Testing Continuous Double Auctions with a Constraint-based Oracle

Roberto Castañeda Lozano[1,2], Christian Schulte[1], and Lars Wahlberg[2]

[1] KTH – Royal Institute of Technology, Sweden
{rcas,cschulte}@kth.se
[2] Cinnober Financial Technology AB, Stockholm, Sweden
{roberto.castaneda,lars.wahlberg}@cinnober.com

Abstract. Computer trading systems are essential for today's financial markets where the trading systems' correctness is of paramount economical significance. Automated random testing is a useful technique to find bugs in these systems, but it requires an independent system to decide the correctness of the system under test (known as *oracle problem*). This paper introduces a constraint-based oracle for random testing of a real-world trading system. The oracle provides the expected results by generating and solving constraint models of the trading system's continuous double auction. Constraint programming is essential for the correctness of the test oracle as the logic for calculating trades can be mapped directly to constraint models. The paper shows that the generated constraint models can be solved efficiently. Most importantly, the approach is shown to be successful by finding errors in a deployed financial trading system and in its specification.

1 Introduction

A financial market is a system that allows buyers and sellers to trade and exchange items of value. Nowadays, all major financial markets use computer systems to support their trading activities [1]. *TRADExpress*, developed by Cinnober Financial Technology AB, is an example of such a system. *TRADExpress* is deployed in different markets around the world, such as the London Metal Exchange, Alpha Trading Systems, or the Hong Kong Mercantile Exchange.

The predominant trading mechanism in financial markets are *continuous double auctions* [2]. In a continuous double auction, trade orders are entered continuously and matched against each other as soon as their constraints are satisfied. The numerous trading strategies render the order matching of *TRADExpress* complex. At the same time, system failures can cause serious economic damage: according to Cinnober's risk assessment, a failure halting *TRADExpress* is estimated to incur losses in the order of several million US dollars.

Automated random testing, as a complement to more systematic testing techniques, is useful in finding bugs in trading systems [3]. However, it presents an inherent problem, commonly known as the *test oracle problem*: an independent system called *test oracle* is needed to automatically decide the correctness of the

output generated by the system under test. The usual goals for designing test oracles are correctness, completeness, accuracy, and low development cost [4].

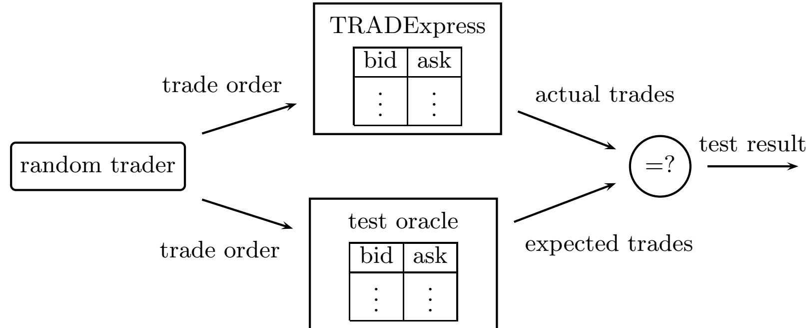

Fig. 1. Random test with the constraint-based test oracle

This paper uses constraint programming to solve the test oracle problem that arises for random testing of continuous double auctions. We present a constraint model of the *TRADExpress*' continuous double auction derived from a set of informal system requirements, and use the model as a test oracle for automated random tests as shown in Fig. 1. The relevance of our approach is illustrated by the number of defects found in the requirements during the modeling process and failures detected during the execution of the random tests.

Related work. Constraint programming has typically been used for automatic test data generation from formal specifications of the system under test [5, 6]. Nevertheless, the case where formal specifications are not available has not been thoroughly explored by the constraint programming community, apart from some efforts in the field of hardware testing [7]. In the area of financial services, the test oracle problem has typically been approached by developing simple failure detection mechanisms [3]. The lack of more complete test oracles has prevented the large-scale application of random testing to trading systems. To the best of our knowledge, there are no previous attempts to design a test oracle that covers the complete functionality of a real-world trading system.

