented in Coq as shown below. We do not expect the reader to fully comprehend the following without going through the formal definitions of the various quantities that appear in the previous tool documentation [1]. In what follows, B' and A' are obtained from B and A by adding (or removing) w from B and A depending on whether the accompanying command is Buy or Sell (or delete).

```
not (matchable hat_B hat_A).

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).
```

```
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).
```

```
Matching M B' A'.
hat_B === (odiff B' (bids M B')).
hat_A === (odiff A' (asks M A')).
```

The main result in [6] can roughly be summarized as follows.

Theorem 1. *Let P_1 and P_2 be two online algorithms such that each of them satisfies the above three properties. Then, on the same list of instructions as input, at each point in time, P_1 and P_2 will generate the same matchings.*

Thus, the above three properties can be used as specifications for continuous double auctions.

A program is also provided in [6], namely Process_instruction, and the following theorem is formally proved.

Theorem 2. *Process_instruction satisfies the above three properties.*

The exchange maintains two logbooks. All the instructions received by the exchange are kept in an order book (sorted by their timestamps) and the corresponding matchings that get generated are maintained in a trade book. As mentioned earlier, a verified program such as Process_instruction can also be used to build automated checkers that detect violations in trade logs of existing exchanges in an offline mode (like at the end of the day of trading). In fact, the above theorems work in a slightly more general setting where the id of a buy or sell order can be equal to the id of an immediately preceding delete instruction. This allows one to implement Update instructions. Using this slightly general model, [6] implements a checker that can handle more complex orders like Updates, Immediate-or-Cancel orders, market orders, and stop-loss orders by adding a preprocessing step where complex instructions are converted into the three primitive types: Buy, Sell, and Delete.

Our results hold for this general setting, and all their results still apply in our faster implementation of Process_instruction.

3 Our contributions

We provide a new and efficient implementation of Process_instruction, which we call eProcess_instruction, which stores the set of resident bids and asks as red-black trees instead of lists. We expected changing lists to red-black trees in the previous implementation to be a straightforward task. But unfortunately, it was not so. While the new implementation needed little innovation (like keeping two trees with different keys instead of one, as we will see later), proving the correctness needed both conceptual and technical work. If one carefully observes the specifications for the online process obtained in the previous work [6], it is formulated in terms of lists. Thus, it is not directly possible to prove the correctness of the implementation that uses trees; one option would be to recast the specifications in terms of trees, but then one would need to prove that the two specifications are equivalent in some sense. Instead, and this is our conceptual innovation, we prove that for any order book, the new and old processes have precisely the same outputs at each point in time, piggybacking on the formal correctness of the previous implementation. Note that we do not prove that our algorithm satisfies the specification directly. Our proof needs to delicately factor in the definition of a 'structured' order book, which we explain later.

On the technical side, we found that the semantic guarantees provided for red-black trees in the standard library implementation of Coq have serious omissions, making the standard library implementation of red-black trees unsuitable for black-box use. Interestingly, we could not find other works using this standard library implementation. We work through the standard library implementation of red-black tees and prove the required guarantees. Note that, through this approach, the running time guarantees of the standard library implementation are thus retained. Others can benefit from our proofs when working with the standard library implementation of red-black trees.

To describe our main results more formally, we first need the following definition.

Definition 1 (cda_tree, cda_list)**.** *Given an order book I, and a natural number $k \leq \mathsf{length}(I)$, let* $\mathsf{cda\_list}(I, k)$ *denote the k^{th} matching output by Process_instruction, when it is run on the order book I, that is, the matching outputted when it processes the k^{th} instruction from I. Similarly,* $\mathsf{cda\_tree}(I, k)$ *represents the k^{th} matching outputted by eProcess_instruction when run on the order book I.*

We are now ready to state our main result.

Theorem 3. *For all order books I, and natural numbers $k \leq \mathsf{length}(I)$,*
$\mathsf{cda\_tree}(I, k) = \mathsf{cda\_list}(I, k)$.