The study of continuous double auctions for other purposes than testing and verification has attracted interest from different areas. Several articles present theoretical models formulated using constraint programming [8] or closely related techniques such as integer programming [9]. However, aspects like order priority, which are common issues in most existent trading systems [10, Ch. 2, Appx.], have not been previously considered in these research models.

Main contributions. The main contributions of this paper are as follows:

- a constraint-based model, for the first time, of a non-theoretical continuous double auction;
- results that show the success of a constraint programming approach to the test oracle problem in financial systems, including the detection of several documentation defects and two failures caused by relevant bugs in the system under test;
- empirical evidence of the feasibility and cost-effectiveness of using formal models in the verification of these systems.

Plan of the paper. Section 2 explains how the continuous double auction works in *TRADExpress*. Section 3 defines the constraint programming model used as a test oracle. Section 4 gives measures related to the main characteristics of the test oracle and results obtained from the execution of random tests. Section 5 concludes and discusses future perspectives.

2 *TRADExpress* and the Continuous Double Auction

This section introduces the main elements related to the continuous double auction, and shows the details of the *TRADExpress* implementation which are necessary to understand the developed model.

The order book. The component of a trading system where different trade orders referring to a certain item are stored and matched is called *order book*. The order book is usually represented as a table with two columns called *sides*. These columns respectively contain the buy (*bid*) orders, and the sell (*ask*) orders sorted in descending priority order (see Fig. 2).

Bid orders	**Ask orders**
b_0 : bid order with highest priority	a_0 : ask order with highest priority
⋮	⋮
b_n : bid order with lowest priority	a_m : ask order with lowest priority

Fig. 2. An order book

Trade orders. A trade order is a request to buy or sell a certain quantity of a financial item. A *TRADExpress* order has the following main attributes:

- Side: bid or ask. The *opposite side* of “ask” refers to “bid” and vice versa.

- Quantity q: amount of units to trade.
- Minimum quantity mq: minimum part of the order quantity that must be matched for the order to be allowed to trade.
- Limit price lp: worst price (highest if the order is a bid, lowest if the order is an ask) that the order can accept in a trade. It can be either fixed by the trader or derived by the system from the state of the order book: *market price*: the order gets a limit price so that it can match any other order in the order book; *pegged price*: the order gets the same limit price as the best order with a fixed limit price on its side.
- Duration: validity period of the order. Ranges from no limit to a single match opportunity, where the order gets an opportunity to trade and is immediately canceled if it does not succeed.

An order is written as q ($\geq mq$) @ lp, where lp is determined in one of the three ways mentioned above. These attributes can be combined to form the most common order types (listed in Table 1).

Table 1. Common order types and their attribute values

Order type	mq	lp	**Duration**
Market	0	market price	one match
Limit	0	fixed	trading day
Fill-or-kill	q	fixed	one match
Fill-and-kill	0	fixed	one match
All-or-none	q	fixed	trading day
Pegged	0	pegged price	trading day

To encourage orders that increase the liquidity of the order book, the following prioritization criteria, sorted by precedence, are applied to each side:

1. Limit price: orders with better limit prices (higher for bid orders, lower for ask orders) get higher priority.
2. Minimum quantity: orders without minimum quantity get higher priority.
3. Time of entry: orders entered earlier get higher priority.

Matching mechanism. Continuous double auction is the applied trading mechanism during most of the execution time of *TRADExpress*. In a continuous double auction, traders are allowed to enter, update, and cancel orders in the order book at any time. Whenever the trading rules allow it, bid and ask orders are matched into trades.

The attributes of an order and its position in the order book determine if the order can be matched against orders on the opposite side. In a valid matching, minimum quantity constraints must always be satisfied, and the limit prices between matched orders must be compatible. Order priority is enforced

by allowing an order to match only if all the orders with higher priority and no minimum quantity are matched as well.

Matching in *TRADExpress* is done in two steps. Every time a trade order is entered or updated, a match attempt is performed, where the incoming order is greedily compared with the orders on the opposite side. For performance reasons, the capacity of the match to find trades is limited. Therefore, if the system detects that there are still potential trades, a second match attempt called *re-match* is performed. In the re-match, all possible combinations between orders from both sides are explored (see Fig. 3).

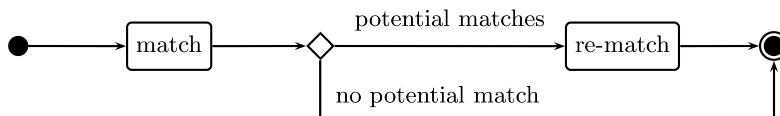

Fig. 3. Two-step matching in the *TRADExpress*' continuous double auction

In some scenarios, an order with high priority might block potential trades where orders with lower priority are involved. This might happen, for example, if the high-priority order cannot add its quantity to any other order on its side for trading against an all-or-none order. In these cases, a cancellation of the high-priority order gives way for the blocked orders to match. Because of this, TRADExpress executes a re-match after an order cancellation.

3 The Test Oracle

To be able to provide the expected trades after every order action, the test oracle holds a model of the state of the *TRADExpress* order book. In this model, orders of all types shown in Table 1 are represented by their quantity q, minimum quantity mq, limit price lp, and position in the order book.

Every time an order is entered, updated, or canceled, the order book model is updated accordingly, and the limit price lp of each order is calculated following the rules given in Section 2. Then, a corresponding order matcher problem is generated and solved with the aid of a constraint programming system. The order book model is updated with the results, and the process is repeated for the re-matcher problem. Finally, the combination of calculated trades for both problems is compared with the output generated by *TRADExpress*. Fig. 4 shows the structure of the test oracle.

3.1 The Order Matching Framework

The order matching framework defines the integer variables that represent the quantity matched between different orders, some auxiliary variables that are useful in the modeling of more advanced rules and constraints that set the basic limitations in how quantities can be distributed.

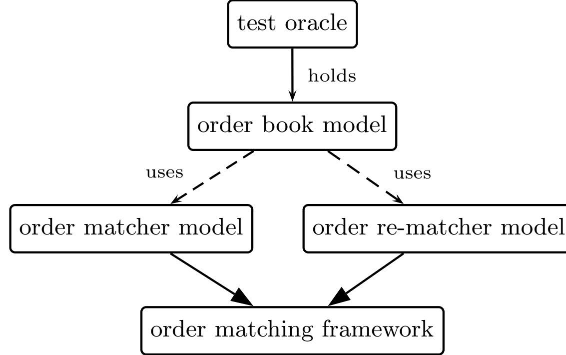

Fig. 4. Structure of the test oracle

Input data. An order book with n bid orders and m ask orders sorted by decreasing priority $(n, m \geq 1)$:

Bid orders	**Ask orders**
$b_0: \; q_{b_0} \; (\geq mq_{b_0}) \; @ \; lp_{b_0}$	$a_0: \; q_{a_0} \; (\geq mq_{a_0}) \; @ \; lp_{a_0}$
$\vdots$	$\vdots$
$b_{n-1}: \; q_{b_{n-1}} \; (\geq mq_{b_{n-1}}) \; @ \; lp_{b_{n-1}}$	$a_{m-1}: \; q_{a_{m-1}} \; (\geq mq_{a_{m-1}}) \; @ \; lp_{a_{m-1}}$

where q_b, mq_b and lp_b represent the quantity, minimum quantity and limit price of the bid order b. For the sake of simplicity, we define the set of bid orders $B = \{b_i \mid 0 \leq i < n\}$ and the set of ask orders $A = \{a_j \mid 0 \leq j < m\}$.

Variables. The following non-negative integer variables represent how the quantities of the given orders are distributed in a match:

	$\boldsymbol{a_0}$	**...**	$\boldsymbol{a_{m-1}}$	**total**
$\boldsymbol{b_0}$	tq_{b_0,a_0}	...	$tq_{b_0,a_{m-1}}$	tq_{b_0}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\boldsymbol{b_{n-1}}$	tq_{b_{n-1},a_0}	...	$tq_{b_{n-1},a_{m-1}}$	$tq_{b_{n-1}}$
total	tq_{a_0}	...	$tq_{a_{m-1}}$	tq

where $tq_{b,a}$ represents the quantity traded between the bid order b and the ask order a. The auxiliary variables tq_b, tq_a, and tq represent the total traded quantity of the bid order b, the ask order a, and between all orders:

$$tq_b = \sum_{a \in A} tq_{b,a} \quad \forall b \in B \quad \text{and} \quad tq_a = \sum_{b \in B} tq_{b,a} \quad \forall a \in A$$

$$tq = \sum_{b \in B} tq_b = \sum_{a \in A} tq_a$$

Constraints. The framework constraints impose the basic rules on how quantities can be distributed in a generic match problem, considering limit prices, maximum and minimum quantities, and order priorities.