This theorem appears in our formalization as follows.

```
Theorem identical_outputs (I : list instruction)(k:nat):
structured I  -> cda_list I k = cda_tree I k.
```

The condition that the order book I is structured captures the fact that the timestamps of the orders in I are increasing, and the ids of all bids and asks are distinct, except if it is preceded by a delete instruction, in which case its id could be the same as the id of the preceding delete instruction. As explained earlier, this relaxation allows one to implement more complex instruction types by converting them into the three primitive types, which is useful for certain exchanges.

eProcess_instruction satisfies the specifications is immediate by combining Theorems 3 and 2. Next, we state the time complexity results we obtain.

Theorem 4. *eProcess_instruction when run on an order book of length n takes $O(n \log n)$ time.*

Theorem 5. *Any algorithm that implements continuous double auctions has running time $\Omega(n \log n)$, where n is the length of the order book.*

Apart from the above results, we add several lemmas that strengthen the guarantees provided in Coq's standard library for red-black trees. This contribution is explained in detail in Section 5.1.

Our formalization of Theorem 3 builds on the earlier formalization of [6]. We add about 1500 lines of new Coq code with about 100 new lemmas, theorems, and definitions. We use Coq's program extraction feature to obtain an Ocaml program for eProcess_instruction. We use Coq 8.12.2 [13] for compiling our code. We randomly generate order books of various sizes using a python script, run the extracted verified OCaml programs of eProcess_instruction and Process_instruction on it, and report the running times. We include the Coq formalization and the scripts that enable a demonstration of running the extracted programs on the randomly generated data as part of the accompanying materials [2].

4 Algorithm for continuous double auctions

We first describe the algorithm for Process_instruction as used in [6] and analyze its running time. The algorithm takes as input the set of resident bids B, the set of resident asks A, and an Instruction τ. Depending on whether τ is a Delete, Buy, or Sell instruction, an appropriate subroutine is called. In the implementation, B and A are kept as sorted lists.

Algorithm 1 Process for continuous double auction

```
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$(B, A, \alpha)$
```

Next, we present the Match_ask and Del_order subroutines as they appear in [6]. Match_bid is symmetric to Match_ask and we do not present it explicitly here.

Algorithm 2 Matching an ask

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

Match_ask takes as input the sets of resident bids and asks B and A (implemented as lists that are kept sorted based on competitiveness), and an ask α. Match_ask first extracts the most competitive bid β from B, which happens to be at the topmost element of the list B, as B is sorted (which takes $O(1)$ time). If the limit price of β is less than that of α, then α is not matchable with any of the orders in B. Thus, β is inserted back in B (this again takes $O(1)$ time, and B remains sorted) and α must become a resident ask, and is inserted in A, which is a sorted list. In the implementation, α is inserted in such a way that the resulting list remains sorted. This takes $O(|A|) = O(n)$ time (assuming there are at most n instructions in total, $|A| + |B| \leq n$) and $\Theta(n)$ time in the worst case (for example, when α needs to be inserted at the middle of the list). This already shows that Process_instruction's implementation takes at least $\Omega(n)$ time.

Else, β is matched with α. If the quantity of β is at least the quantity of α, then α gets completely traded with β. β with its remaining quantity if any is inserted back to B. This entire step takes $O(1)$ time.

Finally, if the quantity of β is less than α, then β gets completely matched with α. We recursively call match_ask with the remaining quantity of α with the resident bids and asks B and A (note that β is not in B anymore, as it was extracted out at the very beginning). The time taken in this entire step is $O(1)$ plus the time taken by the recursive call. The entire recursion can take at most $O(n)$ time, since the size of B decreases by 1 for each recursive call.

The Delete instruction simply searches the id in the resident orders and deletes all elements with that id (note that there cannot be more than one such item). This step also takes $O(n)$ time.