Limit price. Two orders can only match if the limit price of the bid order is greater or equal than the limit price of the ask order:

$$lp_b < lp_a \implies tq_{b,a} = 0 \quad \forall b \in B, a \in A \tag{1}$$

Maximum quantity. Orders cannot trade over their maximum quantities:

$$tq_x \leq q_x \quad \forall x \in B \cup A$$

Minimum quantity. Orders can only trade above their minimum quantities or not at all:

$$tq_x \geq mq_x \vee tq_x = 0 \quad \forall x \in B \cup A \tag{2}$$

Order priority. An order without minimum quantity constraint cannot be bypassed by another order with lower priority:

$$tq_{b_i} < q_{b_i} \wedge \ mq_{b_i} = 0 \implies tq_{b_j} = 0 \qquad \forall b_i, b_j \in B : i < j \tag{3}$$

$$tq_{a_i} < q_{a_i} \wedge \ mq_{a_i} = 0 \implies tq_{a_j} = 0 \qquad \forall a_i, a_j \in A : i < j \tag{4}$$

3.2 The Order Matcher Model

The order matcher model corresponds to the first step in the *TRADExpress*' continuous double auction. In this problem, the incoming order is sequentially matched against the opposite side, starting by the order with highest priority.

Input data. The input data to the order matcher model is the order book inherited from the matching framework and the incoming order c together with its side $C : c \in C, \ C \in \{B, A\}$.

Variables. The order matcher uses only the traded quantity variables inherited from the matching framework (as described in the previous subsection).

Objective function. The solution must maximize the total traded quantity:

$$\text{maximize}(tq) \tag{5}$$

Constraints. The order matcher constraints impose that only the incoming order can trade on its side and the matching is performed sequentially.

Incoming order. The incoming order c is the only one that can trade on its side:

$$tq_x = 0 \;\; \forall x \in C - \{c\} \tag{6}$$

Sequential matching. An order on the non-incoming side whose quantity, added to the accumulated traded quantity of orders with higher priority, still fits into the quantity of the incoming order c cannot be bypassed by orders with lower priority:

$$c \in A \;\wedge \mathrm{atq}(b_i) + q_{b_i} \leq q_c \;\wedge tq_{b_i} < q_{b_i} \implies tq_{b_j} = 0 \;\; \forall b_i, b_j \in B : i < j \tag{7}$$
$$c \in B \;\wedge \mathrm{atq}(a_i) + q_{a_i} \leq q_c \;\wedge tq_{a_i} < q_{a_i} \implies tq_{a_j} = 0 \;\; \forall a_i, a_j \in A : i < j \tag{8}$$

where $\mathrm{atq}(x_i)$ is a function from orders to integers that represents the accumulated traded quantity from x_0 to x_{i-1}:

$$\mathrm{atq}(x_0) = 0$$
$$\mathrm{atq}(x_i) = \begin{cases} \mathrm{atq}(x_{i-1}) + q_{x_{i-1}} & \text{if } \mathrm{atq}(x_{i-1}) + q_{x_{i-1}} \leq q_c \\ \mathrm{atq}(x_{i-1}) & \text{otherwise} \end{cases} \quad \text{for } i > 0$$

Example. Let us consider the following order book, where the incoming order a_0 has received the highest priority on the ask side due to its limit price:

Bid orders	**Ask orders**
$b_0 : 10 \,@\, 10$	$\rightarrow a_0 : 18 \,@\, 8$
$b_1 : 10\ (\geq 10) \,@\, 9$	$a_1 : 30\ (\geq 30) \,@\, 9$
$b_2 : 5\ (\geq 5) \,@\, 9$	
$b_3 : 3 \,@\, 7$	

The only order allowed to trade on the ask side is a_0 (6). To maximize the total traded quantity (5), b_0 must trade all its quantity. Otherwise, the lower priority orders b_1, b_2 and b_3 would not be allowed to match at all (3). Given that b_0 trades all its quantity, b_1 cannot trade due to its minimum quantity constraint (2). Finally, b_2 contributes to the objective by trading all its quantity. The limit price of b_3 is not compatible with that of a_0 (1). Hence, the expected trades are:

$$b_0 \leftrightarrow a_0 : 10 \qquad\qquad b_2 \leftrightarrow a_0 : 5$$

As this example illustrates, the order matcher always finds a single valid solution. This is enforced by the order priority (3, 4) and sequential matching (7, 8) constraints.

3.3 The Order Re-matcher Model

The order re-matcher model corresponds to the second step in *TRADExpress*' continuous double auction (see Fig. 3). This problem is used as a complement to the order matcher when this, due to its restrictive constraints, is not able to calculate all potential trades in the order book. Although both problems share the same basic structure, their variables and constraints differ. Because of this, it is not possible for the order re-matcher to reuse the solution to the order match problem.

In an order re-match, all possible combinations between order quantities are considered. All orders participating in trades must have a limit price compatible with a reference price, called the *equilibrium price*, which is selected to maximize the objective function. Orders without minimum quantity constraints and better limit price than the equilibrium price are called *must trade* orders, because all their quantity must be matched in a valid solution.

Input data. The input data is the order book inherited from the general matching framework.

Variables. Apart from the traded quantity variables inherited from the matching framework, the following variables are added for re-matching:

- A non-negative integer variable eqp, which represents the equilibrium price of the order matching and whose domain is the set of different order limit prices $\{lp_x \mid x \in B \cup A\}$.
- An integer variable im representing the imbalance in a calculated matching:

$$im = \sum_{\substack{b \in B: \\ lp_b = eqp, mq_b = 0}} (q_b - tq_b) - \sum_{\substack{a \in A: \\ lp_a = eqp, mq_a = 0}} (q_a - tq_a) \quad (9)$$

- Non-negative integer variables $tq_0, \ldots, tq_{n+m-1}$, where tq_k represents the total traded quantity between orders whose indexes sum up to k:

$$tq_k = \sum_{\substack{b_i \in B, a_j \in A: \\ i+j=k}} tq_{b_i,a_j} \quad \forall k : 0 \leq k < n+m$$

  The main purpose of these variables is to reflect the total traded quantity in different priority levels. Graphically, each priority level tq_k corresponds to a counter-diagonal in the traded quantity matrix. For example, a re-match problem with $n = m = 3$ has five priority levels. tq_0 corresponds to the darkest diagonal in the following representation:

	a_0	a_1	a_2
b_0	tq_{b_0,a_0}	tq_{b_0,a_1}	tq_{b_0,a_2}
b_1	tq_{b_1,a_0}	tq_{b_1,a_1}	tq_{b_1,a_2}
b_2	tq_{b_2,a_0}	tq_{b_2,a_1}	tq_{b_2,a_2}

Objective function. In contrast to the order matcher problem, for an order re-match problem it is possible to obtain several solutions that maximize the total traded quantity tq. If this happens, it is desirable to get a solution where the unmatched quantity of orders in the equilibrium price level is balanced on both sides of the order book. This can be obtained by minimizing the absolute imbalance $|im|$. If several solutions present the same total traded quantity and absolute imbalance, trades between orders positioned higher in the order book are prioritized by lexicographically maximizing the different priority levels $tq_0, \dots, tq_{n+m-1}$.

In summary, the solution must lexicographically maximize the following tuple of integer variables (note that the absolute imbalance $|im|$ is to be minimized):

$$\text{maximize}(\langle tq,\ -|im|,\ tq_0, \dots, tq_{n+m-1}\rangle) \tag{10}$$

Constraints. The order re-matcher constraints define the effect of the equilibrium price on which orders can trade and impose that the whole quantity from *must trade* orders is always traded.