Algorithm 3 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 $(B, A, \emptyset)$
```

In total the running time of the list implementation for processing one instruction in $O(n)$. Thus, for processing n instructions, it will take $O(n^2)$ time. The main bottlenecks are inserting an order in a sorted list and deleting an order from the list. The recursive step in Match_ask also seems to take $O(n)$ time, but one can do a better analysis for multiple instructions to get an improved amortized running time.

The above online algorithm is used for just processing a single instruction. When we run an online algorithm P repeatedly on an order book I, the output matching generated after processing the k^{th} instruction is given by the following recursive program of [6].

Algorithm 4 Iteratively running a process on an order-book

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

The function cda_list defined earlier is then obtained by setting P to Process_instruction.

4.1 Improved implementation using balanced binary search trees

We next show how the above-mentioned bottlenecks can be removed by using balanced binary search trees (BSTs) to store the resident orders, where insertions and deletions take $O(\log n)$ time.

We store the resident bids and asks as BSTs. Note that in Match_ask the insertion of the ask α in A is done based on competitiveness, whereas in Del_order the deletion is done using the id. We do not know how to implement insertion based on one key (competitiveness) and deletion based on another key (id) in $O(\log n)$ time each in the same BST. Consequently, we will keep two BSTs for the same set of resident asks, one for competitiveness and one for id.

Similarly, we will have two BSTs for the resident bids. With this trick of keeping two trees for the same set of elements, we can bring down the cost of insertion and the cost of deletion from $O(n)$ to $O(\log n)$. However, the time to extract the most competitive order (for example β from B in the first step of Match_ask) increases from $O(1)$ to $O(\log n)$. But, this will not hurt the asymptotic running time.

We always keep the two BSTs for the resident asks (bids) the same. When inserting an order, we insert it in both the trees, which takes $O(\log n)$ time. When deleting an order given by a Delete id command, we first search for the order in the id tree. If we find an order with that id, we get the order's price and time and search for it in the other tree (whose key is determined by price and time), and then delete that order from both trees, which takes $O(\log n)$ time in total. Similarly, when extracting the most competitive ask, we first extract it from the tree that is ordered by competitiveness. Once the most competitive ask is extracted, we have its id, and then we can easily extract it from the id tree. In total, this takes $O(\log n)$ time.

We now prove that the running time of our implementation of the algorithm when it is repeatedly applied to n instructions one after the other is $O(n \log n)$.

Proof of Theorem 4. Since there are n instructions in total, the number of resident orders at any point is $|A| + |B| \leq n$. Thus, each insertion, deletion, and extracting the most competitive order takes $O(\log n)$ time.

First notice that there can be at most n Delete instructions, each taking $O(\log n)$ time, needing $O(n \log n)$ time in total.

Now, we will bound the running time for all Match_ask calls. A similar bound holds for Match_bid.

Match_ask calls can be triggered in two ways. Either the call is triggered by a Sell α instruction or it is a recursive call made inside another Match_ask call. In the latter case, observe that a most-competitive bid β gets completely exhausted and leaves the system while executing the calling Match_ask function. We will say this β triggers the recursive call of Match_ask. Note that a single bid can trigger at most one Match_ask recursive call.

Time taken to execute a Match_ask call (not counting the time taken for executing the recursive calls if any) is the time taken for extracting the most-competitive bid β and the time taken to insert back the modified β in B or the modified ask α in A, which in total is $O(\log n)$. As noted above, a Match_ask is either triggered by a Sell α instruction, or by a bid β. Since there are at most n instructions and at most n bids, the total time taken by all Match_ask calls put together is $O(n \log n)$. Note that it might happen that just a single instruction takes $O(n \log n)$ time by itself. But this cannot happen for every instruction.

Thus, Theorem 4 follows immediately. □

5 Implementation with red-black trees

Our implementation, namely eProcess_instruction, thus has the following type signature:

$$(B, B_{id}, A, A_{id}, \text{Command } w) \mapsto (\hat{B}, \hat{B}_{id}, \hat{A}, \hat{A}_{id}, M),$$

where B and B_{id} are the BSTs for the resident bids with competitiveness and id as the keys, respectively. Similarly, A and A_{id} are the BSTs for the resident asks. The output also contains the four trees for the resident bids and asks after the instruction command w is processed, and the outputted matching M, which is maintained as a list.

In our Coq implementation, eProcess_instruction is defined as follows.