Equilibrium price. An order that has worse limit price than the equilibrium price eqp cannot trade:

$$lp_b < eqp \implies tq_b = 0 \quad \forall b \in B$$

$$lp_a > eqp \implies tq_a = 0 \quad \forall a \in A \tag{11}$$

Must trade quantity. The quantity from all *must trade* orders on both sides must be completely matched:

$$\sum_{\substack{b \in B: \\ lp_b > eqp, mq_b = 0}} (q_b - tq_b) = 0 \qquad \text{and} \qquad \sum_{\substack{a \in A: \\ lp_a < eqp, mq_a = 0}} (q_a - tq_a) = 0 \tag{12}$$

Example. Let us consider the following order book:

Bid orders	**Ask orders**
$b_0 : 10\ (\geq 10)$ @ 19	$a_0 : 6\ (\geq 6)$ @ 12
	$a_1 : 9\ (\geq 9)$ @ 12
	$a_2 : 5$ @ 16
	$a_3 : 5\ (\geq 5)$ @ 17

There are several possible pairs of ask orders that can trade the total quantity of b_0, maximizing the total traded quantity (10):

Pair	eqp	$\lvert im\rvert$
$\{a_0, a_2\}$	16	1
$\{a_1, a_2\}$	16	4
$\{a_2, a_3\}$	17	0

If one of the two first pairs is chosen, the equilibrium price is set to 16, so that a_2 can trade (11). In that case $lp_{a_2} = eqp$, and a_2 adds its unmatched quantity to the imbalance (9). If the pair $\{a_2, a_3\}$ is chosen, the equilibrium price becomes 17, and a_2 becomes a *must trade* order, no longer contributing to the imbalance. Because the must trade constraint (12) is satisfied, and the absolute imbalance $|im|$ is minimized (10), the pair $\{a_2, a_3\}$ is chosen to trade with b_0:

$$b_0 \leftrightarrow a_2 : 5 \qquad\qquad b_0 \leftrightarrow a_3 : 5$$

3.4 Why Constraint Programming?

As mentioned in the introduction, accuracy and low development cost are two essential goals in the design of test oracles. These goals require that the implementation of the test oracle is kept simple and easily traceable to the specification. The main reason for using constraint programming has been therefore the ease of implementing every concept from the matcher and re-matcher models, including logical expressions (3) and lexicographic optimization (10).

In particular, the chosen system (Gecode) has allowed us to express the lexicographical optimization straightforwardly, by adding new lexicographic constraints every time a better solution is found. Another relevant factor in the decision to use constraint programming has been the availability of solvers distributed as object-oriented libraries. This feature has contributed to a simpler and thus less error-prone interface with Cinnober's JUnit-based test framework.

4 Results

We evaluate the constraint programming approach to the test oracle problem from two angles. First, we show the failures detected by the oracle in the execution of random tests against *TRADExpress*. Then, we complete the analysis by giving results related to the design goals of the test oracle.

The constraint models used by the test oracle have been implemented in the constraint programming system Gecode [11], version 3.2.0. The branching strategy for all variables in both models has been to branch on the variable with the smallest domain. The values are selected by splitting the domain into two subsets, exploring the larger subset first. The search has been executed on a single core. All tests and measurements have been performed on a Linux machine with a Quad-Core Intel Xeon 2.5 GHz processor and 4 GB of main memory.

Random tests. We have considered a scenario designed to exploit all capabilities of the test oracle. It includes three traders entering, updating, and canceling random orders of different types. Table 2 shows the designed configuration, with the probabilities of traders and actions being chosen in each cycle.

We have run 500 test cases with different random generator seeds. Each test case comprises 100 order actions each, with prices generated in a range between 10 and 100, and quantities generated in a range between 2 and 50.