```
Definition eProcess_instruction
(B: TB.t)(A: TA.t)(tau: instruction)(B_id A_id : T_id.t):
((TB.t)*(TA.t)*(T_id.t)*(T_id.t)*(list transaction)) :=
match (cmd tau) with
|del => EDel_order B A (id (ord tau)) B_id A_id
|buy => EMatch_bid B A (ord tau) B_id A_id
|sell => EMatch_ask B A (ord tau) B_id A_id
end.
```

Note that since the type signatures of Process_instruction and eProcess_instruction are different, we cannot formally show that they are semantically the same. Thus, we show that when we run them on an order book, they have exactly the same outputs at each point in time. For this, we define a version of Iterated for eProcess_instruction which does exactly the same thing as Iterated to obtain cda_tree and show that cda_tree and cda_list are semantically the same.

Remark 1. *We make use of the Equations plugin [12] for implementing EMatch_ask and EMatch_bid smoothly, which was not critical in the previous implementation. Since our new implementation uses red-black trees, and the remove operation on red-black trees in the standard library implementation is not structurally recursive, several additional proof obligations get generated. These obligations are mitigated by using Equations.*

5.1 Strengthening the standard library guarantees

For our BSTs, we use the standard library implementation of red-black trees [3], which implements insertion, extraction, and deletions in $O(\log n)$ time.

However, we could not use the implementation in a black-box manner. To illustrate this consider the following lemma included in the standard library.

```
Lemma add_spec s x y `{Ok s} :
InT y (add x s) <-> eq y x \/ InT y s.
```

Here, s is a red-black tree (Ok s asserts that), x and y are two elements, add is a function that inserts x in s and returns the resulting tree.

The above lemma states that an element y is *In* the tree after inserting an element x iff y was *Equal* to the element x or y was already *In* the tree before the insertion of x.

Here, *In* and *Equal* correspond to InT and eq in the above lemma, and they do not carry the usual meaning of in and equal. InT a s means that the key of a is in the tree s, and not that a is in s, unless the keys of the elements are the elements themselves.

For our application, the key and the element are not the same; for example, the key is just the id, whereas the element is the entire order.

For using red-black trees, it is easy to imagine situations where one needs the following guarantee. Assume there is an element x whose key is not in the current tree s. Then, x will be part of the tree (add x s). Furthermore, the other elements of (add x s), other than x, are precisely the elements of s. Such a guarantee is not currently available in the standard library.

One thus requires a stronger statement to prove the correctness of the insert operation add, which was missing in the standard library implementation of red-black trees. Similar issues are there with the specification lemmas for other operations like remove. Thus, in our formalization, we had to go through the implementation in detail to prove the stronger results needed in our application. In particular, we show the following where Intree y s means y is an element of the tree s.

```
Lemma add_spec_tree s x y `{Ok s} :
not (InT x s) -> Intree y (add x s) <-> x = y \/ Intree y s.
```

The above lemma captures the situation described above. If the key of x is not in the tree s, then an element y is part of (`add` x s) if and only if either x and y are the same elements or y is an element in the tree s.

Similarly, we strengthen specifications relating to the remove operation and the elements function which outputs a list. They are as follows. Our strengthened lemmas have a suffix of '_tree'.

```
Lemma remove_spec s x y `{Ok s} :
 InT y (remove x s) <-> InT y s /\ ~X.eq y x.

Lemma remove_spec_tree s x y `{Ok s} :
Intree y (remove x s) <-> Intree y s/\ ~X.eq x y.

Lemma elements_spec1 :
forall s x, InA X.eq x (elements s) <-> InT x s.

Lemma elements_spec_tree :
forall s x, List.In x (elements s) <-> Intree x s.
```

Such lemmas derived in our formalization could be useful for future works that use the standard library implementation of red-black trees.

6 Time complexity of continuous double auctions

We now show that continuous double auctions take $\Omega(n \log n)$ time for processing $2n$ orders, proving Theorem 5.

Proof of Theorem 5. We reduce the task of sorting n positive numbers $\{b_1, \cdots, b_n\}$ in decreasing order to continuous double auctions. Our order book consists of $2n$ orders each with quantity 1. The first n orders are bids, whereas the last n orders are asks. The i^{th} bid has a price $b_i > 0$. Each ask has a limit price of 0.

Observe that each pair of bid and ask is matchable. Since the quantity of each order is 1, there will be exactly n matchings produced each of quantity 1. The first matching will be produced at the $n + 1^{\text{th}}$ step, where the most-competitive bid will be picked out. At the next step the second most competitive bid is picked out and so on. Hence, the matchings are produced in the sorted order of competitiveness. Since continuous double auctions prioritize matchable orders by price first (and then by time), the trade book generated will be sorted by price.

We now simply use the folklore sorting lower bound of $\Omega(n \log n)$, to get our result. □

6.1 Experimental findings

We show the running time of Process_instruction (previous work) and eProcess_instruction (current work) on randomly generated datasets using a python script which we enclose in the accompanying materials. Our order books generated are of sizes going from 200 thousand to 10 million. We stopped running the previous implementation on very large datasets as it took an inordinate amount of time. The figures appearing on page 2 represent the following data graphically.

No. of orders	Time (previous)	Time (current)
200,000	1 min 01 s	2 s
400,000	5 min 51 s	4 s
600,000	16 min 43 s	6 s
800,000	33 min 51 s	8 s
1,000,000	53 min 47 s	10 s
1,200,000	1 hr 29 min	12 s
1,400,000	2 hr 07 min	14 s
1,600,000	2 hr 45 min	16 s
1,800,000	3 hr 33 min	18 s
2,000,000	4 hr 20 min	20 s
4,000,000		41 s
6,000,000		1 min 03 s
8,000,000		1 min 27 s
10,000,000		1 min 47 s

As part of the supplementary materials [2], we provide a demonstration that runs both the old and new implementations on two randomly generated datasets and reports the running times. One needs an OCaml compiler to be able to run this demonstration.

7 Conclusions

In this work, we provide an efficient and formally correct implementation for continuous double auctions. This has a drastic impact on the running time as demonstrated by our analysis and leads to fast checkers which are extremely useful for finding errors and monitoring existing exchanges.

Acknowledgement

We wish to thank Mohimenul Kabir of the National University of Singapore for presenting our work at the conference.

References

[1] Formal definitions, theorems, algorithm, and implementation. `https://github.com/suneel-sarswat/cda/blob/main/additional.pdf`, 2022.

[2] Efficient continuous double auction. `https://github.com/ganitsutra/ecda`, 2023.

[3] Andrew W Appel. Efficient verified red-black trees. *url: https://www.cs. princeton. edu/˜ appel/papers/redblack. pdf*, 2011.

[4] Iliano Cervesato, Sharjeel Khan, Giselle Reis, and Dragisa Žunić. Formalization of automated trading systems in a concurrent linear framework. In *Linearity-TLLA@FLoC*, volume 292 of *EPTCS*, pages 1–14, 2018.

[5] M Garg and S Sarswat. Efficient and Verified Continuous Double Auctions. In R. Piskac and A. Voronkov, editors, *Short Papers of the 24th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2023.

[6] Mohit Garg and Suneel Sarswat. The design and regulation of exchanges: A formal approach. In Anuj Dawar and Venkatesan Guruswami, editors, *42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2022, December 18-20, 2022, IIT Madras, Chennai, India*, volume 250 of *LIPIcs*, pages 39:1–39:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[7] Raja Natarajan, Suneel Sarswat, and Abhishek Kr Singh. Verified double sided auctions for financial markets. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 28:1–28:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[8] Grant Olney Passmore. Some lessons learned in the industrialization of formal methods for financial algorithms. In *International Symposium on Formal Methods*, pages 717–721. Springer, 2021.

[9] Grant Olney Passmore and Denis Ignatovich. Formal verification of financial algorithms. In *26th International Conference on Automated Deduction, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 26–41. Springer, 2017.

[10] Suneel Sarswat and Abhishek Kr Singh. Formally verified trades in financial markets. In *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*, volume 12531 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2020.

[11] Securities Exchange Board of India (SEBI). Order in the matter of NSE Colocation, Apr 30, 2019. Order in the matter of NSE Colocation.

[12] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in coq. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

[13] The Coq Development Team. The coq reference manual, release 8.12.2, December 11 2020.

[14] U.S. Securities and Exchange Commision (SEC). SEC Charges UBS Subsidiary With Disclosure Violations and Other Regulatory Failures in Operating Dark Pool. `https://www.sec.gov/news/pressrelease/2015-7.html`, July, 2015.

[15] U.S. Securities and Exchange Commision (SEC). NYSE to Pay US Dollar 14 Million Penalty for Multiple Violations. `https://www.sec.gov/news/press-release/2018-31`, March 6, 2018.

[16] U.S. Securities and Exchange Commision (SEC). SEC Charges NYSE for Repeated Failures to Operate in Accordance With Exchange Rules. `https://www.sec.gov/news/press-release/2014-87`, May 1, 2014.