Table 2. Random traders, actions and probabilities in the executed test cases

Trader	Probability	Action	Probability
A	30%	Enter a limit order	80%
		Update an order	10%
		Cancel an order	10%
B	30%	Enter a marker order	33.3%
		Enter a fill-or-kill order	33.3%
		Enter a fill-and-kill order	33.3%
C	40%	Enter an all-or-none order	40%
		Enter a pegged order	40%
		Update an order	10%
		Cancel an order	10%

Table 3. Total match and re-match hits and expected trades in the test runs

Matching step	Total hits	Total trades
Match	10 895	17 037
Re-match	526	1189

Although we planned a total of 50 000 order actions, premature terminations in several test cases due to the detection of failures have reduced the total order actions to 45 144. Table 3 shows the total match and re-match calculations with positive traded quantity (called *hits*) and their corresponding expected trades.

Table 4. MTBF of the failures detected in the random test cases

Failure	Occurrences	MTBF (order actions)
Pegged order in empty order book	51	885
Incoming low priority order matches	37	1220
Total	88	513

After applying manual analysis to the failing test cases, we were able to identify two types of failures in *TRADExpress* reported by the test oracle. In order to better understand the corresponding risks, we have calculated the individual and total Mean Time Between Failures (MTBF) during the execution of the random tests. The results, expressed in number of order actions, are shown in Table 4. Further detail about each type of failure can be found in [12, Ch. 6]:

A pegged order remains in an empty order book. An invariant of the order book in *TRADExpress* is that a side that does not contain any limit order cannot contain a pegged order, because pegged orders require limit orders to base their price on. This invariant is verified by the test oracle at the end of the two-matching step calculations, and was violated in several test scenarios.

An incoming order without best priority is allowed to match. In some specific situations, constraints 3 and 4 in Section 3.2 were violated by *TRADExpress*, allowing orders without the best priority to match when higher-priority orders did not.

These two failures correspond to bugs in the core of the *TRADExpress* matching logic. Both bugs have been reported and fixed by the time of writing this paper. These bugs had been most likely present in the system for several years. However, the infrequent test cases needed to reproduce them had not been generated in more systematic testing approaches. Previous random testing approaches were not able to detect them either, because of the lack of a complete test oracle [3, Ch. 6]. The detection of these failures can be seen, thus, as empirical evidence of the effectiveness of the constraint programming approach to the test oracle problem.

Furthermore, in the requirements formalization process for formulating the constraint model, we have detected six documentation defects, classified into two categories: ambiguous statements due to the use of a natural language (English); erroneous calculations in illustrative examples.

Main goals of the test oracle. As mentioned in the introduction, the main goals for the design of the oracle have been completeness, correctness, accuracy, and low development cost. Performance has been a secondary goal, because the execution of the random tests does not require manual intervention and is not subject to time pressure.

Completeness. Table 1 shows the main order types supported by *TRADExpress*. A more complete list is given in [13]. The order types *Good till Date*, *Good for Day*, *Good till Canceled*, and *Good until Next Uncross* are special cases of the limit order, differing only in their duration, and are thus covered by the test oracle.

The test oracle covers 10 out of the 13 order types listed in [13], that is, a 77% of the order types supported by *TRADExpress*. Extending the model to support the remaining *Stop-loss*, *Iceberg* and *Dark* order types is left as future work.

Accuracy and correctness. The use of a declarative programming paradigm such as constraint programming reduces the proof of correctness of the continuous double auction model to verifying the specification itself. Furthermore, it makes the oracle more independent from the system under test, which is developed following an imperative paradigm. This independence contributes to the effectiveness of the test oracle, as it reduces the likelihood of sharing design and implementation bugs with the system under test [4].

Development cost. The development of the test oracle, including modeling, implementation and testing, took approximately 200 person-hours. All used technologies, including the constraint programming system, are freely available. We

have estimated the maintainability by counting lines of code [14]. The test oracle has 1946 lines of code, approximately only 20% of the code that implements the modeled functionality in *TRADExpress*. Furthermore, the declarative nature of the model makes it easy to trace changes in the system requirements, which contributes to lower maintenance costs.

Performance. We have measured the average execution time taken by Gecode to solve the different order match and re-match problems generated during the execution of 60 test cases based on the scenario shown in Table 2. Due to the limited accuracy of the time measurement functions in the considered orders of magnitude, the average time of 50 executions has been taken for each match and re-match problem.

In a total of 4256 instances, the order match problem is solved in 161 μs on average, with a coefficient of deviation of 85%. The order re-match problem as a more complex combinatorial problem takes 388 μs on average, with a coefficient of deviation of 61%, for a total of 3534 instances. Some degenerate re-match problems designed to stress-test the oracle take up to 155 ms to be solved, which is still an order of magnitude more efficient than our initial design goal.

5 Conclusion and Future Work

This paper has introduced a constraint-based test oracle for a continuous double auction from a real-life trading system. It has shown that constraint programming meets the particular goals for the design of test oracles, providing accuracy and ease of modeling in a cost-effective way. The significance of this approach is witnessed by finding actual, relevant bugs in a widely-deployed, thoroughly tested trading system. As a side benefit, several defects in the system requirements have been found.

The results obtained in this paper support the importance of using formal models in the development of complex financial systems such as *TRADExpress*. Constraint programming adds the ability to make these models executable with low development cost. This application improves significantly the effectiveness of the testing process, and has raised interest at Cinnober: the company plans to continue the formalization process initiated by this research and to use the test oracle for regression purposes.

An obvious way to improve the effectiveness of the test oracle is to extend the continuous double auction model to cover additional order types supported by *TRADExpress*, as suggested in Sect. 4. Considering a more global perspective, the emergence of open protocols such as the Financial Information eXchange protocol (FIX) [15] opens an opportunity to extend the application presented in this paper to a variety of trading systems. Future work in this direction would include the development of a constraint-based test oracle implementing a relevant subset of a widely adopted protocol such as FIX.

Acknowledgments. The authors are grateful for helpful comments from Mikael Z. Lagerkvist, Carles Tomás Martí and the anonymous reviewers.

References

[1] Wagner, W.H.: Electronic trading: rival or replacement for traditional floor-based exchanges? In: World of Exchanges: Adapting to a New Environment. Euromoney Books (2007)

[2] Friedman, D., Rust, J., eds.: The double auction market: institutions, theories, and evidence. Addison-Wesley (1993)

[3] Höjeberg, N.: Random tests in a trading system: random tests in a trading system using simulations and a test oracle. Master's thesis, School of Computer Science and Communication, KTH Royal Institute of Technology, Sweden (2008)

[4] Hoffman, D.: Using oracles in test automation. Proceedings of Pacific Northwest Software Quality Conference (2001) 90–117

[5] Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. SIGSOFT Softw. Eng. Notes **23**(2) (1998) 53–62

[6] Meudec, C.: ATGen: automatic test data generation using constraint logic programming and symbolic execution. Software Testing Verification and Reliability **11**(2) (2001) 81–96

[7] Bin, E., Emek, R., Shurek, G., Ziv, A.: Using a constraint satisfaction formulation and solution techniques for random test program generation. IBM Systems Journal **41**(3) (2002) 386–402

[8] Ryu, Y.U.: Hierarchical constraint satisfaction of multilateral trade matching in commodity auction markets. Annals of Operations Research **71**(0) (1997) 317–334

[9] Kalagnanam, J.R., Davenport, A.J., Lee, H.S.: Computational aspects of clearing continuous call double auctions with assignment constraints and indivisible demand. Electronic Commerce Research **1**(3) (2001) 221–238

[10] Hasbrouck, J.: Empirical Market Microstructure: The Institutions, Economics, and Econometrics of Securities Trading. Oxford University Press (2007)

[11] Gecode Team: Gecode: Generic constraint development environment. `www.gecode.org` (2006)

[12] Castañeda Lozano, R.: Constraint programming for random testing of a trading system. Master's thesis, School of Information and Communication Technology, KTH Royal Institute of Technology, Sweden (2010)

[13] Cinnober Financial Technology AB: TRADExpress Trading System product sheet. `http://www.cinnober.com/files/TS_ProductSheet_0.pdf` (2010)

[14] Zuse, H.: A Framework of Software Measurement. Walter de Gruyter & Co., Hawthorne, NJ, USA (1997)

[15] FIX Protocol Limited: Financial Information eXchange Protocol. `www.fixprotocol.org` (2010)